NTNU
Innovation and Creativity

Experimentation with inverted indexes for dynamic document collections

Truls Amundsen Bjørklund

Master of Science in Computer Science
Submission date: August 2007
Supervisor: Bjørn Olstad, IDI
Co-supervisor: Øystein Torbjørnsen, FAST Search & Transfer ASA

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The purpose of this project is to investigate the efficiency of various inverted index structures when used for indexing dynamic document collections.

The project should give both theoretical models for the efficiency of the investigated indexes and provide results from experiments that test the validity of the theoretical models and whether these indexes perform better in practice than other well-known structures.

Assignment given: 15. September 2006
Supervisor: Bjørn Olstad, IDI

Abstract

This report aims to asses the efficiency of various inverted indexes when the indexed document collection is dynamic. To achieve this goal, we experiment with three different overall structures: Remerge, hierarchical indexes and a naive B-tree index. An efficiency model is also developed. The resulting estimates for each structure from the efficiency model are compared to the actual results.

We introduce two modifications to existing methods. The first is a new scheme for accumulating an index in memory during sort-based inversion. Even though the memory characteristics of this modified scheme are attractive, our experiments suggest that other proposed methods are more efficient. We also introduce a modification to the hierarchical indexes, which makes them more flexible.

Tf-idf is used as the ranking scheme in all tested methods. Approximations to this scheme are suggested to make it more efficient in an updatable index.

We conclude that in our implementation, the hierarchical index with the modification we have suggested performs best overall. We also conclude that the tf-idf ranking scheme is not fit for updatable indexes. The major problem with using the scheme is that it becomes difficult to make documents searchable immediately without sacrificing update speed.

Preface

This is a master thesis for the Master of Technology program at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU).

I would like to thank my supervisor Øystein Torbjørnsen for valuable feedback and suggestions. His extensive experience has been of great help in all stages of this work. I would also like to thank Nils Grimsmo (IDI) and Hans Christian Falkenberg (Fast Search & Transfer) for fruitful discussions and valuable comments.

Contents

Preface **2**

1 Introduction **15**

2 Background and problem statement **17**
2.1 Notation . . . 17
2.2 Inverted indexes . . . 19
2.2.1 Definition of an inverted index . . . 20
2.2.2 Building inverted indexes . . . 21
2.2.3 Compression . . . 22
2.2.4 Ranking in search engines . . . 23
2.2.4.1 Evaluating ranked queries . . . 24
2.2.5 Zipf's law and Heap's law . . . 25
2.3 The problem . . . 26
2.4 Summary of [Bjø06] . . . 26
2.4.1 Criteria for evaluating updatable indexes . . . 26
2.4.2 Evaluated methods . . . 27
2.4.2.1 Rebuild . . . 27
2.4.2.2 Remerge . . . 28
2.4.2.3 Hierarchical index . . . 28
2.4.2.4 Simple incrementally updatable index . . . 31
2.4.2.5 Methods using B-trees . . . 33
2.4.2.6 Continuous update . . . 35
2.4.2.7 Dual-structure index . . . 36
2.4.3 Further considerations and experiments . . . 37

3 State of the art **39**
3.1 Eager, Piggyback and Batch . . . 39
3.1.1 Comments . . . 40
3.2 Experimentation with B-trees . . . 41
3.2.1 Experiments in [LZW06] . . . 42
3.2.2 Comments . . . 43
3.3 Hybrid methods . . . 44
3.3.1 Geometric partitioning . . . 44
3.3.2 Logarithmic merge . . . 44
3.3.3 A first set of hybrid methods . . . 45

3.3.4 Improving hybrid methods . . . 46
3.3.5 Comments . . . 47

4 Implementation **49**
4.1 Implemented methods and overall choices . . . 49
4.1.1 Rationale behind the methods chosen . . . 50
4.1.2 Why implement a Search Engine from scratch? . . . 51
4.2 The new Brille . . . 52
4.2.1 Overall design principles . . . 52
4.2.2 The architecture of Brille . . . 54
4.2.3 Remerge . . . 57
4.2.3.1 Chosen dictionary . . . 57
4.2.3.2 In-memory accumulation . . . 58
4.2.4 Hierarchical index . . . 61
4.2.5 Naive B-tree index . . . 62
4.2.6 Document manager . . . 63
4.2.6.1 Limiting the main memory requirement . . . 64
4.2.6.2 Adaptation to hierarchical index . . . 67
4.2.6.3 Adaptation to an incrementally updatable index . . . 68
4.2.7 Buffer Pool . . . 69
4.2.8 Implementing a B-tree . . . 71

5 Experiments **73**
5.1 Document collection . . . 73
5.1.1 GOV2 . . . 73
5.1.2 Parsing GOV2 . . . 74
5.2 Obtaining I/O and CPU statistics . . . 74
5.3 Experiments performed . . . 75
5.3.1 Testing updatability . . . 75
5.3.2 Testing search performance . . . 76
5.3.3 Tested configurations . . . 77
5.3.4 Test output intrusiveness . . . 78

6 Efficiency model **79**
6.1 Model of computation . . . 79
6.1.1 Disk performance . . . 80
6.1.2 CPU performance . . . 81
6.1.2.1 Estimation based on a dumped dictionary . . . 82
6.1.2.2 Estimation based on running the application . . . 83
6.1.2.3 Resulting estimates . . . 86
6.1.3 Some notes on the experiment environment . . . 86
6.2 Constructing partial indexes . . . 87
6.2.1 In-memory accumulation . . . 87
6.2.1.1 Estimating time spent accessing disk . . . 87
6.2.1.2 Estimating time spent processing . . . 90
6.2.2 Sorting and flushing . . . 91

6.2.2.1 Estimating time spent accessing disk 91
6.2.2.2 Estimating time spent processing 92
6.2.3 The phase as a whole 92
6.3 Merging indexes 93
6.3.1 Time spent accessing disk 93
6.3.1.1 Estimate 1 94
6.3.1.2 Estimate 2 95
6.3.2 Time spent processing 95
6.4 Remerge 96
6.4.1 Off-line construction 97
6.4.1.1 Merging indexes with estimate 1 98
6.4.1.2 Merging indexes with estimate 2 99
6.4.2 Immediate merge 99
6.4.2.1 Disk estimate 1 101
6.4.2.2 Disk estimate 2 101
6.4.2.3 Resulting estimates 102
6.5 Hierarchical index 103
6.5.1 Expected merges 104
6.5.2 Constructing partial indexes 104
6.5.3 Processing merges 107
6.5.4 Time spent accessing disk in merges 108
6.5.5 Expected times for hierarchical index construction 109
6.6 Naive B-tree index 110
6.6.1 The size of a complete B-tree index 111
6.6.2 Reading and parsing documents 113
6.6.3 Constructing the index 113
6.6.4 Disk accesses in the naive B-tree index 114
6.6.5 Estimates for $N = 300000$ 115
6.7 Expected search performance 118
6.7.1 Remerge 119
6.7.2 Hierarchical index 120
6.7.3 Naive B-tree index 122

7 Results 125
7.1 Experiment environment 125
7.2 Remerge 126
7.2.1 Update speed 126
7.2.2 Update latency 128
7.2.3 Search performance 130
7.2.4 Actual results versus estimates 132
7.2.4.1 Off-line construction 132
7.2.4.2 Immediate merge 136
7.2.4.3 Search performance 136
7.2.5 Discussion of results 139
7.2.6 Chosen configuration 141
7.3 Hierarchical indexes 142

7.3.1 Update speed . . . 143
7.3.2 Update latency . . . 146
7.3.3 Search performance . . . 148
7.3.4 Actual results versus experiments . . . 149
7.3.4.1 Update speed . . . 149
7.3.4.2 Search performance . . . 154
7.3.5 Discussion of results . . . 155
7.3.6 Chosen configuration . . . 157
7.4 Naive B-tree index . . . 157
7.4.1 Update speed . . . 157
7.4.2 Update latency . . . 159
7.4.3 Search performance . . . 160
7.4.4 Actual results versus estimates . . . 162
7.4.4.1 Update speed . . . 162
7.4.4.2 Search performance . . . 164
7.4.5 Discussion of results . . . 166
7.4.6 Chosen configuration . . . 167
7.5 Comparison of the chosen configurations . . . 167
7.5.1 Updatability . . . 168
7.5.2 Search performance . . . 170
7.5.3 Conclusion . . . 170
7.6 Intrusiveness of output . . . 172
7.7 Possible future improvements of implementation . . . 173

8 Conclusion **177**
8.1 Further work . . . 178

A Terms searched for in experiments **182**
A.1 Frequent terms . . . 182
A.2 Terms with medium frequency . . . 184
A.3 Terms with one occurrence . . . 187

B Logs from selected experiments **190**
B.1 Remerge as off-line construction . . . 190
B.2 Remerge with immediate merge . . . 223
B.3 Hierarchical index . . . 267
B.4 Naive B-tree index . . . 314

C Code **331**
C.1 brille . . . 331
C.1.1 brille.BrilleDefinitions . . . 331
C.1.2 brille.BTreeFeedingThread . . . 333
C.1.3 brille.BTreeIndexMaster . . . 335
C.1.4 brille.DelFromBPThread . . . 339
C.1.5 brille.HierarchicIndex . . . 341
C.1.6 brille.HierarchicIndexMaster . . . 343

C.1.7 brille.Indexer . . . 352
C.1.8 brille.NotImplementedException . . . 353
C.1.9 brille.PerformanceTestingOfBrille . . . 353
C.1.10 brille.RemergeIndexMaster . . . 358
C.1.11 brille.Result . . . 365
C.1.12 brille.ResultHeap . . . 366
C.1.13 brille.Searcher . . . 368
C.1.14 brille.SearchResultHandle . . . 368
C.1.15 brille.SearchResultMerger . . . 369
C.2 brille.btree . . . 372
C.2.1 brille.btree.BTree . . . 372
C.2.2 brille.btree.BTreeHandle . . . 388
C.2.3 brille.btree.Entry . . . 389
C.2.4 brille.btree.EntryFactory . . . 391
C.2.5 brille.btree.InternalEntry . . . 392
C.2.6 brille.btree.InternalEntryFactory . . . 392
C.2.7 brille.btree.Node . . . 393
C.2.8 brille.btree.Update . . . 406
C.2.9 brille.btree.UpdateThread . . . 407
C.3 brille.buffering . . . 409
C.3.1 brille.buffering.Buffer . . . 409
C.3.2 brille.buffering.BufferFIFOQueue . . . 414
C.3.3 brille.buffering.BufferPool . . . 416
C.3.4 brille.buffering.BuffPool . . . 419
C.3.5 brille.buffering.FlushingThread . . . 421
C.3.6 brille.buffering.LinkedListElement . . . 422
C.3.7 brille.buffering.NewBufferPool . . . 424
C.3.8 brille.buffering.Pair . . . 428
C.3.9 brille.buffering.Pin . . . 428
C.3.10 brille.buffering.UndirtyThread . . . 429
C.4 brille.dict . . . 431
C.4.1 brille.dict.BTreeSearchResultHandle . . . 431
C.4.2 brille.dict.Dictionary . . . 432
C.4.3 brille.dict.DictIteratorHeap . . . 433
C.4.4 brille.dict.ExtendableDocTermEntry . . . 434
C.4.5 brille.dict.FullBTreeIndex . . . 435
C.4.6 brille.dict.FullBTreeIndexInternalEntry . . . 438
C.4.7 brille.dict.FullBTreeIndexInternalEntryFactory . . . 441
C.4.8 brille.dict.FullBTreeIndexLeafEntry . . . 442
C.4.9 brille.dict.FullBTreeIndexLeafEntryFactory . . . 446
C.4.10 brille.dict.InMemPartialIndex . . . 447
C.4.11 brille.dict.MergerThread . . . 461
C.4.12 brille.dict.MergeSortedListDictEntry . . . 463
C.4.13 brille.dict.SortedListDictEntry . . . 464
C.4.14 brille.dict.SortedListDictionary . . . 466

C.4.15 brille.dict.SortedListDictIterator . . . 478
C.5 brille.docman . . . 481
C.5.1 brille.docman.BlackList . . . 481
C.5.2 brille.docman.DocEntry . . . 482
C.5.3 brille.docman.DocEntryFactory . . . 484
C.5.4 brille.docman.DocMan . . . 485
C.5.5 brille.docman.DocManager . . . 485
C.5.6 brille.docman.FixRankingsThread . . . 491
C.5.7 brille.docman.FreeThread . . . 493
C.5.8 brille.docman.InternalDocEntry . . . 494
C.5.9 brille.docman.InternalDocEntryFactory . . . 496
C.5.10 brille.docman.StaticDocManager . . . 496
C.6 brille.inv . . . 502
C.6.1 brille.inv.IndexEntry . . . 502
C.6.2 brille.inv.InvListIterator . . . 504
C.6.3 brille.inv.InvListMergerHeap . . . 506
C.6.4 brille.inv.MergeInvListIterator . . . 507
C.7 brille.locking . . . 510
C.7.1 brille.locking.LockManager . . . 510
C.8 brille.utils . . . 512
C.8.1 brille.utils/AlternativeGOV2FileParser . . . 512
C.8.2 brille.utils.ByteArr . . . 518
C.8.3 brille.utils.ByteArrayList . . . 521
C.8.4 brille.utils.Document . . . 523
C.8.5 brille.utils.DoubleArrayList . . . 525
C.8.6 brille.utils.FreeBSDIOStats . . . 527
C.8.7 brille.utils.FreeBSDIOStatsGatherer . . . 532
C.8.8 brille.utils.FreeBSDLibAccesser . . . 533
C.8.9 brille.utils.GOV2FileParser . . . 533
C.8.10 brille.utils.IntArrayList . . . 539
C.8.11 brille.utils.IOStats . . . 542
C.8.12 brille.utils.IOStatsGatherer . . . 543
C.8.13 brille.utils.LinuxIOStats . . . 544
C.8.14 brille.utils.LinuxIOStatsGatherer . . . 546
C.8.15 brille.utils.LongArrayList . . . 548
C.8.16 brille.utils.NoMoreDocsException . . . 550
C.9 Extra code for getting I/O-statistics on FreeBSD . . . 551
C.9.1 Header-file . . . 551
C.9.2 Implementation . . . 551

List of Figures

2.1 Example documents 21
2.2 Index for example documents 22
2.3 Overall method for building inverted indexes 22
2.4 Index with difference lists 23
2.5 Example hierarchy with method 1 and $K = 3$ 29
2.6 Example hierarchy with method 2 and $K = 4$ 30
2.7 Number of disk accesses required in a search as a function of index size 32
2.8 A naive B-tree index for the example documents 33
2.9 Snapshot of the index on disk when continuous update is used 35

4.1 Generic class diagram for Brille 54
4.2 Sequence diagram for adding a document in remerge and hierarchical indexes . 55
4.3 Sequence diagram for adding a document in the naive B-tree index 56
4.4 Sequence diagram for searching 57
4.5 In-memory accumulation of index 60
4.6 Code for the test_arraylist class 65
4.7 Code for the test_intarraylist class 66
4.8 Results when running the test_arraylist class 66
4.9 Results when running the test_intarraylist class 67
4.10 The new and improved buffer pool 70

6.1 Result from testing the maximum bandwidth from disk 80
6.2 Example results from a baseline run 86
6.3 All expected merges while building a hierarchical index with $K = 2$ and $T = 1$ for 1 million documents 105
6.4 Expected construction time for naive B-tree index as a function of hit ratio . . . 117
6.5 Resulting hierarchical index with $K = 2$, $T = 1$ and $N = 1$ million 120

7.1 Average construction time with remerge for indexes with various size 127
7.2 Sample standard deviation for construction in remerge with indexes of various size 127
7.3 Read and searchable documents over time using off-line construction, 16kB buffers and 1 million documents 129
7.4 Read and searchable documents over time using remerge with immediate merge, 4kB buffers and 1000000 documents 129
7.5 Average time spent performing 100 searches for terms with low frequency in indexes constructed by remerge 130

7.6 Average time spent performing 100 searches for terms with medium frequency in indexes constructed by remerge . . . 131
7.7 Average time spent performing 100 searches for terms with high frequency in indexes constructed by remerge . . . 131
7.8 Read and searchable documents over time using remerge with immediate merge, 4kB buffers and 10000000 documents . . . 140
7.9 Read and searchable documents over time using remerge with immediate merge, 16kB buffers and 10000000 documents . . . 141
7.10 Average construction times for hierarchical indexes with various sizes($B = 16$KB) . . . 144
7.11 Average construction times for hierarchical indexes with various sizes($B = 4$KB) 144
7.12 Sample standard deviation for construction of hierarchical indexes with various sizes ($B = 16$KB) . . . 145
7.13 Sample standard deviation for construction of hierarchical indexes with various sizes ($B = 4$KB) . . . 146
7.14 Read and searchable documents over time using hierarchical index, $K = 2$, $T = 1$, 16kB buffers and 1 million documents . . . 147
7.15 Read and searchable documents over time using hierarchical index, $K = 2$, $T = 4$, 4kB buffers and 300000 documents . . . 148
7.16 Read and searchable documents over time using hierarchical index, $K = 4$, $T = 1$, 4kB buffers and 1 million documents . . . 149
7.17 Read and searchable documents over time using hierarchical index, $K = 4$, $T = 4$, 16kB buffers and 1 million documents . . . 150
7.18 Average time spent performing 100 searches for terms with low frequency in hierarchical indexes. The figure on top is for $B = 16$KB, and the lower figure is for $B = 4$KB . . . 151
7.19 Average time spent performing 100 searches for terms with medium frequency in hierarchical indexes. The figure on top is for $B = 16$KB, and the lower figure is for $B = 4$KB . . . 152
7.20 Average time spent performing 100 searches for terms with high frequency in hierarchical indexes. The figure on top is for $B = 16$KB, and the lower figure is for $B = 4$KB . . . 153
7.21 Average construction times for naive B-tree index with various index sizes . . . 158
7.22 Sample standard deviation for construction of a naive B-tree index with various size . . . 159
7.23 Read and searchable documents over time using naive B-tree index with 4 feeding threads, 4kB buffers and 100000 documents . . . 160
7.24 Read and searchable documents over time using naive B-tree index with 1 feeding thread, 16kB buffers and 300000 documents . . . 161
7.25 Average time spent performing 100 searches for terms with low frequency in a naive B-tree index . . . 162
7.26 Average time spent performing 100 searches for terms with medium frequency in a naive B-tree index . . . 163
7.27 Average time spent performing 100 searches for terms with high frequency in a naive B-tree index . . . 164

7.28 Read and indexed documents over time in a naive B-tree index 167
7.29 Average time spent constructing indexes with the chosen configurations 168
7.30 Plot showing the update latency for the chosen configurations. The top left figure shows remerge as off-line construction. The top right figure shows remerge with immediate merge, while the bottom left shows the hierarchical index. All these first three figures have the same range on the x-axis. The bottom right figure shows the plot for the naive B-tree index 169
7.31 Average time spent performing 100 searches for terms with low (top figure), medium (middle figure) and high (bottom figure) frequency in the compared structures. 171

List of Tables

2.1 Lengths of the example documents using the tf-idf weighting scheme 25

6.1 Parameters for disk performance 81
6.2 Parameters for CPU performance 86
6.3 Expected sizes of index files for the complete index after each batch is added . 100
6.4 Time spent processing and accessing disk while merging for different estimates in remerge with immediate merge 102
6.5 Time spent processing and accessing disk while making partial indexes searchable 107
6.6 Time spent processing and accessing disk while merging for different estimates in a hierarchical index with $K = 2$ and $T = 1$ 109
6.7 Statistics for various groups of terms 119
6.8 Expected average time for searches for terms with various frequency ($N = 1$ million) 121
6.9 Adjusted statistics for the groups of terms 122
6.10 Expected average time for searches for terms with various frequency ($N = 300000$) 123

7.1 Characteristics for experiment environment 125
7.2 Average time spent constructing indexes with remerge 126
7.3 Final estimates for time spent constructing an index with $N = 1$ million documents with off-line construction 132
7.4 Estimated total size of files read and written in the merge in off-line construction 134
7.5 Average number of disk operations during merges in off-line construction . . . 135
7.6 Average amount of moved data in disk operations during merges in off-line construction 135
7.7 Final estimates for time spent constructing an index with $N = 1$ million documents using remerge with immediate merge 136
7.8 Expected time spent searching for terms with different frequencies in remerge . 137
7.9 Average time spent searching for terms with different frequencies in remerge . 137
7.10 Average time spent constructing hierarchical indexes 145
7.11 Final estimates for time spent constructing a hierarchical index with $N = 1$ million documents 150
7.12 Expected time spent searching for terms with different frequencies in a hierarchical index with $K = 2$ and $T = 1$ 154
7.13 Average time spent searching for terms with different frequencies in a hierarchical index with $K = 2$ and $T = 1$ 154
7.14 Average time spent constructing naive B-tree indexes 158

7.15 Average time spent constructing naive B-tree indexes 162
7.16 Expected average time for searches for terms with various frequency ($N = 300000$) 164
7.17 Search performance in naive B-tree indexes ($N = 300000$) 165
7.18 Comparison of construction times for indexes with and without output 173

Chapter 1

Introduction

This report explores different inverted indexes for dynamic document collections. A document collection is considered dynamic if it is subject to change. The possible changes include adding, deleting and updating documents.

Several index structures for dynamic document collections have been proposed in the literature. However, there are not many reported results from experiments comparing them [LZW06]. This report aims to contribute with additional experiment, and the focus is on adding new documents. Although deletions are supported in some of the implemented structures, we do not test this in the experiments. We also propose some slight modifications to some existing methods.

The contributions of this report include:

- Results from experiments with three overall index structures for dynamic document collections are presented. The results are compared to estimates for each of the structures. The estimates are calculated based on an efficiency model developed in this report.
- A modification of the scheme for how a partial index in memory is accumulated in sort-based inversion is proposed.
- A modification to hierarchical indexes constructed with method 2 from [OvL80] is presented. The modification makes the method more flexible. By changing a variable one can make the update speed faster, but doing so sacrifices search performance.
- The effects of using the tf-idf ranking scheme in updatable indexes is considered. Some approximations are introduced in the ranking scheme to make the overhead of maintaining document lengths for all documents less significant.

The report is organized in 8 chapters. Chapter 2 provides a basic introduction to the subject at hand, and gives a summary of the author's previous work. Chapter 3 introduces the most recent developments in the field, and Chapter 4 explains how we have implemented the search engine used in the experiments in this project. Chapter 5 introduces the experiments conducted. The

efficiency model for the implemented index structures is developed in Chapter 6. The actual results are presented and analyzed in Chapter 7, and we conclude in Chapter 8.

Chapter 2

Background and problem statement

This report is based on the work in the author's $9th$ semester project [Bjø06], and the theory underlying this work is therefore similar. This chapter gives a summary of all the notation used in this report, and a brief introduction to the subject. It also explains the problem at hand, before a short summary of the findings in [Bjø06] is presented.

2.1 Notation

Most of the notation used in this report is listed below. Less frequently used notation is introduced as needed. The meaning of the variables listed here will be introduced in the text as well. The intent of including the list is to provide efficient look-up for the reader.

N - the number of documents in a document collection. This variable will be used with subscripts to denote subsets of N.

n - the total number of terms in a document collection.

t_{cpu} - the time spent processing. This variable is typically used with additional subscripts to denote the phase considered.

t_d - time used accessing disk. This variable is sometimes used with subscripts to denote the phase considered.

t_s - time used searching for the correct position on disk.

t_t - the inverse bandwidth between disk and memory.

b - a given number of bytes

d_j - the document vector for document number j.

q - a query vector.

$|v|$ - the length of vector v.

$f_{i,j}$ - the frequency of term i in document j.

n_i - the number of documents in the collection containing term i.

occ_i - the total number of occurrences of term i.

V - The number of unique terms in a document collection. This variable will also be used with subscripts to denote the number of unique terms in a subset of the document collection.

w - a given term.

K - a variable defining how sizes of indexes in a hierarchical index grow.

B - the buffer size.

M - the amount of main memory available. This variable will sometimes be used with subscripts to denote specific parts of the main memory, for example a part reserved for caching index files.

t_{nr} - a term number, which is a number chosen to represent a given term.

d_{nr} - a document number, which is a number chosen to represent a given document.

z - the number of index entries there is room for in an in-memory index.

T - the maximum number of searchable partial indexes in a hierarchicial index.

t_p - the time used to parse one term, and add it to the HashMap of terms and occurrences for its document.

t_{am} - the time used to add parsed documents to the memory resident index, measured in seconds per byte.

t_{se} - the average time used for one operation in a multi-way merge of several lists.

t_{ur} - the time used to update the length of document based on a single index entry.

t_{lu} - the average time used to perform a look-up in a sorted list dictionary.

t_{bi} - the average time used to insert a new entry in a B-tree.

t_{bs} - the time used to perform a search in a B-tree.

$n_{d,u}$ - the average number of unique terms per document.

$n_{d,t}$ - the average total number of terms per document.

l_e - the average length of the term of one index entry.

l_u - the average length of unique terms.

s - the number of bytes used to store an object. Subscripts are used to denote the object considered.

$nodes$ - the number of nodes in a B-tree. Different subscripts will be used to denote specific B-trees.

$nodes_{leaf}$ - the number of leaf nodes in a B-tree. Different subscripts will be used to denote specific B-trees.

$|V|$ - the size of a dictionary file.

$|I|$ - the size of an inverted file.

d_a - a number of disk accesses.

m - the number of indexes merged in a given merge.

o - a given number of operations. Subscripts are used to denote the kind of operation.

pi - an expected number of partial indexes.

he - the height of a B-tree.

h - the hit ratio, which is the fraction of pin requests to the buffer pool that ask for an already cached part.

t_{search} - the time spent performing a search in an index. This variable is used with various subscripts to denote the type of index structure.

2.2 Inverted indexes

Large-scale search engines allow a user to search in a large collection of documents efficiently. In order to achieve this, the document collection is indexed before the user is allowed to search. Indexing involves creating a data structure in which the search is carried out, instead of performing a brute-force search through the complete document collection.

There exist many kinds of index structures, each with its own characteristics. Inverted indexes have proven to be the indexing method of choice for large-scale search engines [LZW06]. They are thus the focus of this report. The following subsections provide a definition of an inverted index and explain a method for building them. An introduction to compression of inverted indexes is given and basic ranking of documents in search engines is also explained. This information can be found in further detail in [Bjø06] and in several textbooks [WMB99, BYRN99].

2.2.1 Definition of an inverted index

A basic inverted index usually consists of two parts [Bjø06]:

- A dictionary
- An inverted file

The dictionary is a data structure which contains all searchable words, called terms, in the document collection. It also contains a mapping to where the inverted list for each particular term is found in the inverted file. Several data structures have been proposed for the dictionary. The most common ones are sorted lists of terms, tree structures and structures based on hashing.

An inverted file contains inverted lists for all terms in the dictionary. An inverted list is a list of the occurrences of the term it represents. If the list only gives the documents which contains the given term, the structure is called a document-level inverted file. If it contains all occurrences in all documents, it is called a word-level inverted index. Inverted indexes may be built with other granularities as well, however word-level inverted indexes are the most efficient ones if one is to support phrase queries. This report only considers inverted indexes with word-level granularity.

A word-level inverted file may consume as much space as 50 to 100 percent of the size of the document collection [WMB99]. Because a large-scale search engine should enable the user to search in very large document collections, the inverted index may not fit in main memory. Usually the inverted file is stored on disk, and it is also possible to store parts of the dictionary on disk [RE04].

Even though the space capacity of disks has increased steadily, the access time has not decreased proportionally. The following model is widely used when considering disk accesses [Bra, TGMS94]. The model says that in order to transfer b consecutive bytes of data from or to a disk, the time spent accessing the disk will follow Equation 2.1.

$$t_d = t_s + t_t \cdot b \tag{2.1}$$

The equation explains that the time used to transfer b consecutive bytes consists of a time used searching for the correct position, and the transfer time. The transfer time is linear to b with t_t, the inverse bandwidth, as the constant. The time spent searching for the correct position is usually modelled as a constant as well. It involves moving the disk arm to the correct cylinder, and waiting until the correct sector is under the head. It should be noted that $t_s >> t_t$ [Bra]. This implies that it is much more efficient with a few large consecutive transfers than several small scattered ones.

When the inverted file is stored on disk, it should be made sure that a search involves as few disk accesses as possible. If we make sure that each inverted list is stored sequentially, searching for one term will typically involve one single disk access. The order of the documents within an inverted list has been thoroughly investigated. The most common solution is to sort them according to document number, even though sorting on frequency of the term within the document

has been proved to result in more efficient retrieval [PZSD96, GWC04]. This report will only consider sorting the inverted lists on document numbers. It is also common to store the inverted lists for different terms sorted lexicographically on the terms.

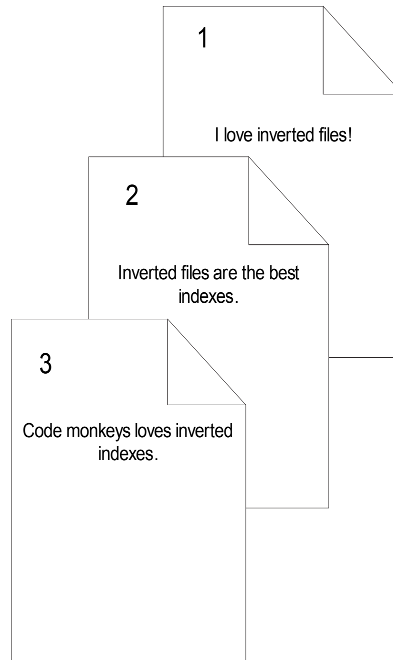

Figure 2.1: Example documents

The example documents shown in Figure 2.1 will be used throughout the rest of this chapter. An inverted index for these documents with document-level granularity is shown in Figure 2.2. The numbers in the upper left corner of the documents are the document numbers, and the numbers in the inverted file in Figure 2.2 denotes the document number of the document that contains the given term. The dictionary in the example index is a sorted list of all terms.

2.2.2 Building inverted indexes

There exist many methods for building inverted indexes, but when the index can not fit in main memory, most of them are quite similar in spirit [BYRN99]. They usually follow the overall method outlined in Figure 2.3.

The variations over this overall method may differ on several aspects. First they may differ in how they treat the dictionary. It is possible to have only one dictionary with references to all inverted lists for each term in different indexes, or one may flush a dictionary together with each inverted file.

There are also several ways to accumulate the partial index in memory, and the two main approaches are presented here. The first is to accumulate a list of triplets consisting of a term,

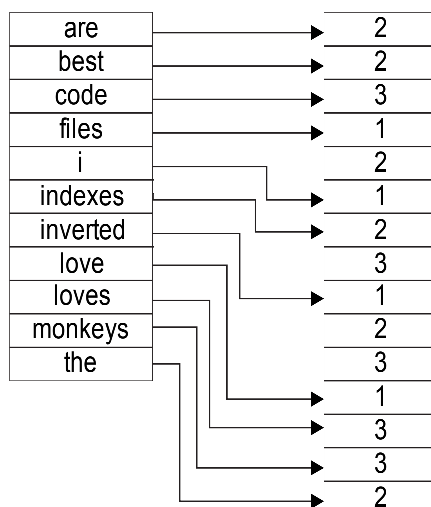

Figure 2.2: Index for example documents

1: create a partial index in memory
2: **while** there are more documents to be indexed **do**
3: **if** the memory is exhausted **then**
4: flush the current partial index
5: create a new partial index in memory
6: **end if**
7: Add a new document to the current partial index
8: **end while**
9: flush the current partial index
10: merge all partial indexes together to one index with a multi-way merge based on a priority queue

Figure 2.3: Overall method for building inverted indexes

a document number, and a sorted list of the occurrences of the term within the given document [WMB99]. When the memory is exhausted, a version of this list sorted on term first and then document number is flushed to disk, which gives an inverted file. The other well-known approach is to construct an in-memory dictionary and accumulate occurrences in either linked lists or lists with array doubling. These approaches and variants thereof will be discussed in more detail in Chapter 4.

2.2.3 Compression

Compression is widely used in inverted indexes. It might reduce the size of an inverted index to approximately $10-15\%$ of the size of the indexed document collection [WMB99]. Apart from the obvious advantage of limiting the disk space used, compression typically also reduce the time spent constructing an index and querying it, because of the smaller amount of data moved

between disk and memory.

At a first glance, it is not obvious how one can compress an inverted file. The main idea behind compression is to use few bits to represent likely values, and more bits to represent unlikely values. For a given inverted list it seems like all values of document numbers are equally likely, however a slight change in the representation eliminates this problem. If a document number in an inverted list is represented as the difference between its value and the previous one, smaller values will be more likely. This can be seen from Figure 2.4, which contains a representation of the index shown in Figure 2.2 but where the document numbers are coded as differences, often called d-gaps.

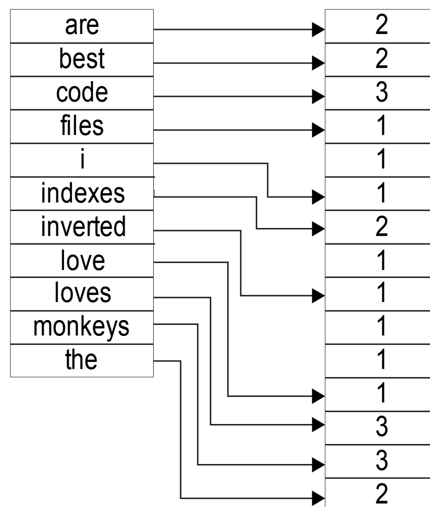

Figure 2.4: Index with difference lists

Looking at Figure 2.4, we notice that there are more 1's here than in Figure 2.2. The inverted list for "files" is an example. It consists of two 1's in Figure 2.4, while it consisted of a 1 and a 2 in Figure 2.2.

The most common compression schemes for inverted lists are Golombs code [Gol66] and Elias code [Eli75]. Even though Golombs code results in a compression scheme which is optimal under some constraints, it is shown in [SWYZ02] that using a compression scheme where all integers consume an integral number of bytes results in faster query evaluation.

Compression is not implemented in the experiments in this report, and we therefore do not go into further detail here.

2.2.4 Ranking in search engines

Ranking of results is what makes information retrieval systems different from data retrieval systems [BYRN99]. Even though the focus of this report is not on ranking, a search engine is an

information retrieval system and should thus facilitate ranking. There are many different ranking schemes, but only one basic scheme within the so-called vector-space model is considered here.

In the vector space model, each document can be viewed as a vector that contains one entry per unique term in the document collection, and the same applies to queries. A common way to calculate the relevance of a document for a given query is to calculate the angle between the two vectors using the dot-product. A formula for such a calculation is shown in Equation 2.2.

$$sim(d_j, q) = \frac{d_j \cdot q}{|d_j||q|} \tag{2.2}$$

Because we are only interested in finding a ranking that discriminates between the documents, dividing by the length of the query is usually omitted.

Each entry in the vectors may be the number of occurrences of the given term in the document or query, but a measure called tf-idf has proven to be a better alternative. Tf-idf is short for *term frequency - inverse document frequency*. The term frequency describes that the more frequent a term is in a document, the more it is assumed to be relevant to the contents of the document. The inverse document frequency says that when there are many documents containing a given term, the term is not likely to be well fitted for distinguishing a relevant document from an irrelevant one. The entry in a document vector for document j when using tf-idf may be calculated as in Equation 2.3.

$$d_{j,i} = f_{i,j} \cdot \log \frac{N}{n_i} \tag{2.3}$$

Equation 2.3 says that the weight given to document j from term i is given from the frequency of term i in document j times the logarithm of the number of documents in the collection divided by the number of documents in the collection containing term i.

Although there are reported results suggesting that the overhead of using tf-idf instead of simpler strategies is not justified by better rankings in the results [GF04], tf-idf will be used in all implementations in this report as it is the textbook example of a ranking strategy [BYRN99, WMB99, GF04].

2.2.4.1 Evaluating ranked queries

A naive implementation of the tf-idf ranking scheme will lead to an inefficient solution. It is obviously quite simple to calculate the numerator in the fraction of Equation 2.2 at query time by merging the inverted lists for each of the query terms.

As noted above, the length of the query in Equation 2.2 is usually discarded, and the only remaining value to be calculated is the document length. Performing this calculation at query

time is very expensive, and it should be pre-calculated. Equation 2.4 gives the formula for calculating the value for document d_j.

$$|d_j| = \sqrt{\sum_{i=1}^{V} \left(f_{i,j} \cdot \log \frac{N}{n_i} \right)^2} \tag{2.4}$$

The variable V in Equation 2.4 denotes the number of unique terms in the document collection. The lengths of the example documents from Figure 2.1 are given in Table 2.1.

Document number	Length of document (tf-idf)
1	1.6057
2	1.9874
3	1.9456

Table 2.1: Lengths of the example documents using the tf-idf weighting scheme

A search for the term ”files” in the example document collection would give higher relevance to document 1 than document 2, because document 1 is shorter. We could also note that a search for ”inverted” will give a relevance score of 0.0 for all documents, because the inverse document frequency will be 0.

As noted above, the lengths of documents in the collection is typically pre-calculated. Maintaining such values when the document collection changes will be discussed in Chapter 4.

2.2.5 Zipf’s law and Heap’s law

Zipf’s law and Heap’s law describes the occurrences of words in corpuses of natural language. They provide a basis for giving estimates of the efficiency of various methods for indexing and updating inverted indexes.

Zipf’s law was originally stated in [Zip49], and states that in large document collections, the frequency of a word is inversely proportional to its rank. The rank is defined as the position of the term in a list of all terms sorted by descending frequency. This can be expressed as in Equation 2.5 [CP90].

$$f(w)r(w) = z \tag{2.5}$$

$f(w)$ and $r(w)$ in Equation 2.5 are the frequency and rank of word w respectively, and z is a constant.

Heap’s law is often used to estimate the size of the vocabulary [BYRN99]. It is given in Equation 2.6.

$$V = k \cdot n^{\beta} \tag{2.6}$$

In Equation 2.6, V is the size of the vocabulary and n is the number of terms in the document collection. β and k are constants, and their values depend on the document collection at hand. [Fre02] reports results from experiments suggesting that typical values of these constants for large web collections are: $\beta = 0.57$ and $k = 16.24$.

2.3 The problem

As noted in Section 2.2.1, the inverted file is usually stored sequentially on disk. Maintaining such a property is difficult when the document collection is updated.

Three kinds of updates might occur in a document collection.

1. A document is added
2. A document is deleted
3. A document is updated

As noted in [Bjø06], one will change the index correctly in case 3 if one deletes the old version of the document, and insert the new one.

This report considers different ways of facilitating efficient updates in inverted indexes. The focus is on adding documents, however significant choices in the implementation is made to ensure that a future implementation of deletions does not involve too much work.

2.4 Summary of [Bjø06]

[Bjø06] is a survey of methods suggested for solving the problem introduced in Section 2.3. The main part of the report is a qualitative comparison of the suggested methods, where the methods are evaluated according to some predefined criteria. It also contains results from experiments with some of these methods. This summary will focus on the qualitative analysis, because the experiments performed in [Bjø06] had limited applicability due to disk caching in the operating system. We will first go through the criteria used in the evaluation of the different methods. Then the considered methods will be explained, and finally a short summary of the results of the evaluation will be given.

2.4.1 Criteria for evaluating updatable indexes

The following criteria for evaluating proposed methods for handling updates in document collections were used in [Bjø06]:

- **Ability to support all kinds of updates:** As noted in Section 2.3, one is able to support all kinds of updates if insertion and deletion is supported. Some of the methods evaluated do not support deletions out of the box.
- **Update speed:** At which rate updates can be reflected in the index is called the update speed. In many cases it is beneficial for the update speed to batch updates. Both update speed when batching updates and for single updates is considered.
- **Update latency:** The update latency is the amount of time, in the worst case, between the time when the update is requested and the time when the index reflects the update.
- **Search speed:** The search speed is measured as the number of disk transactions needed to search for a single term, and low values are obviously preferred. The inverted index outlined in Section 2.2.1 typically requires one disk access per query term.
- **Ability to support searches while updating:** This criteria aims to assess whether or not it is possible to support searches while updating, and if so, to what extent the search performance suffers from the ongoing update process.
- **Storage overhead:** This is the amount of extra space used when compared to a compact representation of an index for a given document collection. Storage overhead might be needed to enable support for searching while the index is being updated, or the method might over-allocate to reserve room for more entries at the end of inverted lists. Either way, a limit on the amount of storage overhead is wanted, and a small overhead is preferred.
- **Recoverability:** Recoverability is concerned with whether a failure will destroy our current index and force us to build it from scratch. If not, the index is recoverable and that is obviously a beneficial feature.

2.4.2 Evaluated methods

This section will give an introduction to the methods evaluated in [Bjø06], as well as a short resume of the results of the evaluation. The methods evaluated are characterized as belonging to two different groups, methods based on rebuilding the index and incrementally updatable methods. The first three methods evaluated belong to the first group, while the last four methods are incrementally updatable.

2.4.2.1 Rebuild

This method is probably the simplest one as it just rebuilds the complete index from scratch every time there is a given number of updates pending [LZW04]. It takes advantage of the fact that there exist very efficient methods for constructing an index for a document collection, for example the multi-way merge mentioned in Section 2.2.2.

Despite efficient methods for constructing inverted indexes, rebuilding an index for each update in a large document collection is clearly not feasible, and updates should be batched.

Evaluation
The rebuild method is capable of supporting all kinds of updates, and the search speed is essentially unchanged from an inverted index for a static document collection. The update speed and latency however, are dependent on the size of the complete index. This is a substantial drawback if the document collection is large.

The method is capable of supporting searches while updating if one is prepared to store an old version of the index while the new one is being built, but this introduces space overhead. The question of recoverability is not as important here, because it should be fairly straightforward to recover an index where the complete index is flushed to disk, and redoing a rebuild is not more expensive than processing updates.

2.4.2.2 Remerge

This method is quite similar to the complete rebuild, but takes advantage of the index already built. When new documents arrive, they are added to a partial index in memory which is flushed if it becomes full. When a batch of updates is added, the partial indexes are merged with the existing index on disk. By maintaining a list of documents to be deleted, the entries in the index for these documents can be filtered out during the merge. The method is tested in [LZW04].

Evaluation
Remerge supports insertions, and deletions can be supported by keeping a list of deleted documents as noted above. The advantage of this method compared to rebuild is that documents that are unchanged since the last merge will not need to be parsed and written to a partial index a second time. This may result in a more efficient update speed and latency, but the result is reversed if significant fractions of the old index are deleted.

It is simpler to lower the update latency here than in merge rebuild, because it is possible to allow searches in the partial indexes. This will on the other hand lower the search speed, because searches in more than one index may be required.

If one uses a merge method where the old index is kept while merging, this method supports searching while updating. It is quite common in construction methods to keep a copy of the old index, although methods with less space overhead exist. Allowing searches while updating will thus typically introduce a space overhead equal to the size of the complete index.

2.4.2.3 Hierarchical index

If we do not merge new updates into the main index in merge rebuild, but rather keep all flushed partial indexes and perform searches in each of them, we will have a much more efficient update process. Such a scheme would punish the search time severely though, and hierarchical indexes

are a compromise between these two schemes.

Hierarchical indexes were first properly introduced in [OvL80], and has since been studied with some variations in several other texts [BC05a, LMZ05]. Here we will focus on the presentation given in [OvL80], which presents two different methods. Common to both methods is that they maintain several indexes of various sizes, and a complete search should involve all these indexes. Just like with the two previous methods, updates may be accumulated in main memory, and when it becomes full, they are merged into the current hierarchy of indexes.

The organization of the hierarchy differs slightly between the two methods, leading to different trade-offs between search efficiency and update efficiency. Method 1 has a hierarchy containing indexes with maximum sizes of K^i for integer values of i. It may hold up to $K-1$ indexes of each maximum size. The size may be quantified in different manners, but a common measure is the number of postings in the inverted file. A size of 1 is typically a number of postings that will fit in main memory, although lower update latencies may be enabled by choosing the size of 1 to be smaller.

Adding documents in method 1 proceeds as follows: New documents are added to an in memory index until a size of 1 is reached. When this happens, the partial index is merged into the hierarchy. It is first flushed. If this leads to a situation where there are K indexes with a maximum size of 1, these are merged into an index with maximum size K. It is merged into an already used one if there exist one that can include it, or into a new one if not. If we now have K indexes of maximum size K, these are merged into an index of size K^2, and so on. Figure 2.5 shows an example hierarchy using method 1 when $K=3$.

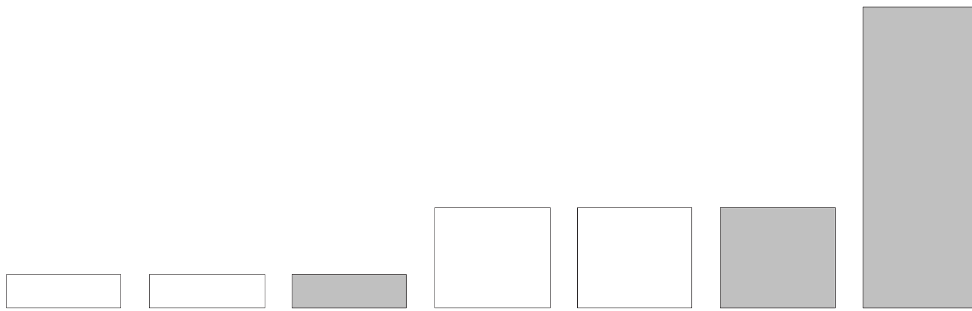

Figure 2.5: Example hierarchy with method 1 and $K = 3$

Method 1 leads to an average insertion time for an index with n postings, $I_1(n)$, as given in Equation 2.7. $P_s(n)$ is the time used to create a static index of size n.

$$I_1(n) = O\left(\frac{1}{\log(K)}\log(n)\right)\frac{P_s(n)}{n} \tag{2.7}$$

Equation 2.8 gives an upper bound on the relationship between the number of disk accesses required to search in a hierarchical index from method 1 and the number of disk accesses required to search a static index. $Q_s(n)$ is the amount of disk accesses required to perform a given search

in a static index of size n.

$$Q_1(n) = O\left(\frac{K}{\log(K)}\log(n)\right)\frac{Q_s(n)}{n} \tag{2.8}$$

Looking at equations 2.7 and 2.8, it is obvious that by increasing K we can get an average insertion time as close as we want to $\frac{P_s(n)}{n}$. Setting K to infinity is actually equivalent to flushing all partial in memory indexes in a normal inverted index construction without merging them at the end. It is thus clear that by increasing K we obtain a lower average insertion time, but sacrifice search speed.

Method 2 differs from method 1 in that it only keeps one index of each size. As in method 1, postings may be accumulated in memory until a threshold for a size of 1 is reached. At that time the partial index is merged into the hierarchy. If there is no index in the hierarchy with size 1, it will be flushed and set into the hierarchy at this position. Otherwise, it is merged into index i in the hierarchy. i is the smallest position where the resulting index from merging the new partial index and all indexes at positions smaller than or equal to i in the hierarchy, is smaller than the maximum size for position i. The different possible maximum sizes of the indexes in the hierarchy in method 2 are just as in method 1. An example hierarchy for method 2 with $K = 4$ is shown in Figure 2.6.

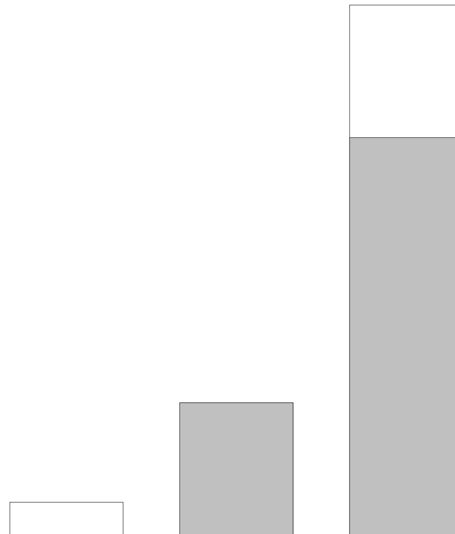

Figure 2.6: Example hierarchy with method 2 and $K = 4$

The equations for average insertion time and upper bound on disk accesses in a query for method 2 are given in equations 2.9 and 2.10.

$$I_2(n) = O\left(\frac{K}{\log(K)}\log(n)\right)\frac{P_s(n)}{n} \tag{2.9}$$

$$Q_2(n) = O\left(\frac{1}{\log(K)}\log(n)\right)Q_s(n) \tag{2.10}$$

We note that setting K to infinity in method 2, makes it similar to remerge. We also note that setting K equal to 2 in both methods makes them similar [Gri05], and the evaluation is only carried out for such a setting in [Bjø06]. Deletions may be supported by constructing a ghost structure as suggested in [OvL80], or by maintaining a delete list in memory where merged indexes filter out occurrences from these documents.

Evaluation
All update operations are supported in all hierarchical methods, and thus also for the special case with $K = 2$. The average insertion time becomes as given in Equation 2.11 for this special case.

$$I(n) = O(\log(n))\frac{P_s(n)}{n} \tag{2.11}$$

The best case for insertion in this method is when the current in memory index can fit in the position for an index of size 1 in the hierarchy, which is supported only by flushing it. The worst case is that all indexes in the hierarchy will have to be merged, giving a complete remerge. The update speed is just better than the two previous methods on average, but the worst case is almost equivalent to remerge. To which extent this method batches updates depends on how we quantify the size of an index, and on how a size of 1 is defined. We note that choosing the size of 1 to be quite large is likely to give a higher update speed, by sacrificing update latency (if we do not allow searches in the partial index in memory).

The upper bound on the number of disk accesses required to perform a search in this special case of a hierarchy is given in Equation 2.12.

$$Q(n) = O(\log(n))Q_s(n) \tag{2.12}$$

If we assume that a search in a single index requires one disk access, we can give a plot of the relationship between the index size and disk accesses required, as shown in Figure 2.7 [Bjø06]. Figure 2.7 also plots the worst case given in Equation 2.12 for reference.

This method is capable of supporting searches while updating, and this issue will be studied in more detail in Chapter 4. The storage overhead is again dependent on the method used for building the inverted indexes, and to which extent we will support searches during processing of updates. As in the previous two methods, recoverability is not a big issue here.

2.4.2.4 Simple incrementally updatable index

The first incrementally updatable index takes advantage of the fact that when adding a new document, only the inverted lists representing terms that occur in the document need to be

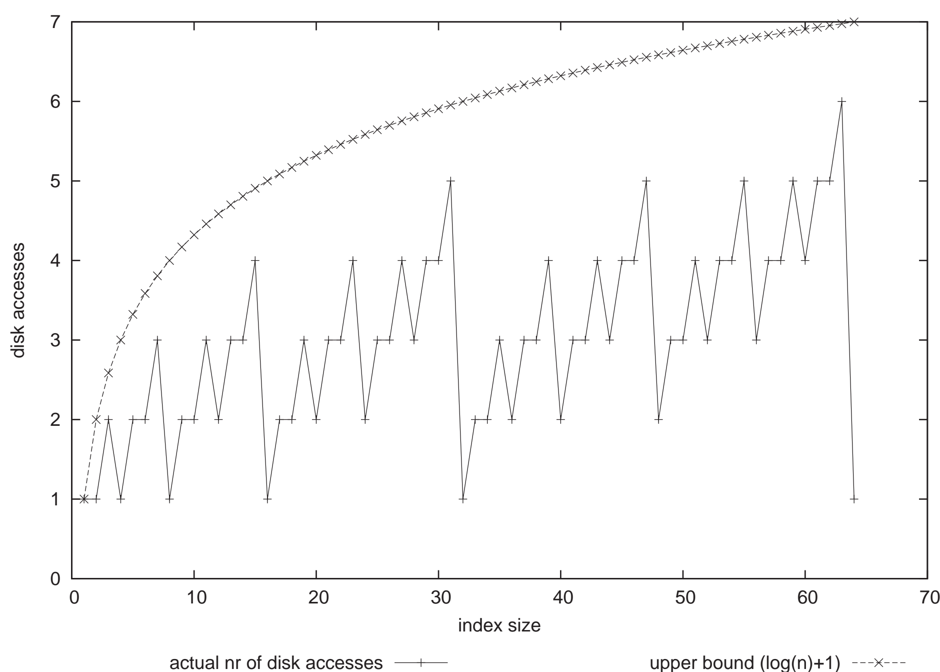

Figure 2.7: Number of disk accesses required in a search as a function of index size

modified [LZW04]. This simple approach thus looks up these lists, updates them, and writes them back to disk. They may have to be written to another position on the disk if there is no more room after their current position. It may not be necessary to read the complete list if we know that the new document has a higher document number than all previous ones.

From Equation 2.6 stating Heaps' law, we note that it is beneficial to batch updates in such a scheme because the number of inverted lists that require update grows as fast as the vocabulary of the batched document collection. Heaps' law states that this relationship is sublinear.

Evaluation
Simple incrementally updatable indexes are capable of supporting both insertions and deletions. As noted above, batching updates requires fewer list updates per document and is thus likely to be beneficial, although this compromises update latency. [Bjø06] contains an argument saying that the number of disk accesses required for an update in this method is more or less independent of the size of the inverted index, although it is obvious that the amount of moved data is not. The update latency is dependent on the way we batch updates as noted above, and another important aspect is whether or not this method is capable of supporting several update operations at once. If it is not, a new update will have to wait for the current batch to finish before it can be processed. In a basic implementation it is likely that two updates can not be processed at once, and it is thus hard to give an exact bound on update latency.

The search speed for this method is equivalent to that of a static inverted file, but some care must be taken to support searches while updating. [Bjø06] gives a general bound on storage overhead for this method dependent on the size of the longest inverted list, but concludes that a thread running in the background to minimize external fragmentation is likely to give the method a

decent performance with respect to storage overhead in practice. To which extent the method is recoverable depends on the implementation, but it is definitely a very desirable feature for such a method.

2.4.2.5 Methods using B-trees

[CP90] introduces several methods based on B-trees capable of handling updates. The first solution is called a naive B-tree index, and is an index where both the dictionary and the inverted file are represented in one single B-tree with keys consisting of a term and a document number. According to typical characteristics of a B-tree, this should give us a search time in $O(\log_B(n))$, where n is the number of postings in the index, and B is the size of the buffers used. Note that O-notation is used. The correct logarithm is actually B divided by the average number of bytes needed to represent a term, a document number and a pointer to another B-tree node. To be precise, we should also replace n with the number of leaf nodes in the B-tree, but because we use O-notation, this does not matter. An update requires us to perform one insert per distinct term in a document, leading to one operation with logarithmic complexity per term. A naive B-tree index for our example documents is shown in Figure 2.8.

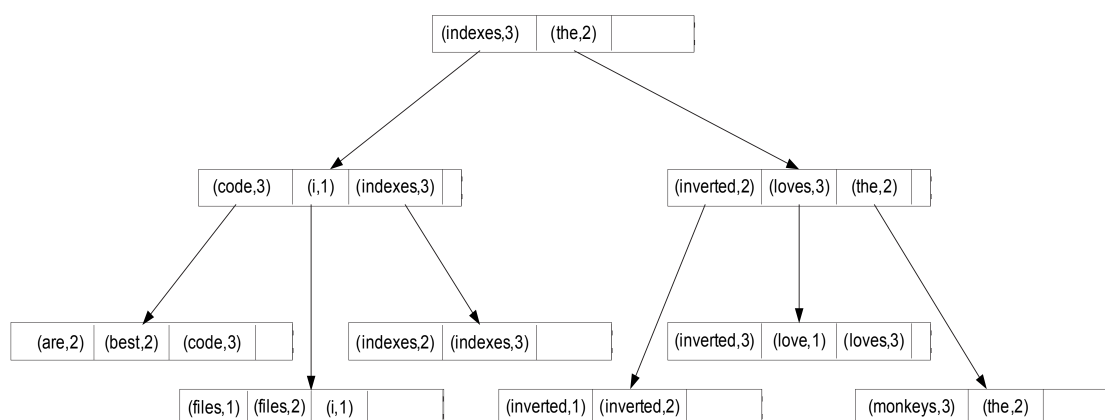

Figure 2.8: A naive B-tree index for the example documents

The asymptotic complexity is equal to the number of disk accesses needed to perform each of the operations when caching is not used. When caching the upper nodes in the B-tree, the number of disk accesses needed becomes $O(\log_B(n) - \log_B(M_c))$, where M_c is the amount of main memory available for caching.

Two optimizations to this scheme are presented in [CP90]. The first one is based on the observation that when considering update speed only, it is more beneficial to use the main memory available to cache updates rather than caching the B-tree. The only part cached from the B-tree is the path from the root to the currently processed leaf. Based on an estimate of the vocabulary size based on Zipf's law, it is shown that this scheme leads to higher update speed. In [Bjø06] it is noted that search speed is likely to suffer severely though, which makes this approach significantly less attractive.

The second proposed improvement is a space optimization. The idea is to avoid having multiple entries in the B-tree with the same term. This is achieved by grouping all occurrences of a term in different documents into one entry. This implies that the term itself can be used as the primary key. It is shown in [CP90] that this will typically reduce the space requirements by 50%. The problem is, however, that some entries might become very large using this approach, and may thus not fit within a single B-tree node. A suggested solution to this problem is to store the entries in a separate file, making this scheme similar to an incrementally updateable inverted index with a B-tree as the dictionary. A technique known as pulsing is also introduced in [CP90]. This technique reserves space for some occurrences in the entry for a given term. When this buffer is full, the entries are moved to the separate file.

[Bjø06] notes that there are several possible schemes obtained by grouping different numbers of entries for a given term. These schemes are also likely to be more space efficient than a naive B-tree index, and may also be constructed so that no separate file is needed.

When choosing to store the entries in a separate file, one has to choose how to store the inverted lists within the file. [Bjø06] refers to studies from [FJ92] which shows that storing all inverted lists contiguously on the disk is often the most efficient solution for both searches and space overhead. By reserving extra space at the end each time the list is relocated, the amortized complexity of appending a new entry on an inverted list is $O(1)$.

Evaluation
Two different methods using B-trees is evaluated in [Bjø06], the naive B-tree index, and the space optimization. The organization of the inverted file recommended in [FJ92] is used in the space optimization.

All update operations are supported by both evaluated methods. All evaluations carried out in [Bjø06] assume that the dictionary structure can fit in main memory. As opposed to the previously discussed methods, this assumption is not vital for a B-tree because the structure is well fitted for storage on disk.

Based on a worst case estimate of the amount of space used by naive B-tree index, it is argued in [Bjø06] that it is likely that each update in the index will require more than one disk access. The space optimization on the other hand, is more likely to require only one disk access based on the assumption that a normal dictionary for the collection can fit in memory. The pulsing technique available for the space optimization is also likely to result in higher update speed on average.

The naive index has one important advantage compared to the space optimization. When the list is relocated in the space optimization, the complete list must be read and written. In the naive B-tree index on the other hand, the correct position to insert a new entry is found directly through the insert in the B-tree. This may have a positive effect on update speed.

B-trees are widely used structures in databases, and it is well known how to ensure serializable histories in them. This makes it easy to perform concurrent updates in the tree, which is likely to lower the update latency. To extend this to the space optimization, some modification is needed to introduce locking in the inverted file as well. Using such a locking scheme, it is easy

to support searches while updating as well.

Using the same assumptions as above, [Bjø06] concludes that a search in an index using the space optimization is likely to use only one disk access. When storing several entries for a term in the naive index, they may be stored on adjacent blocks in the b-tree, but we have no guarantee that these blocks are sequential on disk. Assuming that an update requires 2 disk accesses in the naive index, a search for a term that is spread out over p nodes, will require $p + 1$ disk accesses. This makes the space optimization more attractive for performing efficient searches and it also has the benefit of not requiring to read the redundant information stored in the naive B-tree.

Because the worst case space efficiency in a B-tree is $\frac{1}{2}$ ($\frac{2}{3}$ on average), it is clear that the dictionary in the space optimization will consume more disk space than a compact representation. The inverted file will be similar to the one for the simple incrementally updatable method, and the overall storage overhead is thus likely to be within reasonable bounds. [CP90] reports that the space optimization is about 50% more efficient than the naive index with respect to space usage. The storage overhead in the naive index is thus expected to be significant.

With respect to recoverability, the methods based on B-trees have a clear advantage compared to other methods. Recoverability for B-trees is also a well studied field and a form of logical logging is likely to yield good performance while at the same time enforcing recoverability.

2.4.2.6 Continuous update

Continuous update is the name used in [Bjø06] to describe a method introduced in [CCB94] and [CC95]. It is not strictly speaking an incrementally updatable approach, but the authors argue that it solves some of the problems with methods based on B-trees. Figure 2.9 describes how this method works.

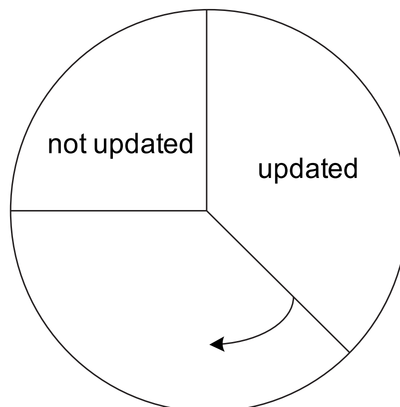

Figure 2.9: Snapshot of the index on disk when continuous update is used

An update process goes through the index at given intervals, and updates to occur are kept in lists. The file containing the inverted lists may be considered as a circular buffer as Figure 2.9 suggests. The update process starts with the first inverted list, reads it, merges in the updates

from the update lists, and writes it back out after the last inverted list. This process continues until all inverted lists are processed and a run is completed. How often a new run is started is a tuneable parameter, and the effects on search performance from this choice is discussed in [CC95].

Evaluation
Continuous update supports both insertion and deletion, but the update speed is again dependent on the size of the complete index. If there are few updates pending when the update process begins, the cost per update will be large. With many updates pending, the cost of each update is relatively smaller, just like with the other methods based on rebuilding the index.

If one insists on processing a new or deleted document completely within one update cycle, the update latency using this method is the time between initiations of two update cycles plus the time spent performing one update cycle. If we do not impose such a restriction, and continously run the update cycle, the worst case latency is the time spent on one update cycle.

The search speed using this method is almost equivalent to a static inverted file. The equivalence is valid if we choose never to let an inverted list that will fill more than the rest of the file store its first postings at the end of the file. It is easy to allow searches while updating the index if we do not insist on consistent retrieval. We only keep the copy of the inverted list to be moved to the end until we have written the updated version. When the new version is written to disk, the pointer in the dictionary is changed in an atomic operation. Insisting on consistent retrieval introduces a significant space overhead as well as some implementation difficulties if we want to support searches while updating.

Some of the modifications suggested in the evaluation introduce a slight space overhead. The file we have allocated to the index should be able to accommodate the complete index we eventually get, or we should be able to allocate new disk chunks adjacent to the current chunk. Recoverability may be supported in this method and [CC95] suggests checkpointing on a regular basis to be able to restart from the last checkpoint in case of a failure.

2.4.2.7 Dual-structure index

A dual-structure index partitions the inverted lists into two sets: long ones and short ones [TGMS94]. The long lists are processed like in the simple incrementally updatable index, while the short lists are stored in buckets. When searching for a term we first look it up in the dictionary for long lists. If it is not found here, we use a hash function, $hash(w)$, to decide in which bucket the inverted list (if it exists) is stored. This scheme allows us to maintain a far smaller dictionary, but may also require an unnecessary disk access when searching for terms that do not exist in the document collection.

In the updating process we follow the same overall scheme as sketched for searches. An update in a bucket might make the bucket overfull. When that happens, the longest list in the bucket is transformed into a long list. The distinction between long and short lists is assumed to be well fitted for information retrieval because of Zipf's law, which essentially state that there are a few

long lists that will grow very rapidly, while a few short ones will almost never grow.

[TGMS94] performs simulations where various policies for handling the long lists are compared. They conclude that even though no method is definitely better than the others, the method that keeps all lists stored continuously all the time is promising in most areas. This conclusion is analogous to the conclusion in [FJ92].

Evaluation
The dual structure index is the only method studied that does not support deletions out of the box. The problem is that there is no policy for migrating long lists back to short ones. The method can undoubtedly be extended to fix this problem, as discussed in [Bjø06].

[TGMS94] does not discuss the choice of a dictionary structure in detail, but it seems highly likely that the dictionary structure can fit in main memory, because we only need a dictionary structure for the frequently occurring terms. Following this assumption, either the bucket or the list for updated terms will have to be read in, updated, and written back out. This leads to a maximum of 2 disk accesses per updated term. The latency for this method is similar to the simple incrementally updatable method. The same is true for considerations regarding the ability to support searches while updating.

A search only requires one disk access, as is the case with a static inverted file, and it is certainly not a drawback that the dictionary used is smaller than for a static inverted file. A complete analysis of the storage overhead introduced by this method is complex, but it is concluded in [Bjø06] that it is likely to be in the same order of magnitude as the simple incrementally updateable index. [TGMS94] does not consider fault tolerance, but that their algorithms and data structures are constructed so that it is possible to restart after a failure.

2.4.3 Further considerations and experiments

[Bjø06] also mentions a technique called landmarks. While all evaluated methods supports updated documents by deleting the old version and inserting the new one, [LWP+01] shows that updates occurring in documents are usually small and clustered. To handle such updates efficiently they propose to insert landmarks in the text [LWP+03]. This technique makes it possible to insert a term at a position in the document without having to update the references for all terms after the inserted one in the text.

When [Bjø06] was written, the only reported experiments comparing updatable index structures were found in [LZW04]. They compare rebuild, remerge and the simple incrementally updatable approach. They argue that all methods evaluated in [Bjø06] are inefficient because they predate the invention of the fastest building techniques for static indexes.

They conclude that with large document collections and relatively small batches, the simple incrementally updatable method performs best, but that in other settings remerge is the best method. They see optimizations to the in-place method as a promising area for future work. [Bjø06] reports that it is peculiar to state that the other proposed incrementally updatable ap-

proaches will perform poorer than the simple method tested in [LZW06]. It is believed that this deserves a more thorough investigation.

Experiments were carried out in [Bjø06] to test remerge versus various configurations of the hierarchical index (which was not tested in [LZW04]). The results of these experiments had very limited applicability though, because of problems with disk caching in the operating system, and no results from these experiments are reported here.

The conclusion of [Bjø06] is that far more experimentation should be carried out in this field to get a clearer view of the strengths and weaknesses of the different methods. The methods considered most interesting to test are the ones based on B-trees with the possible improvement mentioned in Section 2.4.2.5. The dual-structure index is also considered to be an interesting starting point for further refinements.

Chapter 3

State of the art

As noted in Chapter 2, [Bjø06] concluded that more experiments are required within the field of updatable indexes to get a clearer view of the advantages and disadvantages of the different techniques. Fortunately, several authors have seen this need and there have been several publications reporting results from experiments during the last year. Through these articles, the author became aware of a technical report [CH98] from several years back that unfortunately was missed in [Bjø06], even though it presents interesting ideas.

This chapter will give an overview of the recent developments in this field, by going through each of the contributions. Each section will explain the overall approaches in the contributions, before the author comments on the presented ideas.

3.1 Eager, Piggyback and Batch

[CH98] is the first article known to the author to focus on an important aspect also mentioned in [Bjø06], the update latency. Their overall goal is to be able to reflect changes in the document collection as soon as possible. In order to solve this problem, they consider three different overall approaches:

- **Eager:** When a document is added or deleted, the inverted lists representing terms contained in this document are updated immediately.
- **Piggyback:** When a new document is added, the update is eagerly applied to all currently cached inverted lists, and a transient index is constructed to describe the updates that has not yet been applied to the index on disk. The updates in the transient index are applied to the real index when a query for the term is submitted, in which case the list is read anyway. A deleted document is handled by marking it as deleted in a list. Documents marked as deleted will eventually be filtered out from the index on disk.
- **Batch:** Added documents are added to a transient index, but none of these updates are

applied to the index immediately. During periods with low load, the transient index is merged into the real index. By searching in both the transient and the real index, queries will have an updated view of the collection between these merges. Deletion is handled just like in the piggyback version.

It is assumed that both transient indexes and the hash-based dictionary used can fit in main memory, but that the list of words for each hash value is stored as a linked list on disk. Concurrency control when there is only one thread performing updates is quite simple and based on a timestamp method which, according to the results of some experiments, obtains far superior performance compared to an approach based on locking [CH98]. The index used is an example of something in between a word-level inverted file and document-level inverted file, because the occurrence of a term within different sections in the document is recorded.

Results from experiments reported in [CH98] show that the piggyback and batch methods are far more efficient than the eager approach when it comes to handling updates. Piggyback is slower than the other two when it comes to search speed because it has to perform updates while searching, while the batch method is significantly slower at searching while the batch of updates is being processed.

The authors conclude that the piggyback approach seems to be the best method because it spreads the updating cost over a longer period of time without severely degrading search performance [CH98].

3.1.1 Comments

When explaining the poor results obtained for the eager method, the authors mention a naive implementation which iterates through all terms in the dictionary. This is done by accessing each value of the hash function, and following the disk-resident linked list of each of the terms with the given value. If the currently checked term exists in the document, its inverted list is obtained and written back out. This leads to a very low variance in insertion and deletion times as a function of the size of the inserted/deleted document.

How the linked lists of all terms with a given hash value is stored on disk is not commented in [CH98]. If it is stored as a linked list on disk it may require one disk access per term in the dictionary regardless of whether the term exists in the document. It appears that such an implementation should not give sufficient proof to the claim that the eager approach is a bad solution.

It also seems like the focus of [CH98] is on processing boolean queries. If they were to support a ranking scheme like tf-idf, it is likely that some additional processing is needed for every added document using the piggyback or batch method. This issue will be discussed when considering the implementation of other methods in Chapter 4.

According to the experiments performed in [CH98], the batch and piggyback methods seems to have very good performance, but the disadvantages of these methods are not mentioned. It

is said that piggyback only updates lists when a search for the term is processed. If the system during a period of time has to handle a lot of updates, while the query load is low, the memory requirements become very high. It is not in any way guaranteed that a list with a lot of updates represents a term frequently searched for. In addition, some terms are probably never searched for, and will thus consume memory throughout the whole lifespan of the index. A backup strategy in case the memory becomes full should thus be incorporated in the method.

For the batch method, it is said that the batch can be processed while the load on the system is low. Some systems experience high load 24 hours a day, and even if not so, the memory may become full long before a period with low load sets in.

Regardless of the commented issues, the focus on update latency in [CH98] seems very reasonable. The piggyback method is not commented elsewhere, but is certainly an interesting path. Schemes with some of the same characteristics have been applied with great success in other fields [ADHN06].

3.2 Experimentation with B-trees

[Bjø06] called for experimentation with updatable structures where a B-tree was used as the vocabulary structure. Such experiments are carried out in [LZW06]. The paper is a follow-up on the experimentation performed in [LZW04]. Rebuild, remerge and various forms of in-place update is compared. In-place update has become the de facto standard term in the literature to denote what was called incrementally updatable methods in [Bjø06], even though there are more aspects to in-place updates which will be discussed later in this section.

Common to all tested methods is that added documents are accumulated in an in-memory index. This index is searchable by using a method for accumulating partial indexes described in [HZ03], a technique that will be discussed in Chapter 4. This implies that a document is searchable immediately after it is added to the in-memory index.

Once a predefined number of documents are added to the in-memory index, the added documents are incorporated into the main index. How this is performed varies between the different methods. The rebuild strategy only initiates a complete rebuild of all added documents. To enable searching while updating, a copy of both the previous index and the in-memory index is kept. This strategy obtains quite poor results in all experiments in [LZW06].

The next strategy tested is remerge, which is explained in Section 2.4.2.2. The accumulation is just as for rebuild, and the resulting in-memory index is merged with the main index when the memory is exhausted.

The last method tested is basically the space optimization from [CP90] with the B-tree as the vocabulary, as explained in Section 2.4.2.5. Initial tests found a naive implementation of this method to have quite modest performance according to the authors, and two improvements where implemented. The first is pulsing, as introduced in [CP90] and explained in Section 2.4.2.5. The second is over-allocation for the inverted lists, but the authors argue that the moti-

vation for their over-allocation policy is not to make room for more than the current entries as in [TGMS94] or the more advanced approach introduced in [SC05]. On the other hand, their motivation is to avoid many small entries in their free map. It is argued in [LZW06] that an efficient implementation of a free map is essential for the overall performance. According to the authors, over-allocation to avoid small entries makes it easier to achieve decent performance in the free map.

The B-tree method is argued to be an in-place update method, because one does not merge in the new batch once it is processed, but rather looks up the terms present in the batch, and appends the new entries at the end of the inverted list for these terms. If there is not enough room for them, the complete list is moved to a different location. This differs slightly from incrementally updatable approaches in [Bjø06]. Whereas new entries are always appended at the end of the inverted lists here, they are typically merged with existing lists in incrementally updatable methods discussed in [Bjø06]. The effects of the approach used here is discussed in Section 3.2.2.

3.2.1 Experiments in [LZW06]

Experiments on these structures were carried out with two different collections, starting with either a small or a large collection for which an initial index was built. Both initial indexes where then updated with approximately 1 GB of additional data. The authors mention, as is also noted in [Bjø06], that no earlier experiments with such structures are reported in the literature.

Their results from the experiments with the small initial collection show that the rebuild-method is inefficient for all but unrealistic scenarios [LZW06]. The naive implementation of the in-place approach performs even worse than rebuild as opposed to the results in [LZW04]. The authors argue that the reason is that parts of the vocabulary structure is now stored on disk. For larger collections on the other hand, the naive in-place becomes competitive, at least when the number of buffered documents is reasonably low.

It turns out that the first optimization to the in-place solution, pulsing, had a positive effect on the update speed. Concerns about the space overhead introduced by keeping short inverted lists in the vocabulary structure were not confirmed, supposedly because the vocabulary structure did not consume a significant part of the complete index [LZW06].

Using the two optimizations together provides very limited benefits compared to using only pulsing, and it is concluded that the potential benefits of over-allocation are small. The average update speed varies between 1 second and 0.1 seconds as a function of the amount of buffered documents in this case.

The same overall results are obtained in the tests with the large collection, and it is concluded that the in-place method is the most interesting method despite the fact that it is slightly less efficient than remerge and leads to fragmentation. The in-place method does not require several copies of the index, and the authors see this as a very desirable feature. They would therefore prefer this method if it can be made slightly more efficient, and see optimizations to an in-place

scheme as a promising area for future work.

3.2.2 Comments

[LZW06] provides the first real experimentation with updatable structures, which is an important step forward. Their average update speed seems to be reasonably good, at least when the document collections are small. But when the document collection consists of approximately 25 million documents, a second is typically needed per document when the batches are not very large. Evaluating the quality of these values is difficult without reference values, but some general comments are in order.

First, it should be noted that only insertions are tested in [LZW06]. Methods for supporting deletions are discussed at the end, but this would possibly require changes in the in-place method. As noted above, new entries in an inverted list are appended to the existing one on disk. To preserve the property that the inverted lists should be stored with increasing document numbers, all new documents must have higher document numbers than all previous ones. To achieve the best possible compression when the lists are represented with d-gaps, the document numbers should have the smallest range possible. It would thus be beneficial to reuse document numbers a while after a document has been deleted, but this is not possible using this method without merging in the new documents instead of appending them. Another solution is of course to not compress the most newly added documents until they are relocated, but either of these solutions will affect performance, and their impact should be considered.

A second comment is in order regarding update latency. Update latency is clearly a focus of [LZW06], and because they are capable of searching in the in-memory index, they argue that the documents are instantly available. It is obviously true that once they are ready to insert a new document, it will be searchable very fast. But the update latency should be measured from the time the document was found to be updated until it is searchable. Such a setting will have to be simulated, and finding a realistic experiment is not straight-forward. We can for instance give an estimate on the average update-latency given a maximum number of arriving documents per second for a given method. Such an experiment would have been interesting.

Finally, the claim in [LZW06] that the documents added to the in-memory index are immediately searchable is also interesting. The ranking scheme used in the experiments is not commented. To which extent ranked queries are immediately available when a document is added to the in-memory index is also uncertain. If tf-idf ranking is used, adding the document to the in-memory index is not sufficient to make it searchable, because the document length will also have to be computed. More elaboration regarding this issue will be given in Chapter 4.

3.3 Hybrid methods

Two hybrid approaches where a distinction between short and long lists is incorporated into different kinds of hierarchical indexes have also been proposed recently [BC06, BCL06]. Both of these papers use the hierarchical methods that were only described as subsets of the overall hierarchical method in Section 2.4.2.3. We will therefore give a short introduction to these variants here, because especially one of them is an important basis for understanding a proof in [BCL06].

3.3.1 Geometric partitioning

This method is proposed in [LMZ05]. It is quite similar to method 2 in [OvL80], described in Section 2.4.2.3. A size of one is defined as the number of postings that can fit in main memory, and the various indexes has maximum sizes defined almost as in [OvL80]. The only difference is that the smallest index is of size $K-1$, while the next is of size $(K-1)K$ and so on. This scheme enables the authors to give a bound on the cost of building an index online as given in Equation 3.1.

$$\Theta\left(N\left(\frac{N}{M}\right)^{\frac{1}{K}}\right) \tag{3.1}$$

When $K = 2$, this method is referred to as Sqrt merge in [BCL06]. Note that this not contradict the results referred to in Section 2.4.2.3, because of the slight difference in sizes for each partition, and because the result here takes the size of the batches into consideration.

3.3.2 Logarithmic merge

This method is outlined in [BC05a], and a more elaborate explanation is given in [BC05b]. This method is essentially method 2 of [OvL80], which is also presented in Section 2.4.2.3. The value of K is set to 2 here as well. In essence, this makes the method similar to the Sqrt merge method mentioned above, as noted in [Gri05]. But the authors of [BC05a] manage the merges so that only two indexes are merged at any time, giving a cost of building the complete index online as given in Equation 3.2.

$$\Theta\left(N\log\left(\frac{N}{M}\right)\right) \tag{3.2}$$

[BC05b] also supports deletions in this scheme, in the fashion suggested for hierarchical indexes in [Bjø06], except that they incorporate delete lists in another feature originally developed for supporting multi-user desktop search.

Experiments from a desktop search environment are also reported, suggesting that the logarithmic merge seems to be a good choice for many different settings regarding the number of queries per update operation [BC05b].

3.3.3 A first set of hybrid methods

A first set of hybrid methods, incorporating a distinction between long and short inverted lists in the above mentioned hierarchical methods is presented in [BC06]. Recent experiments have shown that in-place methods are usually slightly slower than merge-based methods. The merge-based methods have the advantage that the disk access pattern is strictly sequential, and thus short inverted lists are merged without requiring a disk access for each of them. The in-place methods, on the other hand, are likely to be more efficient when processing long inverted lists, because they do not require relocating the list each time if over-alloction is used.

Following the line of thought given above, it seems likely that there exists a certain threshold value for the length of inverted lists that should be processed in-place, while the rest should be merged. This scheme is similar to the presented methods in [TGMS94]. The main invention in [BC06] is that a hierarchical approach is used as the merge-based method. The hierarchical method chosen is the logarithmic merge presented above with $K = 2$.

The authors argue that a custom implementation of an in-place update scheme is not trivial. To make the implementation simpler, one relies upon the file system. Each long inverted list is represented in a single file. When new entries should be added to this file, the new entries are appended to the file. One thus relies on the file system to limit the amount of fragmentation and keep each inverted list stored more or less contiguous on disk.

For the immediate merge method used, called remerge in Section 2.4.2.2, various limits on the size distinguishing long from short inverted lists are used. For the hierarchical approach, it is not trivial to keep track of the complete number of occurrences in the complete inverted file because it is usually fragmented over several indexes. The strategy used takes the decision of whether a list is short or long when the first partial index is flushed.

Another possible improvement is also presented. When the memory is exhausted after building a partial index in main memory, parts of the accumulated postings should be appended to long lists, while the rest should be merged with the on-disk short lists. The merging process requires reading and writing the complete on-disk index. It is thus suggested that merging should be delayed as long as possible. Therefore, when the memory is exhausted, the long lists are appended to the appropriate files, and deleted from memory. If the amount of free memory now becomes below a given threshold, one starts accumulating more postings in memory. If it is above the given threshold, the merge with the on-disk index for short lists is initiated. This strategy requires a smaller number of merges, and the strategy is called partial flush.

Experiments show that while only constructing partial indexes without ever merging them is the most efficient method for index construction, the best version of the hybrid method with logarithmic merge is not much worse. It is substantially better than a normal logarithmic merge,

which is in turn better than immediate merge.

Partial flush has a positive effect on indexing performance. Although the hybrid method used with remerge is slightly faster at queries than hybrid logarithmic merge, it is substantially slower at index construction. The authors thus conclude that the hybrid logarithmic merge is the overall best method. They also note that relying on the file system for the allocation strategies for the long inverted lists is risky because it is completely unknown what is done. Future work is to implement a custom way of handling the long inverted lists.

3.3.4 Improving hybrid methods

[BCL06] presents some improvements to the techniques discussed above. The authors argue that there are two main problems with the previous hybrid methods.

1. Handing over control of the in-place updates to the file system makes it difficult to analyze, and it is uncertain whether the inverted lists are actually stored contiguously.

2. With a certain limit discriminating between long and short inverted lists, some lists may use a lot of merge operations to achieve this limit. When they eventually become long, the next merge will append the new entries for these terms to the long list, even if it would be much more efficient to merge the new entries into the merge-part of the index. This problem is more severe if the main memory available is limited, because this implies a larger number of small appends on the long inverted lists.

These problems are dealt with in [BCL06]. To fix the second problem, the limit on the length of inverted lists considered long is not set as a global value. Whether a resulting inverted list from merging indexes is stored as a long or short list is dependent on its length, not the length of the complete inverted list obtained by concatenating all inverted lists for this term. This implies that a complete inverted list may be scattered between its long list part, and several short list parts.

All long lists are stored in a single file. This file is not organized with contiguous lists that are relocated. When a new inverted list should be stored in the file for long lists, it is just appended to the end of the file. This solution requires a structure for keeping track of all segments of all inverted lists within the file for long lists. This structure will never be very large, because there are quite few terms with such a length if one chooses a high number of entries to define a long list. The authors in [BCL06] thus conclude that this is a structure that will fit in main memory.

It is obvious that this scheme sacrifices search speed to enable faster updates, but it is argued that if the value used to define a large inverted list is high, the overhead is not substantial [BCL06]. The reasoning behind the claim is that when the threshold for long inverted lists is large, each part of the inverted list in the file for long lists will be long. The disk accesses involved when switching from one part of an inverted list to another in the file, will thus not constitute a significant part of the time spent accessing the disk in a search.

The number of disk accesses required to build a complete index with n postings is proven in [BCL06] to be linear in n when using this strategy together with logarithmic merge. The proof builds on the fact that as n increases towards infinity, the number of short lists will go towards zero while the number of long lists will increase. It is also shown that applying the same strategy to immediate merge does not have a similarly positive effect, although the asymptotic number of disk accesses is slightly reduced.

[BCL06] also reports results from experiments testing their strategy. The collection used in the experiments is the same as the large one used in the experiments in [LZW06], TREC GOV2. Various configurations with the improved and old hybrid schemes, and different basis structures like remerge, logarithmic merge and sqrt merge is tested. As a baseline, building all partial indexes without merging them for the complete document collection is also tested. This method is referred to as No merge. Almost regardless of the amount of memory available, their baseline is capable of constructing the index with approximately 25.2 million documents in 4.66 hours. The other methods are slower, but the new hybrid method with logarithmic merge is the fastest one and builds the complete index in 6.86 hours. The average search time in this structure is much faster than with no merge, 1.2 versus 4.8 seconds.

It is also shown that the hybrid methods have different performance depending on the length defined to make a list long or short. As the authors expected, the new improved hybrid techniques are less dependent on the amount of memory available compared to the old hybrid scheme. The reason is that whether a list is considered long is defined at each merge, not after a given percentage of the collection has been indexed, as was the case in [BC06].

The hybrid method with remerge is, not surpisingly, the most efficient method for query processing. Its most efficient configuration is when the limit for long lists is set to infinity, which is basically the same as a normal remerge. The new hybrid technique with logarithmic merge is not an improvement when it comes to search speed. It is actually less efficient than all other configurations except no merge.

The results reported in [BCL06] are shown to be quite consistent with computed expected times.

3.3.5 Comments

As a first comment it can be noted that the reported results seem quite good. Building all partial files for a complete index of the GOV2-collection in less than 5 hours is impressive. The newly proposed technique is also quite fast when it comes to updating.

That the new hybrid approach with logarithmic merge is fast compared to the baseline strategy is not strange if one takes into account the choices made in the implementation. The logarithmic merge is quite efficient for updates in itself, as we know from Equation 2.7. In addition, choosing to relax the restriction that long lists are to be stored contiguously will obviously have positive effects on the update speed. The search efficiency does not seem to be quite as impressive though.

Update latency is not commented on in [BCL06], and this remains an open but nevertheless important question. It is said that the Okapi BM25 [RWB99] is used for ranking. This does not require such a complex computation of document length as tf-idf. It is therefore plausible that documents are searchable as soon as they are added to the in-memory index. According to a technical report describing the memory management in their overall implementation, it is not unlikely that these documents are immediately searchable [BC05c].

Chapter 4

Implementation

This chapter gives an introduction to the implementation used as a basis for all experiments in this report. It starts with a presentation of the methods chosen for testing, and explain why they were chosen. All the code used in the experiments in this project was eventually written from scratch, and the reason why is explained in Section 4.1.2.

Section 4.2 will give an overall introduction to the implemented search engine called Brille [HCG+06]. Brille was originally developed in a course called *TDT4215 Knowledge in Document Collections* at NTNU, but only the name is still the same. All the code is rewritten, and it is included in Appendix C.

4.1 Implemented methods and overall choices

There are several important aspects when considering which methods to test. First and foremost, one should believe that testing the chosen methods will give a further insight. We also prefer choosing methods we expect to be efficient. Another important aspect is to choose methods with manageable implementation complexities for the scope of the given project. Based on such considerations, the chosen methods for this report are:

1. Remerge

2. Hierarchical index method 2

3. Naive B-tree index

The next subsection will give a further explanation of why these methods where chosen, and the following will explain the rationale for choosing to implement the overall solution from scratch.

4.1.1 Rationale behind the methods chosen

The reasons for choosing to implement remerge are quite simple. First of all, it is a basic method. This means that it is fairly straight-forward to implement, but it is still reported to have decent performance in several experiments [LZW04, LZW06]. Implementing remerge involves implementing a standard construction algorithm for inverted files, like the one outlined in Section 2.2.2. A basic construction algorithm is a quite interesting baseline test, and getting both methods through one implementation seems like a good idea.

The hierarchical methods have quite a lot in common with remerge. The accumulation in memory is equivalent, and the only difference is essentially when to merge and what files to merge. When the choice of implementing a hierarchical method was made, the author was not aware of the experiments performed in [BC06] and [BCL06]. It was thus not known to the author that such methods with suggested improvements had been tested quite thoroughly in the literature. Because hierarchical methods had received some attention in [BC05a] and [LMZ05], they seemed like promising methods to test. The results from [BC06] and [BCL06] proves that the methods are indeed interesting, and a basic version of the scheme still seems like a good basis for comparison.

Hierarchical method 2 was chosen over method 1 because we believed that the sacrifice of query performance in method 1 is not acceptable in practical systems. As will be shown in Section 4.2, some implementation choices are made to enable a lower average update latency in method 2 by sacrificing search speed there as well. By using such an implementation, it was believed that implementing only method 2 was reasonable.

The choice of implementing the naive B-tree index from [CP90] probably seems more strange. [Bjø06] concluded that the structure where the B-tree is the dictionary is probably the most interesting one to test. Implementing that method is not straight-forward though. Apart from having to implement a B-tree, one needs to implement free space management in the inverted file on disk. This was thought to be a quite complex operation, and the comments in [LZW06] confirms this assumption.

Several authors have noted that a method updating the inverted list of each distinct term in each document is likely to have very bad performance [LMZ05, LZW06]. Even if this is the worst case performance of a naive B-tree index, it has some beneficial features as well.

Compared to methods building a partial index in memory, the naive B-tree index will have more free memory to be used for buffering the index. Because a web collection is assumed to be Zipf distributed, we know that some terms are very common. When we are buffering significant parts of the index, it is likely that we are at least buffering the parts containing the most common terms. This implies that inserting new entries in the inverted lists for the common terms will usually not involve any disk accesses. When we have more memory available for buffering, we are able to cache larger parts of the index, and it is thus likely that a smaller fraction of the inserted terms requires a disk access.

Furthermore, because the naive B-tree index is simply a B-tree, producing serializable histories

under concurrent accesses in it is a well studied field. Such a feature is likely to have a positive effect on update latency because an update can be initiated as soon as it comes in. We therefore choose to support serializable histories.

In Section 3.2.2 we noted that appending new entries at the end of an inverted list required us to know that all incoming document numbers are larger than the previous ones. The naive B-tree-index does not require any ordering of the input, and replacing deleted document numbers is thus not a problem. No matter what length the inverted list has and regardless of size of the document number for the inserted document, only the part where the new entry should be inserted is accessed.

Another motivation for testing this method is that creating a B-tree producing serializable histories under concurrent accesses is an essential part of the method originally thought to be the best in [Bjø06]. The naive B-tree index is an example of an eager updating scheme according to the classification in [CH98]. As their experiments with eager updates seem to have tested a quite naive approach, it seems reasonable to test one that is not quite as naive. The naive B-tree index is in our opinion likely to be more efficient than the method tested in [CH98], especially because not all entries in the dictionary are accessed for each added document. The author has not been able to find any reported results from experiments with the naive B-tree index. The reason can be that it has shown modest performance. Testing a method that may prove to be inefficient is not necessarily a bad idea in a master's thesis, because one might gain important experience.

Following the given argument, the naive B-tree index was chosen as the last method to test despite its inherent drawbacks including a quite large space overhead and a possibly slow indexing speed.

4.1.2 Why implement a Search Engine from scratch?

To experiment with indexes, one needs an implementation to experiment with. There are several open-source search engines available, and we could either choose to extend one of these, or to implement our own from scratch. Lucene from Apache is one of the most prominent open-source search engines[1]. The initial choice in this report was narrowed down to choosing whether to extend Lucene, or the search engine the author participated in developing in the course *TDT4215 Knowledge in Document Collections* at NTNU, called Brille.

When one is to modify the index structures of a search engine, it is likely that many other parts of the application are affected. It is therefore tempting to do this in a system one already knows, because it might be very time consuming to get familiar with code written by others. Based on such an argument, Brille was decided on to be the basis of the implementation in this report. As noted in [Bjø06], Brille suffers from some drawbacks. This was part of the reasons why the experimental results in [Bjø06] where damaged by disk caching in the operating system.

The main problems with using Brille in its original form for this report was:

[1] http://lucene.apache.org/

- Brille had support for a lot of ranking schemes, clustering algorithms, collocations and so on. These features are not of particular interest in this report, and the maintenance costs associated with keeping them when changing the index structure are probably not justified.

- Because Brille did not perform any buffering of the index by itself, we relied upon the operating system to perform it for us. As is shown in [Bjø06], the operating system typically did this job quite well, but relying on it makes the results very sensitive to other load on the system. It is also generally tempting to have control over such behavior in an application.

- The document manager was memory based in Brille, making its scalability questionable.

The implementation thus started with fixing the second problem, by creating a buffer pool. Eventually it turned out that the maintainability costs for the other parts of Brille became a significant workload because of large changes to the way disk accesses where handled, and how the index works. It was therefore decided to delete all the old code, and build everything from scratch. Such a solution also has the benefit that it is easier to distinguish what parts of the code that should be considered as part of this report.

4.2 The new Brille

This section aims to give an overall explanation of how Brille works, and explain some of the more important choices in the implementation and why they were made.

The section starts with an overview of the design principles used throughout the implementation. These principles lead to limitations regarding the scope of the implementation. Then, the overall structure of Brille is presented, after which all of the implemented methods are presented. Some of the various other features in Brille with their implementations are represented in the end.

4.2.1 Overall design principles

When the development of Brille started, some initial guidelines for the whole implementation were decided upon. The intent of the different guidelines may either be to limit the scope of the implementation, or to help ensure that the implementation is of descent quality. This section will explain these principles and why they were chosen:

- **All implementation is done in Java:** Java is the author's language of choice. It therefore seemed like a natural choice, even though it is a common opinion that C++ is a more efficient language when it comes to processing. Regardless of whether this is true or not,

it is certainly more efficient to implement something in a language one already knows, and we therefore chose Java.

- **Using Java NIO buffers:** The *java.nio* library has been made to provide efficient I/O in Java. Because a search engine typically is an I/O-intensive application, using *nio* seems like an obviously good idea.

- **Managing buffers in a buffer pool:** Instead of relying on the operating system to perform caching of the index files, a buffer pool is to be created within the application. This will give more control as to what is currently buffered, and it might be an area for future work to make it better adapted to the problem at hand.

- **Have a concious approach to using main memory:** This principle can actually be considered to be a requirement to create a search engine that scales reasonably well. This principle has for instance been the basis of a choice on how to implement the in-memory accumulation of inverted files, as described in Section 4.2.3.

- **Supporting the tf-idf ranking scheme:** Ranking is an important concept in search engines as most queries are ranked [LZW06]. To support ranking, a ranking scheme must be chosen, and there are several available. Because the vector space model and tf-idf are introduced as the most common basic one in both [BYRN99] and [WMB99], it seemed like a natural choice. Section 4.2.6 will give insight into the consequences of this choice.

- **Do not support index compression:** Index compression is widely used both to limit the space usage of the index, and to obtain faster construction and query evaluation, as mentioned in Section 2.2.3. If the mentioned operations should become faster, their performance have to be bound by the disk usage. According to [WMB99], this is usually true for search engines. Despite these obvious advantages, implementing compression requires some extra amount of effort. This is especially true in the naive B-tree index, because the author has not got any particular experience with compression in such a structure. It thus seems like a reasonable limitation of scope not to implement it.

- **Do not support deletions, but make sure that a future implementation of deletion will not require too much changes:** It is a common restriction within this field to only consider a monotonically growing document collection [LZW06, BC06, BCL06]. We will limit the scope of this project with the same restriction, but will not make any choices that makes a future implementation of deletion significantly more complex.

- **The index should be a word-level inverted index:** Word-level granularity is the most commonly used one within this field [LZW06, BC06, BCL06], and this choice obvious. We do not choose to implement support for phrase queries though, because it is not believed to add significant further insight about the search performance of an index structure. Searching for phrases only involves a slightly more sophisticated merging process in memory over searching for multiple terms.

4.2.2 The architecture of Brille

This section aims to give an introduction to the architecture of Brille, without going into much detail. The idea is to give the reader an overall understanding of the system.

We will first give a class diagram describing how the system typically looks like regardless of the index structure used. Such a diagram is given in Figure 4.1. The class diagram shows that there are two interfaces that are implemented by all masters, *Indexer* and *Searcher*. There exists no class called *Master* in the source code for Brille, but there is one master for each index structure.

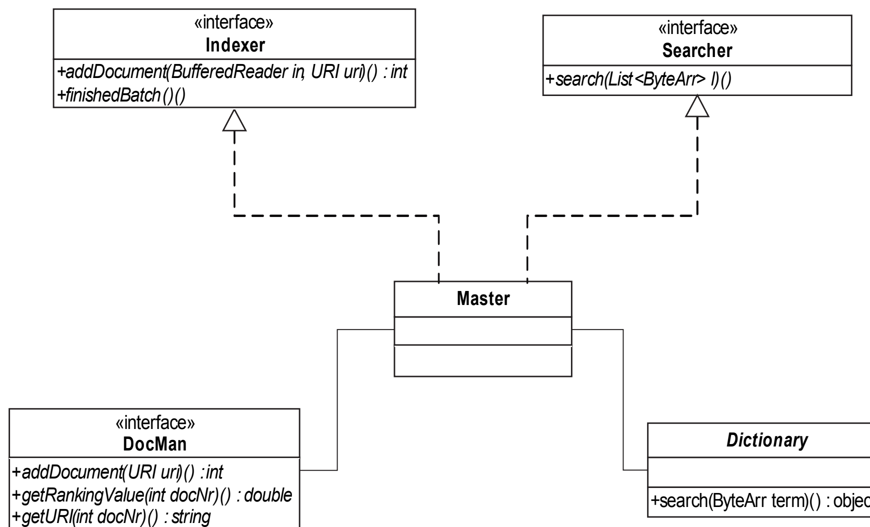

Figure 4.1: Generic class diagram for Brille

This master contains methods to add a new document to the index, and to finish a batch of documents. The method for finishing a batch is called when you want to make all added documents searchable. The master also contains a method for searching for a list of terms. This method returns the results ranked according to standard tf-idf ranking.

Figure 4.1 also contains one other interface and one other class. The interface is called *DocMan* and represents that each Master has a document manager. The primary tasks of the document manager is to have an overview of the relationship between document number and document URI for all documents, and to store the length of each document as calculated in the tf-idf ranking scheme. It provides method to add a new document, manipulate the tf-idf-length of documents and retrieve these lengths.

The *Dictionary* is an interface which declares methods available in all different dictionaries. The most important method here is the one to search for a term, which returns an iterator over all hits in the given dictionary. While a *RemergeMaster* has only one dictionary, a *HierarchicIndexMaster* may have several. For the naive B-tree index, the dictionary is actually also the complete inverted index. These distinctions will be made clearer in the next subsections.

To give the reader a clearer view of the flow in the system for both indexing and searching, some sequence diagrams will be presented next. To avoid making these diagrams to abstract, we will now make a distinction between the naive B-tree index on one hand, and remerge and the hierarchical index on the other. We will first concentrate on remerge and the hierarchical indexes, and Figure 4.2 contains a sequence diagram representing what happens when a document is added.

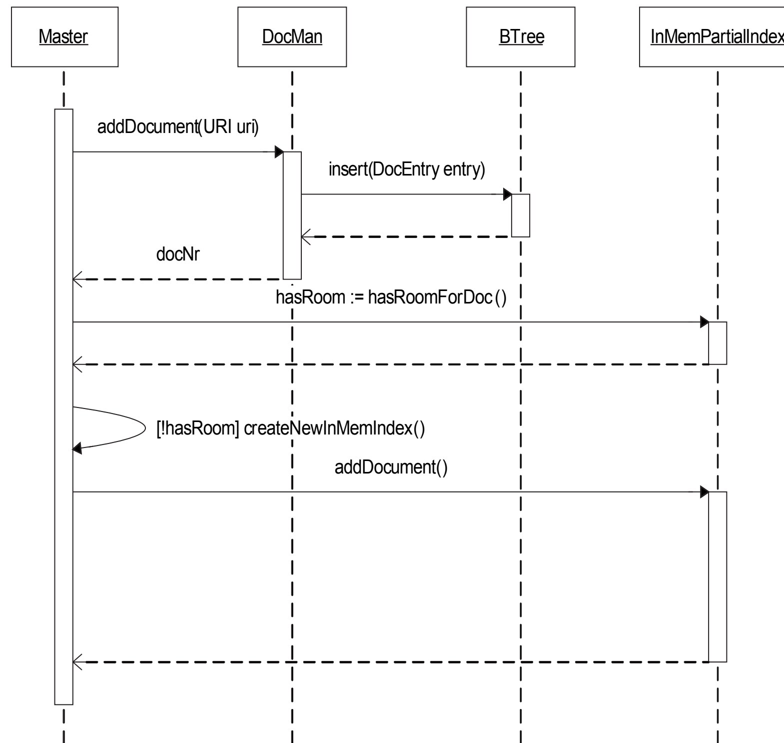

Figure 4.2: Sequence diagram for adding a document in remerge and hierarchical indexes

As can be seen from Figure 4.2, the document is first added to the document manager, where the URI is stored, and the document number for this document is returned. Note that the document manager inserts entries in a B-tree. This will be detailed in Section 4.2.6.

Next, it is checked whether the current *InMemPartialIndex* can accommodate the current document to be added. There is actually no method call that tests specifically for this in the implementation, it is incorporated into the *addDocument*-method in *InMemPartialIndex*. The method call is included here for clarity. If there is no more room in the current in-memory index, a new one is created. At this point, the two methods differ. The current in-memory index will be flushed and the hierarchical method will make the documents added to the old in-memory index searchable, while remerge will not when it builds an index off-line. Either way, a new in-memory index is created, and the new document is added to it.

In the naive B-tree index, the process of adding new documents is slightly different, as can be seen in Figure 4.3. The document is first added to the document manager as before. Each distinct term in the document is then added to the *FullBTreeIndex*. Adding the index entry for one term initiates two operations on the B-tree. This issue will be commented in Section 4.2.5.

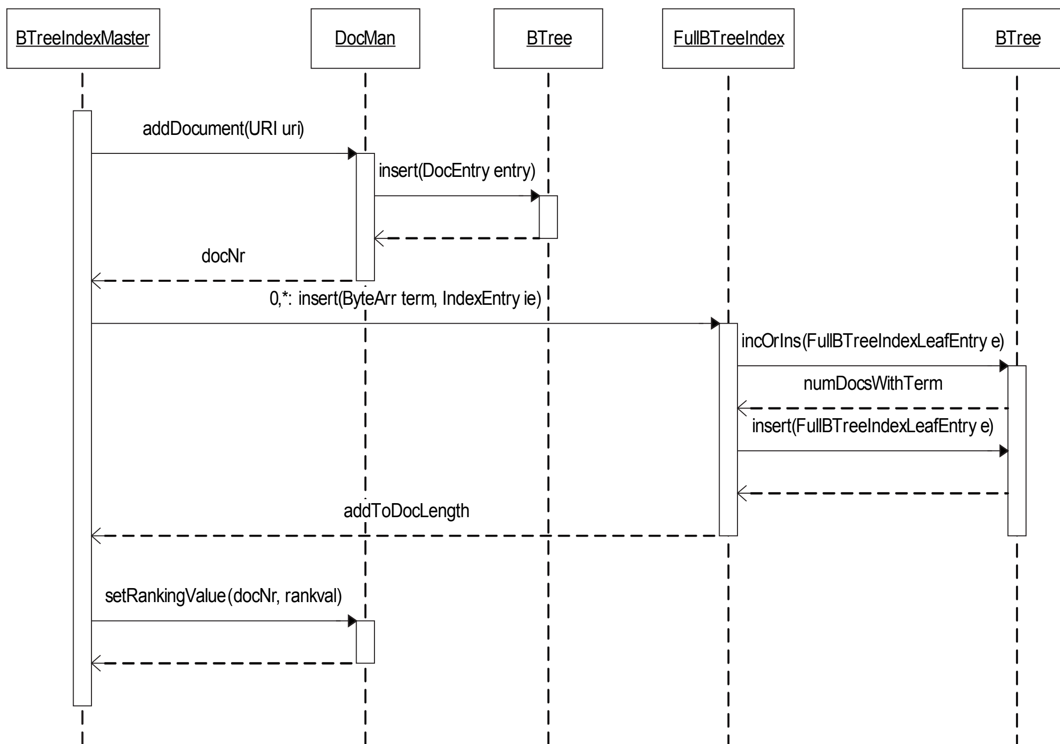

Figure 4.3: Sequence diagram for adding a document in the naive B-tree index

While adding the distinct terms in the document to the naive B-tree index, the value to be added to the length of the document for each term is returned and aggregated. Eventually we end up with the tf-idf length, which is set in the document manager. This will lead to an approximation of the tf-idf score, and this issue will be discussed in Section 4.2.6.3.

In Figure 4.4, a sequence diagram describing searches in Brille is given. This process is more or less similar for all indexes, and we thus only give one general sequence diagram for this operation. All searches in Brille are ranked ones. One may search for several terms, but phrase queries are currently not implemented even though the index structure supports it.

The first thing that happens in a typical search is the actual search for the given terms. Each of these searches returns a *SearchResultHandle* which is a handle capable of iterating over the results for a particular term. All these handles to search results are then added to a *SearchResultMerger*. A *SearchResultMerger* is capable of returning all ranked results from this search by performing a multi-way merge based on a priority queue of the handles. Some searches request the top X ranked results. If so, these are gathered in a priority queue, as described in [WMB99].

When we have found the results to return, we look up these documents URIs in the document manager. The results are then returned sorted by decreasing tf-idf relevance.

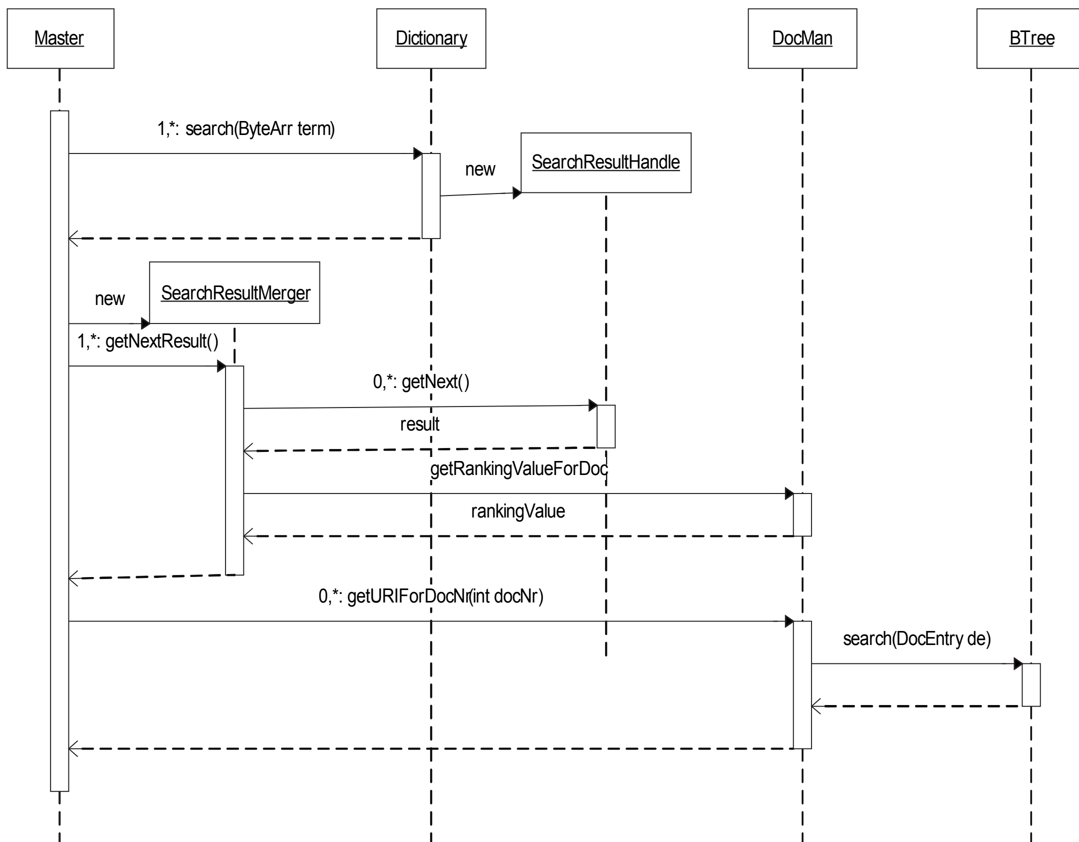

Figure 4.4: Sequence diagram for searching

4.2.3 Remerge

In remerge, new documents are added to an in-memory index. When main memory is full, this partial index in memory is written to disk. When using this method to construct an index, as introduced in Section 2.2.2, several such partial indexes are written to disk before a multi-way merge at the end makes all the documents searchable. When using remerge as an updatable method, each partial index is merged with the main index immediately. The updatable version was introduced in Section 2.4.2.2.

As mentioned in Section 2.2.2, different versions of this method typically differ on which dictionary they use, and how the in-memory index is accumulated. These are the two aspects we will discuss here, while the reader is referred to the source code in Appendix C for all further details.

4.2.3.1 Chosen dictionary

When choosing what kind of dictionary to use, there are several aspects to consider. First of all, we want a fairly efficient structure. We believe that it also is a good idea to use a structure

which is quite easy to understand. We do not assume that the dictionary can fit in main memory, and we thus need to store it on disk. It is therefore important to choose a structure that enables caching the parts stored on disk in a buffer-pool, and provide look-ups with a small number of disk-accesses. Several different structures have been used in the literature [LZW06, RE04], and we have no reason to believe that some are definitely superior over the other.

To make the chosen structure easy to understand, Brille uses a dictionary that is essentially a sorted list of the terms. A search in a sorted list is performed through a binary search. Because we want to store the dictionary on disk, it is important to avoid the random accesses caused by a binary search on a disk based structure. We therefore divide the sorted list into blocks where each block has the same size as a buffer in the buffer pool. The last term in each block is stored in a list that is always kept in memory as long as the dictionary is searchable. We may thus binary search the in-memory list first, before the buffer possibly containing the term search for is read in.

A binary search is then performed in the correct block to obtain the dictionary entry. This structure is similar to the structure outlined in [RE04]. Note that terms have different lengths. To enable a binary search in the blocks, we need to store some information telling us where the different terms start and how long they are. This is done by storing a list of pointers at the end of the block, while the terms are stored starting at the start of the block. These two lists will grow towards each other, and the block is considered full when they will collide if we store the next term in between.

An advantage of this structure is that it only requires a maximum of 1 disk seek per look-up in the dictionary. It is also easy to understand, and scales reasonably well with respect to main memory consumption.

It should be noted that a quite different strategy is used in [BCL06], when making a dictionary for the terms with inverted lists that are shorter than a given length. They actually do not keep all terms in the dictionary. When several short inverted lists are stored within one disk block, only the lexicographically smallest term within that disk block is stored in memory. When searching in such a dictionary, a binary search in the list in memory is performed. The disk block pointed to from the entry found in the binary search is read in. The given disk block typically contains several inverted lists, and one has to read all of them to find the one for the term searched for. The advantage with this structure is that the dictionary is smaller, especially when only storing short inverted lists. In addition, the possible disk access introduced within the dictionary is avoided. The disadvantage on the other hand, is that one needs parse more data from the disk. Because we do not discriminate between long and short inverted lists in Brille, the benefits of this structure are not thought to be significant in our setting. We therefore chose the more traditional approach introduced above.

4.2.3.2 In-memory accumulation

The next aspect to consider in the implementation of remerge is how the partial index in main memory is accumulated. [HZ03] gives an overview of the most promising proposed methods

in this field, and they also come up with a new suggestion which they argue is better than the previous ones. We will give an introduction to the new method they propose and the old method that is the fastest one for construction.

The fastest old method is called the sort-based approach, and it proceeds as follows. When documents are parsed, triplets of the form $(t_{nr}, d_{nr}, f_{t_{nr},d_{nr}})$ are stored in memory, where t_{nr} is the term number, d_{nr} is the document number and $f_{t_{nr},d_{nr}}$ is the frequency of the term with term number t_{nr} in the document with number d_{nr}. When the memory is full, these triplets are sorted, typically using a form of Quick sort, and written out as a partial inverted index. When all documents are added, a multi-way merge process, as described in Section 2.2.2, is performed.

Notice that the traditional sort-based approach consider term numbers, not the actual terms. This implies that we need a mapping between the actual terms and the term numbers. It also implies that the final index will have inverted lists stored sorted on their associated term numbers, not lexicographically on their terms. Both these problems are fixed if we choose to store the actual terms instead, but it might lead to a slightly poorer utilization of main memory.

The new method introduced in [HZ03] is called the single-pass approach, and works by keeping a small inverted index in memory. It consists of a dictionary which is typically a hash map, and each term in the dictionary maps to a byte vector. When this byte vector becomes full, a new and larger one is allocated to allow more entries to be added. When the memory is full, this partial index is written to disk. This process goes on until the final multi-way merge.

Results from experiments reported in [HZ03] show that the single-pass approach is slightly faster than the sort-based approach, but there are more aspects than speed to consider when implementing a method.

When using memory to accumulate indexes, it is important to be able to control the amount of memory used. Because a significant part of the available memory is used by NIO buffers in our implementation, it seems like a natural approach to use these for storing the partial index as well. This approach will also make it simple to evaluate the amount of memory used for the partial index.

If we are going to use the buffers as memory for the partial index, we have a different basis for considering which method to implement. To implement the single-pass approach, we have to implement a memory allocation-routine for the buffers, and find a way to minimize fragmentation. This is certainly feasible, but according to [LZW06] it is quite complicated to implement an efficient allocation routine within a file on disk, and this is essentially the same problem. In addition, implementing a dictionary within the buffers seems quite cumbersome, and it is likely to be significantly simpler to keep it out of the buffers. Keeping the dictionary outside the buffers would introduce more uncertainty regarding the amount of memory used at any given time.

To keep the implementation reasonably simple, the sort-based approach was thus chosen. It should be noted though, that two papers the author was not aware of when this method was chosen, describes a method which should be faster, while at the same time being simpler to implement in our setting [BC05c, LL07]. The method in [BC05c] keeps buckets with similar

sizes. All terms in the dictionary get one bucket allocated initially. When the bucket for one term runs full, a new one is allocated, but the list is not relocated. A pointer is rather stored in the old bucket to the new one, and we end up with a linked list of buckets for each term in the dictionary. This leads to far simpler memory management, and according to experimental results the speed improvement is approximately 30%. It is quite likely that this would have been the best approach to implement in Brille, but such an implementation job was not initiated when the project was approaching its deadline.

Even if the sort-based approach was chosen, a straight-forward implementation of it is not the only option, and we ended up with a slight modification in this project. The author has not been able to find any presentation of this modification elsewhere.

When using sort-based inversion, there are typically three phases that lead to the construction of a new partial index. First, the new documents are read in one by one. Each document is parsed and its triplets are stored. When the memory becomes full, the triplets are sorted. Assuming that we can accommodate z triplets in memory, the asymptotic complexity of this operation is $O(z \log(z))$. And finally, the partial index is written to disk. In most implementations, the first and last phases are probably I/O-bound, while the second is CPU bound. It thus seems like a good idea to make two of these phases overlap to achieve a speed-up.

It is not straight-forward to make the first phase overlap with the second, but it is quite simple to make the second and third overlap. This could for instance be done by implementing a sorting routine that scans through the triplets z times, finding the smallest one at each run. This would enable us to know when the first buffer is full, and it can be written to disk while the triplets to go in the next buffer are found. Such a scheme is not likely to be efficient though, because of the $O(z^2)$ complexity of the sorting routine.

Instead, we introduce a scheme in Brille that both allows us to overlap these two phases and achieves a better asymptotic complexity for sorting the triplets. In addition, the triplets are actually not triplets any more, and the memory usage is hence more efficient. We also use the actual terms instead of term numbers, to avoid the problems mentioned previously, even though this might lead to poorer space utilization. Next follows an introduction to how our method works.

Figure 4.5: In-memory accumulation of index

When accumulating documents in the in-memory index, we basically have an array of buffers. Such a structure is represented in Figure 4.5. When new documents come in, their terms are stored lexicographically from the beginning of the array together with a list of the occurrences of the term in the document. The document number together with pointers to where this document starts and ends is stored at the end of the array. The pointers are denoted as st_i for the start of document i, and e_i for the end of document i in the figure. This process continues until the two lists meet. At this point, the memory is considered full, and we start the second phase, which now includes the two last phases in the standard sort-based approach.

Because we know that the terms within each document are sorted, we only need to compare the first term for each document to find the first entry to be flushed. We can actually build a priority queue with one entry per document and perform a multi-way merge in it. By using a standard representation of a priority queue in an array, this priority queue is actually represented in the list of documents at the end of our array. By performing the multi-way merge in-place, we are using only constant memory apart from the buffers for the in-memory index.

This process will have an asymptotic complexity for the sorting equal to $O(z \log(N_p))$, where N_p is the number of documents in the partial index. This complexity is better than $O(z \log(z))$ as long as there is more than one distinct term in each document on average. As empty documents are not added to the priority queue, it can never have worse complexity than the sorting in standard sort-based inversion. The method does not need to store triplets, but rather the terms together with a list of occurrences. On the other hand, it needs to store pointers to where this document starts and ends in the array. This scheme will lead to better memory utilization if the number of distinct terms in each document is larger than 2 on average, which is a highly likely scenario. In addition, it makes it simple to overlap the I/O in the third phase of standard sort-based inversion with the computation in the second step, which is also beneficial.

Compared to the alternative approaches discussed in [HZ03] and [BC05c], it is uncertain how our strategy performs. But because it is likely to be more efficient than standard sort-based inversion, it is likely to be reasonably close to the single-pass method. A further comparison of these strategies is deferred to future work.

4.2.4 Hierarchical index

The hierarchical method will reuse part of the code from the remerge method. Specifically, it will use the same method for accumulating a partial index in memory. When and which indexes to merge on the other hand, is inherently different and that is what we will focus on here. Section 4.2.6.2 will describe how the tf-idf ranking scheme is implemented for the hierarchical index.

When implementing a hierarchical index, one has the basic choice of whether to implement method 1 or 2, or similarly the sqrt merge or logarithmic merge [LMZ05, BC05a, BC06]. Method 2 gives the best search performance while sacrificing update performance while it is the other way around for method 1. While implementing method 1, we have discovered that it is actually possible to optimize it for low update latency by changing some parameters. We thus chose to implement method 2, but with a slight modification to enable better update latency. How this is accomplished is described next.

When a partial index is to be merged with the hierarchy of indexes in a straight-forward implementation of method 2, it will initiate a merge of all small indexes in the hierarchy that can not fit themselves and all smaller indexes in their position. Such a merge may actually be a merge of all indexes. If we can not add another document until this process finishes, the update latency may become large.

The implementation in Brille follows another principle: There is only a background thread that

is allowed to perform merges into the hierarchy. This implies that when a partial index is full, it is written to disk, and its ranking values are updated in the document manager, as described in Section 4.2.6.2. The partial index is then added as a small index, in which searches may be performed. The added documents are thus searchable. The background thread will note that a new small index is added, and this will initiate a merge just as in a regular implementation of method 2.

While a merge is processed, new documents may be added, and their partial index might be written to disk. If a large merge is processed and a lot of new documents come in, we might get a large number of small indexes. Having a lot of small indexes will sacrifice search performance significantly, and we thus introduce a new tuneable parameter, called T. This parameter describes the maximum number of small indexes that can be part of the searchable indexes at any given time. When a partial index is flushed, it checks whether it may create another partial index. If it can not, because the number of partial indexes is equal to T, it will wait until it is notified by the background thread performing merges. When it is notified, a new partial index is created.

Note that setting $T = 1$ makes this modification similar to a standard implementation of method 2. Setting T to be larger than 1 will give lower update latency, but sacrifices search speed.

By using such a method, it is believed that we get the same flexibility as we would have gotten from implementing both method 1 and 2. We therefore choose to implement this special version of method 2, and skip method 1.

4.2.5 Naive B-tree index

The main difficulties when implementing a naive B-tree index is to implement the B-tree. An important requirement for the B-tree is that it supports keys with variable length. Some insight into how the B-tree is implemented will be given in Section 4.2.8. Here we will focus on how the part of the ranking performed at search time can be processed efficiently, while Section 4.2.6.3 will describe how the tf-idf lengths of documents are maintained. We will also introduce some restrictions on what can be stored in the B-tree.

According to the tf-idf ranking scheme with the formula given in Equation 2.2, we need to know the number of documents containing a term searched for to calculate the rank given to the particular document from the given term. As should be obvious from Figure 2.8, a search in the naive B-tree index for a particular term will have to traverse all occurrences of a particular term to find this number. Because this would require two traversals of all hits for each search, it would be quite inefficient. We will therefore try to avoid it.

To do so, we introduce one extra entry per unique term in the B-tree. This entry contains the term and has a document number of -1. Because of the low document number, this entry will be found at the beginning of the list for this particular term. It will thus be the first entry we return from a search for the term. In this entry, there are no actual occurrences of terms included, but rather two integers. One of them represents the approximate number of documents in the

collection that contains the given term, and the other the total number of occurrences of the term in the document collection.

Using such a scheme, we may use these numbers in the ranking and will thus not need to traverse the occurrences twice or keep them in memory. The disadvantage of the approach is that we need to insert two values in the B-tree for every distinct term in every document. This is likely to have a significant impact on the construction time for the index, but we consider efficient searches to be more important. In addition, we will never be completely sure that the calculated ranking values are exactly correct. The insert of an entry describing the occurrence of a term in a document involves updating the number of occurrences of the term first, and then inserting the actual entry. If a search goes on while this entry is inserted, it might read the updated number of documents containing the entry, but not the entry describing the occurrence of the term in the document currently being added. The ranking will therefore possibly be slightly wrong, but following the argument that tf-idf is a heuristic, the error is not considered significant.

It is not possible to store an entry in a B-tree that consumes more space than a node. Brille uses a word-level index, and some terms are very common. The term ”the” occurs more than 11000 times in one of the documents in the collection used in the experiments in this project. To avoid that the solution fails on such inputs, we chose to put a maximum length of the terms indexed, and also a maximum length on an entry to be stored. Terms that are longer than the maximum length is not indexed. If an entry is longer than the maximum, the last occurrences are not listed, but we still store the actual number of occurrences.

4.2.6 Document manager

As mentioned previously, the document manager has two main tasks:

1. Maintain the mapping between document numbers and URIs.
2. Have an overview of the tf-idf length of the different documents.

We keep in mind our basic principle of limiting the main memory consumption. A solution which keeps all this information in memory will not follow that principle, and some of it should be stored on disk. To decide what to store in main memory and what to store on disk, we need to consider the access pattern for the information stored in the document manager.

Brille only supports ranked queries. A typical ranked query will ask for the top ten ranked documents, although it is possible to retrieve all results. If one asks for the top ten documents, all documents containing any of the terms searched for will have their ranks calculated. To do so, we need to know the tf-idf length of the document. When returning the top ten results, we also need the URIs for these documents. This scheme is depicted in Figure 4.4. We note that we may need the tf-idf length for a lot of documents, but that we only need the URI for a few of them. If a user asks for all documents containing any of the terms, we will actually need the URI for all matched documents, but this is not considered a likely scenario.

Based on the discussion given above, it seems like a natural choice to store the tf-idf lengths of documents in main memory, while storing the mapping from document number to URI in a disk-based structure. The disk-based structure used in Brille for this purpose is a B-tree. This choice was quite obvious because a B-tree has been implemented for other purposes as well, and is thus easily available. The structure used to store the tf-idf lengths of documents in memory is discussed in Section 4.2.6.1.

To provide some insight into how the document manager maintains the length of the different documents in a quite basic setting, we will now give a quick overview of how it works for the remerge method. In a situation where we first build an index of reasonable size, the document lengths are easily calculated while the partial indexes are merged. If we now add another batch of documents, consisting of some partial indexes, we will have new lengths for the new documents. But the lengths of the documents in the old index will also have to be updated to be correct. This can be seen from the idf-part in formula 2.3 where N and n_i are present.

We want to be able to search in the old index while merging the new one, and we hence need to keep the old value in the document manager. To be able to calculate the new values at the same time, a shadow copy of list of document lengths is kept. When we switch to the new index, the current list of document lengths is switched with the shadow copy and we thus have correct document lengths for the new index. It should be noted that the lengths of the old documents will not change significantly when new ones are added to an already large index. The overhead of calculating these extra values is relatively small though, because the whole index is read and written anyway. We therefore chose to calculate all lengths from scratch at each merge.

It should be obvious that this scheme does not work in the same way for the other two methods, and the issues introduced by them are discussed in sections 4.2.6.2 and 4.2.6.3.

4.2.6.1 Limiting the main memory requirement

We have decided to store the tf-idf document lengths in memory, and we thus need to choose a structure to store them in. They are accessed when we know the document number we want to obtain the length for, and they have to be extendable because we want to be able to add more documents. An array with array-doubling therefore seems like a natural choice of structure.

Java contains a class *ArrayList* in its API. When using it to store floating point numbers, a natural choice would be to use an *ArrayList<Double>* or *ArrayList<Float>* object. It is probably not a good idea to use *ArrayList*s with simple types, a fact the following small experiment will show.

To show the memory usage and speed of an *ArrayList<Integer>* versus a self-made implementation of array doubling for integers, the programs shown in figures 4.6 and 4.7 where run.

These programs where run on the same computer used for running the actual experiments in this report. Its specification is given in Chapter 7. As can be seen from the code, one of the classes uses *ArrayList<Integer>*, while the other one uses *IntArrayList*, a class included in Brille and provided in Appendix C.8.

```
import java.util.*;

public class test_arraylist{

  public static int doWasteWork(){
    int a =0;
    for(int i=0;i<1000000;i++) a+=i;
    for(int i=0;i<1000000;i++) a-=i;
    return a;
  }

  public static int calcSum(ArrayList<Integer> l){
    int sum = 0;
    for(int i=0;i<l.size();i++) sum+=l.get(i);
    return sum;
  }

  public static void runGC(){
    for(int i=0;i<100;i++) System.gc();
  }

  public static long memoryUsed(){
    runGC();
    return Runtime.getRuntime().totalMemory()-
      Runtime.getRuntime().freeMemory();
  }

  public static void main(String[] args){
    int w = doWasteWork();
    int addelem = 1000000;
    long start = memoryUsed();
    ArrayList<Integer> l = new ArrayList<Integer>();
    for(int i=0;i<addelem;i++) l.add(i);
    long used = memoryUsed()-start;
    System.out.println("added "+addelem+" elements, and used "+used+
        " bytes of memory.");
    System.out.println("That means approximately "+(used/addelem)+
        " bytes per element.");
    int sum = calcSum(l);
    int write=sum-w;
    System.out.println("This is only printed to make sure nothing is"+
        " optimized away. The meaning of life is not 42, it is "+write);
  }
}
```

Figure 4.6: Code for the test_arraylist class

Both programs start out with performing some unnecessary work. This is done to make sure the java virtual machine is done initializing before we start allocating any memory. Then, the list used in the given program is populated with 1 million integers, and we measure the amount of memory this occupies. Some calculations are performed in the end to ensure that no significant parts of the code are removed by optimization. The results from running these programs and timing them with *time* in Linux, is given in figures 4.8 and 4.9.

As can be seen from the results, each integer consumes 32 bytes of memory in an *ArrayList<Integer>*, while it only uses 4 bytes in an *IntArrayList*. Using 4 bytes to represent an integer is expected. We will now try to explain why an *ArrayList<Integer>* uses a lot more.

The first reason is that an *ArrayList* is only capable of storing objects. We will therefore not

```
public class test_intarraylist{

  public static int doWasteWork(){
    int a =0;
    for(int i=0;i<1000000;i++) a+=i;
    for(int i=0;i<1000000;i++) a-=i;
    return a;
  }

  public static int calcSum(IntArrayList l){
    int sum = 0;
    for(int i=0;i<l.size();i++) sum+=l.get(i);
    return sum;
  }

  public static void runGC() throws Exception{
    for(int i=0;i<100;i++) System.gc();
  }

  public static long memoryUsed() throws Exception{
    runGC();
    return Runtime.getRuntime().totalMemory()-
      Runtime.getRuntime().freeMemory();
  }

  public static void main(String[] args) throws Exception{
    int w = doWasteWork();
    int addelem = 1000000;
    long start = memoryUsed();
    IntArrayList l = new IntArrayList();
    for(int i=0;i<addelem;i++) l.add(i);
    long used = memoryUsed()-start;
    System.out.println("added "+addelem+" elements, and used "+used+
        " bytes of memory.");
    System.out.println("That means approximately "+(used/addelem)+
        " bytes per element.");
    int sum = calcSum(l);
    int write=sum-w;
    System.out.println("This is only printed to make sure nothing is"+
        " optimized away. The meaning of life is not 42, it is "+write);
  }
}
```

Figure 4.7: Code for the test_intarraylist class

```
$ javac test_arraylist.java && time java test_arraylist
added 1000000 elements, and used 32003080 bytes of memory.
That means approximately 32 bytes per element.
This is only printed to make sure nothing is optimized away.
The meaning of life is not 42, it is 1783293664

real    0m18.096s
user    0m17.939s
sys     0m0.126s
```

Figure 4.8: Results when running the test_arraylist class

```
$ javac test_intarraylist.java && time java test_intarraylist
added 1000000 elements, and used 4023552 bytes of memory.
That means approximately 4 bytes per element.
This is only printed to make sure nothing is optimized away.
The meaning of life is not 42, it is 1783293664

real    0m2.250s
user    0m2.222s
sys     0m0.025s
```

Figure 4.9: Results when running the test_intarraylist class

use a regular *int* data type, but rather an *Integer* object. In addition to storing the object, the actual array where the array doubling is performed, is a list of pointers to these objects. On a 64-bit architecture like the one in our experiments, these pointers will consume 8 bytes of memory each. Because the doubling factor in *ArrayList* is approximately 1.5, each *Integer* object consumes between 20 and 24 bytes of memory. We will not go into a discussion of how this memory is used, but note that the overhead is significant.

We also see that the timings suggests that the *IntArrayList* class is way more efficient. This is partly because it does not need to allocate as much memory, and partly because an array of integers is typically stored contiguosly in memory while the Integer objects may be spread randomly. This makes it way more efficient to use an array of integers because of caching effects.

We conclude that through making our own implementations of array doubling for simple types, we both get a more memory efficient solution and a faster solution. A *DoubleArrayList* is thus used to store both the document lengths and the shadow copy. Even though this will occupy a significant amount of main memory, it will now only be approximately 8 bytes per document in each list. For 10 million documents, these two lists will occupy approximately 76.3MB of main memory, and this is thought to be reasonable.

4.2.6.2 Adaptation to hierarchical index

As noted above, it is not feasible to update the tf-idf document lengths in the same fashion as for remerge in the other indexes. This section will provide an introduction to the problems caused by the tf-idf scheme for a hierarchical index, and what choices were made in Brille to solve these problems.

According to the tf-idf ranking scheme, the document length for each document should be calculated as shown in Equation 2.4. We note that this formula says that the length of each document is dependent on both the total number of documents, and on the number of documents containing the terms in this document. This actually implies that the lengths of all documents should change when a single new document is added. In the hierarchical index, we should thus

recalculate all document lengths when a new batch of documents from memory is added as a small index. Doing this is likely to lead to quite modest performance, and we would like to avoid it.

Tf-idf is basically a heuristic used to approximate which documents that are likely to match the users query. We consider it to be reasonable to change this heuristic slightly to achieve better performance. In Brille we follow the following principles for tf-idf lengths in the hierarchical index:

1. A document gets its length (re)calculated when it is:
   (a) part of a merge.
   (b) flushed as a small index for the first time.
2. When a (re)calculation of the length of a document is performed, it will always use the number of currently searchable documents as N. It will count the number of occurrences of a term, n_i, as the number of occurrences within the currently processed documents and the largest current index(if the largest index is not part of the current merge).

By following these rules, it is believed that the document lengths are kept reasonably up to date. At the same time, the approach does not make it impossible to make the newly added documents searchable within reasonable amounts of time.

4.2.6.3 Adaptation to an incrementally updatable index

The same problem as for the hierarchical index occurs for the naive B-tree index as well, and it has possibly even more impact here. The naive B-tree index should be able to add one and one document to the index, and recalculating the tf-idf lengths for all other documents as well would be prohibitive performance wise.

As mentioned in Section 4.2.5, there is an additional entry in the B-tree for each unique term in the naive B-tree index. This entry contains the number of occurrences of the term. When the entry for a term in a document is inserted, one first updates the entry containing the number of occurrences. This is done in a special operation in the B-tree called *incOrInsert*. This method returns the updated or inserted entry, and we will thus know how many documents that contains the inserted term. Because we also keep track of the number of active documents in the document manager, it is actually quite simple to calculate the tf-idf length of the new document.

We thus have a scheme for giving each new arriving document its correct tf-idf length, but the problem is that as more and more documents are added, the lengths for the old documents deteriorate. To cope with this problem, Brille has a background thread that on regular basis reads through the whole index and updates all document lengths. The implementation makes sure that no documents that are currently being added get their lengths updated.

As for the hierarchical index, this implementation choice is made to enable better performance while sacrificing the exact heuristics of tf-idf ranking.

4.2.7 Buffer Pool

As mentioned in Section 4.2.1, it was decided early on that Brille should perform buffering from the index files by implementing a buffer pool. This section explains the implementation of such a structure.

The initial implementation of a buffer pool in Brille was quite naive. It managed free and dirty buffers in arrays with array doubling. When pinning a buffer, one searched through all buffers to see if one had the part asked for buffered. If not, one of the buffers was flushed if it was dirty, pinned to the new part, and filled with data from the part asked for. The source code for this implementation is found in the *BufferPool* class in Appendix C.3.

During initial testing, the old version of the buffer pool proved to be a significant performance bottleneck, especially when the application is run with a lot of buffers. This should not come as a surprise because pinning and unpinning buffers is a very common operation in all tested methods. Experiments not reported in this report showed that Brille was able to index approximately 3 times as many documents per second by switching to a new and improved implementation. We will now give an introduction to how the new implementation of the buffer pool works. The code for this solution is also found in Appendix C.3, and a drawing of a snapshot of how the structures in the buffer pool might look at a given time is shown in Figure 4.10.

The new version of the buffer pool consists of three main data structures. The most important one is a *HashMap* that maps a pair of integers to a buffer. The pair of integers represents a file number and a part number. Each index file gets a number when it is added to the buffer pool, and if one wants a part from a given file one needs to know the file number. The part number obviously describes which part of the file you want to access. If one wants part number i, one gets the bytes between $B \cdot i$ and $B \cdot (i + 1)$. If a buffer contains a given part from a given file, it is found with this pair as a key in the *HashMap*.

If a buffer is not currently pinned, it is also found in one of the doubly linked lists in the buffer pool. One of the lists represents buffers that are dirty, meaning that they contain changes that are not yet written to disk, while the other list contains buffers that are identical to their disk-based version.

When another part of the application wants to pin a buffer, several scenarios might occur. If a buffer contains this part at the moment, the given buffer is returned. If this buffer is part of one of the lists, it is removed from it. Note that in order to be able to call a method on an object to remove it from a list, the *LinkedList* class in the API in Java can not be used, because the elements in such a linked list only contains pointers to the actual objects stored in the list. Brille thus contains an implementation of a doubly linked list where the objects themselves are linked.

If there is no buffer currently containing the wanted part, the first buffer in the list of non-dirty buffers is removed from the queue, and its reference in the *HashMap* is removed. It is then inserted in the *HashMap* with the wanted part. The correct part is read into the buffer and the buffer is returned. Because the first buffer in the list of not dirty buffers is used, this buffer pool uses an LRU replacement policy. There are most certainly more sophisticated possibilities, but

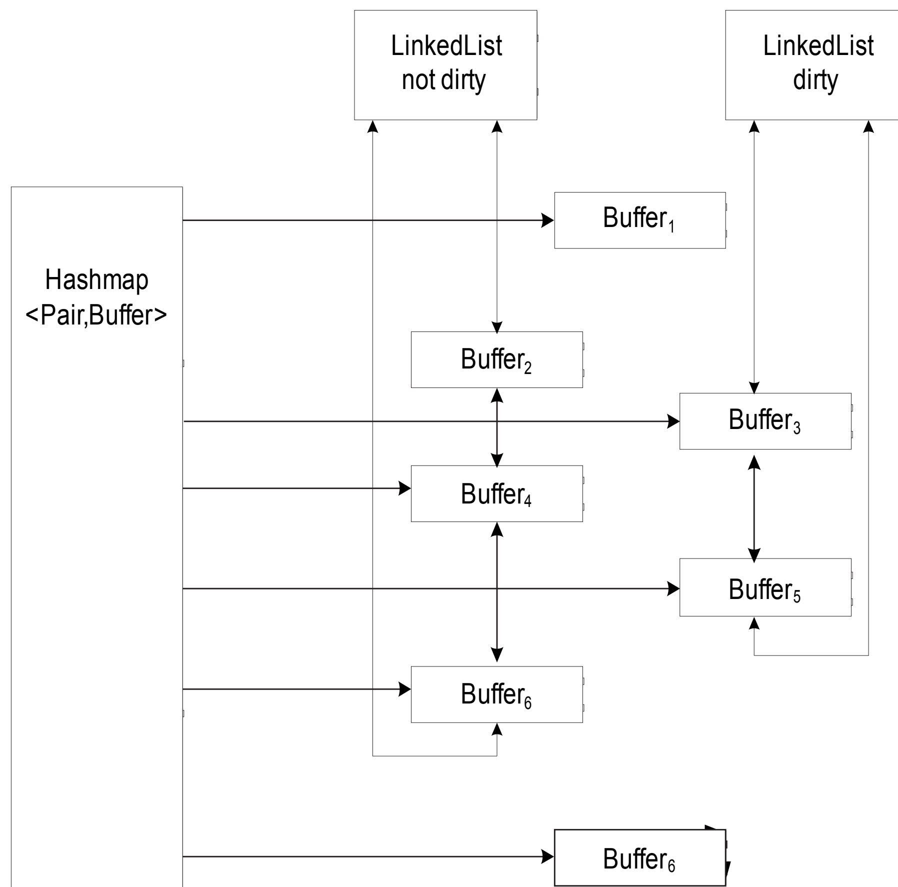

Figure 4.10: The new and improved buffer pool

that is not tested in this report.

To make sure that dirty buffers are eventually flushed, a background thread checks for dirty buffers on a regular basis. When the linked list of dirty buffers is non-empty, the first one is removed, written to the correct *FileChannel*, and inserted at the end of the non-dirty list. A *FileChannel* is an object in *java.nio*. When writing sections of a file to it, it will eventually write it to the actual file. One is capable of forcing it to write out all changes to the file by a calling a specific method, but that is never done in Brille. Not doing so introduces some uncertainty regarding the number of disk accesses performed. It is likely that it is capable of merging several writes. Finding an efficient pattern for when to force the *FileChannel* to flush all its changes is deferred to future work.

To make this explanation clearer, we take a look at the example in Figure 4.10. In the snapshot given in this figure, we can conclude that buffers 1 and 6 are currently pinned because they are not part of any of the linked lists. When they are unpinned they will be appended to either the dirty or not dirty linked list.

We also see that all buffers except buffer 2 buffers a part from a file. We can see that buffer 2 does not contain any part because it has not got any reference to it from the *HashMap*. If a

request for a pin on a part that is not currently buffered is issued, buffer 2 will be used to fulfill this request.

This new approach will require one look-up in the *HashMap* for each request to pin a buffer, and it will also typically remove the buffer from one of the linked lists. It might also require removing and entry from the *HashMap* and inserting a new one. All of these operations have average asymptotic complexities of $O(1)$. This is far more effecient than the linear complexity of the previous buffer pool.

Having a good average asymptotic complexity is very important in a buffer pool because the methods are mostly synchronized, and it is important to not let other threads wait longer than necessary. Its implementation is not straight-forward though, and there has been several bugs due to synchronization issues in this new version of the buffer pool throughout the development.

4.2.8 Implementing a B-tree

The B-tree is an important structure in Brille, because it is both the main index in the naive B-tree index and it is also an important part of the document manager. The way the B-tree is used imposes several requirements on its implementation. Most importantly, we need to support insert, update and inc-or-insert. To make the implementation ready for a future implementation of deletion, we also chose to support delete and dec-or-delete. Dec-or-delete is the opposite operation of inc-or-insert, and should be used when an entry is removed from the inverted list for a given term in the naive B-tree index.

To enable adding new documents while performing searches, both the B-tree in the document manager and the index need to produce serializable histories. Doing so will also enable adding several documents at the same time in different threads in the index. We thus chose to ensure that the B-tree produces serializable histories, and this requirement is fulfilled by making a so-called linked B-tree with a form of tree locking [BHG].

The B-tree supports deletion of entries, but if we want to support deletion of nodes, the locking scheme will have to change quite significantly. We thus chose to not support deletion of nodes. This choice has several implications. The negative effect is that we can no longer guarantee that the space utilization in a B-tree is at worst $\frac{1}{2}$ and on average $\frac{2}{3}$, because all entries in an existing node might be deleted. This scenario is considered quite unlikely in a search engine though, because the size of the index is assumed to increase rather than decrease. This issue will not have any impact in our experiments anyway, because deletions are not tested. The positive effect of not deleting nodes is that the lock contention is less significant, and that the implementation is simpler and thus likely to be more robust.

Both the keys and the data part of the entries may be of variable sizes in the B-trees in Brille, and our implementation has to support this. This requirement is fulfilled by making the structure of the nodes quite similar to the blocks in the sorted list dictionary used in remerge and the hierarchical index.

We will not go into more detail about the implementation of the B-tree in Brille. There are several interesting aspects when implementing a B-tree, but they are not thought to be of particular interest in this report.

Chapter 5

Experiments

When experimenting with indexes one need a document collection to index. The one used in our experiments is introduced in Section 5.1. Section 5.2 explains how Brille obtains information about the utilization of the I/O subsystem and the CPU, while Section 5.3 ends the chapter with an introduction to the experiments performed.

5.1 Document collection

The document collection used in the experiments in this report is the GOV2 text collection from the TREC Terabyte track[1]. GOV2 was chosen because it is the most commonly used collection in related work [BC06, BCL06, LZW06].

This section will provide some statistics for GOV2 and explain how the documents in it is stored. How the collection is parsed in Brille will also be explained.

5.1.1 GOV2

The GOV2 collection consists of approximately 25.2 million documents, mostly html or text documents. The size of the complete collection is 426 GB. The experiments in this report will use various subsets of this collection.

The documents in the GOV2 collection are organized in a particular fashion. The complete collection is partitioned into 273 different directories, with approximately 92000 documents in each directory on average. These directories contain 100 files, and each file thus contains slightly under 1000 documents on average.

Each of the files has a structure resembling an XML-file. It is not proper XML however, because

[1] http://www-nlpir.nist.gov/projects/terabyte/

there is no single root element. Most of the documents are html, while some are in text format. If the document was in a binary format originally, it is represented in a text format in the collection. Brille has to be able to parse the documents, and the following subsection explains how it is done.

5.1.2 Parsing GOV2

Each document in the xml-like GOV2 files starts with some meta-information about the document. The only information extracted about the documents is the content between the $< DOCNO >$ and $< /DOCNO >$ tags. This string is used in Brille as the URI for the document. The rest of the parsing only considers the actual documents.

Tokens are collected from a document as all sequences of alpha-numeric characters. This means that all sequences of English characters and digits are considered to be terms. No stop words are removed, and Brille will end up with a complete index of all terms in text files.

For html files however, the situation is slightly different. The content within tags in such files is not indexed. We do not index scripts and CSS code either. This means that the content between $< script >$ and $< /script >$ or $< style >$ and $< /style >$ is also removed. After removing these segments, the tokenization in html files are just as in text files.

5.2 Obtaining I/O and CPU statistics

There are typically two different resources that may be the bottleneck in a search engine: CPU and the I/O subsystem. Memory is typically not a bottleneck because all algorithms in a well engineered search engine uses constant memory. To get a better understanding of which resource that is the bottleneck in various phases of indexing and searching, we need to measure the utilization of both resources. Appendix C.8 contains code to gather statistics from the I/O subsystem under Linux and FreeBSD, and statistics from the CPU under Linux. The reason why FreeBSD is included is that it was originally planned to run the experiments under FreeBSD because of problems making Linux run on the available hardware. These problems were eventually fixed, and we will only explain the Linux implementation here.

No tool available in Linux known to the author is capable of measuring the utilization of the processor or the I/O subsystem between a start time and a later defined end time. It was therefore necessary to include code capable of gathering such statistics in Brille. Fortunately, there are several useful features under */proc/* in Linux. The main inspiration for the current implementation of this part is in the *iostat* tool which is available on most *nix-systems.

The information gathered about the disk utilization is found in */proc/diskstats*. This file contains information about the number of processed read and write operations for each disk as unsigned integers. The basic idea of the implementation is to read these values at the beginning and end of each phase, and to subtract the start values from the end values. The counters in */proc/diskstats*

might overflow the unsigned int they are stored in, and Brille will give correct statistics if these values are not overflowed more than once during a phase. Information about the CPU utilization is found in a similar way by obtaining values from counters in */proc/stats*.

The information we gather from */proc* is only available in 2.6-kernels. In earlier kernel versions, the same information can be found within *sysfs*, but this is not supported in Brille.

5.3 Experiments performed

When experimenting with the implemented methods, there are two main aspects of each method we want to test, its efficiency when it comes to handling updates, and its search performance. How each of these aspects is tested is explained in the following two subsections. The last subsections will explain the different configurations tested for each method and introduce a last set of experiments carried out to consider the intrusiveness of the output given in the other experiments.

5.3.1 Testing updatability

It is not straightforward to test updatability, and different ways of conducting the experiments have been presented in the literature [LZW06, BCL06]. As mentioned in previous chapters, both update speed and latency are of interest. Testing update latency is probably the most complicated of these two, because it brings up a lot of questions. In our experiment setting, we already have the document collection available, and the most natural way to test update latency is probably to do some sort of simulation of the arrival of different documents. One possible approach is to decide upon a specific feeding rate for documents and measure the average latency for the different methods with this given rate. Choosing the rate is quite tricky though, because a high rate will benefit the methods that batches documents while a slow one will benefit the naive B-tree index.

Rather than to conduct experiments giving a precise answer to the question of which methods that are superior over the others with a given feeding rate, we wish to illuminate both positive and negative aspects of all methods. To do so, we let all methods index as fast as they can, and note when the documents are read in versus when they are searchable. This will give us two graphs, one representing the feeding and one representing the time when different documents are searchable. The area between these two graphs will describe the update latency; while the time spent constructing the final index will describe the update speed.

This experiment will be run with various numbers of documents in the final index. The numbers chosen are: 100000, 300000, 1 million and 10 million documents. The naive B-tree index is too slow to index 10 million documents within reasonable amounts of time, so these experiments were not completed. Because only one computer was available for carrying out the experiments, the limited time frame of this project only allowed the largest index size for each index to be

built 1 time, while the smaller experiments were run 10 times each to be able to measure the variability in the results as well as average speed.

In all experiments with updatability, the documents used are the lexicographically first in GOV2. It is assumed that these do not differ significantly from the others in the collection.

We try to measure the utilization of disks and CPUs in all different phases of construction in each of the methods, in the way described in Section 5.2.

5.3.2 Testing search performance

To test search performance, we test searching for various terms in indexes with different amounts of documents. Because of the Zipf distribution assumed for the terms in the collection, it is likely that the number of occurrences of each term varies significantly.

It is also likely that it takes more time to search for a very common term than a rare one because the amount of data read from disk is typically larger. To consider the effects of searching for terms with varying amounts of postings, search performance is tested with 3 different groups of terms. To find the groups of terms the index for the first 1 million documents was built and the dictionary was dumped to disk. The most common term in this collection is ”the” with 41813109 occurrences in total, which is almost 42 on average per document.

The first group consists of frequently occurring terms, and contains the 100 most frequent ones in this collection. These terms have an average of 3353112.56 occurrences in the document collection consisting of the first 1 million documents from GOV2. The least frequent terms have only one posting each, and this applies to more than half the vocabulary. 100 of these terms were randomly chosen to be the group of the least frequently appearing terms. The last group of terms consists of the ones that comes the closest in number of postings to the square root of the average number of postings per term for the most frequent terms.

By choosing these three groups we hope the capability of the different structures to process searches for terms with various frequency is tested. In all processed search queries we only want the top ten ranked documents to be returned, but this will nonetheless require us to read the complete inverted list for the term searched for. Searching for multiple terms at the same time is not tested because the expected time for such a search has a clear relation to a search for a single term. To search for x terms at once is equivalent to performing x searches, one for each of the terms, minus the time spent retrieving the URIs for $10 \cdot (x - 1)$ documents. Note that this calculation assumes that all searches has at least 10 matches.

All groups of terms have been randomly permuted. The rationale behind such an operation is to make sure the terms within one group are not sorted lexicographically, because this could be a significant benefit for the caching in the dictionaries. All groups of terms are listed in appedix A, in the order in which they are searched for in all experiments. For each index size, the 100 searches within one group are performed one time each. We measure the time spent performing all 100 searches. It is important that we do not search for the same term several times, because

of caching effects.

5.3.3 Tested configurations

We have chosen some methods to test, but each method can be configured to work in many different ways. Testing all different configurations for each method is not feasible. The reader may get an overview of the different possible configurations by looking at the interface *BrilleDefinitions* in Appendix C.1.

All experiments are run with the same amount of memory allocated to the Java virtual machine, and all experiments use the same amount of buffers. Several authors have reported results suggesting that the amount of main memory allocated to the application has reasonably low impact on efficiency [HZ03, BCL06]. Choosing not to vary this parameter thus seemed reasonable. For the methods batching updates we also chose to always allow the method to use the same number of buffers to construct the in-memory index, namely half the buffers in the buffer pool. Brille is a multi-threaded application and it is possible to configure different sleep times for different threads, but testing several such values is assumed to be more time consuming than it is interesting. We thus leave all values as shown in the *BrilleDefinitions* interface included in Appendix C.1.

The remerge method may be run either as an off-line construction method, or by merging with the main index at given intervals. We chose to test it with both these configurations. As opposed to the experiments carried out in [BCL06], we perform the merge at the end in the off-line construction method. For the method that merges partial indexes with the main index at given intervals, one has the choice of when to perform the merge. We choose to do so every time the memory is filled with the partial index. Doing it more often could have resulted in better average update latency, but update efficiency is likely to suffer, and to keep the number of configurations manageable we do not test other configurations for remerge.

The size of each of the buffers in the buffer pool might be varied, and we try two different configurations for all methods, namely 4Kb and 16Kb.

There are more configurable variables in the hierarchical index than in remerge. In particular, it is obviously interesting to vary K, which describes by which rate the maximum sizes of the different indexes increase. We choose to test with $K = 2$ and $K = 4$. The variable T introduced in our method is also likely to have an impact on the results, and we test with $T = 1$ and $T = 4$. Notice that $T = 1$ will make the hierarchical index similar to a standard implementation of method 2 from [OvL80].

The naive B-tree index does not have very many variables, but it is obviously interesting to see if using more feeding threads will lead to more efficient index construction. We therefore vary the number of feeding threads between 1 and 4. Otherwise, we only vary the buffer size like in the other methods.

This leaves us with the configurations for the different methods listed below:

- Remerge:
  - Tested as an off-line construction method and with immediate merges with the main index.
  - Buffer size: 4096 and 16384
- Hierarchical index:
  - K: 2 and 4
  - T: 1 and 4
  - Buffer size: 4096 and 16384
- Naive B-tree index:
  - Number of feeding threads: 1 and 4
  - Buffer size: 4096 and 16384

5.3.4 Test output intrusiveness

As mentioned in Section 5.3.1, we measure the utilization of both disks and CPU in each phase of index construction. The results are stored on disk, and outputting them will obviously have an impact on the results in the experiments. It is important however that the impact is not too significant. Otherwise, the results from the experiments can not be trusted. To get a better view of the impact of these measurements on the results, we run one configuration with each overall method once, to compare the construction time with the obtained average for this configuration. Each index structure has its own pattern of when it outputs results from a phase. Testing one configuration per index structure is therefore believed to be sufficient to consider to which extent the output is intrusive.

We chose to build the index with 300000 documents in all these experiments, and we always use the largest buffer size, namely 16kB. Remerge is run as an off-line construction method, and the naive B-tree index is run with 4 feeding threads. The hierarchical index is run with $K = 2$ and $T = 4$.

Chapter 6

Efficiency model

This chapter provides estimates of the expected performance of the different index structures, based on I/O and CPU usage. To obtain such estimates, we need to know some basic quantities. Section 6.1 will introduce the quantities used in our calculations.

Section 6.2 will estimate the time spent constructing a partial index in memory, and Section 6.3 considers merging indexes. The following sections will provide estimates for each of our tested structures. We will only calculate the exact expected values for the largest index size where the test for the particular structure is run 10 times. We could have calculated for all possible sizes, but we believe the presentation is clearer when we only consider one size. Section 6.7 will provide estimates for the expected time used to process the searches tested in this report.

As will be clear from Section 6.1, the underlying model in this chapter will be partly dependent on our experiment environment and partly on our implementation. The developed model is thus not a reasonable framework for validating the quality of our implementation, but it will hopefully help analyzing the results obtained in the experiments. This may in turn provide guidelines for future improvements of the implementation.

6.1 Model of computation

In search engines there are typically two possible bottlenecks, disks and the central processing unit(CPU). As explained in Section 5.2, memory is typically not a bottleneck in search engines because all the algorithms use constant memory. To achieve reasonable estimates on performance, both I/O and CPU is taken into account.

Obtaining theoretical estimates for disk performance is quite straight-forward, but estimating theoretically the average time needed to parse a term on a given processor is not. We will therefore use more practical methods to estimate CPU performance, which for instance involves running parts of our implementation. By doing so, we do not end up with a strictly theoretical model, but it will hopefully provide some insight nonetheless. We will first consider disk

performance in Section 6.1.1, before we look at CPU performance in Section 6.1.2.

6.1.1 Disk performance

Equation 2.1 introduced the most commonly used model for disk accesses, and it is repeated in Equation 6.1 for convenience.

$$t_d(b) = t_s + t_t \cdot b \tag{6.1}$$

To obtain estimates on disk performance, we need to find values for the disk access time, denoted t_s, and the inverse bandwidth, t_t. There are several ways to determine these values, but for the average disk access time we do as in [HZ03] and trust the disk vendor[1]. According to the vendor, the average time spent moving the disk arm from one position to another is 8.9 ms on the disks in our experiment environment. In addition, we will have to wait until the correct sector is under the disk head. Because our disk rotates 7200 rounds per minute, it will spend approximately 8.33 ms on each rotation. On average, we have to wait half a rotation after the disk head has moved to the correct position. We will thus have an average $t_s = 13.1$ ms.

Several available third party applications are capable of measuring transfer speed for files with different sizes and different buffer sizes, for example IOzone[2] and Bonnie++[3]. Because our main goal is to establish an upper bound on the bandwidth, we choose a simpler approach. Using the below command with a large file as input, we are able to get an estimate on how fast it is possible to read a large, and probably sequentially stored, file from disk.

```
time dd bs=128k if=<largefile> of=/dev/null
```

The above command copies the large file to */dev/null* and outputs the amount of megabytes transferred per second. This command was run several times, but the variability was low. The result from one run is shown in Figure 6.1.

```
20096+1 records in
20096+1 records out
2634055680 bytes (2.6 GB) copied, 32.7171 seconds, 80.5 MB/s

real    0m32.719s
user    0m0.008s
sys     0m0.924s
```

Figure 6.1: Result from testing the maximum bandwidth from disk

As can be seen from Figure 6.1, the measured bandwidth was 80.5 MB/s. We know that there is typically at least one disk seek involved in the timings. According to our estimates, the disk

[1] http://www.samsung.com/Products/HardDiskDrive/
[2] http://www.iozone.org
[3] http://sourceforge.net/projects/bonnie/

seek would take approximately 13.1 ms. This does not constitute a significant part of the time overall, and we do not bring it into the calculations. We thus end up with the model parameters for disk performance shown in Table 6.1.

Variable	**Estimate**
t_s	$13.1 \cdot 10^{-3}$ s
t_t	$1.2 \cdot 10^{-8}$ s/byte

Table 6.1: Parameters for disk performance

6.1.2 CPU performance

As mentioned above, estimating CPU performance is not straight-forward. It is possible to know the cost of assembly operations on a processor, but it is not feasible to estimate the number of single operations performed while constructing an inverted index. We thus need to consider more abstract operations. At a higher level of abstraction, it is easier to estimate the number of different operations performed, but it is harder to estimate the cost of each of them.

It seems like the only sound way to estimate the cost of high-level operations is to run our own implementation, and try to measure the amount of time used by different parts of the code. This approach is used in several other texts in the field as well [WMB99, HZ03]. Unfortunately, such timings can never be very accurate in a multi-threaded application, because the thread containing the code currently being timed might be suspended. In addition, it is important to avoid measuring the time spent accessing the disk. To be able to state that we actually obtain a model, and not only the results from experiments, we would also like to minimize the runs used to obtain the estimates.

Before we decide which experiments to run to determine the expected CPU running times for different operations, we have to decide which operations we want to consider. Following the arguments given above, the operations should be high-level enough to ensure that we are able to estimate the number of times they occur. They should not be too high-level however, because it is very difficult to avoid measuring disk accesses and other parts of the code when measuring high-level operations. With these considerations in mind, the following operations were chosen as the basis for our CPU performance model:

- t_p - the time used to parse one term, and add it to the *HashMap* of terms and occurrences for its document.
- t_{am} - the time used to add parsed documents to the memory resident index, measured in seconds per byte.
- t_{se} - the average time used for one operation in a multi-way merge of several lists. In other words, this is the constant for an extract minimum operation from a priority queue of iterators, which is known to have an asymptotic complexity of $O(\log(s))$, where s is the size of the priority queue.

- t_{ur} - the time used to update the length of document based on a single index entry.
- t_{lu} - the average time used to perform a look-up in a sorted list dictionary.
- t_{bi} - the average time used to insert a new entry in a B-tree. Note that we choose not to consider the height of the B-tree or the size of the entry.
- t_{bs} - the time used to perform a search in a B-tree.
- $n_{d,u}$ - the average number of unique terms per document.
- $n_{d,t}$ - the average total number of terms per document.
- l_e - the average length of the term of one index entry. An index entry will be created per unique term for each document, and l_e describes the average length of the term an index entry represents.
- l_u - the average length of unique terms.

It might seem inconsistent to measure seconds per term for t_p, and seconds per byte for t_{am}. It will introduce additional uncertainties in our estimates to change one of them though, and we prefer more reliable estimates. A similar approach is used in [HZ03].

The remaining part of this section will explain how we estimate these variables. We manage to obtain estimates for all of them in just one baseline run with our implementation, and by using the data gathered in Chapter 5 to choose which terms to search for.

6.1.2.1 Estimation based on a dumped dictionary

In Chapter 5, we dumped a dictionary for the collection with 1 million documents, to consider which terms to search for in our experiments. For each term in the dumped dictionary, there is information about the number of documents containing the term, and the total number of occurrences. We are thus able to estimate the last four variables in the list above from this dump.

The average length of the term of one index entry can be found as shown in Equation 6.2. The equation shows that l_e is equal to the sum over all terms, of the length of the term, $l(w_i)$, times the number of documents containing the term, n_i, divided by the total number of index entries.

$$l_e = \frac{\sum_{i=1}^{V} l(w_i) \cdot n_i}{\sum_{i=1}^{V} n_i} \tag{6.2}$$

By substituting values from the dumped dictionary in Equation 6.2, we obtain $l_e = 6.01$. The average length of unique terms is calculated as in equation 6.3.

$$l_u = \frac{\sum_{i=1}^{V} l(w_i)}{V} \tag{6.3}$$

Substituting values from the dumped dictionary into Equation 6.3 gives us an estimated $l_u = 7.92$. We note that the value estimated for l_e compared to the one for l_u suggests that the more common terms are actually shorter than the less common ones.

Because we know that the document collection, represented by the index we dumped the dictionary for, contains $N = 1$ million documents, we can find the average number of terms per document as shown in Equation 6.4.

$$n_{d,t} = \frac{\sum_{i=1}^{V} occ_i}{N} \tag{6.4}$$

The variable occ_i in Equation 6.4 represents the total number of occurrences of term i. Likewise, we can find the average number of unique terms per document as in Equation 6.4.

$$n_{d,u} = \frac{\sum_{i=1}^{V} n_i}{N} \tag{6.5}$$

By substituting values from our dumped dictionary into the two above equations, we obtain the following estimated values: $n_{d,t} = 946.14$ and $n_{d,u} = 279.95$.

Note that we could have used Heap's law to estimate the average number of unique terms in a document based on the average number of terms it contains. Heap's law is only considered applicable for large document collections, and applying it per document does not seem like a sound approach. We will use this law for some estimates in the following sections however.

6.1.2.2 Estimation based on running the application

The rest of the variables chosen to be the basis for our CPU performance model are estimated empirically. We will now explain how we measure to obtain the estimates. t_p is easily estimated by recording the number of terms parsed, and the amount of time spent doing so. This happens in a thread by itself, and it is obviously possible that this thread is suspended occasionally. It is suspended when it waits for various other threads to perform work needed by this thread. To limit the effects of such events on the timings, we stop and start the timings before and after calls to $wait()$ on various objects. We also stop and resume timings before and after segments of code that may cause disk accesses. Even though there are still several possible causes of errors in our estimates, this seems to be the best we can achieve with reasonable amounts of effort.

There is only one part of the code where parsed documents are added to the memory resident index, and we thus have no choice regarding where to measure to obtain an estimate for t_{am}.

This operation happens in parallel with parsing the documents, but in another thread. We thus have the same problem with the measurements as in the previous case, but because this is the only place in the code this operation is performed, we do not have other options. Because the processor has two cores, it is possible to run two threads at once, and it is thus theoretically possible that the measurements will lead to reliable estimates.

Sorting through a multi-way merge is a quite common operation when using remerge or a hierarchical index in Brille. When flushing a partial index, we perform a multi-way merge on the entries from different documents. It is also used when merging different indexes, both to merge dictionary entries and to merge inverted lists. This gives us a choice of where to obtain our estimate of t_{se}, and a decision should be reached based on where it is likely to obtain the most reliable estimates. When flushing partial indexes, the only other thread that definitely running, is the thread flushing dirty buffers from the buffer pool. It is quite straight-forward to measure the time spent sorting in this phase, because the sorting is the only CPU intensive operation performed. The sorting process may however, be forced to wait due to lack of buffers in the buffer pool. We allow using half the buffers to accumulate the memory resident index. Some of the remaining buffers are used to cache the B-tree in the document manager, and we will probably use the rest to flush the partial index. Because the overall need for buffers is relatively low, at least when the B-tree in the document manager is small, we assume that the overhead involved with pinning new buffers is not overwhelming.

Measuring one of the other sorting processes is more complicated. First of all, the code performing the merges is spread over different classes, and the amount of disk I/O involved is higher. We therefore choose to estimate t_{se} by measuring the time spent flushing a partial index, and divide by the number of entries to sort times the logarithm of the number of documents. We keep in mind though, that the reliability of this estimate is questionable due to the possible need to wait for flushing when pinning new buffers.

Updating document lengths is also a frequent operation in Brille. Usually when this happens, a lot of other operations go on at the same time however. The length is typically updated when merging partial indexes, or when the thread updating document lengths iterates over the naive B-tree index. In both of these cases there are a lot of disk accesses involved and measuring the exact part of the time used for updating the document lengths is difficult. Fortunately, there is one code segment where the process of updating document lengths is more isolated. This is when we want to make a single partial index searchable. This occurs in the hierarchical index when an index is flushed and in remerge when adding all files to be indexed only involves one partial index. Because the partial index has just been flushed, it is quite likely that most parts of it is cached in the buffer pool. When we also take care not to measure the time spent pinning buffers from the buffer pool, it seems quite likely that we are able to obtain a reasonable estimate for t_{ur}.

We thus choose to make a baseline run adding the exact number of documents that can fit in a single partial index, and use remerge as the method. Choosing this amount of documents is necessary to obtain a reasonable estimate on t_{ur}, as mentioned above. Fortunately, all of the previously mentioned variables can also be estimated during such a run. The sorting in a multi-way merge is estimated when we flush our partial index. It is actually an advantage that we

only measure the time spent in one flush, because the B-tree in the document manager is quite small when relatively few documents have been added. This makes the estimates more reliable as the amount of buffers currently in use is not exhaustive. We are also capable of estimating the first two variables, as parsing of documents and adding them to the memory resident index will obviously take place when we construct a partial index.

The only sound way to find an estimate for the time spent searching in a sorted list dictionary, t_{lu}, is to perform searches and time them. We have chosen not to discriminate between different sizes of the dictionary for this variable. It is likely that there is a difference between searching in dictionaries for large and small indexes, but according to Heap's law, the vocabulary grows significantly slower than the total number of terms in the index. In addition, the complexity of a binary search is logarithmic, which also makes the differences between different sizes less significant than if it was linear. The index constructed in the baseline run depicted above is not large. This might not give us a fair average look-up time, but we can at least be quite sure that the dictionary is cached in the buffer pool when the index is just constructed. We therefore try searching for the same terms as we do in the experiments, but measure only the average time spent searching in the dictionary. We do a complete round of 300 searches 1000 times. Cache effects are not a problem here, because we wish to estimate the processing time.

The only variables not estimated yet are those considering the efficiency of the B-tree. During the run depicted above, the only operations on a B-tree are the ones adding the mappings between document numbers and URIs in the document manager. Adding slightly more than 60000 documents involves adding the same number of entries in the B-tree. This number is considered too small to obtain a reliable estimate on the average time needed to insert an entry. To insert more entries into the B-tree and obtain more reliable estimates, we thus add several more such mappings, until we reach a total of 1 million inserted entries. Timing the insertion of 1 million insertions will hopefully give us a more reasonable estimate of insertion time. The reason for using the B-tree in the document manager instead of the naive B-tree index is that it gives us a smaller probability of having to read in nodes from the disk. The insertions from the document manager is ordered according to document number, and the next insertion is thus always in the last node on the leaf level. We will therefore follow the same path down through the B-tree each time, and it seems quite unlikely that it will be necessary to stop caching these nodes in the buffer pool.

When we have constructed a B-tree containing 1 million entries, it is straight-forward to test searching in it. We estimate t_{bs} with 100000 searches in the constructed B-tree. Note that it is intuitively likely that the time spent performing searches and insertions in a B-tree are dependent on the size of the B-tree, because their complexities are asymptotically linear to the height of the B-tree. We therefore considered testing both insertions and searches in B-trees of various sizes. It turned out that it was hard to extract a pattern when testing with heights up to 4 however. It is also an advantage to keep the calculations based on the model relatively simple, and it therefore seemed reasonable to restrict the estimates to a single operation.

The baseline run introduced above was run 10 times, producing reports as the example in Figure 6.2. Based on the reports from all runs, the estimates were calculated, and the results are given in the following subsection.

```
[java] Used 81495356000 nanoseconds to parse 49624458 terms.
[java] Used 112191789000 nanoseconds to flush the new docs in inmemindex.
  There are now 64335 docs flushed.
[java] Used 11015449000 nanoseconds adding rankings for 16032955 entries.
[java] Used 15956032000 nanoseconds adding 393207293 bytes to memory
  resident index.
[java] Created the index in 3 minutes and 43 seconds.
[java] Used 3113530000 nanoseconds searching for 300000 entries in
  the dictionary
[java] Used 85643928000 nanoseconds adding 1000000 entries to B-tree
[java] Used 1447276000 nanosecond searching for 100000 entries in B-tree
```

Figure 6.2: Example results from a baseline run

6.1.2.3 Resulting estimates

The results from the estimations introduced in the previous subsections, gives us the estimated values presented in Table 6.2.

Variable	Estimate
t_p	$1.62 \cdot 10^{-6}$ s
t_{am}	$4.12 \cdot 10^{-8}$ s
t_{se}	$4.87 \cdot 10^{-7}$ s
t_{ur}	$7.14 \cdot 10^{-7}$ s
t_{lu}	$1.05 \cdot 10^{-5}$ s
t_{bi}	$8.64 \cdot 10^{-5}$ s
t_{bs}	$1.58 \cdot 10^{-5}$ s
$n_{d,u}$	279.95 terms
$n_{d,t}$	946.14 terms
l_e	6.01 bytes
l_u	7.92 bytes

Table 6.2: Parameters for CPU performance

6.1.3 Some notes on the experiment environment

Even though the experiment environment is formally introduced in Chapter 7, we mention some details here that are essential for obtaining estimates with some resemblance to reality. First of all, the computer used in the experiments has two processor cores. This makes it possible for two threads to run in parallel. Making estimates about the level of parallelism actually obtained is inherently hard, but we should at least be aware of it while calculating.

The GOV2 collection used in the experiments is quite large. We therefore use a single empty disk to store all its 426 GB. The other disk on our experiment computer contains the operating

system and all index files created by Brille. It is thus theoretically possible to read documents from one disk, while writing index files to the other.

6.2 Constructing partial indexes

As explained in Chapter 4, the in-memory accumulation proceeds in two phases. In the first phase, the documents are read, parsed and added to the in-memory index stored in buffers. The other phase involves sorting the occurrences of different terms and constructing a dictionary and an inverted file. This section aims to give an estimate of the time we expect to spend constructing a partial index, based on the framework introduced in the preceding section. We will first consider each of the two phases individually, before we consider constructing partial indexes as a whole in Section 6.2.3.

6.2.1 In-memory accumulation

During accumulation in memory, the documents are read and parsed in one thread; while another thread stores the parsed documents in the buffers in main memory. This phase will proceed as long as we can fit the next document in the reserved buffers. To estimate the work needed to fill the memory, we should estimate the number of documents added. A document with average length will occupy the number of bytes in the in-memory index, s_d, as given in Equation 6.6.

$$s_d = 12 + (l_e + 6) \cdot n_{d,u} + 4 \cdot n_{d,t} \tag{6.6}$$

Equation 6.6 describes that for each document, its entry into the priority queue used for merging occupies 12 bytes. For each unique term, we use 2 bytes to represent the length of the term, while 4 additional bytes are used to represent the number of occurrences. For each occurrence of a term within a document, 4 bytes are used to describe its term number. By inserting our average values we can conclude that an average document will use $s_d = 7158.76$ bytes. Brille allows the in-memory index to fill 375 MB. On average, we thus have room for approximately $N_p = 54928$ documents. Given such a batch of documents, the following two subsections will estimate the time needed for disk accesses and processing for the batch.

6.2.1.1 Estimating time spent accessing disk

To obtain an estimate on the time spent accessing the disk during this phase, we need to consider how the documents in GOV2 are stored. As mentioned in Chapter 5, the collection is partitioned into 273 directories, where each directory contains 100 files. The average size of such a file is approximately 16 MB, and they contain 923.27 documents on average. We can thus conclude that the average number of files we need to read in this phase is as given in Equation 6.7.

$$\left\lceil \frac{54928}{923.27} \right\rceil = \lceil 59.5 \rceil = 60 \tag{6.7}$$

From Equation 6.7 we can conclude that we on average will need to access 60 different files, and read 59.5 of them in each accumulation phase. The amount of read data from the disk is thus easily calculated, but estimating the number of disk accesses is not as straight-forward. A file system will usually try to store single files sequentially. It is therefore likely that we are able to read a single file with just one disk access. Even if we might need to access the disk each time we fill our buffer used for reading, we choose to assume that it is possible to read the file sequentially. With on average 60 disk accesses and $59.5 \cdot 16$ MB of read data, the estimated time spent reading from disk in this phase is given in Equation 6.8.

$$60 \cdot t_s + 59.5 \cdot 16 \cdot 2^{20} \cdot t_t = 12.76 \tag{6.8}$$

We thus estimate that it will take approximately 12.76 seconds to read the documents to be indexed from the disk containing the collection. There are also some accesses to the other disk during this phase. All the new documents get their document number and URI stored in the B-tree in the document manager.

In our implementation of a buffer pool, we have a so-called flushing thread which is responsible for writing the contents of dirty buffers to a *FileChannel*, as mentioned in Section 4.2.7. The flushing thread starts at regular time intervals, and writes out all dirty buffers that are not in use at the moment. During the accumulation phase, there are several insertions in the B-tree, and it is hard to estimate the number of times a buffer is written out before its final version is written to disk. We know that as a minimum, each buffer is flushed once. In order to keep these calculations simple, we assume that each buffer is flushed only once, and that this flush requires the disk to seek for the correct position. We thus have to calculate the average number of buffers needed to store a B-tree with 54928 document entries. We assume an average space efficiency of $\frac{2}{3}$ for a B-tree, even though we can not guarantee the lower limit on space efficiency of $\frac{1}{2}$ for our implementation, as noted in Section 4.2.8.

It is actually quite straight-forward to estimate the number of nodes at the leaf level in the B-tree of the document manager, because the size of each entry when we index the GOV2 collection is 22 bytes. 4 of these bytes are used to store the document number, and 2 to store the length of the URI. All URIs are actually 16 bytes long in the GOV2 collection, and we thus end up with 22 bytes per leaf entry. In addition, the B-tree node will store a pointer to the entry, making the total space consumption for each entry 26 bytes. The space consumption for internal entries in the given B-tree is 12 bytes, because they store the document number, a pointer to a node, and a pointer to the entry within the node.

When using a buffer with size B to represent a B-tree node, the number of leaf nodes in the B-tree is as given in Equation 6.9.

$$nodes_{leaf,dm} = \left\lceil \frac{26 \cdot N_p \cdot 3}{2 \cdot B} \right\rceil \tag{6.9}$$

Assuming the same space efficiency for internal nodes, each node will on average contain $p_n = \frac{B}{18}$ pointers. This approximation applies to all nodes except the root, which may have as few as 2 pointers. Carrying out a similar calculation to what is done in [Bjø06], it is quite easy to calculate the approximate total number of nodes in the B-tree as given in Equation 6.10

$$nodes_{dm} = 1 + \sum_{i=0}^{\infty} \left\lfloor \frac{nodes_{leaf,dm}}{\left(\frac{B}{18}\right)^i} \right\rfloor \approx 1 + nodes_{leaf,dm} \cdot \sum_{i=0}^{\infty} \frac{1}{\left(\frac{B}{18}\right)^i} = 1 + nodes_{leaf,dm} \cdot \sum_{i=0}^{\infty} \left(\frac{18}{B}\right)^i \tag{6.10}$$

We recognize the geometric series in Equation 6.10, and end up with the approximate number of nodes given in Equation 6.11.

$$nodes_{dm} \approx 1 + nodes_{leaf,dm} \cdot \frac{B}{B - 18} \tag{6.11}$$

Our experiments are run with $B = 4096$ and $B = 16384$. When $B = 4096$, the number of leaf nodes is $nodes_{leaf,dm} = 523$, and the total number of nodes is approximately 526. We thus get approximately 526 disk accesses and slightly over 2 MB of written data according to our assumptions. This will require approximately 6.92 seconds spent accessing the disk according to our model. With $B = 16384$, we estimate to spend 1.76 seconds.

Because we use an LRU replacement policy in the buffer pool, it seems likely that we might have to read in some nodes while inserting in the B-tree. Because of the way we assign document numbers to documents, this is quite unlikely however. Each new document receives a document number that is strictly larger that the previous. This implies that it will be inserted to the far right at the leaf level in the B-tree. The path from the root down to rightmost node in the leaf level is thus accessed for every insert. We therefore consider it unlikely that these buffers will be replaced.

Note that the above calculation is based on the first added partial index, but only slightly more nodes will be added in later batches because the number of nodes in the top of the tree grows slowly. For simplicity we assume that the number of nodes added in each batch is similar.

We thus have estimates for all disk operations in this phase, but we should keep in mind that the flushing of the nodes for the B-tree does not necessarily happen before the next phase, because the buffer pool does not guarantee that the buffers are flushed instantly. We should thus consider whether flushing these buffers, in addition to the disk operations in the next phase, might become the bottleneck. We will consider this aspect in Section 6.2.3.

6.2.1.2 Estimating time spent processing

The most important processing operations involved in accumulating a partial index in memory is parsing the terms and adding the occurrences to a *HashMap* for each document, and adding the partial documents to the in-memory index. These operations take place in different threads, but the threads may obviously have to wait for each other. We will calculate estimates for each of the operations, and consider how we can estimate the time spent doing both of them at the end of this section.

We have calculated the average number of documents contained in one memory resident index, N_p. We also know the average number of terms in each document. Multiplying the product of these two values with our estimate of the time spent parsing each term, and adding it as an occurrence, we end up with an estimated number of seconds for this operation as given in Equation 6.12.

$$N_p \cdot n_{d,t} \cdot t_p = 84.9 \tag{6.12}$$

We thus estimate using approximately 84.9 seconds to parse all terms and record their occurrences. This is a quite interesting result, because regardless of the time spent doing the rest of the processing, we are now in a position to estimate that this phase is bound by CPU. This contradicts our initial beliefs introduced in Section 4.2.3.2.

The other part of processing involves adding documents to the in-memory index, and to insert their mapping between document number and URI in the document manager. In order to keep the things simple, we assume that we will be reasonably close to fill all the buffers set aside for accumulating an index, meaning that there are 375 MB to add. We use the number of documents added to estimate the time spent inserting entries in the B-tree. The calculations are performed in Equation 6.13.

$$375 \cdot 2^{20} \cdot t_{am} + N_p \cdot t_{bi} = 20.95 \tag{6.13}$$

Adding the documents to the in-memory index is estimated to take 20.95 seconds, and is thus more efficient than parsing them.

Because our processor has two cores, it is not reasonable to sum these two values to obtain the final estimate for the processing involved in this phase. It is not likely that we can use just the most costly operation either, because it is dependent on some I/O before it can initiate, and we must add the final documents to the index after they have been parsed. These aspects are considered too cumbersome to estimate though, and this leads us to a final estimate of spending approximately 84.9 seconds processing in this phase. We keep in mind though, that the actual number is probably slightly higher.

6.2.2 Sorting and flushing

The next phase during construction of partial indexes is when the index entries are sorted and flushed to disk. For this phase to be done, we need to have sorted all entries, and unpinned all buffers. This does not imply that all buffers will have to be flushed before the phase is ended, but we will estimate the time needed to do so nonetheless, and use the estimate in the next subsection.

6.2.2.1 Estimating time spent accessing disk

During this phase, all disk accesses are to the disk storing the index files created by Brille. Disk accesses are required when we flush the dictionary and the inverted file for this partial index. We will first estimate the size of each of these files.

The dictionary contains all unique terms in the collection of the estimated N_p documents in this partial index. To obtain an estimate on the number of unique terms in such a collection, we use Heap's law, which is introduced in Section 2.2.5. We have an estimate on the number of terms in this collection through the number of documents in the collection multiplied by their average number of terms. We will also have to determine which constants to use in Heap's law. Because GOV2 is an example of a large web collection, we choose to use the constants suggested in [Fre02]. We thus end up with the calculation of vocabulary size given in Equation 6.14

$$V_p = 16.24 \cdot (N_p \cdot n_{d,t})^{0.57} \tag{6.14}$$

Substituting our estimated values into Equation 6.14, we obtain the estimated $V_p = 406034.62$. Each entry in the sorted list dictionary will consume space equal to the length of its term in addition to 18 bytes used to store the pointer into the inverted file and some statistics. We have an estimate on the average length of unique terms, and we can thus estimate that the space occupied by the dictionary is $|V_p| = 10.04$ MB.

We know that for each document, the number of bytes used in the inverted file, s_{di}, is as given in Equation 6.15.

$$s_{di} = 8 \cdot n_{d,u} + 4 \cdot n_{d,t} \tag{6.15}$$

By substituting our estimated variables into Equation 6.15 and multiplying with the estimated number of documents, we calculate that the size of an average inverted file for a partial index is approximately $|I_p| = 315.51$ MB.

Having obtained estimates for the size of each file, we know the amount of data that should be written to disk during this phase. We also need to determine the number of disk accesses. As noted above, the write operations to disk in this phase are performed by the flushing thread in the buffer pool. At regular time intervals, the flushing thread goes through all dirty buffers

and writes them to a *FileChannel*. This happens in the order in which the buffers have been unpinned, and we do not know exactly when they are actually flushed to disk. Because of the uncertainties regarding when the different buffers are actually flushed, we have to make some conservative estimates. The inverted file is substantially larger than the file for the dictionary, and it thus grows faster. This implies that every time we unpin a buffer for the dictionary, the last and next unpinned buffers are most likely to be part of the inverted file. A conservative estimate is to assume that each time we write a buffer to a *FileChannel*, it will be flushed fast. It thus seems reasonable to assume that a disk seek is required each time we switch what file we write to. With such an assumption, each buffer written to the dictionary will require 2 disk accesses. One disk access is needed to move the disk arm to the correct position in the file for the dictionary, while the other one is needed to move it back to the inverted file again. How many buffers the complete dictionary consists of is dependent on the buffer size, and we can estimate the number of disk accesses as 2 times the size of the complete dictionary divided by the buffer size. This gives us the estimated time spent accessing the disk given in Equation 6.16.

$$t_d = 2 \cdot \left\lceil \frac{|V_p|}{B} \right\rceil \cdot t_s + (|V_p| + |I_p|)t_t \tag{6.16}$$

When using a buffer size of 4 kB, we get the $t_d = 71.46$ seconds, and for a buffer size of 16 kB, $t_d = 20.94$ seconds.

6.2.2.2 Estimating time spent processing

The only main contributor to processing time during this phase is the sorting of the index entries. We know the number of documents to merge and we can find an estimate on the number of index entries to sort, and are thus able to estimate the worst-case number of operations performed in the priority queue of iterators. Because we have an estimate of the constant in such operations, t_{se}, we are able to estimate the time spent processing during the sorting phase as given in Equation 6.17.

$$t_{cpu,s} = N_p \cdot n_{d,u} \cdot \log(N_p) \cdot t_{se} \tag{6.17}$$

Based on Equation 6.17, we estimate the number of seconds spent processing during the sorting phase to be $t_{cpu,s} = 117.91$.

6.2.3 The phase as a whole

In the last two subsections we have calculated estimates for the time spent constructing partial indexes based on our framework of CPU and disk performance. The estimates show that in both phases, the processor is actually the bottleneck in our implementation. The processing performed in the sorting phase is dependent on the first phase, meaning that it can not initiate

before the first phase is completed. This implies that the estimated time spent constructing one partial index is $84.9 + 117.91 = 202.81$ seconds.

The flushing to the disk containing the files created by Brille will, if it was the only process running, spend $71.46 + 6.92 = 78.38$ seconds accessing the disk if the buffer size is 4 kB. When the buffer size is 16 kB, the estimate is $20.94 + 1.76 = 22.70$ seconds. In both cases, the processing time will be the bottleneck. The same is true for the disk where the indexed documents are read from, because reading in the documents is approximated to take 12.76 seconds.

We thus conclude that in our implementation, the expected time used to construct a partial index when filling the allowed memory completely is 202.81 seconds. The quite surprising result that processing time seems to be the bottleneck will be commented on when we obtain the actual results in Chapter 7.

6.3 Merging indexes

To merge indexes is a frequent operation in both remerge and hierarchical indexes. Even though there are slight differences between merges with different numbers of indexes and indexes with different sizes, we choose to develop common estimates for merging indexes. The rationale behind such a choice is to keep this section within reasonable length. Furthermore, it is believed that even with such a choice, we will still get a fair idea about the bottlenecks in our system. The following two subsections will consider the two possible bottlenecks, disk I/O and CPU.

6.3.1 Time spent accessing disk

During a merge, we will obviously have to read all partial inverted files and their dictionaries, and write the complete new index to disk. All of these operations take place at the same disk in our experiment environment. The size of the resulting inverted file is equal to the sum of the sizes of the inverted files we merge. The size of the dictionary is not calculated that easily however, because the smaller dictionaries are likely to have several terms in common. Assuming that the complete number of terms in all indexes to merge is n, we can use Heap's law to estimate the size of the vocabulary, as given in Equation 6.14. Remembering from Section 6.2.2.1 that each occurrence in the dictionary consumes 18 bytes in addition to the term itself, we can estimate the size of the complete dictionary, $|V|$, to be as given in Equation 6.18.

$$|V| = 16.24 \cdot n^{0.57} \cdot (18 + l_u) \tag{6.18}$$

We are thus capable of estimating the total amount of moved data between disk and memory. We also need to estimate the number of disk accesses however. This is slightly more complicated because there are several effects that might play a significant role. We will first consider the

expected number of disk accesses needed to read the indexes to merge. All the dictionaries and inverted files will be read sequentially, but we may possibly need to read from different files for each new pinned buffer. This makes the access pattern far from linear. Because our buffer pool will only read one buffer at a time, we might expect the number of disk accesses needed to be the sum of the number of buffers needed to be read each index. Such an assumption is used in a comparison of different construction methods in [HZ03].

There are some effects we should probably take into consideration however. The operating system will usually employ some form of read-ahead, meaning that it will cache the content of the disk following the section just read. The same kind of caching is usually performed in the disk controller. It is therefore likely that the number of disk accesses will be smaller. The problem is however, that such effects are inherently hard to estimate. Even if it is possible to get a reasonable overview of the caching in an open-source operating system, it is usually not known exactly how the disk controller works [Bra].

Because of the problems involved with giving a precise estimate, we will give two separate estimates. One of the estimates follows the strategy in [HZ03], while the other one will assume that the access pattern is actually sequential. Even if we know that the access pattern is not sequential, the effects mentioned above are likely to decrease the number of disk accesses significantly. If we actually read large parts of the files sequentially, the overhead introduced by the disk accesses will not be significant, and this estimate might be appropriate.

6.3.1.1 Estimate 1

If we assume that one disk access is required for each buffer read from each of the m partial indexes to be merged, we need the number of disk accesses given in Equation 6.19 to read in all indexes to be merged.

$$d_{a,r} = \sum_{i=1}^{m} \left(\left\lceil \frac{|V_i|}{B} \right\rceil + \left\lceil \frac{|I_i|}{B} \right\rceil \right) \tag{6.19}$$

We must also need to write the complete merged index to disk, with the size of the dictionary and inverted file as calculated above. The dictionary is typically far smaller than the inverted file, and it seems reasonable to assume that we will require one disk access per buffer in the dictionary as well. We might obtain better results for the inverted file however. If we are merging several indexes with approximately the same size, it is likely that they have inverted lists for terms within approximately the same lexicographical range in one buffer. It is therefore likely that we are able to flush several buffers to the inverted file sequentially, which requires only one disk access. Despite such observations, we choose to keep this estimate conservative, and follow the same approach as in [HZ03]. This implies that we assume that each buffer written to the complete inverted file will also require one disk access. The total number of expected disk accesses is given in Equation 6.20.

$$d_a = \left\lceil \frac{|V|}{B} \right\rceil + \left\lceil \frac{\sum_{i=1}^{m} |I_i|}{B} \right\rceil + \sum_{i=1}^{m} \left(\left\lceil \frac{|V_i|}{B} \right\rceil + \left\lceil \frac{|I_i|}{B} \right\rceil \right) \tag{6.20}$$

We are now able to give an estimate of the time spent accessing the disk with estimate 1, and it is given in Equation 6.21.

$$t_{d,m,1} = d_a \cdot t_s + \left(|V| + \sum_{i=1}^{m} (|V_i| + 2 \cdot |I_i|) \right) t_t \tag{6.21}$$

6.3.1.2 Estimate 2

The amount of moved data is similar in estimates 1 and 2, but the number of disk accesses differs. In estimate 2, we assume that only 1 disk access is needed per file read or written. Because each partial index consists of one dictionary and one inverted file, such a scheme would require 2 disk accesses per partial index, and further 2 to write out the final index. This makes the computation for estimate 2 quite straight-forward, and it is given in Equation 6.22.

$$t_{d,m,2} = (2 \cdot m + 2) t_s + \left(|V| + \sum_{i=1}^{m} (|V_i| + 2 \cdot |I_i|) \right) t_t \tag{6.22}$$

6.3.2 Time spent processing

Both the dictionaries and the inverted files are merged during a normal merge. The dictionaries are merged to find the unique terms. For each unique term, the inverted lists from the indexes that contain the given term are merged. We thus have two levels of merge operations. We will consider the merge of the dictionaries first. The number of entries to be merged is equal to the sum of the number of entries in each of the merged dictionaries. Assuming that index number i contains $N_i \cdot n_{d,t}$ terms in total, we can estimate the number of entries in each dictionary with Equation 6.14. The number of entries in the priority queue is equal to the number of indexes to be merged, denoted m. The worst-case number of operations needed to perform the merge of dictionaries, o_{md}, is given in Equation 6.23.

$$o_{md} = \sum_{i=1}^{m} 16.24 \cdot (N_i \cdot n_{d,t})^{0.57} \cdot \log(m) \tag{6.23}$$

We also merge the inverted lists for each unique term. Such merges will also be performed in a priority queue, and it seems natural to estimate that the cost of such an operation is dependent on the number of occurrences of the term, and the number of merged indexes which contain

the given term. There is one important observation here however. When indexes are merged in Brille, we will always have partitioned the document numbers between the different indexes. In addition, all indexes will contain a continuous range of document numbers. This implies that during the merge in the priority queue, we first empty one index, before we start emptying another. We will thus only have the worst-case logarithmic complexity of extracting the minimum index entry each time an index is empty.

For common terms, the above observation will give the merge an average complexity that is almost linear. For terms that occur only once or twice, we never have to merge more indexes than there are occurrences of the term. Even if we for some terms are likely to obtain the worst-case complexity, it seems reasonable to assume that the logarithmic factor in the merge can be discarded. We make this assumption, and the expected number of operations in the merge of inverted lists is given in Equation 6.24.

$$o_{mi} = \sum_{i=1}^{V} n_i = n_{d,u} \cdot \sum_{i=1}^{m} N_i = n_{d,u} \cdot N \tag{6.24}$$

While the merge is performed, new and updated tf-idf document lengths are also computed. Each index entry will contribute, and we have estimated the time needed to update the length of a document based on one index entry. We can then estimate the processing time spent merging indexes as given in Equation 6.25.

$$t_{cpu,m} = t_{se} \cdot \log(m) \cdot \sum_{i=1}^{m} \left(16.24 \cdot (N_i \cdot n_{d,t})^{0.57}\right) + (t_{se} + t_{ur}) \cdot n_{d,u} \cdot N \tag{6.25}$$

With Equation 6.25 as a basis, we are able to estimate the time spent processing a merge based on the total number of documents in the merge, N, the number of documents in each of the indexes to merge, N_i, and the number of indexes to be merged, m.

6.4 Remerge

In the last two sections, we have estimated the time spent constructing partial indexes and merging them. We are thus able to estimate the time spent constructing an index with the off-line version of remerge, and with the version where new partial indexes are immediately merged with the main index. We will do so in the following two subsections. As for the other overall methods, we will only calculate exact estimates for the largest index size where we test with 10 runs in the experiments.

6.4.1 Off-line construction

When constructing an index with off-line construction, we first construct partial indexes until all documents are indexed. The partial indexes are eventually merged into a large searchable index. This section will estimate the time spent during such a construction. When considering the time spent in the merge, we will calculate values using both estimate 1 and 2. Regardless of which of those estimates we use, the same partial indexes are constructed, and we start by calculating the expected time for this phase.

In Section 6.2, we calculated the expected time spent constructing one partial index. In order to use the values calculated there, we need to know the expected number of partial indexes constructed when building an index for 1 million documents. We have an estimate for the number of documents there is room for in one partial index, N_p, and it is thus easy to calculate the expected number of partial indexes, pi. Such a calculation is shown in equation 6.26.

$$pi = \frac{N}{N_p} = \frac{1000000}{54928} = 18.21 \tag{6.26}$$

We note that the expected number of partial indexes is 18.21. We have calculated the estimated time spent constructing a full partial index, but it is not likely that the time spent constructing one that is not full scales linearly with size. The process of constructing a partial index consists of two phases. The first phase consists of parsing documents, and adding them to an in-memory index. We noted that parsing the documents was the dominant factor in that phase. The complexity of the process is linear to the number of documents, if we assume that they all have the same length. The complexity of the second phase is not linear to the number of documents however, because sorting the entries was assumed to be the dominant factor here. The asymptotic complexity of the sorting routine used is $O(z \log(N_p))$, where z is the number of index entries, and N_p is the number of documents in the partial index. Because our estimates indicate that z grows almost 280 times as fast as N_p, we will actually assume that the process of constructing partial indexes scales linearly with the amount of documents added. The rationale behind such an assumption is to keep the calculations simple, and the belief that the impact is insignificant. We thus estimate the time spent constructing partial indexes to be as given in Equation 6.27.

$$18.21 \cdot 202.81 = 3693.17 \tag{6.27}$$

According to Equation 6.27, we expect the time spent constructing partial indexes to be 3693.17 seconds, or slightly more than an hour. When all the partial indexes have been created, they are merged to form a large searchable index. The time spent performing such a merge was estimated in Section 6.3, and we will use both of our estimates for time spent accessing disk in the following two subsections. First, we will calculate the processing time for such a merge however, because it is needed to obtain the final estimates for both estimation methods.

From Equation 6.25, we know how to calculate the expected time spent processing during a merge. All variables in the equation are already known, except for N_i for all partial indexes, but

these values are easily estimated. The first 18 partial indexes are assumed to contain approximately 54928 documents each, and this leaves us with 11296 for the last partial index. We thus end up with the estimated time spent processing during a merge as shown in Equation 6.28.

$$t_{cpu,m} = t_{se} \cdot \log(19) \cdot \left(18 \cdot 16.24 \cdot (54928 \cdot n_{d,t})^{0.57} + 16.24 \cdot (11296 \cdot n_{d,t})^{0.57}\right) + (t_{se} + t_{ur}) \cdot n_{d,u} \cdot 1000000 \quad (6.28)$$

By substituting values for the variables in Equation 6.28, we obtain the estimated $t_{cpu,m} = 351.68$ seconds. For each of the estimates for time spent accessing disk in the following two subsections, we must consider whether the obtained estimate is larger or smaller than the estimated $t_{cpu,m}$. The largest of the two will be our estimate for the time spent merging indexes, because we assume that disk accesses and processing can overlap.

6.4.1.1 Merging indexes with estimate 1

The time spent accessing disk when merging indexes in estimate 1 is given in equations 6.20 and 6.21. We have to calculate the size of the various index files to obtain the estimated time spent accessing disk. We have estimated the size of the dictionary and inverted file for a partial index in Section 6.2.2.1, to be 10.04 and 315.5 MB respectively. The last partial inverted file is $0.21 \cdot 315.5$ MB, which is equal to 66.25 MB. The size of the last partial dictionary is estimated as in Equation 6.29.

$$|V_p| = 16.24 \cdot (11296 \cdot n_{d,t})^{0.57} \cdot (18 + l_u) \quad (6.29)$$

Equation 6.29 follows the same assumptions as Equation 6.18, and estimates the size to be 4.07 MB. The size of the complete dictionary is estimated to 52.47 MB by substituting 1 million for 11296 as the number of documents in Equation 6.29. The size of the final inverted file is the sum of the sizes of the partial files, and is thus 5746.26 MB, or approximately 5.61 GB.

Because the expected number of disk accesses is dependent upon the size of the buffers used, we will have to calculate values for both of the buffer sizes used in our experiments, 4 KB and 16 KB. Substituting our estimated values into Equation 6.20, we estimate that the number of disk accesses with the small buffer is 3001914. For the large buffer, our estimate becomes 750494 disk accesses. The total size of all files is approximately 11727.77 MB. Based on these calculations, the expected number of seconds spent accessing disk when using a small buffer is calculated in Equation 6.30, and the same value for the larger buffer size is given in equation 6.31.

$$t_{d,m,1,4096} = 3001914 \cdot t_s + 11727.77 \cdot 2^{20} \cdot t_t = 39472.64 \quad (6.30)$$

$$t_{d,m,1,16384} = 750494 \cdot t_s + 11727.77 \cdot 2^{20} \cdot t_t = 9979.04 \tag{6.31}$$

According to the above calculations, we should use almost 11 hours merging the partial indexes with the small buffer size, and approximately 2 hours and 47 minutes with the larger buffer size. The estimated processing time is significantly lower, and we can conclude that if estimate 1 is the most valid approach, merging indexes is definitely I/O bound.

According to estimate 1, it should take 43165.81 seconds to construct the complete index when the small buffers are used, and 13672.21 seconds with the large buffers.

6.4.1.2 Merging indexes with estimate 2

Estimate 2 is simpler to calculate than estimate 1. We already know the total size of the files. The number of disk accesses is assumed to be 40, because there are 19 partial indexes. Because this method does not distinguish between using different buffer sizes, the only needed calculation is shown in Equation 6.32.

$$t_{d,m,2} = 40 \cdot t_s + 11727.77 \cdot 2^{20} \cdot t_t = 147.73 \tag{6.32}$$

Using estimate 2, we estimate to only use 148.09 seconds accessing disk while merging indexes. If this estimate is valid, we expect this phase to be CPU bound, because we estimated using 351.68 seconds processing the merge. According to estimate 2, it should take 4044.85 seconds to construct the complete index.

It should be noted that neither estimate 1 nor estimate 2 is likely to give a correct view of the actual time spent accessing disk while merging. They are probably upper and lower limits for the actual time spent accessing disk. We discuss the validity of both approaches when we discuss the actual results in Chapter 7.

6.4.2 Immediate merge

When all partial indexes are immediately merged with the main index, making the newly added documents searchable, remerge is expected to give lower update latency. It will however, probably need more time to construct the complete index. The partial indexes constructed are just the same here as for the off-line construction, and we thus estimate that the process of constructing the partial indexes to take 3693.17 seconds. We need to calculate the expected time spent performing the merges as well to obtain an estimate for the complete construction with this configuration.

We will first consider the processing involved in each of the merges. When we make the first partial index searchable, we do not need to merge it. Brille will then pin buffers with all parts of

this recently flushed index, and update the lengths of all documents. The time spent processing for this batch is thus as given in Equation 6.33.

$$t_{cpu,m,1} = N_p \cdot n_{d,u} \cdot t_{ur} = 10.98 \tag{6.33}$$

According to Equation 6.33, we will use 10.98 seconds updating the document lengths for the first batch.

The following merges will merge $m = 2$ indexes each. One of the indexes will contain N_p documents, while the other will contain the documents added before this batch. The last batch will also have 2 indexes to merge, but the smallest one will contain 11296 documents according to our previous estimates. We are thus able to calculate the estimated processing time for each merge, based on Equation 6.25. The results are shown in the second column in Table 6.4.

To find our estimates for the time spent accessing disk during the different merges, we need to know the size of the indexes. We estimate the size of each inverted file as the sum of the sizes of the merged inverted files. The sizes of the dictionaries are estimated with Heap's law, as in Equation 6.14. The results are shown in Table 6.3.

Batches	**Dictionary (MB)**	**Inverted file (MB)**
1	10.04	315.51
2	14.90	631.02
3	18.77	946.53
4	22.12	1262.04
5	25.12	1577.56
6	27.87	1893.06
7	30.43	2208.57
8	32.84	2524.08
9	35.12	2839.59
10	37.29	3155.10
11	39.37	3470.61
12	41.37	3786.12
13	43.31	4101.63
14	45.17	4417.14
15	46.99	4732.65
16	48.75	5048.16
17	50.46	5363.67
18	52.13	5679.18
19	52.47	5746.26

Table 6.3: Expected sizes of index files for the complete index after each batch is added

With the sizes of all intermediate index files calculated, we are able to calculate the expected time used to access disk during the merges. As for remerge, we will estimate with both methods 1 and 2. The calculations are explained in the following two subsections. We will compare each

of the estimates with the obtained estimate for processing time, and provide a summary of the results in Section 6.4.2.3

6.4.2.1 Disk estimate 1

All merges in remerge with immediate merge include 2 indexes. According to estimate 1, the amount of moved data is equal to the size of all files to merge, in addition to the size of the resulting index. We also need one disk access for each buffer read and written for each index file. In all but the last batch, the smallest index will be equal to the resulting index after the first batch given in Table 6.3.

When the first batch is added, there will not be any merge, and we will only update the document lengths of all documents added. In the current implementation, this involves pinning a buffer for each part of the index file. Because the B-tree in the document manager is quite small at this point, it is not likely that significant parts of these buffers have gotten their content replaced. It is therefore not likely that we have to read significant parts of the file. We will not make any such assumptions however, and carry out the calculations for all estimates as previously. We will consider whether this is a sound approach when discussing the actual results.

For the first batch, we thus have to read the complete file while updating the document lengths. We can calculate the time spent accessing disk after batch 1 with estimate 1, $t_{d,m,1,1}$, as shown in Equation 6.34.

$$t_{d,m,1,1} = \left(\left\lceil \frac{10.04 \cdot 2^{20}}{B} \right\rceil + \left\lceil \frac{315.51 \cdot 2^{20}}{B} \right\rceil \right) \cdot t_s + (10.04 + 315.51) \cdot 2^{20} \cdot t_t \qquad (6.34)$$

When we substitute the buffer sizes from our experiments, 4 KB and 16 KB, into equation 6.34, we obtain $t_{d,m,1,1} = 1095.88$ and $t_{d,m,1,1} = 277.05$ respectively.

The expected times spent performing the other merges according to estimate 1, is calculated as shown in equations 6.20 and 6.21. The results are shown in the third and fourth column of table 6.4.

6.4.2.2 Disk estimate 2

Estimating the time spent accessing disk with estimate 2 is quite simple now that we have calculated the sizes of all files to read and write. After the first batch, we will only have to read 2 files, the inverted file and dictionary for the partial index. For all later batches, estimate 2 assumes that we will require 6 disk accesses in each merge. 4 of these disk accesses are used to read the indexes to merge, while the last 2 are used to write out the resulting index. The calculations required to obtain our estimates are thus straight-forward, and the results are shown in column 5 in Table 6.4.

6.4.2.3 Resulting estimates

Batch	$t_{cpu,m}$	$t_{d,m,1,4096}$	$t_{d,m,1,16384}$	$t_{d,m,2}$	Est 1 (4KB)	Est 1 (16KB)	Est 2
1	10.98	1095.88	277.05	4.12	1095.88	277.05	10.98
2	37.33	4366.06	1103.78	16.40	4366.06	1103.78	37.33
3	55.89	6519.57	1648.19	24.45	6519.57	1648.19	55.89
4	74.44	8668.00	2191.32	32.48	8668.00	2191.32	74.44
5	92.97	10813.53	2733.72	40.50	10813.53	2733.72	92.97
6	111.50	12957.02	3275.59	48.51	12957.02	3275.59	111.50
7	130.02	15098.99	3817.10	56.52	15098.99	3817.10	130.02
8	148.54	17239.85	4358.32	64.52	17239.85	4358.32	148.54
9	167.06	19379.77	4899.29	72.52	19379.77	4899.29	167.06
10	185.57	21518.88	5440.07	80.52	21518.88	5440.07	185.57
11	204.08	23657.31	5980.68	88.51	23657.31	5980.68	204.08
12	222.59	25795.16	6521.13	96.50	25795.16	6521.13	222.59
13	241.10	27932.56	7061.47	104.49	27932.56	7061.47	241.10
14	259.60	30069.48	7601.68	112.48	30069.48	7601.68	259.60
15	278.11	32205.99	8141.81	120.47	32205.99	8141.81	278.11
16	296.61	34342.17	8681.85	128.45	34342.17	8681.85	296.61
17	315.11	36477.97	9221.78	136.43	36477.97	9221.78	315.11
18	333.61	38613.50	9761.65	144.42	38613.50	9761.65	333.61
19	337.33	39045.76	9868.54	146.05	39045.76	9868.54	337.33
				Sum:	405797.45	102585.02	3502.44

Table 6.4: Time spent processing and accessing disk while merging for different estimates in remerge with immediate merge

The first 5 columns of Table 6.4 show all estimates calculated for the different merges during a run with 1 million documents. As previously, we assume that I/O and processing can overlap while we are merging indexes. The final estimates for both estimate 1 and 2, are thus the maximum of the estimated time spent processing and the estimated time spent accessing disk. The estimated time spent accessing disk is dependent on the buffer size used in estimate 1. The last 3 columns of Table 6.4 show the resulting estimates.

It should be noted that according to estimate 1, all merge processes are I/O bound, while all merges are CPU bound according to estimate 2. We will have to evaluate the validity of each of these approaches when we report the actual results in Chapter 7.

The bottom line in Table 6.4 shows the total time we expect to use merging files with each of the estimates. By adding the number of seconds we expect to spend constructing partial indexes, we obtain the final estimates for each of our estimation methods. According to estimate 1, we will use 409490.62 seconds, or nearly 114 hours building the complete index with small buffers. With larger buffers, the estimate becomes 106278.19, or approximately 29.5 hours. According to estimate 2, the process is much more efficient and takes only 7195.61 seconds, or slightly less than 2 hours.

6.5 Hierarchical index

Significant parts of the construction phase for hierarchical indexes are similar to construction in remerge. New documents are added in batches, and some of the partial indexes are merged occasionally. When and what indexes to merge, depends on the parameters K and T, as explained in Section 4.2.4. We will not consider setting $T = 4$ or $K = 4$ in this section. The rationale behind this choice is that we know that our estimates for time spent accessing disk while merging are probably inaccurate. It is therefore very difficult to estimate how many small indexes that will be part of a merge when we allow adding more than 1 at a time. We defer a more thorough investigation of the effects of changing T to further work, after we have developed more precise estimates for disk accesses while merging indexes. The reason why we do not consider varying K is that we want to avoid making this section too long. We believe that estimating the expected time with $K = 2$ will help us determine to which extent our estimates are accurate. Performing the same calculations for $K = 4$ is not believed to add significant insight.

To develop estimates for hierarchical indexes, we should consider the differences between a hierarchical index and remerge in more detail. The construction of partial indexes is almost exactly similar. The only difference is that after each partial index has been created, the document lengths for the documents within the given index are calculated. This process is similar to the process carried out in remerge with immediate merge when the first batch is made searchable. When there are already several indexes in the hierarchy, we will perform a lookup in the dictionary for the largest one for each unique term in the newly added batch. This is done to calculate more accurate tf-idf document lengths, as explained in Section 4.2.6.2.

It might require disk accesses to look up in the largest dictionary, because we do not know how large parts of it that is buffered at a given moment. The dictionary is accessed sequentially because the look-ups are ordered lexicographically on the terms. It might seem reasonable to assume that the dictionary is read in a single sequential read. There is one problem with such an assumption however, that the partial index just created might have to be read at the same time. We thus end up with a situation which to some extent is similar to a merge of different indexes, except that we do not need to write out anything to the disk. To know the size of the dictionary file read, we need to know how the hierarchy looks like when a new partial index is made searchable.

To estimate the cost of the merges in the hierarchical index, we must know which merges we expect to occur. If this is known, we can estimate the size of the largest index when each new partial index is made searchable. We will therefore present which merges we expect to occur in Section 6.5.1. Section 6.5.2 will calculate the time we expect to spend to make all partial indexes searchable. In Section 6.5.3, we will estimate the time spent processing for the expected merges, and Section 6.5.4 estimates the time spent accessing disk in the same merges. We will provide a short summary of the results in Section 6.5.5.

6.5.1 Expected merges

This section will present which merges we expect in a hierarchical index with $K = 2$ and $T = 1$ when constructing an index with 1 million documents. To define a size of 1 in the hierarchy, Brille uses the number of buffers we are allowed to fill with an in-memory index. When a new index is constructed, either as a new partial index or as a result of a merge, we count the number of buffers it occupies. This determines at which position in the hierarchy we will put the index. It therefore seems likely that we are able to fill a size of 1 almost completely each time we fill the memory, and we make this assumption.

When we assume that each partial index nearly fills a size of 1, the evolution of the hierarchy with $K = 2$ is similar to counting in binary digits, as noted in [OvL80]. When the first partial index is constructed, it is added as a small index. We only allow one small index in the hierarchy where we search, and the process of adding new documents will thus have to wait. The partial index added as a small index is already searchable, and since there is no index at the first position in the hierarchy, the index is just moved there. We assume that the time used removing an entry from one list, and inserting it in another is negligible. The same thing will happen each time there is no small index in the hierarchy when we add a new one, and this is actually true in half the cases. On the other hand, a merge will always take place when the smallest index in the hierarchy is non-empty, and we have added a new partial index.

We thus end up with 9 actual merges, and Figure 6.3 shows all of them in correct order.

In Figure 6.3, it should be noted that the index to the far left in all hierarchies is strictly not part of the hierarchy. It represents the additional small index we allow searches in when $T = 1$. For the case with $T = 4$, we would have 4 such extra indexes.

As can be seen from Figure 6.3, the first merge will involve two indexes of size 1, and create an index of size 2. The next merge will merge two indexes of size 1 and one of size 2. The resulting index from this merge is expected to have size 4. We note that merges 1, 2, 4 and 8 will involve the largest index constructed up until now. We thus do not need to access the largest index to calculate more correct document lengths in these merges. The other five merges on the other hand, require accesses to the largest dictionary. This issue will be commented in the following subsections. We also note that we have estimated the size of both the dictionary and the inverted file for indexes that are merged from different numbers of full batches when estimating performance for remerge with immediate merge in Section 6.4.2. Those estimates are found in Table 6.3.

6.5.2 Constructing partial indexes

As noted above, we have already estimated the expected time used to construct a single partial index. In Section 6.2, we estimated that it takes 202.81 seconds to construct and flush a full partial index, and we have estimated that it takes 42.59 seconds to construct the one in the 19th batch. We also need to estimate the time spent making each partial index searchable, and that is

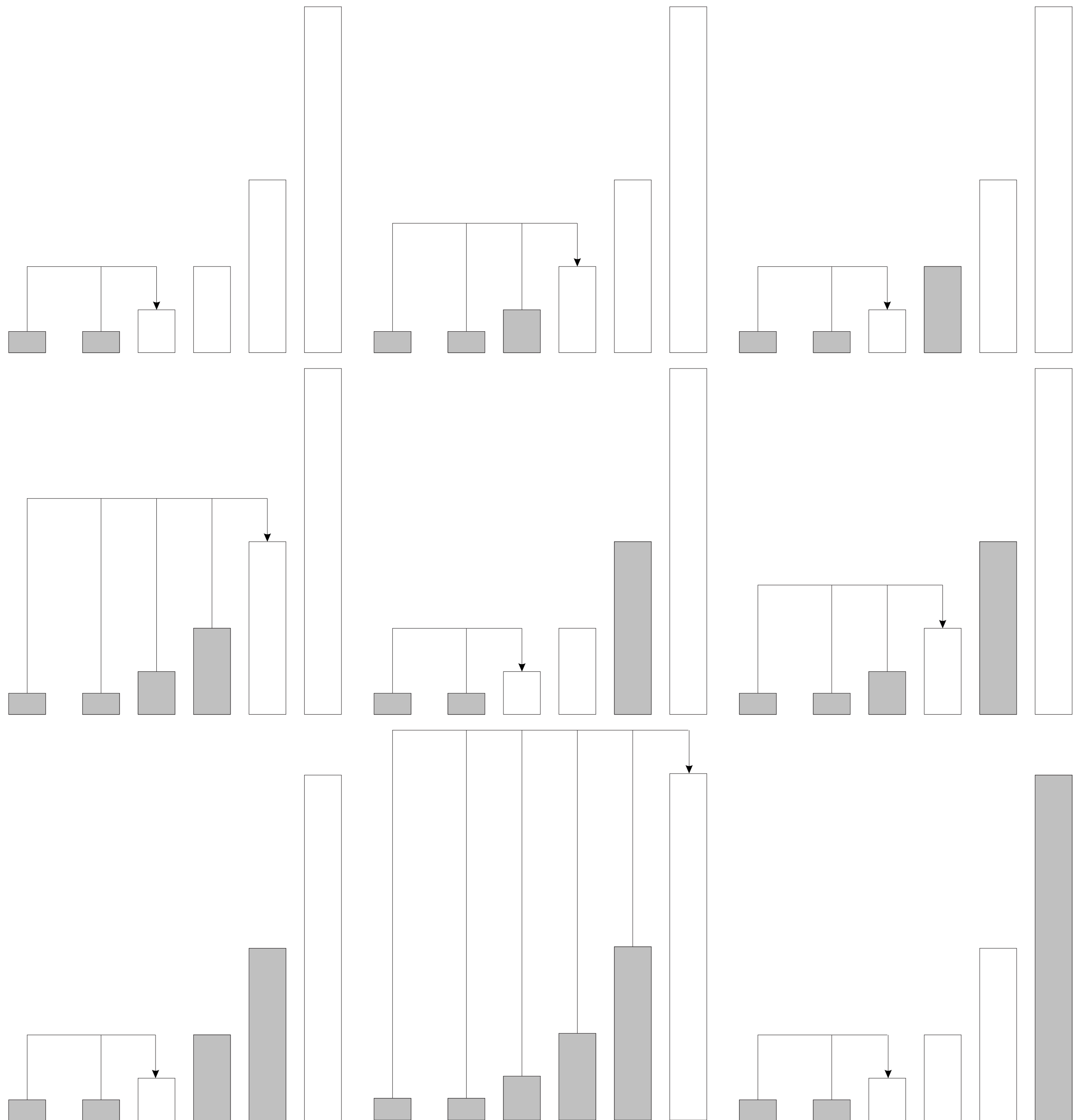

Figure 6.3: All expected merges while building a hierarchical index with $K = 2$ and $T = 1$ for 1 million documents

the focus of this section.

If there are any indexes in the hierarchy, making a partial index searchable involves a look-up in the largest dictionary for each unique term in the partial index. This will probably require us to read in the dictionary as well. Because there is both processing and disk I/O involved, we will calculate estimates for both, and choose the largest within each estimation method as our final estimate.

The first time we make a partial index searchable, there is no index in the hierarchy. This situation is thus similar to the first batch in remerge with immediate merge, and we have already calculated all necessary values in the first row in Table 6.4. When the second batch is added, there is one index in the hierarchy, the first partial index added. For the third and forth batches, the largest index in the hierarchy is merged from 2 batches. For batches 5 to 8, the largest index is merged from 4 batches. For the next 8 batches, the largest index is of size 8, and for the last 3 batches, it is of size 16.

The expected processing time for all batches except the first and last are similar, because our estimate for time spent looking up in a dictionary is independent of its size. We can estimate the processing time for all batches except the first as shown in Equation 6.35.

$$t_{cpu,sp} = N_p \cdot n_{d,u} \cdot t_{ur} + V_p \cdot l_{lu} \tag{6.35}$$

We note that the first part of Equation 6.35 is similar to Equation 6.33. The second part on the other hand, represents the look-ups in the dictionary. We need to perform one look-up for each distinct term in our partial index, and V_p is estimated as shown in Equation 6.14. The estimated time spent processing while making each batch searchable is shown in the second column in Table 6.5.

We also need to estimate the time spent accessing disk while making the partial indexes searchable. The files we possibly need to read are the dictionary file for the largest dictionary in the hierarchy, and the partial index we have just flushed. As noted in Section 6.4.2, it is not likely that we have to read the complete partial index, but we choose to make the conservative assumption that we need to do so. We can thus estimate the time spent accessing disk while making a partial index searchable, $t_{d,sp}$, for both disk estimates. The calculation for estimate 1 is shown in Equation 6.36, and the calculation for estimate 2 is shown in Equation 6.37. The obtained results are shown in the third, forth and fifth column of table 6.5.

$$t_{d,sp,1} = \left(\left\lceil \frac{|V_p|}{B} \right\rceil + \left\lceil \frac{|I_p|}{B} \right\rceil + \left\lceil \frac{|V_l|}{B} \right\rceil \right) \cdot t_s + (|V_p| + |I_p| + |V_l|) \cdot t_t \tag{6.36}$$

$$t_{d,sp,2} = 3 \cdot t_s + (|V_p| + |I_p| + |V_l|) \cdot t_t \tag{6.37}$$

$|V_l|$ in the above equations, is the size of the dictionary in the largest index in the current hierarchy. We have noticed above which batches that are added with different sizes of the largest index in the hierarchy. The expected size of the dictionary in an inverted index merged from a given number of batches is found in Table 6.3.

We have now calculated all required values to estimate the total time spent constructing all partial indexes and making them searchable. Constructing the partial indexes was estimated take 202.81 seconds for each of the first 18 batches, and 42.59 seconds for the 19th. We thus expect to spend 3693.17 seconds to construct all partial indexes. The table above contains the different estimates for how much time we will spend making the partial indexes searchable.

Batch	$t_{cpu,sp}$	$t_{d,sp,1,4096}$	$t_{d,sp,1,16384}$	$t_{d,sp,2}$	**Est 1 (4KB)**	**Est 1 (16KB)**	**Est 2**
1	10.98	1095.88	277.05	4.12	1095.88	277.05	10.98
2	15.24	1129.68	285.60	4.26	1129.68	285.60	15.24
3 and 4	15.24	1146.04	289.73	4.32	1146.04	289.73	15.24
5 to 8	15.24	1170.34	295.88	4.41	1170.34	295.88	15.24
9 to 16	15.24	1206.43	305.00	4.55	1206.43	305.00	15.24
17 and 18	15.24	1259.98	318.53	4.75	1259.98	318.53	15.24
19	3.99	400.81	101.33	1.54	400.81	101.33	3.99
				Weighted sum:	21771.21	5504.02	274.05

Table 6.5: Time spent processing and accessing disk while making partial indexes searchable

By adding the time spent constructing partial indexes with the different estimates for making them searchable, we end up with our final estimates. For estimate 1 with small buffers we expect to spend 25464.38 seconds constructing all partial indexes and making them searchable. The expected times for estimate 1 with large buffers and estimate 2, are 9197.19 and 3967.22 seconds respectively. We again note that we end up with much better expected performance if estimate 2 is closest to reality. In such a case, we would be able to construct all partial indexes and make them searchable in approximately 1 hour and 6 minutes. If we use the small buffer size, estimate 1 expects it to take more than 7 hours.

These times could have been appropriate estimates on the total time spent constructing a hierarchical index if we set $T = 4$, never had to wait for merges of small indexes, and if the merging process did not slow down the construction of partial indexes significantly. We have decided to estimate the time spent constructing the hierarchical index when $T = 1$ however. We will therefore have to wait for an index to be merged into the hierarchy every time a new one is created. We thus have to calculate the expected time spent merging as well.

6.5.3 Processing merges

Figure 6.3 gives an overview of the merges we expect to occur while constructing the hierarchical index for 1 million documents. This section will estimate the expected time spent processing during each of these merges. As noted in Section 6.5.1, 4 of the 9 merges include the currently largest index in the hierarchy, and will thus not need to access the largest dictionary to calculate more correct tf-idf document lengths. The remaining 5 on the other hand, will require an access in the largest dictionary for each unique term in the merged index. We will first calculate the expected processing time for the four merges that do not require dictionary look-ups.

When no look-up in the largest dictionary is required, the merges in the hierarchical index are just like the merges considered in Section 6.3. Their expected processing time can thus be calculated as shown in Equation 6.25. This applies to merges 1, 2, 4 and 8, and the results are shown in the second column in Table 6.6.

When a look-up is required for each unique term, the expected time spent processing in a merge

is as shown in Equation 6.38.

$$t_{cpu,m} = t_{se} \cdot \log(m) \cdot \sum_{i=1}^{m} \left(16.24 \cdot (N_i \cdot n_{d,t})^{0.57}\right) + (t_{se} + t_{ur}) \cdot n_{d,u} \cdot N + 16.24 \cdot (N \cdot n_{d,t})^{0.57} \cdot l_{lu} \tag{6.38}$$

We notice that the only difference between Equation 6.38 and Equation 6.25 is the last part, which estimates the cost of the look-ups. The results when estimating time spent processing during the merges with look-ups in the largest dictionary are also shown in the second column of Table 6.6.

6.5.4 Time spent accessing disk in merges

This section will estimate the time spent accessing disk during the merges introduced in Section 6.5.1. 4 of the expected merges will not involve look-ups in the largest dictionary in the hierarchy, because the currently largest index is part of the merge. These merges are thus handled just like standard merges, which was discussed in Section 6.3. Their estimated time spent accessing disk are calculated as shown in equations 6.20 and 6.21 for estimate 1, and as in Equation 6.22 for estimate 2.

The remaining 5 merges requires accesses to the largest dictionary in the hierarchy, with size denoted $|V_l|$. Estimate 1 will, as usual, assume that only B bytes at a time can be read from the file. Estimate 2 on the other hand, assumes a sequential read with only 1 single disk access for the complete file. The calculation for estimate 1 in this case, is shown in equations 6.39 and 6.40, while Equation 6.41 shows the calculation for estimate 2.

$$d_a = \left\lceil \frac{|V_l|}{B} \right\rceil + \left\lceil \frac{\sum_{i=1}^{m} |I_i|}{B} \right\rceil + \left\lceil \frac{|V_p|}{B} \right\rceil + \sum_{i=1}^{m} \left(\left\lceil \frac{|V_i|}{B} \right\rceil + \left\lceil \frac{|I_i|}{B} \right\rceil \right) \tag{6.39}$$

$$t_{d,m,1} = d_a \cdot t_s + \left(|V_l| + |V_p| + \sum_{i=1}^{m} (|V_i| + 2 \cdot |I_i|) \right) t_t \tag{6.40}$$

$$t_{d,m,2} = (2 \cdot m + 3) t_s + \left(|V_l| + |V_p| + \sum_{i=1}^{m} (|V_i| + 2 \cdot |I_i|) \right) t_t \tag{6.41}$$

Based on the above equations, estimates for the time spent accessing disk during the merges have been calculated. The results are shown in the third, fourth and fifth column of Table 6.6.

The last 3 columns shows the resulting estimates when using the different methods to estimate time spent accessing disk. The resulting estimate is the maximum of the time we estimate to

Merge	$t_{cpu,m}$	$t_{d,m,1,4096}$	$t_{d,m,1,16384}$	$t_{d,m,2}$	**Est 1 (4KB)**	**Est 1 (16KB)**	**Est 2**
1	37.33	4366.06	1103.78	16.4	4366.06	1103.78	37.33
2	74.96	8688.79	2196.59	32.58	8688.79	2196.59	74.96
3	43.66	4440.52	1122.61	16.69	4440.52	1122.61	43.66
4	149.99	17295.87	4372.5	64.78	17295.87	4372.5	149.99
5	43.66	4476.62	1131.73	16.83	4476.62	1131.73	43.66
6	84.36	8799.35	2224.54	33.01	8799.35	2224.54	84.36
7	43.66	4476.62	1131.73	16.83	4476.62	1131.73	43.66
8	299.6	34453.0	8709.89	128.94	34453.0	8709.89	299.6
9	43.66	4530.16	1145.26	17.03	4530.16	1145.26	43.66
				Sum:	91526.97	23138.6	820.89

Table 6.6: Time spent processing and accessing disk while merging for different estimates in a hierarchical index with $K = 2$ and $T = 1$

spend processing, and the time we estimate to spend accessing disk. As for remerge, when estimate 1 is used, all merges are I/O bound. If estimate 2 is an accurate estimate on the other hand, processing is the bottleneck.

6.5.5 Expected times for hierarchical index construction

We are now able to give precise estimates for the time we expect to spend constructing a hierarchical index for 1 million documents with $K = 2$ and $T = 1$. In Section 6.5.2, we calculated the expected time spent constructing the partial indexes and making them searchable. Because we estimate for $T = 1$, we will have to wait for all merges. Table 6.6 contains the results from our calculation of expected time for performing the merges. Adding the values for each estimation method, we obtain our final estimates for constructing the complete index. We expect it to take 116991.35 seconds when using the small buffers according to estimate 1. 116991.35 seconds is approximately 32 hours and 30 minutes. Estimate 1 expects the process to be more efficient with larger buffers. For $B = 16$ KB, our estimate becomes 32335.79 seconds, or slightly under 9 hours. According to estimate 2, we will spend only 4788.11 seconds, which is approximately 1 hour and 20 minutes. We will compare these expected values with the obtained results in Chapter 7.

As noted at the beginning of this section, we do not calculate precise estimates for other configurations. We will however, give a less formal explanation of what we expect.

Setting $K = 4$ will typically improve the search efficiency, but the construction speed will typically suffer. The search efficiency will improve because there are on average fewer indexes in the hierarchy, and the construction speed will decrease because the merges are larger on average.

How much time we expect to spend building the whole index when $T = 4$ depends heavily on which of our two estimates of disk performance that is most accurate. According to estimate 1, the merges are definitely disk bound. We noted in Section 6.2 that the whole process of

constructing partial indexes is CPU bound. If we allow building new partial indexes during a merge, it is thus likely that the two processes can run in parallel without degrading each others performance. This implies that the merges will run constantly, while we add the next small indexes to the hierarchy. It is thus likely that the time spent constructing partial indexes will have limited effect on the total construction time.

If estimate 2 is accurate on the other hand, the merges are expected to be CPU bound. Because the construction of partial indexes is also CPU bound, we need a good utilization of our dual-core processor to obtain a significantly faster construction. The construction of partial indexes uses several threads, and we thus consider a significant improvement in construction time unlikely in this case.

6.6 Naive B-tree index

The naive B-tree index differs quite a lot from the other two methods, because it does not go through phases, but rather keeps doing the same throughout its lifecycle. The naive B-tree index processes one document at a time. The document is first parsed, a process that, just like for the other methods, goes on in a thread on its own. The mapping between the document number and the URI of the document is then inserted into the B-tree in the document manager. There are two insertions into the B-tree index for all unique terms in the document, and the tf-idf length for the document is calculated while these entries are inserted in the B-tree.

We will try to give estimates for both time spent accessing disk and time spent processing for this structure as well, but there are some complicating factors here. When accessing a B-tree, we will search downwards from the root. In each node, we perform a binary search which will tell us which node to access next. The buffer containing the given node is then pinned. If this buffer is not currently cached, we will have to read it into a buffer. Because the binary search that determines which buffer to pin was just finished, it seems unlikely that this read can be carried out while we are processing. It is thus likely that the processing will have to wait while we are reading in the next buffer. We therefore do not assume that the operations on the B-tree and the possible reads can overlap. For the B-tree in the document manager, we did not consider this aspect. Because we only insert documents with strictly increasing document numbers in our experiments, we will follow the same path down from the root to a leaf at each insert in the B-tree in the document manager. It is therefore unlikely that we will need many reads overall. We thus believe that it makes sense to consider this aspect for the naive B-tree index, and not for the B-tree in the document manager.

The above observation implies that we can not assume that processing in the B-tree and reads from disk can go on in parallel. We also know that the constructed B-tree will have to be written to disk. Read and write operations on a disk can not go on in parallel either.

In addition, as for the other index structures, the documents are parsed in a thread on its own. This will of course require both disk accesses and processing, as explained in Section 6.2.1. The documents to index are stored on another disk than the constructed index however, and we

thus assume that reading them will not interfere with the disk accesses to the index files. When there is only one feeding thread for the index, it also seems likely that the processing performed when parsing documents can go on in parallel with the insertions in the B-tree.

Based on the above assumptions, there are 3 different processes that might be the bottleneck during construction of a naive B-tree index for N documents:

- Reading in documents and parsing them.
- Constructing the index for the parsed documents, and reading in the necessary buffers in the B-tree.
- All disk operations performed on the disk with the index files. These operations include the read operations mentioned under the previous item.

In order to obtain estimates for each of these possible bottlenecks, we need to know the size and height of the complete B-tree index. We estimate these values in Section 6.6.1. We will go through each of the possible bottlenecks in sections 6.6.2, 6.6.3 and 6.6.4 respectively. Section 6.6.5 will give a summary of our findings and provide estimates for the expected time used to construct the index with 300000 documents. The estimates will be given as a function of hit ratio, h. The hit ratio defines the fraction of the accessed buffers that do *not* require a read operation. We have thus chosen not to estimate an average hit ratio, because we do not find a reasonable basis for providing such an estimate. The hit ratio will in addition be dependent on the size of the index, which complicates the aspect even further.

To avoid too much complexity in our estimates, we will not consider using more than 1 feeding thread for the index. It should also be noted that there is a background thread in Brille that updates the document lengths of all documents at regular intervals. This will of course cause additional disk accesses and processing time, but we will not consider its effects here. We note that this possibly introduces a significant inaccuracy in our estimates, but it is left out to keep this section within reasonable length.

6.6.1 The size of a complete B-tree index

Each unique term in each document will lead to an entry in the naive B-tree index. We know the average number of unique terms, and the average number of terms in one document. Based on these values, we are able to calculate the average number of occurrences of each unique term in each document. This will again enable us to estimate the expected average size of each entry in the B-tree index, denoted s_e. The calculation of s_e is shown in Equation 6.42.

$$s_e = l_e + 12 + 4 \cdot \frac{n_{d,t}}{n_{d,u}} \tag{6.42}$$

The rationale behind Equation 6.42 is that the term and the document number will have to be stored. In addition, we use 2 bytes to store the length of the term and 2 bytes for the length of

the occurrence list. The entries in the occurrence list consume 4 bytes each. The last 4 bytes are used to store the pointer in the node to where the given entry is stored. Substituting our estimates into the equation, we obtain an estimated average $s_e = 85.62$ bytes. We will have $N \cdot n_{d,u}$ entries with this average size in our B-tree.

There is also another type of entry in our naive B-tree index, namely the ones storing statistics about the occurrences of each term. There will be one such entry per unique term in the collection. This entry will store the actual term, in addition to 20 bytes. It uses the first 12 bytes just like the normal entries, and always has 2 stored occurrences. The occurrences represent the number of documents containing the term and the total number of occurrences of the term in the indexed document collection. The number of unique terms is estimated according to Heap's law, as in Equation 6.14.

With N documents in the collection and assuming a space efficiency in the B-tree of $\frac{2}{3}$, we can estimate the number of leaf nodes as we did in Equation 6.9. The calculation for our current setting is given in Equation 6.43.

$$nodes_{leaf,bt} = \left\lceil \frac{\left(s_e \cdot N \cdot n_{d,u} + (l_u + 20) \cdot 16.24 \cdot (N \cdot n_{d,t})^{0.57}\right) \cdot 3}{B \cdot 2} \right\rceil \tag{6.43}$$

To estimate the total number of nodes, we also need to know the average size of internal entries in the B-tree. Each internal entry contains a term, a pointer to a node and 6 extra bytes. The length of the term is not as easily estimated here. We know that there are $n_i + 1$ entries in the B-tree for term i. Our estimates for l_u and l_e suggest that the common terms are on average shorter than the less common ones, because l_u is larger than l_e. Since we now have one extra entry per term, we might believe that the average length of the terms stored in upper levels of the B-tree are longer than l_u. On the other hand, the most common terms typically also have the largest number of occurrences in each document, making their entries larger. This makes it more likely to find common terms in higher levels of the B-tree. We thus estimate that l_e is a reasonable value for the average length of terms in the internal entries in the B-tree as well.

On average, this should make the size of an internal node equal to 16.01 bytes. When we take into account that each node has an average space efficiency of $\frac{2}{3}$, the average entry will occupy approximately 24 bytes. We thus get the approximate number of nodes given in Equation 6.44.

$$nodes_{bt} \approx 1 + nodes_{leaf,bt} \cdot \frac{B}{B - 24} \tag{6.44}$$

According to the above calculations, the average branching factor for each internal node is $\frac{B}{24}$. The height of the B-tree, denoted he, is defined as the number of pointers one has to follow from the root to reach a leaf node. With $nodes_{leaf}$ leaf nodes, the average height is calculated as shown in Equation 6.45.

$$he = \left\lceil \log_{\frac{B}{24}}(nodes_{leaf}) \right\rceil \tag{6.45}$$

We note that the number of nodes we need to access to insert or search for an entry in a B-tree with height he is $he + 1$.

6.6.2 Reading and parsing documents

Reading and parsing documents for the naive B-tree index proceeds just like for the other methods tested in this report. In Section 6.2.1, we estimated the expected time spent reading and parsing documents, and adding them to an in-memory index. We concluded that the bottleneck in such a phase was parsing the documents, while reading them was far more efficient. Both reading and parsing documents scales linearly with the number of documents. It is therefore clear that parsing the documents will be the bottleneck in the naive B-tree index as well. We can thus estimate the time spent reading and parsing documents as shown in Equation 6.46.

$$N \cdot n_{d,t} \cdot t_p \tag{6.46}$$

6.6.3 Constructing the index

A thread responsible for adding the parsed documents to the index performs several operations. It inserts the mapping between URI and document number in the B-tree in the document manager, and inserts one entry and increment or insert another entry for each unique term in the document. It also calculates the document length for the inserted document. This section will estimate the expected processing time needed to perform these operations, in addition to the possible time needed to access disk to read in needed parts of the index files.

According to our previous estimates of time spent inserting the mapping between document number and URI in the document manager, it should take $N \cdot t_{bi}$ seconds to add the mappings for all documents. Likewise, calculating the rank for each document is expected to take $N \cdot n_{d,u} \cdot t_{ur}$ seconds in total.

Finally each index entry is inserted into the B-tree index, along with an inc-or-insert operation for the entry describing the occurrences of this term. We have not estimated the average cost of an inc-or-insert operation, but its locking scheme is identical to an insert. It therefore seems likely that its cost is also quite similar to the cost of an insert. We make this assumption, and thus expect to spend $2 \cdot N \cdot n_{d,u} \cdot t_{bi}$ seconds inserting into the tree. As mentioned in the beginning of this section, we might have to wait while reading in a block that is not currently cached. The number of times we have to read in a block depends on the hit ratio. As mentioned above, it is difficult to estimate the hit ratio.

Even if we do not have an estimate for the hit ratio, we can calculate the expected time used to construct a naive B-tree index with the hit ratio as a variable. We know that we can calculate the expected average height of the B-tree in a naive B-tree index as shown in Equation 6.45. If we assume that the B-tree has the same size through the whole construction phase, we can

easily estimate the number of nodes accessed. Note that the height will obviously grow during construction, making this assumption incorrect. The height of a B-tree grows slower and slower however. It will thus grow fast at the beginning, and will probably quickly reach its final height. The assumption is thus not as crude as it may seem. With such an assumption, we can estimate the total number of B-tree nodes accessed as $2 \cdot N \cdot n_{d,u} \cdot (1 + \log_{\frac{B}{24}}(nodes_{leaf}))$.

Denoting the hit ratio by h, we get the estimated time spent in a thread adding parsed documents to the index as shown in Equation 6.47.

$$t_{cpu,bt} = N \cdot (1 + 2 \cdot n_{d,u}) \cdot t_{bi} + N \cdot n_{d,u} \cdot t_{ur} + 2 \cdot N \cdot n_{d,u} \cdot (1 + he) \cdot (t_s + B \cdot t_t) \cdot (1 - h) \quad (6.47)$$

6.6.4 Disk accesses in the naive B-tree index

The B-tree is updated 2 times per unique term in each document. When entries are inserted or updated, the buffers representing the updated nodes are eventually unpinned and appended to the queue of dirty buffers in the buffer pool. At regular intervals, the queue of dirty buffers is traversed, and they are all written to their respective *FileChannel*s. Before a dirty buffer is written to its *FileChannel*, it might be pinned again, and is thus no longer part of the queue. In addition, we do not know exactly when the changes in a *FileChannel* are written to disk. It is therefore very hard to estimate the exact number of times the buffer representing a node in the B-tree is written to disk before its final version is represented at disk. It is even harder to estimate the amount of disk accesses these I/O operations will lead to, because the *FileChannel* might be able to write out several changes parts in only one disk access.

In Section 6.2.1.1, we concluded that during a complete run, we will at least write all nodes in the B-tree to disk once. The best imaginable case is that all buffers representing the nodes are written to disk in one single access, but we note that this behaviour is very unlikely. If we can not store the complete B-tree in memory, it is probably impossible. The worst case is that all buffers are written to disk each time there is a change in the node the buffer represents, and if each write requires a disk access.

How we should obtain a reasonable estimate on the amount of time spent writing in the B-tree index is at best unclear. We thus only make a choice, and will have to evaluate whether this choice was appropriate or not when we get the results from the actual experiments in Chapter 7. We therefore assume that each buffer in the B-tree index will be written only once, but that each of these writes will require a disk access. It is very likely that each buffer will be written more than once, but it is difficult to quantify, as explained above. On the other hand, it is unlikely that a disk access is required for each of the actual writes, so the estimated number of disk accesses is thus not necessarily that far from reality. In addition, all these writes are within the same file. A file system will usually try to store a single file sequentially on the disk, and the disk accesses within one file will thus typically not be as costly as an average one on the disk.

The above assumption has another important benefit, namely that it makes the calculation of our estimated time spent accessing disk quite simple. The estimated time spent writing to the

B-tree index is calculated as shown in Equation 6.48.

$$nodes_{bt} \cdot t_s + nodes_{bt} \cdot B \cdot t_t \tag{6.48}$$

In addition to the disk writes considered above, we have to store the mappings between document number and URI at the same disk as our B-tree index. Using the same assumptions as in Section 6.2.1.1, we estimate the size of the B-tree in the document manager as shown in equations 6.49 and 6.50.

$$nodes_{leaf,dm} = \left\lceil \frac{26 \cdot N \cdot 3}{2 \cdot B} \right\rceil \tag{6.49}$$

$$nodes_{dm} = 1 + nodes_{leaf,dm} \cdot \frac{B}{B - 18} \tag{6.50}$$

We have previously made the same assumptions about writes to the B-tree in the document manager as we did for the B-tree index above. We make the same assumption here.

To obtain the estimated time spent accessing the disk with the index files, we also need to consider the possible reads from the B-tree index. The time spent performing these, as a function of the hit ratio, was calculated in the previous section. We do not consider possible reads from the B-tree in the document manager because each new document number is strictly larger than the previous, as explained in Section 6.2.1.1.

With the above assumptions, we estimate the time spent accessing the disk containing the index files to be as given in Equation 6.51.

$$t_d = (nodes_{dm} + nodes_{bt} + 2 \cdot N \cdot n_{d,u} \cdot (1 - h) \cdot (1 + he)) \cdot (t_s + \cdot B \cdot t_t) \tag{6.51}$$

6.6.5 Estimates for $N = 300000$

We have now calculated estimates for all of the possible bottlenecks in the naive B-tree index introduced at the beginning of this section. All estimates are a function of N and h. As explained above, we will not estimate h, but rather plot a graph representing the expected time spent constructing an index with 300000 documents for varying h. The reason why we estimate with $N = 300000$ for the naive B-tree index, is that this is the largest index size for which we have run the experiment 10 times.

Our final estimate is, for a given hit ratio, the maximum time we expect to spend in any of the 3 different possible bottlenecks. We will now carry out the calculations for each of the possible bottlenecks. The resulting value for reading and parsing documents is as shown in Equation 6.52.

$$N \cdot n_{d,t} \cdot t_p = 459.82 \tag{6.52}$$

Parsing all documents is thus expected to take approximately 459.82 seconds. The time spent adding all documents to the index when we use $B = 4096$ is calculated below with the answer in Equation 6.53.

$$nodes_{leaf,bt} = \left\lceil \frac{\left(s_e \cdot N \cdot n_{d,u} + (l_u + 20) \cdot 16.24 \cdot (N \cdot n_{d,t})^{0.57}\right) \cdot 3}{B \cdot 2} \right\rceil = 980631$$

$$he = \left\lceil \log_{\frac{B}{24}}(nodes_{leaf}) \right\rceil = 3$$

$$t_{cpu,bt} = N \cdot (1 + 2 \cdot n_{d,u}) \cdot t_{bi} + N \cdot n_{d,u} \cdot t_{ur} + 2 \cdot N \cdot n_{d,u} \cdot (1 + he) \cdot (t_s + B \cdot t_t) \cdot (1 - h) =$$
$$14598.49 + 8834652.25 \cdot (1 - h) = 8849250.74 - 8834652.25 \cdot h \tag{6.53}$$

Equation 6.54 gives the expected time spent adding documents to the index when $B = 16384$.

$$nodes_{leaf,bt} = \left\lceil \frac{\left(s_e \cdot N \cdot n_{d,u} + (l_u + 20) \cdot 16.24 \cdot (N \cdot n_{d,t})^{0.57}\right) \cdot 3}{B \cdot 2} \right\rceil = 245158$$

$$he = \left\lceil \log_{\frac{B}{24}}(nodes_{leaf}) \right\rceil = 3$$

$$t_{cpu,bt} = N \cdot (1 + 2 \cdot n_{d,u}) \cdot t_{bi} + N \cdot n_{d,u} \cdot t_{ur} + 2 \cdot N \cdot n_{d,u} \cdot (1 + he) \cdot (t_s + B \cdot t_t) \cdot (1 - h) =$$
$$14598.49 + 8933724.98 \cdot (1 - h) = 8948323.47 - 8933724.98 \cdot h \tag{6.54}$$

From the above calculations, we can already estimate that the time spent processing to parse the documents will never be the bottleneck, regardless of the actual value of the hit ratio, h. To consider whether the time spent adding documents to the index or the time spent accessing the index files is the bottleneck, we calculate our estimates for the latter below. We start with the case where $B = 4096$, and the answer for this case is given in Equation 6.55.

$$nodes_{bt} = 1 + nodes_{leaf,bt} \cdot \frac{B}{B - 24} = 986412$$

$$nodes_{leaf,dm} = \left\lceil \frac{26 \cdot N \cdot 3}{2 \cdot B} \right\rceil = 2857$$

$$nodes_{dm} = 1 + nodes_{leaf,dm} \cdot \frac{B}{B - 18} = 2871$$

$$t_d = (nodes_{dm} + nodes_{bt} + 2 \cdot N \cdot n_{d,u} \cdot (1 - h) \cdot (1 + he)) \cdot (t_s + \cdot B \cdot t_t) =$$
$$13008.23 + 8834652.25 \cdot (1 - h) = 8847660.48 - 8834652.25 \cdot h \tag{6.55}$$

Here follows the same calculation when $B = 16384$, and the answer is given in Equation 6.56.

$$nodes_{bt} = 1 + nodes_{leaf,bt} \cdot \frac{B}{B-24} = 245519$$

$$nodes_{leaf,dm} = \left\lceil \frac{26 \cdot N \cdot 3}{2 \cdot B} \right\rceil = 715$$

$$nodes_{dm} = 1 + nodes_{leaf,dm} \cdot \frac{B}{B-18} = 718$$

$$t_d = (nodes_{dm} + nodes_{bt} + 2 \cdot N \cdot n_{d,u} \cdot (1-h) \cdot (1+he)) \cdot (t_s + \cdot B \cdot t_t) =$$
$$3274.12 + 8933724.98 \cdot (1-h) = 8936999.10 - 8933724.98 \cdot h \quad (6.56)$$

The calculations above suggest that adding documents to the index is the bottleneck. We note however, that even with our optimistic estimate regarding the number of times the contents of a node is written to disk, the time spent accessing disk is estimated to be relatively close to the time spent adding the documents to the index. This is especially true when we use the smallest buffers. We will keep this optimistic assumption however, but keep in mind that is relatively likely that constructing naive B-tree indexes is disk bound, at least when the size of the index becomes reasonably large.

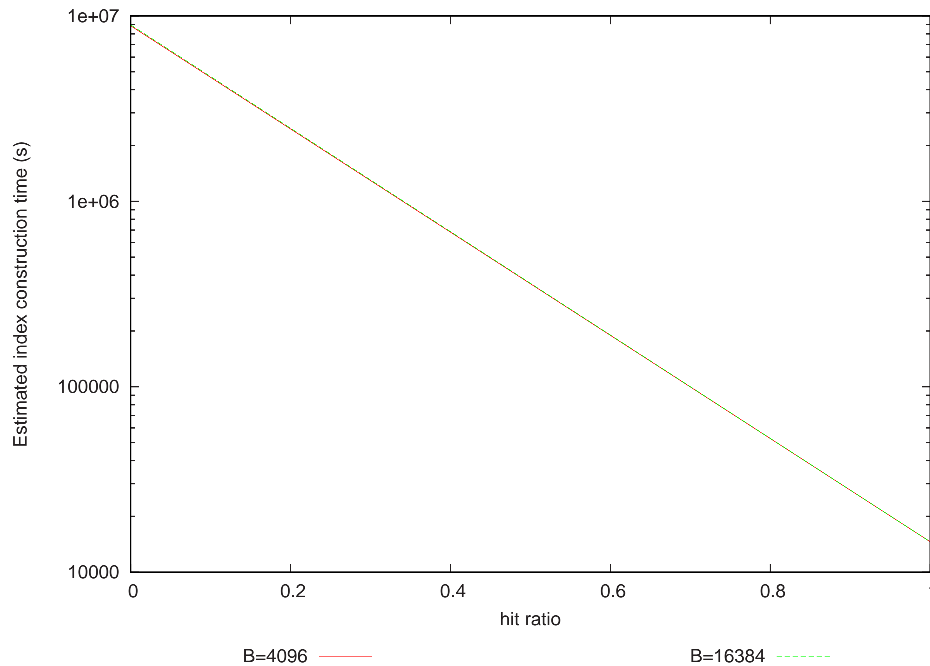

Figure 6.4: Expected construction time for naive B-tree index as a function of hit ratio

Figure 6.4 shows the resulting estimates for constructing a naive B-tree index with 300000 documents. These estimates show that it is crucial to have a decent hit ratio to obtain reasonable performance. With a hit ratio of 1.0, we are able to index approximately 20.55 documents per second according to our estimates. If there was no buffering involved and all accessed nodes had to be read from disk at every access, we would not be able to index more than approximately

0.034 documents per second regardless of the buffer size used. We should also note that even if the height of the B-tree is similar for both buffer sizes. The B-tree with $B = 16$KB will reach this height later in the construction process. The estimates for this case are thus more pessimistic than the estimates for time spent constructing with small buffers.

We have decided not to estimate the hit ratio, but we note that there are some aspects that make it likely that it is relatively high. Because of the assumed Zipf distribution of terms in the collection, there are some very frequently occurring terms. When new documents are added, they usually contain several of these frequent terms. Inserting the resulting index entries involves one access at the beginning of the list of entries for the term, and one at the end. For frequent terms, it seems very likely that the paths to the nodes to update are already cached. It is of course also likely that the nodes at the top of the tree are cached most of the time, because they are accessed by many operations.

It should also be noted that the hit ratio is dependent on the size of the complete B-tree index. When we have free buffers to cache the complete index, the hit ratio will be 1.0. When the index size grows beyond this limit however, the hit ratio will decrease. How much it will decrease depends on the replacement policy. We thus expect the number of documents indexed per second to drop gradually as the size of the index increases beyond the total size of the buffers in our buffer pool.

We have not estimated the time spent constructing a naive B-tree index when the number of feeding threads is set to 4 instead of 1, but we will discuss it in a less formal way now. We concluded above that the time spent constructing the index seems to be the major bottleneck. By using several feeding threads, we might be able to do some of this work in parallel. Because we have a processor with 2 CPUs, we can never expect to run more than two such threads simultaneously though. It might seem like there is no point in using 4 feeding threads, but we expect this to have an effect when the hit ratio drops below 1.0. When this happens, some of the feeding threads will from time to time be forced to wait while the next node they should access is read from disk. When we have multiple feeding threads, it is likely that some of them can perform useful work while the others are waiting for disk operations. We thus expect the performance to improve when we use 4 feeding threads.

6.7 Expected search performance

This section will estimate the expected time spent performing a search in each of the tested index structures. We test searching for terms with varying number of hits in this report, and we expect the search performance in all structures to be dependent on the frequency of the terms. We will therefore provide estimates for all three groups of tested terms in each index structure.

In all tested searches, we only retrieve the top 10 ranked results. We thus typically look up 10 mappings between document number and URI in the document manager. If there are fewer than 10 hits, we will only look up the number of actual hits. Because it is unlikely that the complete B-tree in the document manager is cached when we perform the searches, looking up

the mappings between document number and URI will typically involve some disk accesses. We will not estimate the cost of these however, because this part does not discriminate between the different kinds of index structures. This means that our estimate for time spent searching for terms with various frequency is likely to be lower than the actual time spent, but we will hopefully be able to estimate the difference in performance between the various structures.

For each of the index structures tested in this report, we will estimate the search performance for the same configuration for which we estimated construction time in the previous sections of this chapter. To be able to estimate time spent searching for terms with various frequency, some statistics for the different groups of terms are given in Table 6.7.

Frequency of terms	**Average** n_i	**Average** occ_i
high	349684.82	3362359.01
medium	1088.17	1849.79
low	1.00	1.00

Table 6.7: Statistics for various groups of terms

The following subsections will estimate the time spent performing a search for terms with various frequency in all tested index structures.

6.7.1 Remerge

Regardless of whether we use remerge as an off-line construction method or with immediate merge, the search performance is equivalent. A search in the structure involves a look-up in the dictionary, which will give us a pointer into an inverted file from where we will have to read. The read inverted list is then processed to find the top 10 ranked documents. The amount of processing involved for each index entry is small, there are only a few additions and multiplications. We thus make the simplifying assumption that after the initial look-up in the dictionary, the disk operations are the bottleneck.

We have estimated the time it takes to perform a look-up in the dictionary, t_{ul}. In addition, the look-up might involve a disk access and a read of one buffer. This will happen if the disk block containing the entry in the dictionary for the term searched for is not currently cached. Next, if the term occurs in the document collection, its inverted list is read. We assume that the list is read in one disk access. How many bytes an inverted list consists of, b, is dependent on n_i and occ_i for the particular term. We can calculate b as shown in Equation 6.57.

$$b = 8 \cdot n_i + 4 \cdot occ_i \tag{6.57}$$

The rationale behind Equation 6.57 is that for each document containing a term, we store the document number and the length of the occurrence list as integers. The occurrences of the term within a document are also stored as integers.

If we assume that all searches involves a disk access in the dictionary, we can estimate that the time spent searching for a term in remerge is as given in Equation 6.58. Note that in our implementation, we will always read an intregral number of buffers. This means that we will typically read slightly more than b bytes when reading an inverted list. We choose to not consider this aspect however, because its effect on the overall time spent is relatively small compared to the disk access.

$$t_{search,r} = t_{lu} + 2 \cdot t_s + (b + B) \cdot t_t \tag{6.58}$$

Note that assuming that each search requires a disk access in the dictionary is the same as expecting worst-case performance in each search. We choose to make this assumption, even though it is not likely to be accurate for the average case. By using the statistics given in Table 6.7, we are now able to calculate the expected time for a term with average frequency within each of our three groups of terms. The results are shown in Table 6.8. Note that these calculations are only applicable for the case when we have an index with $N = 1$ million documents.

6.7.2 Hierarchical index

To estimate the search performance in the hierarchical index, we will consider how the hierarchy will look like when the construction phase is done. By adding the 19th batch after the final merge in Figure 6.3, we end up with the hierarchy shown in Figure 6.5.

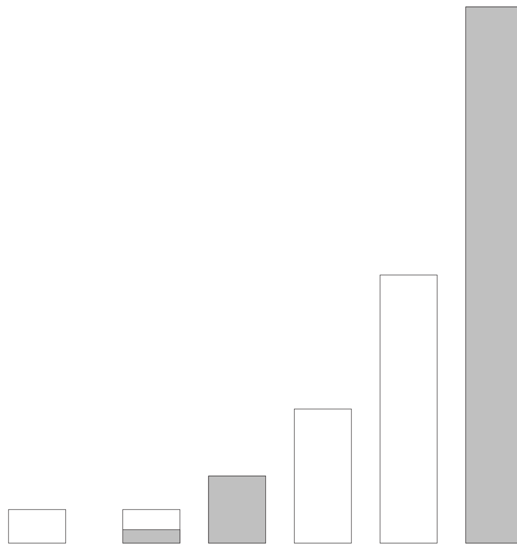

Figure 6.5: Resulting hierarchical index with $K = 2$, $T = 1$ and $N = 1$ million

We notice from figure 6.5 that there are 3 non-empty indexes in the hierarchy. Each of these three indexes are similar to the single searchable index in remerge, except that these indexes are smaller. To search for a term in this hierarchy, we have to perform a look-up in each of

the dictionaries. We again assume worst-case performance, namely that this look-up involves a disk access. For all of the indexes that contains the term searched for, we need to access the inverted list. If we assume that all indexes in the hierarchy contains occurrences of the term, we will have to perform three times as many disk accesses as with remerge. We will not read more data from the inverted files than we did with remerge however. This means that we can calculate b just like we did in Equation 6.57.

For the terms with high and medium frequency, we assume that there exist occurrences for all of these terms in all indexes. For these groups of terms, we can calculate the expected time spent searching as in Equation 6.59.

$$t_{search,h} = 3 \cdot (t_{lu} + 2 \cdot t_s) + (b + 3 \cdot B) \cdot t_t \tag{6.59}$$

For the group of terms with only one occurrence each, it is not reasonable to assume that all indexes contains an inverted list for all terms. When there is only one occurrence of a term in the document collection, it can never have an inverted list in more than one of the indexes in the hierarchy. We have to look up in all dictionaries however, and the expected time spent searching for a term with one occurrence is calculated as in Equation 6.60

$$t_{search,h} = 3 \cdot t_{lu} + 4 \cdot t_s + (b + 3 \cdot B) \cdot t_t \tag{6.60}$$

The results from calculating the expected search time based on equations 6.59 and 6.60 is shown in table 6.8.

Term frequency	**Remerge**		**Hierarchical index**	
	$B = 4$KB	$B = 16$KB	$B = 4$KB	$B = 16$KB
high	0.221	0.221	0.274	0.274
medium	0.026	0.027	0.079	0.079
low	0.026	0.026	0.052	0.053

Table 6.8: Expected average time for searches for terms with various frequency ($N = 1$ million)

Note that the calculation was only carried out for the special case with $N = 1$ million documents, $K = 2$ and $T = 1$. With other configurations, the expected time is different. In general, the maximum number of indexes we need to search in when $T = 1$ is $\lceil \log_K(pi) \rceil + 1$, where pi is the expected number of partial indexes as previously. When T is larger than 1, we might have to search in the additional small indexes as well. We also note that the number of bytes read from all inverted files in a hierarchical index is similar to the number of bytes read from the single index in remerge. This implies that regardless of whether there are 1000 or 1 million occurrences of the term searched for, the difference in expected time between remerge and the hierarchical index is constant. If the frequency of the term is very low, the difference in expected time is even lower, because we may not need to look up inverted lists in all indexes in the hierarchy.

6.7.3 Naive B-tree index

For the naive B-tree index, we will estimate the search performance for an index with 300000 documents, for the same reasons why we estimated construction time with this index size. The statistics in Table 6.7 are thus not correct for this particular case. To find more appropriate values, we assume that all terms are uniformly distributed in the document collection. This implies that we will have 30% of the occurrences of the terms in the first 300000 of the 1 million documents. Table 6.9 shows the adjusted values.

Frequency of terms	**Average n_i**	**Average occ_i**
high	104905.45	1008707.70
medium	326.45	554.94
low	0.30	0.30

Table 6.9: Adjusted statistics for the groups of terms

A search in the naive B-tree index is just like a search in a normal B-tree. It starts from the root, and proceeds downwards through the tree by performing a binary search in each node. We have estimated the time it takes to search in a B-tree in terms of processing speed, t_{bs}. We make the same assumption here as we did for the two other methods, namely that the time spent reading an inverted list from disk dominates the time spent processing it.

When analyzing the other methods in the previous subsections, we assumed worst-case behaviour with respect to the number of disk accesses performed in the dictionary. The worst case number of disk accesses when searching for a single entry in a B-tree is $he + 1$. The root will obviously be cached, at least after the first search, and it is also highly likely that most of the nodes on the next level are cached. Even so, to follow the same overall approach we did for the other methods, we assume that all nodes will have to be read. This implies $he + 1$ disk accesses for each search.

Each entry in a B-tree will be stored within a single node. A search for a term that exists in our B-tree index will however always have to read at least two entries. The first entry is the one storing statistics about the approximate number of documents containing the term, and the total number of occurrences of the term. The following entries will contain the actual entries in the inverted list. Even if our above calculations suggest that the average rare term has 0.3 occurrences in the collection considered, we assume that all terms exist in the index when we calculate. This implies that we can find an upper bound on the number of nodes read from disk as given in Equation 6.61.

$$he + 1 + \left\lceil \frac{b}{B} \right\rceil \tag{6.61}$$

The variable b in Equation 6.61 is the expected number of bytes used to store the inverted list, not including the entry with statistics for the occurrences of the term. The rationale behind Equation 6.61 is that we might need to read $he + 1$ nodes to find the one with the entry with statistics. If the expected number of bytes in the inverted list is larger than zero, we might

have to access $\lceil \frac{b}{B} \rceil$ more nodes to read the whole inverted list. We should note that we have no guarantee that the leaf nodes are stored sequentially on disk, it is actually very unlikely. It therefore seems reasonable to assume that we need one disk access per node read.

We also need to estimate b. For each index entry, there is one entry in the B-tree containing the actual term, the document number, and all occurrences of the term within the document. Each entry also has a pointer to it within the node, to enable binary searches when the entries have variable lengths. As we did when we considered the expected construction time for the B-tree, we will assume that the space efficiency in the B-tree is $\frac{2}{3}$. Based on these considerations, the calculation of b is shown in Equation 6.62.

$$b = \frac{3}{2} \cdot ((l_u + 12) \cdot n_i + 4 \cdot occ_i) \tag{6.62}$$

The calculation of the complete time spent searching for a term in a naive B-tree index is as shown in Equation 6.63.

$$t_{search,bt} = t_{bs} + \left(he + 1 + \left\lceil \frac{b}{B} \right\rceil \right) \cdot (t_s + B \cdot t_t) \tag{6.63}$$

By substituting the average values for each group of terms from Table 6.9 into equations 6.62 and 6.63, we obtain the results shown in Table 6.10.

Term frequency	**Naive B-tree index**	
	$B = 4$KB	$B = 16$KB
high	23.08	7.51
medium	0.092	0.066
low	0.066	0.066

Table 6.10: Expected average time for searches for terms with various frequency ($N = 300000$)

We note that the B-tree is expected to provide slower searches than hierarchical indexes and indexes constructed with remerge. The difference is most significant when searching for frequent terms. We keep in mind that the index we estimate for here contains 300000 documents, while the one considered for the other two methods contains 1 million documents.

Chapter 7

Results

This chapter presents the results from the experiments performed in this project. We start by introducing the experiment environment in Section 7.1. In the following three sections, we go through the results with each of the structures. We will also evaluate to which extent the actual results fit in with our efficiency model developed in Chapter 6.

The results obtained for each method will be used to declare the best tested configurations for each method, as the author sees it. The chosen configurations for each method will be compared in Section 7.5. In Section 7.6, we will discuss the reliability of the results, by evaluating the intrusiveness of the output generated by the test runs. We will consider possible future improvements of our implementation in Section 7.7.

7.1 Experiment environment

CPU:	Intel Core 2 Duo E6700 2.67GHz CPU, FSB1066, 4MB cache
Memory:	2 x 1 GB DDR2 Dual Channel DIMM, PC6400 (DDR2-800)
Disk:	2 x 500 GB Samsung HD501LJ, 7200 RPM, 16 MB, Serial ATA II (NCQ)
OS:	Ubuntu server 7.04
Java version:	5.0

Table 7.1: Characteristics for experiment environment

Table 7.1 gives the characteristics of the experiment environment used in all experiments in this report.

7.2 Remerge

This section provides the results obtained when testing remerge, both as an off-line construction method and with immediate merges. All the planned experiments for this method, as presented in Section 5.3, were completed successfully.

The following subsections compare the different configurations tested for remerge based on update speed, update latency and search performance. Section 7.2.4 compares the obtained results with the estimates developed in Chapter 6, while Section 7.2.5 gives a deeper analysis of interesting aspects in the obtained results. We will then choose one configuration of remerge to compare with representatives from the other tested methods in Section 7.2.6

7.2.1 Update speed

As mentioned in Section 5.3.1, we construct indexes from scratch to measure the update speed for different methods. All but the largest experiments are run 10 times. Based on these 10 runs, we calculate the average time spent constructing the index, and the sample standard deviation. Denoting the time spent constructing the index in run i as x_i, and the average as $\bar{x}$, we calculate the sample standard deviation as shown in Equation 7.1.

$$\sqrt{\frac{1}{10}\sum_{i=1}^{10}(x_i - \bar{x})^2} \tag{7.1}$$

Figure 7.1 shows the average time spent constructing an index with remerge for various sizes of the index and various configurations, and Figure 7.2 shows the calculated sample standard deviation. The average construction times are also given in Table 7.2, to faciliate reading precise values.

Documents	**Off-line construction**		**Immediate merge**	
	$B = 16$KB	$B = 4$KB	$B = 16$KB	$B = 4$KB
100000.0	0h 5m 52s	0h 6m 29s	0h 6m 11s	0h 6m 43s
300000.0	0h 19m 18s	0h 19m 23s	0h 22m 16s	0h 23m 58s
1000000.0	1h 8m 25s	1h 9m 43s	1h 43m 5s	1h 48m 1s
10000000.0	8h 55m 39s	10h 44m 46s	45h 39m 2s	151h 11m 42s

Table 7.2: Average time spent constructing indexes with remerge

We note that off-line construction with large buffers is the fastest configuration for all index sizes. It constructs the index with 10 million documents in less than 9 hours, which implies that more than 311 documents are indexed each second. Using 4KB buffers instead of 16KB makes off-line construction slightly slower on average, but the difference is relatively small. We keep in mind that the largest experiments were only run once. We should thus be careful not to conclude much based on the largest experiments. Even though the difference between

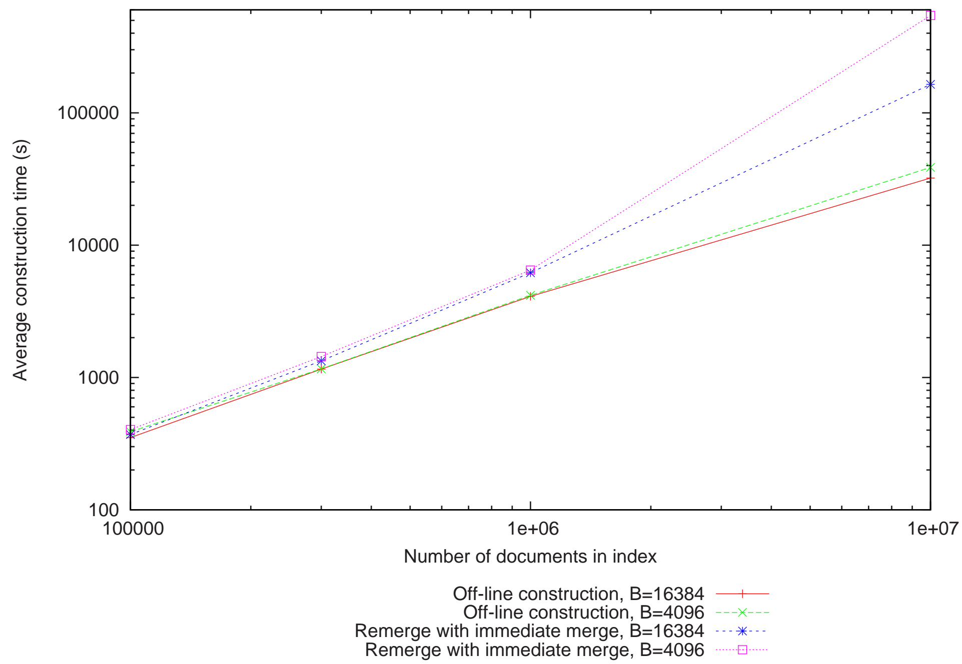

Figure 7.1: Average construction time with remerge for indexes with various size

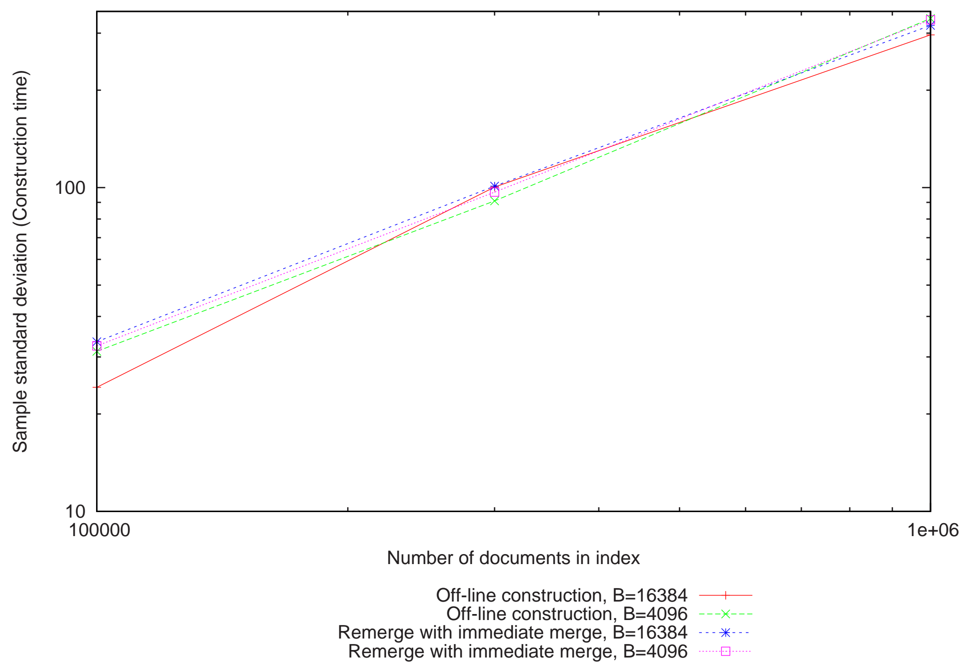

Figure 7.2: Sample standard deviation for construction in remerge with indexes of various size

small and large buffer sizes for off-line construction is nearly 2 hours there, the sample standard deviations from the other index sizes suggest that there is a significant variance.

When using immediate merge, the construction seems to be significantly slower, at least for larger index sizes. This configuration will perform a merge each time we have filled our memory. The total size of all files we merge is thus proportional to the square of the size of a partial index. This quadratic factor is not much noticeable for small indexes, but for large ones, it plays a significant role. We also note that the difference in construction time when using small instead of large buffers is overwhelming for the index with 10 million documents. Regardless of the buffer size used, both versions of remerge with immediate merge will merge the same files. It is possible that merges are faster when using larger buffers, but according to the results with off-line construction, it seems unlikely that the difference is significant enough to fully explain this behaviour. We will analyze the cause of this unexpected result in more detail in Section 7.2.5.

The sample standard deviation is quite similar for all methods. It increases with increasing index size, as expected, and is slightly lower for the fastest methods. There are of course several causes of variability in our implementation. When running a multi-threaded application, the scheduling of threads within the application is an obvious cause. In addition, the operating system and other applications are running on the experiment computer, and this may of course also have effects on the running time of our application.

7.2.2 Update latency

In Section 5.3.1, we decided not to test update latency by simulating a process where new documents arrive at a given rate. We rather let each method index as fast as possible, and note the time passed from the documents are read until they are searchable. We measure disk and CPU utilization in all phases of construction. In the first phase of constructing a partial index, the documents are read. When this phase ends, we record the number of read documents. We thus know when we have read the first batch, but we do not know exactly when each particular document is read. We therefore assume that we read a constant number of documents per second within each accumulation phase.

We also record the time at which documents become searchable. Based on the information about when documents are read and become searchable, we are able to construct two graphs. One of the graphs describes when the documents are read, and the other when they are searchable. The area between these two graphs provides a visualization of the average update latency.

Figure 7.3 shows the graphs describing when documents are read and become searchable when using off-line construction and 16KB buffers. A similar graph for remerge with immediate merge and 4KB buffers is shown in Figure 7.4.

From Figure 7.3, we notice that all documents are read before any of them become searchable. The stair-case function for the graph describing when documents are read shows that documents are read while we fill the in-memory index. The phases where the number of read documents does not increase are when the partial index is written out to disk, and at the end when all partial indexes are merged. The area between the two graphs in Figure 7.3 shows that the average update latency is large. We are also able to state the worst-case update latency for this method,

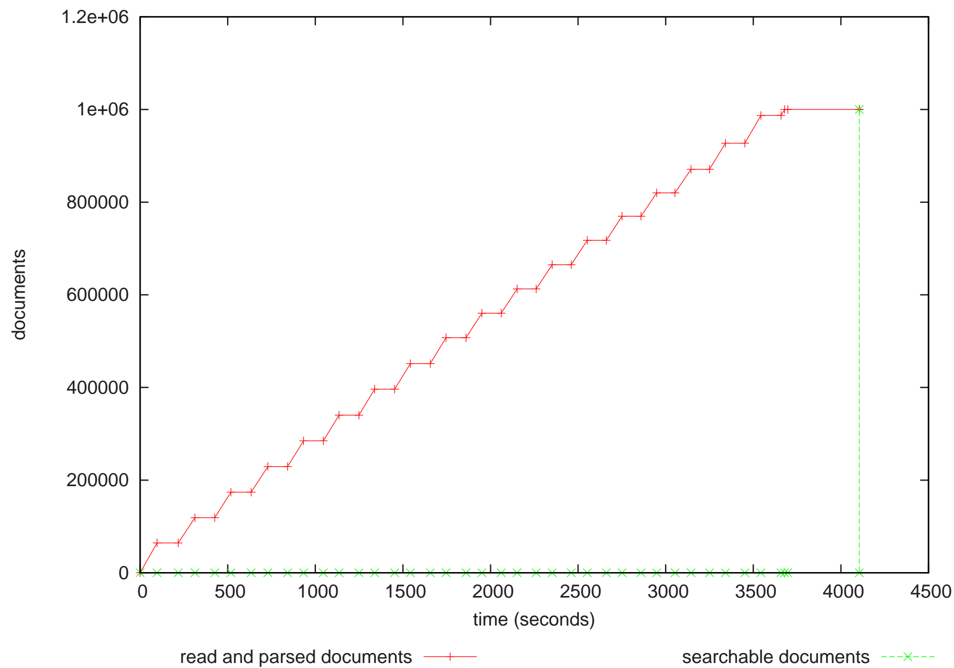

Figure 7.3: Read and searchable documents over time using off-line construction, 16kB buffers and 1 million documents

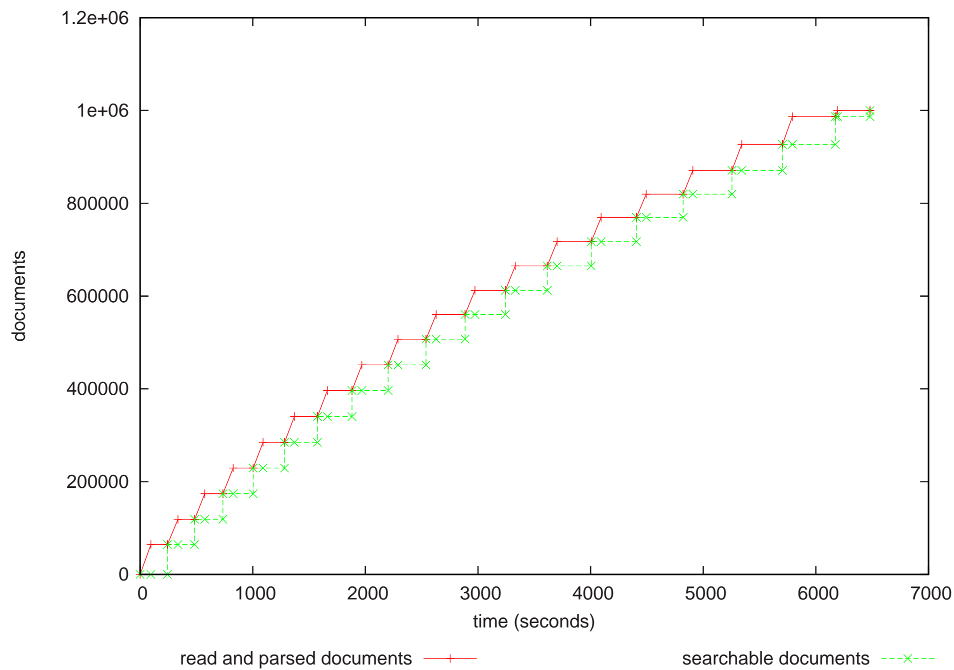

Figure 7.4: Read and searchable documents over time using remerge with immediate merge, 4kB buffers and 1000000 documents

which is actually equal to the total construction time. Even though constructing an inverted

index with off-line construction is reasonably efficient; the update latency is a drawback.

When using immediate merge, the new partial indexes are merged into the main index immediately. When a partial index is merged into the main index, the document lengths are calculated and the documents are searchable when the merge finishes. Figure 7.4 shows that this gives us a much more attractive average update latency. We note however, that the update latency seems to be larger and larger as more and more documents are added. This observation fits well with the results from Section 7.2.1.

7.2.3 Search performance

This section presents the results from the experiments with search performance for the different configurations of remerge. As explained in Section 5.3.2, we test search performance by measuring the time spent performing 100 searches for terms with either high, medium or low frequency. The same searches are carried out in all tested index sizes.

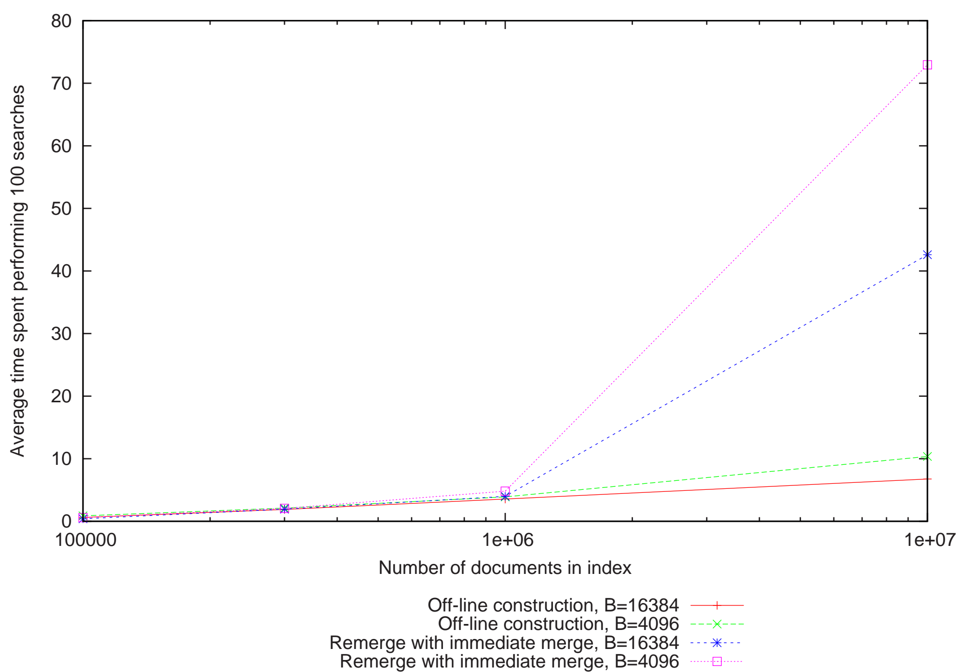

Figure 7.5: Average time spent performing 100 searches for terms with low frequency in indexes constructed by remerge

Figure 7.5 shows the average time spent performing 100 searches for terms with low frequency in various configurations of remerge. Note the logarithmic scale used on the x-axis. For all but the largest index size, the configurations show quite similar performance. For the largest index size, the configurations with immediate merge are significantly slower than the others. We keep in mind that the largest experiments are only run once.

Figure 7.6 shows the results from searching for terms with medium frequency. Again, the con-

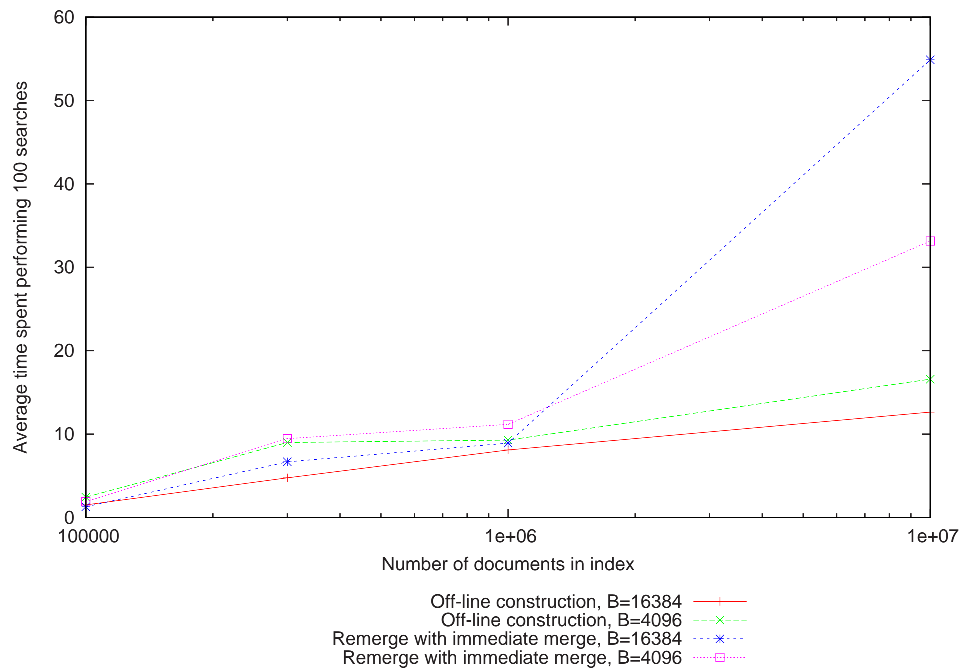

Figure 7.6: Average time spent performing 100 searches for terms with medium frequency in indexes constructed by remerge

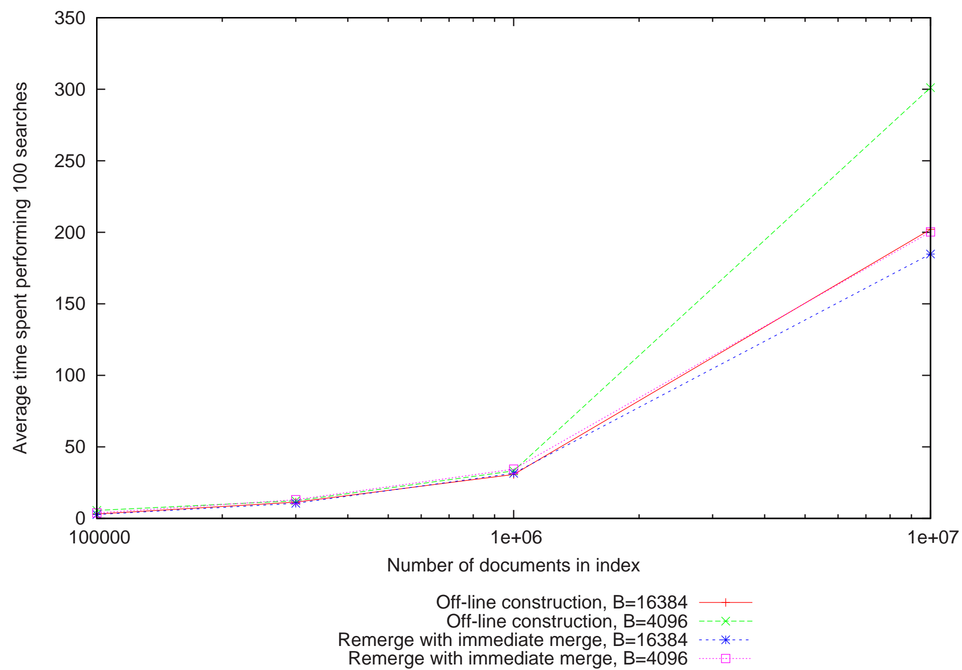

Figure 7.7: Average time spent performing 100 searches for terms with high frequency in indexes constructed by remerge

figurations with immediate merge are slower than the others, especially for the larger indexes. The differences are not that significant in this case however. The average time spent searching for frequent terms in the different configurations is shown in Figure 7.7. In this case, all methods show quite similar performance, with the exception that the configuration with $B = 4$KB in off-line construction is slightly slower than the others. Because the largest experiments are only run once, we should be careful not to overanalyze these results. Without further empirical basis, we thus assume that the slow search speed in the largest index for off-line construction with small buffers is caused by variability. It is interesting to note however, that the methods with immediate merge performs well in this case.

7.2.4 Actual results versus estimates

In Chapter 6, we developed estimates for how much time we expected to spend constructing indexes with the different methods tested. This section will compare the actual results with our estimates. We will first analyze off-line construction; before we consider building indexes with remerge with immediate merge. Section 7.2.4.3 will consider the search performance for all configurations.

7.2.4.1 Off-line construction

In Section 6.4.1, we estimated how much time we would spend constructing an index with $N = 1$ million documents using off-line construction. We used two different estimates for time spent accessing disk during a merge. The first estimate assumed that each buffer read from or written to a file required a disk access. Estimate 2 on the other hand, noted that the operating system and the disk controller will probably employ read ahead. The number of disk accesses is thus probably significantly lower, and estimate 2 assumed that we only use one disk access per partial file.

Both estimates expect the construction of the partial indexes to take approximately 3693.17 seconds. Their estimate for time spent during the merge differs a lot however, leading to different final estimates. The final estimates are given in Table 7.3.

Estimate 1		**Estimate 2**
$B = 4$KB	$B = 16$KB	
11h 59m 25s	3h 47m 52s	1h 7m 25s

Table 7.3: Final estimates for time spent constructing an index with $N = 1$ million documents with off-line construction

It should also be noted that according to estimate 1, the process of merging indexes is disk bound regardless of the size of the buffers, while the calculations in estimate 2 suggest that merges are CPU bound.

The results we should compare these estimates with are found in Table 7.2. We note that constructing an index with 1 million documents takes on average 1 hour, 8 minutes and 25 seconds when we use large buffers, and 1 hour, 9 minutes and 43 seconds with small buffers. It thus seems obvious that estimate 2 is far more accurate than estimate 1. We will not jump to such a conclusion however, but rather investigate how the structure behaves in each of the phases of construction.

To analyze the different phases in construction, we have included a log from one of the experiment runs in Appendix B.1. Even if this log is from a single run, it provides some insight regarding the bottleneck during different phases.

All of our estimates suggest that both the phase accumulating an index in memory, and the phase sorting the entries and writing the index to disk are CPU bound. To consider whether this is true, we look at the CPU utilization in each phase in the log. For the phase accumulating an index in memory, the user time is typically around $75 - 85\%$ in the most active CPU. According to our estimates, it is likely that this CPU is running the thread that reads in and parses documents. In one accumulation phase, the user time is as low as 65% on the most active CPU. This should make us doubt to which extent the phase is actually CPU bound. The time spent waiting for I/O is typically 0 or 1% however. We thus conclude that it is likely that this phase is actually CPU bound. We have carried out some unreported experiments to investigate this further. We changed the organization of how documents are read slightly, and the user time in the CPU was then nearly 100% each time. We are therefore fairly confident with this conclusion.

In some of the phases sorting entries and writing the partial index to disk, the user time on the CPU is reported to be close to 100%. There are some phases however, where the user time is much lower on the most active CPU. An important observation here is that the user time on the other CPU is typically also substantial in these phases. The sum of the user time on both CPUs is typically quite close to 100. It is clearly possible that the thread sorting the entries switches which core it runs on during this phase. It thus seems reasonable to assume that this phase is also in fact, CPU bound. The fact that we have two phases of constructing partial indexes that seems to be CPU bound is discussed in Section 7.7.

As noted above, our estimates expect all phases constructing partial indexes to take approximately 3693.17 seconds, or slightly more than an hour. The actual results show that the process takes approximately 3697 seconds when using large buffers, and 3670 seconds using small buffers. For large buffers, the estimate is thus very close. It is a small surprise that construction of partial indexes is faster when using small buffers however, because it involves 4 times as many calls to the buffer pool to pin new buffers. We will not go into more detail on this issue here, but we present an aspect that might explain the observed behaviour to some extent in Section 7.3.4.1. Despite this slightly unexpected result, we conclude that our estimates for construction of partial indexes seem to be accurate when using off-line construction.

The comparison above suggested that estimate 2 was the most accurate estimate for the time spent merging. According to estimate 2, the merging process is also CPU bound. This seems unlikely when we consider the log in Appendix B.1 however. In the merging phase, the user time on one processor core is 38%, and it is 2% on the other. The CPUs spent 58 and 28% of their time waiting for I/O however. This makes it reasonable to assume that the phase is disk

bound. A further investigation of the number of reads and writes completed on the disk with the index files supports this theory.

This is a quite interesting result. Even though estimate 2 was quite accurate when estimating the time spent merging the partial indexes, its conclusion that the phase is CPU bound is not correct. This suggests that there are actually two problems with our estimates for the time spent merging. We are probably overestimating the time spent processing during the merge, and the actual time spent accessing disk is significantly larger than estimate 2 assumes.

It is not straight-forward to see the cause of the first of these problems. The merge involves a merge of the dictionaries and of the inverted lists. In addition, the document lengths of all documents are calculated. Both the estimate for time spent processing during the actual merges and the estimate for time spent processing to calculate the length of all documents may be inaccurate. We consider it likely that the estimated constant in a multi-way merge, t_{se} is inaccurate, because we had problems finding a reliable estimate for it, as explained in Section 6.1.2.2. The difference between estimated and actual processing time may of course also be explained by the fact that we estimated that a merge of dictionaries will have worst-case complexity on average. Unfortunately, we are in no position to pinpoint the problem at this point. To obtain more reliable estimates for variables like t_{se}, we should probably avoid measuring them in a multi-threaded environment. It is of course possible to extract parts of the code, and only run it in one thread. In future work, we should probably follow such a strategy.

The second problem with our estimates for merging is not a surprise. We indicated in Section 6.3.1 that estimate 1 is probably an upper bound on the time spent accessing disk, while estimate 2 is probably a lower bound. The results presented here support this belief. To enable more accurate estimates in the future however, we should compare the actual number of disk reads and writes to the estimated number of disk accesses for merges of different sizes. According to results of our experiments, we will have 19 batches with sizes close to the estimated ones from Section 6.3.1 when constructing an index for $N = 1$ million documents. For $N = 10$ million documents, there are 140 batches, and the last is about $\frac{1}{3}$ full. When $N = 300000$ documents, we have 6 batches, and the last one has approximately the same size as the last one when $N = 10$ million documents. We are thus able to calculate the estimated size of all files read and written in each of these cases following the same strategy as in Section 6.2.2.1. The results are shown in Table 7.4.

$N = 300000$		$N = 1$ million		$N = 10$ million	
Files read	Files written	Files read	Files written	Files read	Files written
1.71GB	1.68GB	5.79GB	5.66GB	44.31GB	43.13GB

Table 7.4: Estimated total size of files read and written in the merge in off-line construction

To get a clearer idea of the average amount of data read and written in one operation we provide the average number of disk operations with different buffer sizes in Table 7.5.

From Table 7.5, we note that the writes are on average larger than the reads. We calculate the average size of both reads and writes in the different merges in Table 7.6. We should note however, that our implementation does not guarantee that all required disk writes are performed

Buffer size	$N = 300000$		$N = 1$ million		$N = 10$ million	
	#reads	#writes	#reads	#writes	#reads	#writes
4KB	15059.4	5104.1	59578.6	19644.5	1310558.0	336041.0
16KB	14377.1	5260.1	57809.5	19083.8	701248.0	129304.0

Table 7.5: Average number of disk operations during merges in off-line construction

when the phase is done. We only know that all dirty buffers are unpinned in the buffer pool, not that the flushing thread has written them to disk. Because we are not able to find the exact number of disk writes performed after the phase is done, we will assume that all of them are performed within this phase. We keep in mind however, that this assumption may be inaccurate.

Buffer size	$N = 300000$		$N = 1$ million		$N = 10$ million	
	Reads	Writes	Reads	Writes	Reads	Writes
4KB	119.1KB	345.1KB	101.9KB	302.1KB	35.5KB	134.6KB
16KB	124.7KB	334.9KB	105.0KB	311.0KB	66.3KB	349.8KB

Table 7.6: Average amount of moved data in disk operations during merges in off-line construction

Table 7.6 shows that the average size of both reads and writes is significantly higher than our buffer size in all cases. It also seems to be a general tendency that the size of the operations drop when the total size of the partial files to merge increases, even though this observation does not fit for the writes in the largest index with large buffers. We should keep in mind however, that this experiment was only run once, and we should be careful not to draw conclusions on such a limited empirical basis.

We should also note that the estimated sizes of the inverted files are reasonably accurate, but that the sizes of the dictionaries are typically larger in practice than in our estimates. This implies that the average sizes of the reads and writes are in fact slightly larger than Table 7.6 suggests. It is hard to draw conclusions for how the size of the writes behaves in general, especially when we take the observation above into account. We note however, that the average size of the reads is never larger than 128KB. As the number of files to merge increse, the average size of a read decrease. We might get tempted to assume that regardless of the size of a read, the read-ahead in the disk controller and/or in the operating system will ensure that we actually read 128KB. If the amount of I/O to other files is large, we will typically not be able to take advantage of the read ahead to the same extent. The reason is that the contents buffered for a specific file may have been replaced before we access the next buffer. This might explain the drop in the size of average reads as the number of files to merge increases. To be able to make such a conclusion however, we need more thorough experiments. An interesting experiment would be to use 128KB as the buffer size in our implementation. If this makes the average size of reads very close to 128KB, our theory is more likely to be correct. We defer such an investigation to further work.

7.2.4.2 Immediate merge

Section 6.4.2 provided estimates for expected time spent constructing an index for 1 million documents using remerge with immediate merge. The final estimates are repeated in Table 7.7.

Estimate 1		Estimate 2
$B = 4$KB	$B = 16$KB	
113h 44m 51s	29h 31m 18s	1h 59m 56s

Table 7.7: Final estimates for time spent constructing an index with $N = 1$ million documents using remerge with immediate merge

The actual results are found in Table 7.2. We again notice that the expected time according to estimate 1 is too pessimistic regardless of the size of the buffers. Estimate 2 is closer, but this is actually also too pessimistic in this case. When using large buffers, the actual average time spent constructing an index with $N = 1$ million documents is 1 houe, 43 minutes and 5 seconds. When using small buffers, the time spent is 1 hour, 48 minutes and 1 second. The actual results are thus approximately 17 and 12 minutes faster than expected when using large and small buffers, respectively. To analyze the cause of this result, we have included a log from one of the experiment runs in Appendix B.2. The log is from an experiment with $B = 16$KB.

To figure out the cause of the difference between the estimate and the actual time spent, we consider the differences in each phase. In the log in Appendix B.2, the time spent constructing partial indexes is approximately 3837 seconds, which is slightly higher than when using off-line construction. There are several runs where we spend less time constructing partial indexes however. The average over all runs is close to the average when using off-line construction.

The above observation implies that the difference between estimated and observed time spent when constructing an index with remerge with immediate merge is likely to be caused by the merges. This assumption is confirmed by analyzing the log. We spend approximately the same time in the first merges as our estimates from Section 6.4.2 suggest. As the merges become larger and larger however, the difference between estimated and observed time spent seems to grow. For the small 19th batch, estimate 2 assumed that the process was CPU bound, and that it would take approximately 337.33 seconds. In the experiment with the log included in Appendix B.2, this merge takes 259 seconds. As for off-line construction, the disk statistics for the merges shows that they are disk bound. We thus experience the same problems with the estimates for the merges in remerge with immediate merge as we did when using off-line construction. We discussed these problems in the previous section, and will thus not analyze them further here.

7.2.4.3 Search performance

In Section 6.7.1, we calculated the worst-case time we expected to spend searching for a term with average frequency in all the groups of terms we test searching for. We did not consider the look-ups of URIs in the document manager for the matching documents however. We argued

that the search performance for all configurations of remerge were likely to be similar, and the estimated time spent on each search is summarized in Table 7.8.

Frequency	$B = 4$KB	$B = 16$KB
Low	0.026 s	0.026 s
Medium	0.026 s	0.027 s
High	0.221 s	0.221 s

Table 7.8: Expected time spent searching for terms with different frequencies in remerge

We presented the results from the experiments testing search performance in Section 7.2.3. The plots presented there show the average time spent performing 100 searches. To make it easier to compare the estimated values and the obtained results, we repeat the results in Table 7.9. All values are divided by 100 before they are inserted into the table, to obtain the average time spent searching for a single term.

Frequency	Off-line construction		Immediate merge	
	$B = 4$KB	$B = 16$KB	$B = 4$KB	$B = 16$KB
Low	0.039 s	0.036 s	0.048 s	0.040 s
Medium	0.093 s	0.081 s	0.112 s	0.089 s
High	0.332 s	0.307 s	0.344 s	0.315 s

Table 7.9: Average time spent searching for terms with different frequencies in remerge

The are several interesting observations in tables 7.8 and 7.9. The most important is that the estimated values are generally smaller than the observed values. This is especially true in searches for terms with medium frequency, while searches for terms with low frequency are only slightly slower than expected. For the most frequent terms, it is also slower to search in practice than our estimates suggest. As noted above, we have argued that it is irrelevant for the search performance whether the index has been constructed with off-line construction or remerge with immediate merge. This theory is not supported by the results, although the differences are relatively small. We believe that there are three aspects that might help explaining these unexpected results:

1. **Accesses to document manager not included in estimates:** As mentioned above, our estimates for time spent searching for terms did not consider the accesses to the B-tree in the document manager to retrieve the URI for the documents to return. When searching for the terms with medium or high frequency, we will typically look up the URI for the top ten ranked documents. In searches for terms with low frequency, there is never more than 1 document that matches the query. We thus only have to look up one URI. This might require several disk accesses however, because the mappings between document number and URI are stored in a B-tree.

   We always perform the searches for the least common terms first. This will typically imply that no parts of the B-tree in the document manager are buffered when we search for these terms. This is because we have just completed a merge of significant size in both variations of the method. Performing a large merge will typically fill the buffer pool

with parts from the files involved in the merge. If we actually require one disk access per look-up of for each URI in the searches for terms with low frequency, the difference between the estimated and actual value is explained.

For the searches for more frequent terms, we will typically perform 10 look-ups in the B-tree, which might of course involve several disk accesses. We search for the terms with medium frequency before we the high frequency terms. It is thus more likely that we will have several disk accesses in the B-tree when searching for terms with medium frequency. When we start searching for the terms with high frequency, it seems likely that significant parts of the B-tree in the document manager are already buffered.

2. **Unnecessary disk writes:** During the development of Brille, a lot of debugging information has been printed. Unfortunately, there was one single form of debugging output we forgot to remove before running the experiments, namely a line printing the term we initiate a search for. We did not notice this glitch during the first experiments, and when we noticed it, the time-frame of this project did not allow us to rerun all experiments. Because the log from an experiment is stored in a file, these prints will initiate disk accesses, and this may obviously affect the result. This is of course very unfortunate. Instead of removing the statement when we discovered it, we chose to run all experiments with it to avoid discriminating between the various structures.

3. **Disk writes from recent merge not processed when the merge phase ends:** A merge phase ends when all the buffers containing parts of the merged index are unpinned in the buffer pool. This does not imply that they have been written to disk however. We have assumed that the searches are disk bound. Because the small tests of search speed initiate immediately after the construction is done, their performance will suffer if several disk writes are processed after the construction phase is done.

We attempt to analyze to which extent the possible causes mentioned above have any effect on the result by looking at the logs in appendixes B.1 and B.2. In the log for off-line construction, we notice that there are no disk writes while searching for the terms with low frequency. This implies that the two last possible effects mentioned above are probably not dominant here. We assumed in the estimates that there would never be more than 2 disk accesses in each actual search. There are 548 completed reads while searching for the terms with low frequency, in addition to 73 merged reads. The most likely explanation for this high number is that the searches in the B-tree in the document manager require several disk accesses. The number of disk accesses in this phase is quite similar in the log for remerge with immediate merge given in Appendix B.2.

For terms with medium frequency, there are also significantly more disk accesses than we assumed in both configurations, typically around 1100. This is far more than what should be required to perform the actual searches. The reads are thus probably from the B-tree in the document manager, and these reads seem to be the reason why our actual results are significantly slower than expected when searching for terms with medium frequency. We also note that in this phase, there are some write operations as well. Some of these are probably caused by our debug output. The fact that there are significantly more writes in immediate merge than in off-line construction in this phase is not likely to be caused by our debug output however.

This might support our theory that it is the previous merge that causes the searches in the index constructed with immediate merges to be slightly slower. That this supposedly happens when using immediate merge, and not when using off-line construction should imply that there is typically a higher fraction of dirty buffers in the buffer pool after the last merge in remerge with immediate merge than with off-line construction. We do not have specific results supporting such a theory.

In the phase searching for frequent terms, the disk characteristics of both methods are quite similar, and there is a significant number of reads in both methods. As mentioned above, it is unlikely that this phase should include a lot of disk accesses in the B-tree. It is thus not straight-forward to explain the high number of reads. It is possible that we are not able to read the complete inverted lists in one read operation, but we can not draw this conclusion on such a limited empirical basis.

We thus conclude that it is likely that the accesses to the B-tree in the document manager is the most important source of the differences between expected and actual performance. The unfortunate debugging information seems to have limited effect. We are not able to reach a conclusion about the reason why indexes constructed by remerge with immediate merge seem to process searches slightly slower than indexes constructed with off-line construction, but the cause suggested above is at least a possibility.

7.2.5 Discussion of results

The presentation of the results from experiments with remerge given above, has explained most of the unexpected results. There is however, one aspect we believe deserve some more attention. That is the large difference in construction speed between using small and large buffers when constructing an index for 10 million documents using remerge with immediate merge.

It is not straight-forward analyze the causes of this behaviour. While remerge with immediate merge spends approximately 45.5 hours constructing an index with 10 million documents when using large buffers, it more than 151 hours when using small buffers. By switching to large buffers, we are thus able to construct the index 3 times as fast. Even if the other results suggest that it is slightly more efficient to merge when using large buffers, there is definitely another cause of this behaviour. We should find the cause to be able to avoid such problems in future implementations. To do so, we will look at some figures. Figure 7.8 shows plots of when documents are parsed, and searchable for remerge with immediate merge and $B = 4$KB. Figure 7.9 shows the same plot when using $B = 16$KB.

It is obvious that the curves in Figure 7.9 grows much more steadily than the curves in Figure 7.8. This observation is confirmed by analyzing the logs of these two experiments. The major cause of the differences is that some merges are extremely slow when using small buffers, while others are reasonably fast. We need to figure out why we experience such behaviour when using small buffers.

The slow merges in the experiment with small buffers typically have at least 20 times the number

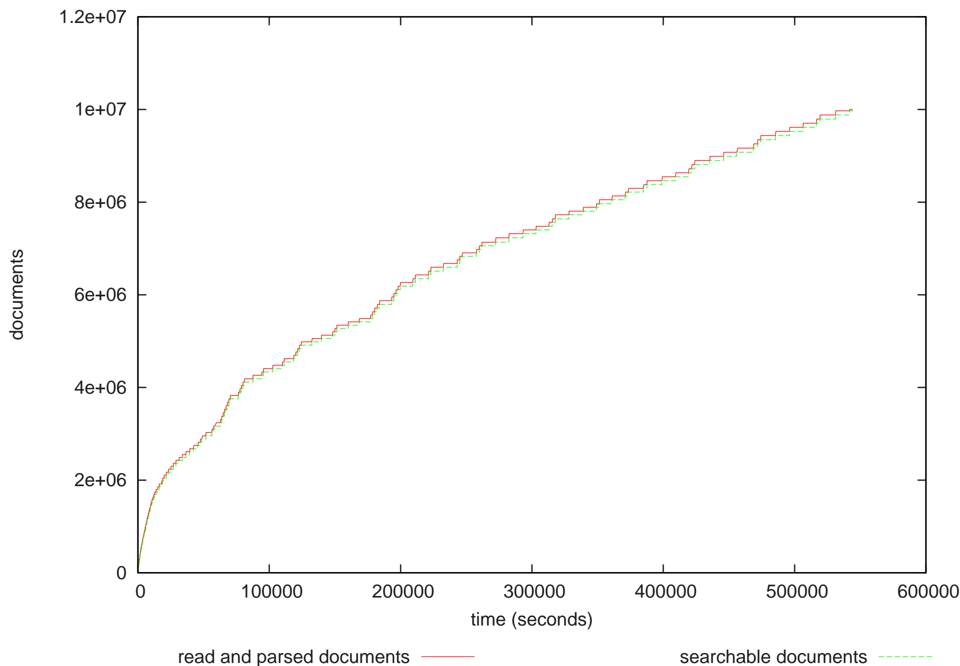

Figure 7.8: Read and searchable documents over time using remerge with immediate merge, 4kB buffers and 10000000 documents

of disk accesses we find in reasonably fast merges with comparable size. It is not straightforward to understand why just by looking at the logs. While this experiment was running, we made an interesting observation however. In a typical run, our application uses between 65% and 75% of the available memory on the experiment computer. During slow merges, this number suddenly increased to more than 96%. This initiated the swap process. The swap process typically used between 20 and 50 percent of the user time on the CPU in such phases. How long the swap process kept running varied, but during this experiment, it had been running for more than 6 hours in total according to the tool *top*.

It is of course surprising that we should require any swapping while running our application. As explained in Section 4.2.1, we have tried to have a conscious approach to memory consumption in the implementation. We thus need to figure out where this memory is used. The only parts of our application that do not use constant memory are the document manager that stores the document length of all documents, and the memory resident part of the dictionary. Storing these values in memory is not a problem in any other configurations, and we thus do not believe that it causes problems for this configuration either.

It is certainly not obvious what causes this excessive use of memory, but we have a theory. As mentioned in Section 4.2.7, we never force a *FileChannel* to flush its changed values to disk. It is possible that the rate at which the small buffers are written to it exceeds its capacity to write the changes to disk. This will force it to store many changes in memory. If it does not force the thread writing buffers to it to wait before swapping is required, we might end up in a hopeless situation. The swapping process will obviously have to choose some parts of

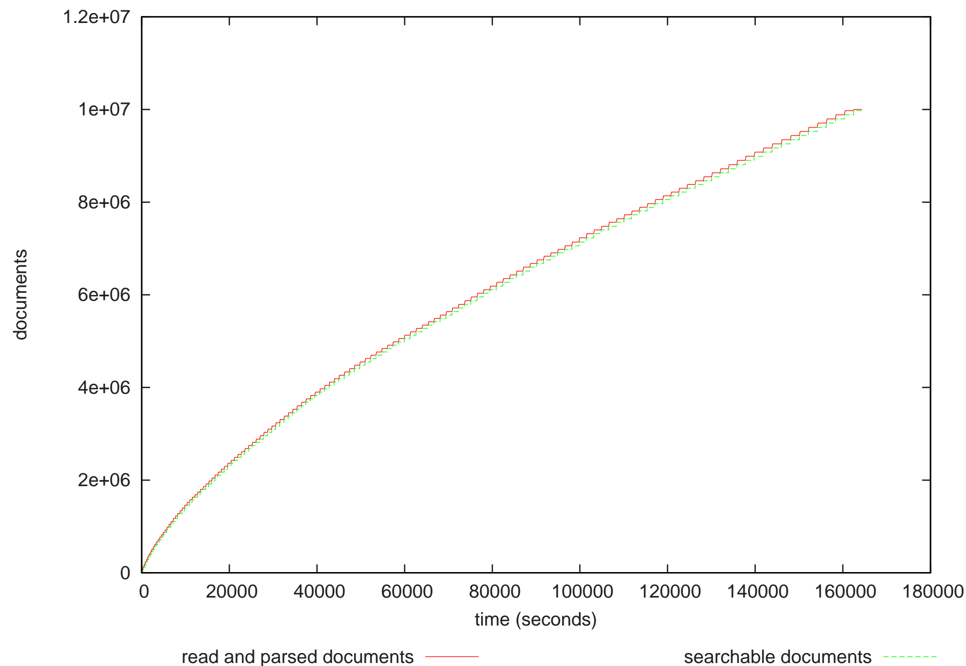

Figure 7.9: Read and searchable documents over time using remerge with immediate merge, 16kB buffers and 10000000 documents

main memory to move to disk. If some of the values stored as changes in the *FileChannel* are chosen, the *FileChannel* will have to read them in before they are written to disk again. This will actually triple the number of disk operations, and the amount of data moved between disk and memory. Even if this explanation might seem far fetched, it is consistent with the disk operations performed at the experiment computer during very slow merges. After the swap process has been running for a while, the merged index files suddenly starts growing again, and the merge continues with a far more decent speed.

If the theory presented above is correct, it was obviously naive to never force the changes in *FileChannel*s to be written out to disk. Even if the theory is wrong, it is definitely a good idea to force the changes to disk from time to time, because it makes it simpler to estimate the occurrence of disk accesses in our implementation. This will obviously be a future improvement of our system.

7.2.6 Chosen configuration

This section aims to choose the configuration for remerge which we believe performs best overall. This configuration will be compared with representatives from the other methods in Section 7.5.

From the presentation of results given above, it is obvious that off-line construction is faster than remerge with immediate merge when it comes to update speed. The indexed documents

are unfortunately not searchable until all documents have been added to the index however, as opposed to when using immediate merge.

This choice between off-line construction and remerge with immediate merge is essentially a choice between update speed and update latency. Because the difference with respect to update speed is relatively small compared to the difference with respect to update latency, we prefer immediate merge. We will however also include the results from off-line construction in the comparison in Section 7.5, because it provides a reasonable baseline for update speed. We choose to use large buffers in both configurations, because this seems to give better update speed and search performance in both cases.

7.3 Hierarchical indexes

This section presents the results from the experiments with hierarchical indexes. Its structure is similar to the previous section.

We were unfortunately not able to run all the planned experiments with the hierarchical indexes. While all configurations worked well for the smaller index sizes, there were problems when indexing 10 million documents. The errors were similar in all the configurations we tested, and the biggest problem was that we were not able to find any clues telling us why the error occurred. We will now explain how the application behaved when the error occurred.

In the configurations we tested, the problem typically occurred after approximately 4.17 million documents had been added to the index. When the problem occurred, our application would never use more CPU. It was thus idle, and eventually became a *defunct* process.

We have tried to investigate this problem quite thoroughly. We expected to find some sort of error message in our system, but we were unfortunately not able to find any. We tried to catch both *Exception* and *Error* around the *run*-method in all running threads, but nothing was thrown. We also tried to evaluate whether the application used a lot of resources when the problem occurred, to consider whether the operating system forced it to become idle because it used too much resources. When the problem occurred, our application used approximately 61% of the memory available on our experiment computer. The user time on the CPU was also reasonably low. It therefore seems unlikely that the operating system would shut the application down. If it did, we should probably have gotten an *Error* in the master thread.

Such a problem can of course have several causes. The most obvious ones are listed below:

- A bug in our implementation
- A bug in the Java virtual machine
- The operating system forces the application to become idle

The most likely cause is of course a bug in our implementation. The only kind of bug we can

think of that would cause our implementation to stop without giving any error messages is a synchronization issue. This does however not fit in well with the fact that the problem occurs at almost exactly the same time in each run. In addition, most of the code run when constructing a hierarchical index is also run in remerge. All configurations run without problems in remerge. If the problem is caused by a bug that is not associated with synchronization, an *Exception* or *Error* should have been thrown in one of the running threads.

We concluded above that it was not likely that the operating system forces the process to become idle and *defunct* because of excessive use of resources. We can not find any other explanation for why the operating system should not allow our process to run.

To evaluate whether this problem is caused by a bug in the *Java virtual machine (jvm)* is not straight-forward. The simplest approach is to test a newer version of the *jvm*, and see if the application works then. We tried with version 1.6, but experienced the same problem.

In conclusion, the most likely cause of this behaviour is a bug in our implementation. It is not straight-forward to come up with what part of our implementation that could cause the process to become *defunct* however. It seems likely that it is some sort of synchronization issue, but this does not fit well with the fact that it happens at almost exactly the same time in each run, as mentioned above. If this is not a synchronization issue, we believe that it is a bug that no sort of error message is thrown in the *jvm*. We believe that it is not very fruitful to speculate more on the possible cause of this error however. Regardless of what causes the problem, the end result is that we are unfortunately unable to run the experiments with the largest index size for hierarchical indexes. We will have to defer a fix for this problem to future work.

7.3.1 Update speed

To test update speed, we construct indexes for document collections with various number of documents. The time spent constructing an index tells us at which rate we can insert new documents into the index.

For hierarchical indexes, we consider the effect of changing several variables in our experiments. We test with the same buffer sizes as for remerge, in addition to varying K and T. This gives us 8 different configurations. To make the presentation as clear as possible, we have partitioned the configurations into two groups based on the buffer size used. Figure 7.10 shows the average construction times for all configurations when using $B = 16$KB, while Figure 7.11 shows the results when using $B = 4$KB. Table 7.10 repeats all the results in a format where it is easier to read the exact values.

From figures 7.10 and 7.11, it is clear that regardless of the buffer size, the variable with the most significant effect on performance is T. Setting T to 4 instead of 1 is more efficient in all tests. This is an expected result, because allowing more small indexes ensures that the thread adding documents to the index does not have to wait as often for merges.

It is not as clear which choice of K that gives the most efficient construction. For the case

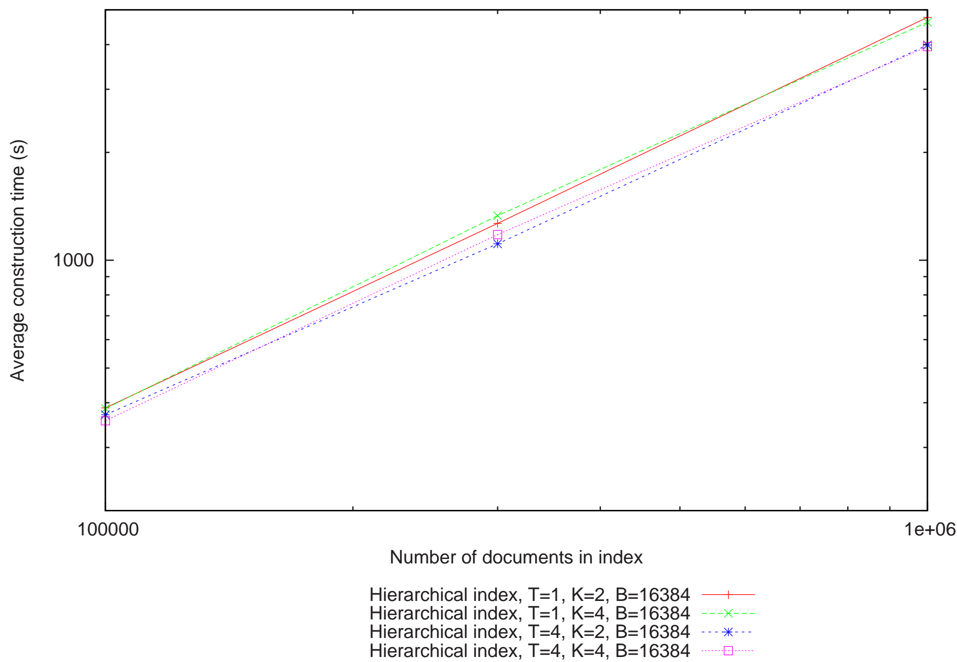

Figure 7.10: Average construction times for hierarchical indexes with various sizes($B = 16$KB)

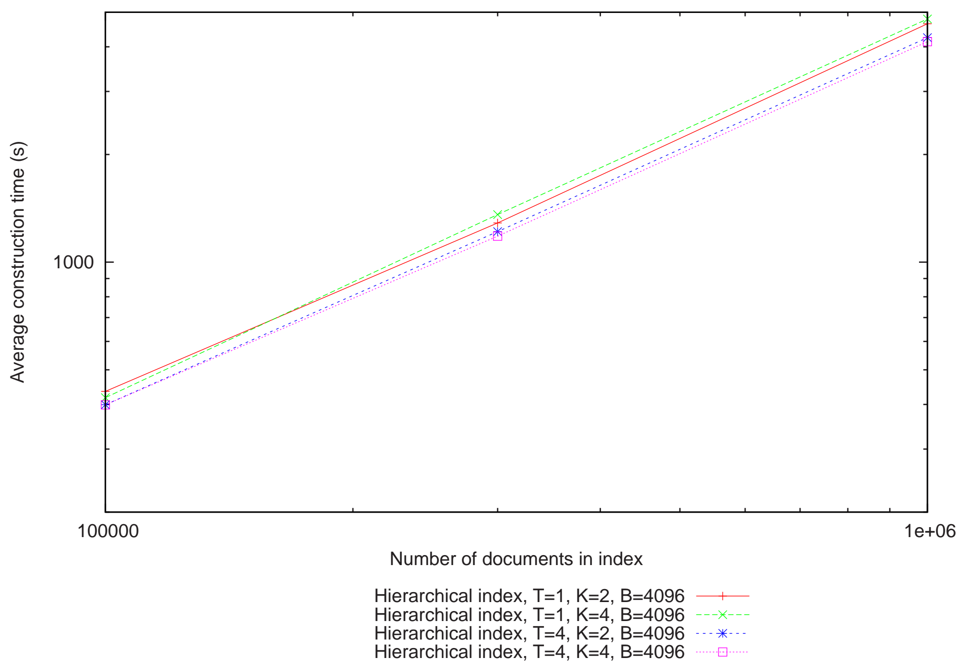

Figure 7.11: Average construction times for hierarchical indexes with various sizes($B = 4$KB)

where $N = 100000$ documents, using $K = 2$ or $K = 4$ is basically equivalent, because there are just 2 batches in this case. For $N = 300000$, $K = 2$ is the most efficient configuration in three cases, while for $N = 1$ million, $K = 4$ is the most efficient configuration in three cases.

Documents	$B = 16$KB				$B = 4$KB			
	$T = 1$		$T = 4$		$T = 1$		$T = 4$	
	$K = 2$	$K = 4$	$K = 2$	$K = 4$	$K = 2$	$K = 4$	$K = 2$	$K = 4$
100000.0	6m 27s	6m 25s	6m 10s	5m 56s	7m 15s	6m 58s	6m 39s	6m 39s
300000.0	21m 7s	22m 12s	18m 31s	19m 39s	21m 27s	22m 38s	20m 17s	19m 42s
1000000.0	79m 26s	77m 1s	66m 30s	65m 48s	77m 20s	79m 47s	70m 44s	68m 57s

Table 7.10: Average time spent constructing hierarchical indexes

The differences are not significant in any of these cases. It is a surprise however, that $K = 2$ is not more efficient than $K = 4$. We will try to explain why we experience this behaviour in Section 7.3.5, by considering the case where $T = 4$ and $N = 1$ million documents. This case is chosen because our results suggests that using $K = 4$ gives better performance regardless of the buffer size used.

Our results also suggest that it is marginally more efficient to use large buffers than small ones, especially when we use $T = 4$.

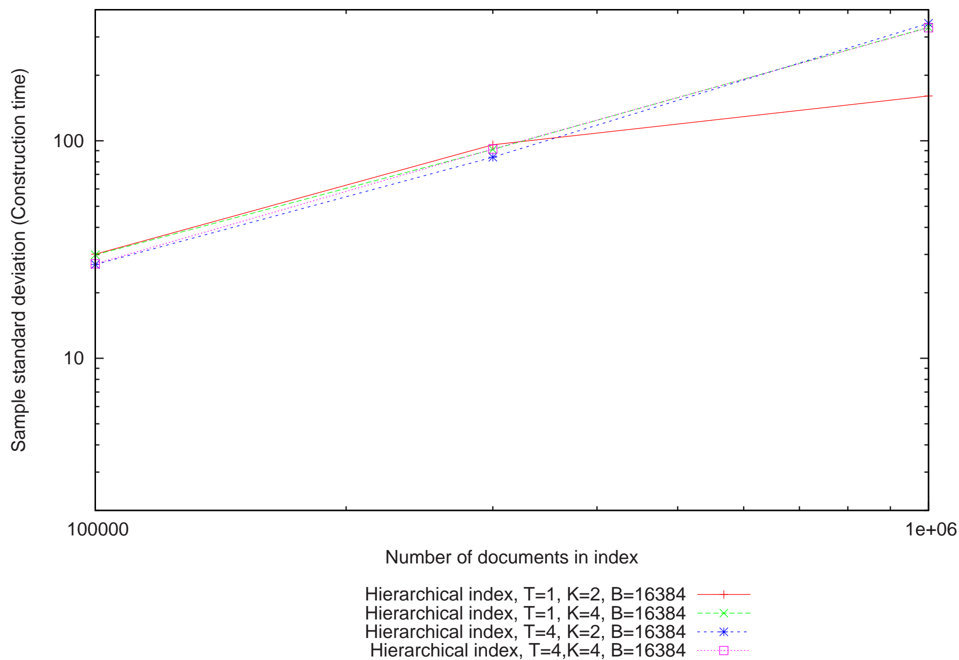

Figure 7.12: Sample standard deviation for construction of hierarchical indexes with various sizes ($B = 16$KB)

To get an impression of the variability in the experiments, Figure 7.12 shows a plot of the sample standard deviation for the configurations with large buffers, while Figure 7.13 shows a similar plot for configurations with small buffers. The sample standard deviation is calculated as shown in Equation 7.1.

We note from the figures that most configurations have similar sample standard deviation, but

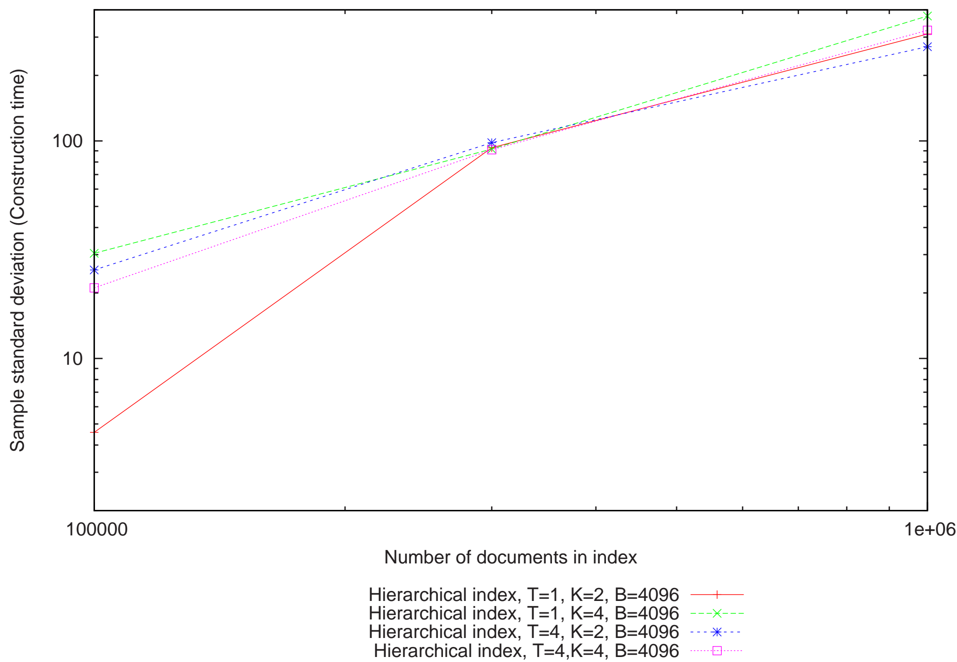

Figure 7.13: Sample standard deviation for construction of hierarchical indexes with various sizes ($B = 4$KB)

that there are two outliers. When running the experiment with $N = 1$ million documents, the configuration with $B = 16$KB, $K = 2$ and $T = 1$ has a sample standard deviation of only 160, while all other configurations in this case have values between approximately 270 and 370. The same configuration with small buffers has a very small sample standard deviation with $N = 100000$ documents as well.

We see no particular reason why the standard deviation should be as small as our measurements suggest in these particular settings. In general however, it is not unexpected that the configurations with $T = 1$ have less variability than when $T = 4$. When $T = 4$, there will more often be several threads running. How these threads are scheduled may have a significant impact on the time it takes to finish construction. When $T = 1$, the thread constructing partial indexes will wait until the merge is done each time a new partial index is added. We will thus never construct partial indexes while we are merging, and the time spent is thus expected to be less dependent on the scheduling. Following this theory however, we should have experienced low variability for all cases where $T = 1$, including when $K = 4$. This is obviously not the case. Even if we can not fully explain the behaviour at this point, we choose not to go into a more thorough analysis.

7.3.2 Update latency

As for remerge, the update latency in various configurations for hierarchical indexes can be visualized by plotting two graphs. One of the graphs shows when the documents are read, while

the other describes when they are searchable. Figure 7.14 shows such plots for a hierarchical index with $B = 16384$, $K = 2$ and $T = 1$.

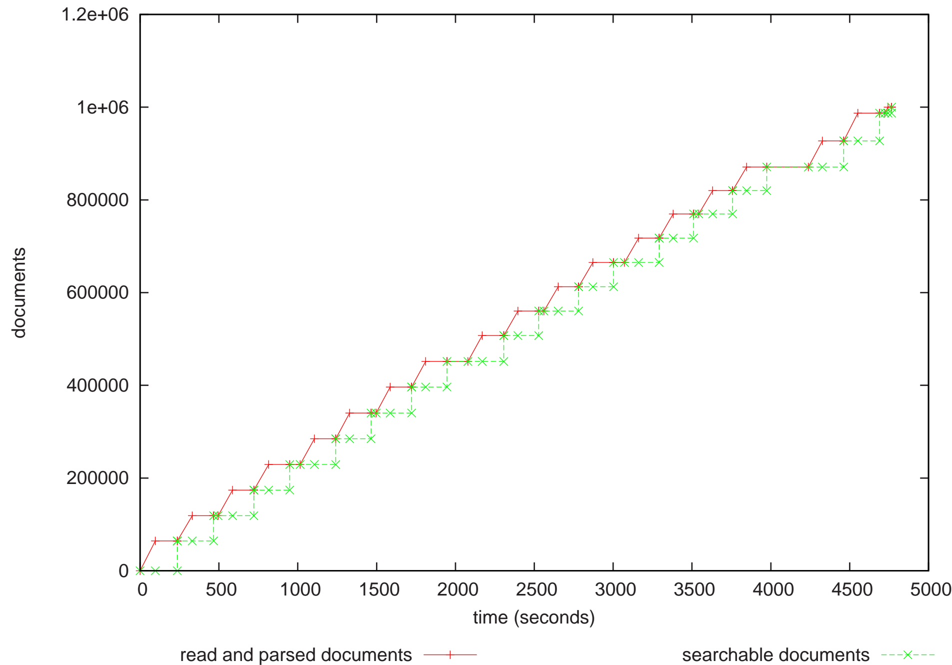

Figure 7.14: Read and searchable documents over time using hierarchical index, $K = 2$, $T = 1$, 16kB buffers and 1 million documents

The plot in figure 7.14, is not unlike the plot for remerge with immediate merge given in Figure 7.4. There is one significant difference however, namely that while remerge with immediate merge will have to wait until a merge is done before the documents are searchable, the documents in a batch are made searchable before the merge in the hierarchical index. When the partial index is constructed, we perform look-ups in the largest dictionary in the hierarchy to calculate reasonably accurate document lengths after the tf-idf ranking scheme. When this process is done, the partial index is searchable. The fact that we do not need to wait for a merge to make documents searchable makes it is much simpler to determine the worst-case update latency for this method than for remerge with immediate merge. The worst-case is the actual case for the first document read in a new batch. This makes the worst-case equal to the amount of time spent reading and parsing all documents in a batch, sorting and writing out the partial index, and updating the document lengths. The time spent in the last phase is dependent on the size of the largest dictionary in the hierarchy however. The worst-case update latency is thus unfortunately not constant, but it is less dependent on the total size of the index than remerge with immediate merge.

The update latency for hierarchical indexes is relatively similar for all configurations. Figure 7.15 shows an example of a construction process when $B = 4$KB, $K = 2$ and $T = 4$. The only difference from Figure 7.14, apart from the fact that $N = 300000$ instead of 1 million, is that the process constructing partial indexes never waits when $T = 4$. The update latency is comparable. Figures 7.16 and 7.17 shows similar plots for construction when $K = 4$. The

conclusion of this section is that the update latency for hierarchical indexes is similar for all configurations and on average lower than for remerge with immediate merge.

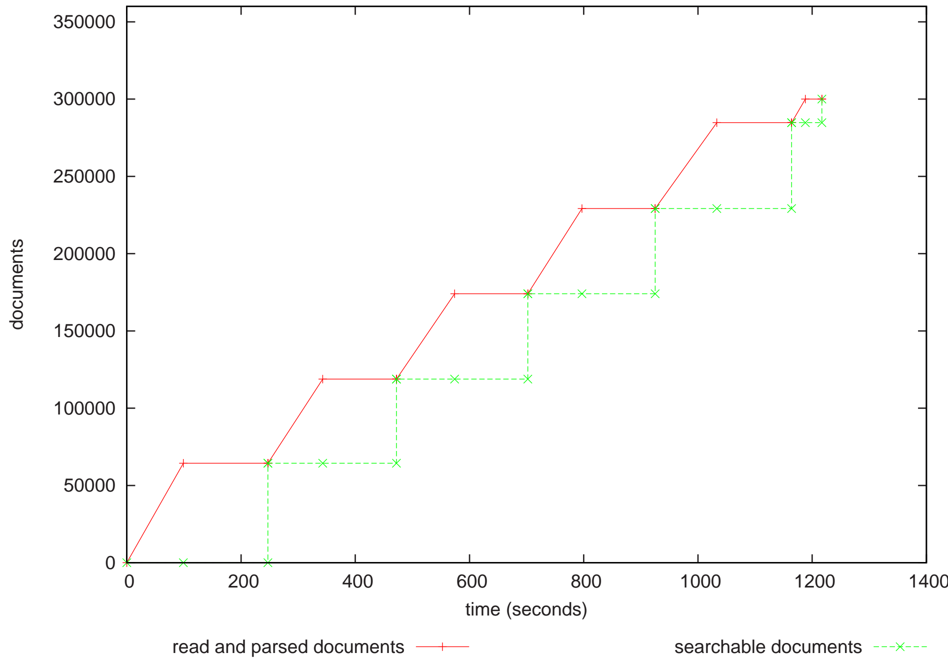

Figure 7.15: Read and searchable documents over time using hierarchical index, $K = 2$, $T = 4$, 4kB buffers and 300000 documents

7.3.3 Search performance

This section will present the results from testing the search performance in various configurations for hierarchical indexes. As explained in Section 5.3.2, we test search performance by measuring the time spent performing 100 searches for terms with various frequencies. The results from these experiments for hierarchical indexes are given in figures 7.18, 7.19 and 7.20. These figures show the average time spent searching for 100 terms with low, medium and high frequency, respectively.

Figure 7.18 shows that the time spent searching for terms with low frequency is quite similar for all configurations, except when $T = 4$ and $K = 4$. This configuration is actually significantly slower than the others for both buffer sizes. The configuration with $K = 2$ and $T = 4$ is also slightly slower than the others in the index with 100000 documents, but for larger indexes, it is similar to the configurations where $T = 1$.

When using large buffers in the hierarchical indexes, we observe the same tendency in searches for terms with medium frequency as we did when searching for terms with low frequency. The difference in time spent performing the search between the index with 300000 documents and the one with 1 million documents is relatively small however. For the small buffer size, it is actually faster to perform the searches in the largest rather than the medium sized index for the

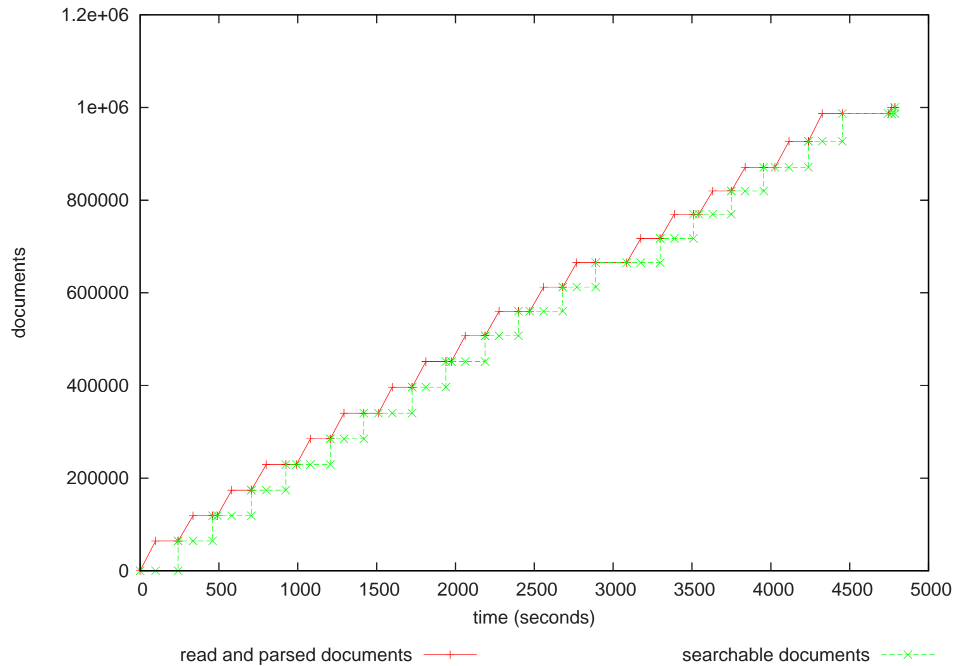

Figure 7.16: Read and searchable documents over time using hierarchical index, $K = 4$, $T = 1$, 4kB buffers and 1 million documents

configuration where $K = 4$ and $T = 1$ and especially for the one where $K = 2$ and $T = 4$. This observation might indicate that the number of occurrences of the terms with medium frequency does not change much between the two index sizes. We believe that the explanation of this behaviour is slightly more complicated however. We will provide an explanation of this behaviour in Section 7.3.5, together with an explanation of why the configuration with $T = 4$ and $K = 4$ is slower than the other configurations.

The general tendencies in Figure 7.20 are very similar to the results from searching for terms with low frequency in Figure 7.18. Again, the configuration with $K = 4$ and $T = 4$ is significantly slower than the others.

7.3.4 Actual results versus experiments

This section will analyze to what extent our estimates for hierarchical indexes are accurate. We will consider the update speed first, before search performance is analyzed in Section 7.3.4.2.

7.3.4.1 Update speed

In Section 6.5, we estimated the time spent constructing a hierarchical index with $N = 1$ million documents when $T = 1$ and $K = 2$. The final estimate differs based on which method we use for estimating the time spent accessing disk, and the results are summarized in Table 7.11.

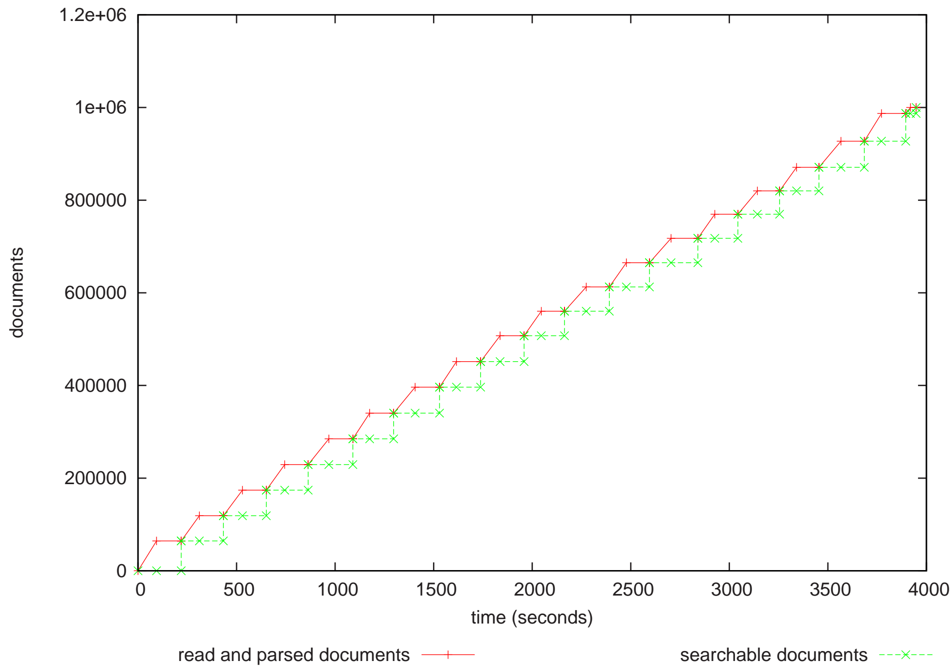

Figure 7.17: Read and searchable documents over time using hierarchical index, $K = 4$, $T = 4$, 16kB buffers and 1 million documents

Estimate 1		Estimate 2
$B = 4$KB	$B = 16$KB	
32h 29m 51s	8h 58m 56s	1h 19m 48s

Table 7.11: Final estimates for time spent constructing a hierarchical index with $N = 1$ million documents

The actual results are found in Table 7.10. When using $B = 16$KB, we on average spend 1 hour, 19 minutes and 26 seconds constructing a hierarchical index with $K = 2$ and $T = 1$. When $B = 4$KB, the average construction time is 1 hours, 17 minutes and 20 seconds.

It is again clear that estimate 2 is much closer than estimate 1 to the actual obtained result. Even if estimate 2 seems to be very accurate, we have included a log from an experiment with a hierarchical index in Appendix B.3. We will go through each of the phases in the log, and compare the time spent with our estimates.

Both the phase accumulating the index in memory and the phase sorting entries and flushing a partial index seem to perform as expected in all phases. By looking at the CPU utilization, it is obvious that both phases are CPU bound. According to estimate 2, the phase making a partial index searchable is also CPU bound. The phase was estimated to spend 10.98 seconds for the first batch, 3.99 seconds for the last batch, and 15.24 for all the other batches. According to the log, these estimates are more or less correct, even though the exact time varies slightly. The actual time spent is typically higher when the largest index in the hierarchy is just constructed. In such cases, we also note that the time spent waiting for I/O is typically noticeable. Even so,

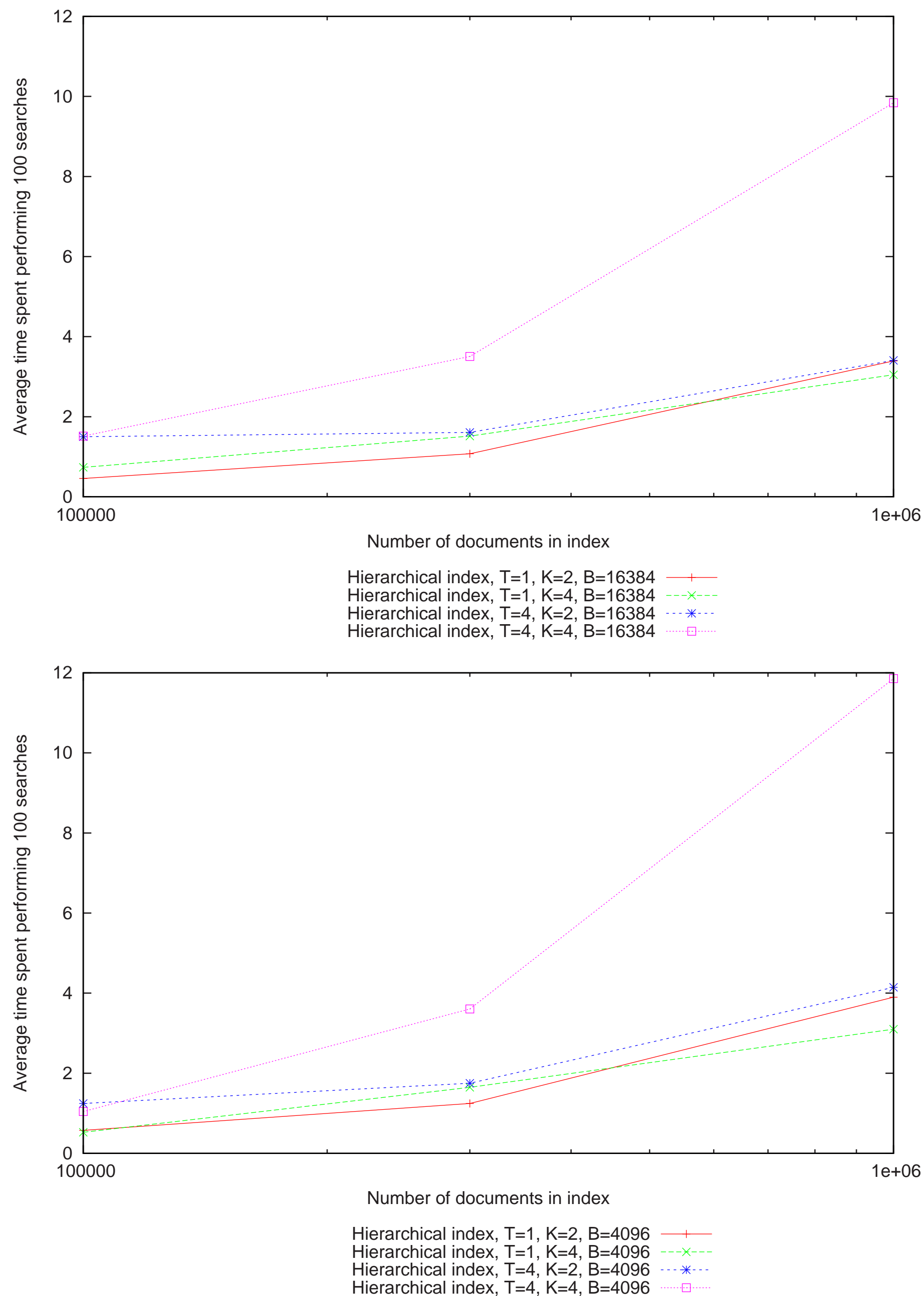

Figure 7.18: Average time spent performing 100 searches for terms with low frequency in hierarchical indexes. The figure on top is for $B = 16$KB, and the lower figure is for $B = 4$KB

both the expected time and the assumption that this phase is CPU bound seem reasonable when we analyze the log.

The total time spent constructing partial indexes and making them searchable is thus as expected. The only phases we have not analyzed yet are the merges. From the log, we see that the time spent during the merges is slightly lower than expected. This observation is similar to

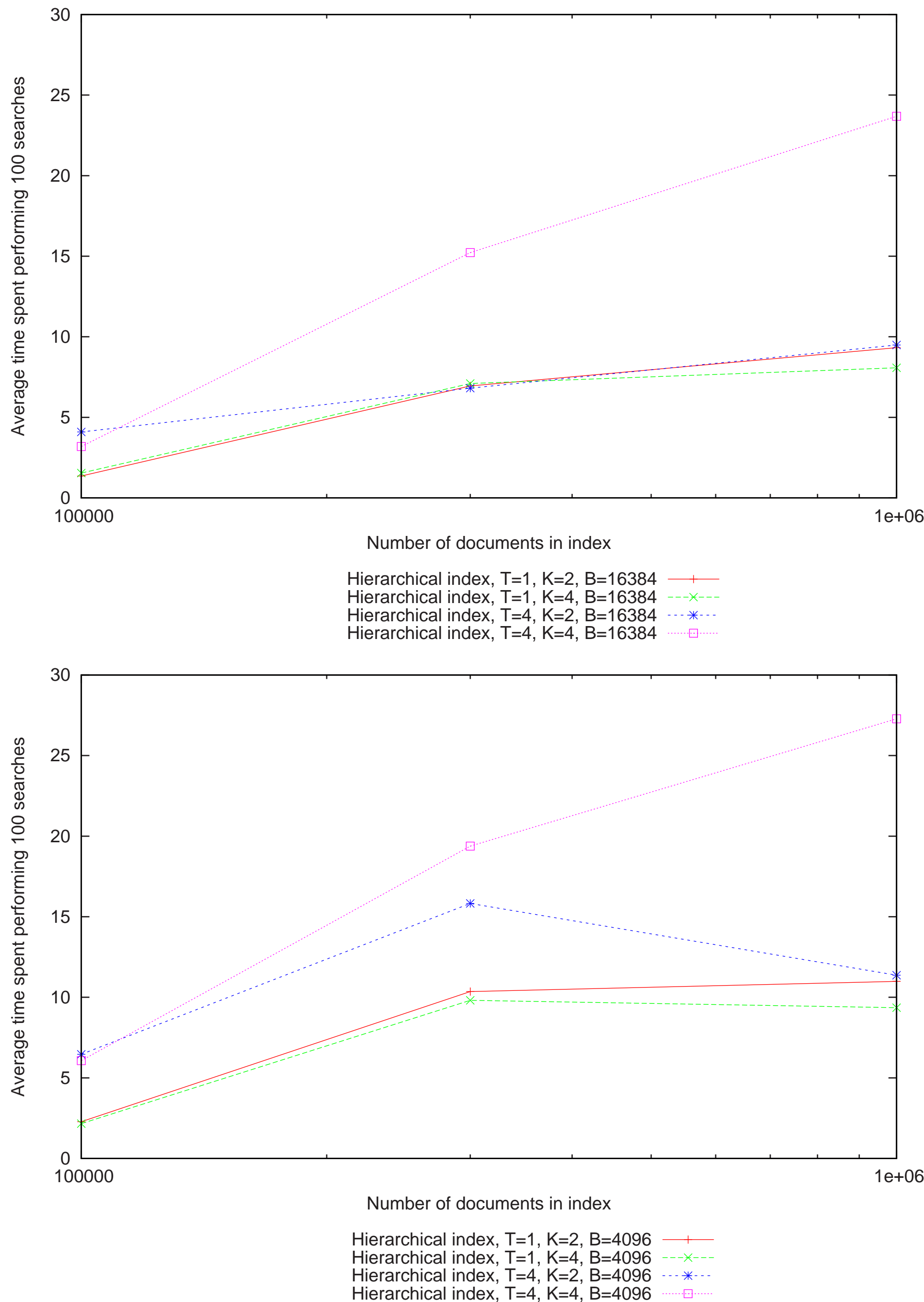

Figure 7.19: Average time spent performing 100 searches for terms with medium frequency in hierarchical indexes. The figure on top is for $B = 16$KB, and the lower figure is for $B = 4$KB

what we noticed for remerge in Section 7.2.4. We observed that the merges in remerge were typically I/O bound as opposed to what estimate 2 assumes. We do not measure disk and CPU utilization in the merges in hierarchical indexes, because when $T = 4$, the merges will go on in parallel with construction of partial indexes. We have no reason to believe however, that the merges should not be I/O bound here as well.

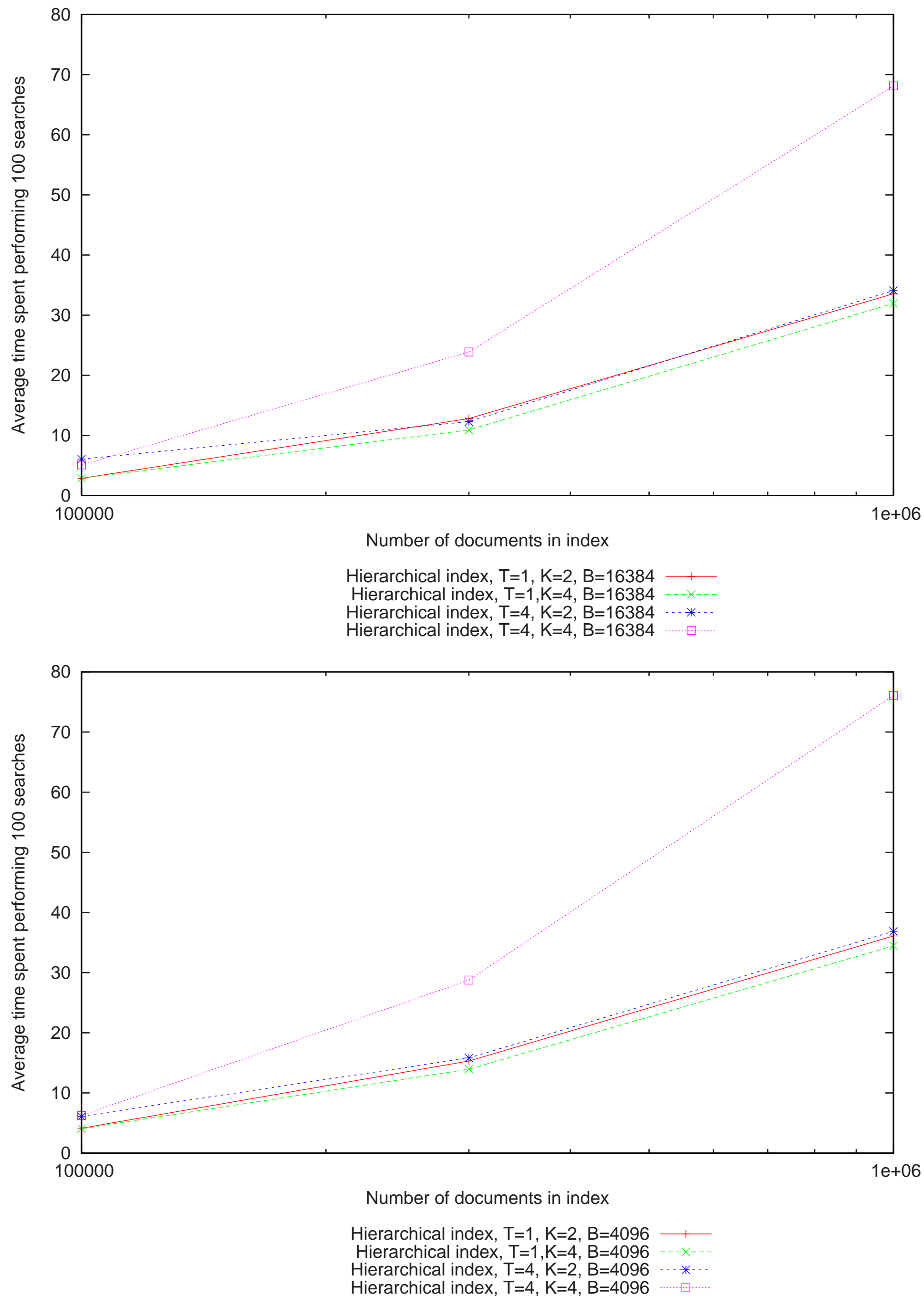

Figure 7.20: Average time spent performing 100 searches for terms with high frequency in hierarchical indexes. The figure on top is for $B = 16$KB, and the lower figure is for $B = 4$KB

We thus conclude that most phases of construction of hierarchical indexes perform as expected, although the merges are marginally faster than expected. We experienced the same behaviour for remerge, and we have analyzed the reasons for this behaviour in Section 7.2.4. It is however slightly unexpected that the configuration with small buffers on average performs better than the configuration with large buffers. In Section 7.3.1, we noticed that the configuration with $B = 16$KB had significantly lower variance than the others in this particular case. By analyzing

the results from each run, we notice that when $B = 4$KB, half the runs are actually slower than the average for $B = 16$KB. Some of the runs with small buffers are very fast however, making the average for this configuration lower than with larger buffers. The fastest runs are typically very efficient in the phase where the entries are sorted and written to disk. We have concluded that this phase is typically CPU bound. The batches contain slightly fewer documents when using small buffers, because of more fragmentation. When the buffers are smaller, we more often waste a few bytes because we can not fit an integer at the end of a buffer if there is less than 4 bytes left. With fewer documents, the sorting is typically more efficient and this may cause the improved efficiency. Even though the differences between the batch sizes here are relatively small, this aspect is generally important. Because of the non-linear complexity of the sorting process, the time spent in this phase is expected to be less than 50% if we reduce the size of the batches by 50%. We will use this observation when we consider possible future improvements of our implementation in Section 7.7.

7.3.4.2 Search performance

This section will compare the estimated search performance for hierarchical indexes given in Section 6.7.2 with the obtained results presented above. The estimates are repeated in Table 7.12, while Table 7.13 shows the actual results. We originally measured the average time spent performing 100 searches for terms with different frequencies. This average is divided by 100 to obtain the average time spent performing a search for a single term given in Table 7.13.

Frequency	$B = 4$KB	$B = 16$KB
Low	0.052 s	0.053 s
Medium	0.079 s	0.079 s
High	0.274 s	0.274 s

Table 7.12: Expected time spent searching for terms with different frequencies in a hierarchical index with $K = 2$ and $T = 1$

Frequency	$B = 4$KB	$B = 16$KB
Low	0.039 s	0.033 s
Medium	0.110 s	0.093 s
High	0.335 s	0.361 s

Table 7.13: Average time spent searching for terms with different frequencies in a hierarchical index with $K = 2$ and $T = 1$

Tables 7.12 and 7.13 show that for terms with medium and high frequency, the actual results are slower than we expected. We noticed the same tendency for remerge in Section 7.2.4.3. We mentioned three possible causes for this behaviour, and concluded that the disk accesses required to read in buffers from the B-tree in the document manager is likely to be the major contributor to the difference. The third possible reason mentioned there is not likely to have any effect for the configuration of the hierarchical index considered here. According to the expected merges we presented in Section 6.5.1, the last batch will not initiate a merge. These expected merges are to a large extent confirmed by Figure 7.14, where we can see that the largest merges

are performed after the 8th and the 16th batch. We thus conclude that the accesses to the B-tree in the document manager is likely to be the most significant cause of the slow search speed, just like for remerge.

For terms with low frequency however, the actual results are faster than our estimates suggested, even though we know that these searches are likely to require disk accesses in the B-tree in the document manager. To find the cause of this behaviour, we analyze the last phase before the searches initiate. This phase makes the small partial index in the 19th batch searchable. This process involves reading the complete partial index for the 19th batch, and looking up all unique terms in the largest dictionary. After this process we may thus expect that the complete partial index from the 19th batch is buffered, in addition to most parts of the largest dictionary.

When searching for terms with only one occurrence, we only have to look up the inverted list in one inverted file. In the others, we stop searching when we have determined that the term is not represented in the index. Because it is likely that both the partial index created by the 19th batch and most parts of the largest dictionary is buffered, we are likely to avoid the two possible disk accesses to these dictionaries. If some of the terms are only found in the 19th batch, we will probably not require a disk access to look up the inverted list either. This may explain why the searches for terms with low frequency are more efficient than expected. This theory is supported by the log included in Appendix B.3. There are less than 400 read operations on the disk containing the index files when searching for terms with low frequency. When we know that there are probably several accesses to the B-tree in the document manager, it is obvious that not all searches in the hierarchy require 4 disk accesses, as we assumed in the estimates. We thus conclude that the reason why the searches for terms with low frequency are faster than expected is probably that the largest dictionary and the smallest partial index are buffered when we start searching.

7.3.5 Discussion of results

We have presented the results from all experiments with hierarchical indexes above. Most of the results are as expected, and some of the unexpected results are analyzed and explained above. There are still a few aspects of the results that are not explained however, and this section will try to explain them. We will focus on the three following observations:

- We expected that using $K = 2$ would give faster construction of hierarchical indexes than with $K = 4$. Table 7.10 suggests that it is the other way around when $T = 4$ and $N = 1$ million documents however.
- When testing search performance, we expected it to be more efficient to use $K = 4$ than $K = 2$. When using $T = 4$, the figures in Section 7.3.3 clearly show that it is more efficient to search when $K = 2$ in our experiments.
- When using $T = 4$ and $K = 2$, the bottom plot in Figure 7.19 shows that it is significantly more efficient to search for the terms with medium frequency in the largest index than in the index with 300000 documents. Even though the time spent is quite similar for the two

index sizes for the other methods, it is unexpected that the time spent decreases for one of the configurations.

We will analyze the causes for the last item above first. By analyzing the log from an experiment with $K = 2$, $T = 4$ and $N = 300000$, we realize that the first three of the expected merges presented in Section 6.5.1 occur in this configuration. There are 6 batches in total, and the last one is quite small. When this index is searchable, a merge of the two smallest indexes will start. This merge finishes just before the searches start. After this merge is completed, most of the buffers in the buffer pool will be used to store the two small indexes that are not searchable any more, and the new index merged from these two. It is unlikely that significant parts of the largest index in the hierarchy or the B-tree in the document manager are buffered. The search performance will obviously suffer, because we will require several disk accesses to read in needed parts of the largest index and the B-tree in the document manager. In addition, we have no guarantee that all the dirty buffers are written out when the phase ends. There might thus be some disk accesses caused by the merge after the merge process has ended. This may also make the search performance slower.

When $N = 1$ million documents however, the last batch in the configuration with $T = 4$ and $K = 2$ will be a small index that is just moved in as the smallest index in the hierarchy. We will thus not have any merge, and the effects mentioned above will not occur in this setting. This explains the fact that it is more efficient to search for terms with medium frequency in the largest index than in the index with medium size.

A similar effect explains why we for all searches see that the configurations with $T = 1$ are more efficient when $N = 100000$. When $T = 1$, we will wait until the merge is done before we start searching, while we will search while processing the merge when $T = 4$.

The first two items above are actually related. The main reason why the construction of the largest hierarchical index with $T = 4$ and $K = 4$ is efficient, is that it does not have to finish the largest merge before the construction phase is completed. Rather than going through each of the merges, we will give a short overview. We keep in mind that the indexes in the hierarchy have maximum sizes of 1, 4, 16 and 64 when $K = 4$. In short, the first 12 batches will be merged to an index with size approximately equal to 12. It is thus placed in the position with maximum size 16. The next 4 batches will fill the index with maximum size 4. We will then add two more; making the total number of batches added 18. This will initiate a large merge of all the indexes currently in the hierarchy. Because $T = 4$ however, we are allowed to construct more small indexes. We thus construct the partial index for the 19th batch and make it searchable. When this process is done, the construction is considered done, even if the large merge is nowhere near finished. We then perform all searches. After all searches are done, the merge eventually finishes.

Based on the explanation given above, it is clear that the configuration with $K = 4$ will have faster construction than with $K = 2$, because it does not need to finish its largest merges during the construction phase. The reason why the search performance in the largest index when $K = 4$ is low, is that there is actually a merge going on while we perform the searches. The merge slows the searches down, because both processes are disk bound.

7.3.6 Chosen configuration

In summary, all the tested configurations for hierarchical indexes have the same update latency. The ones with $T = 4$ are faster at constructing the index, because they never have to wait for merges in our experiments. With larger index sizes however, there will eventually be a merge that will require the process of constructing partial indexes to wait if T is not set to infinity. The search performance typically suffer slightly when using $T = 4$ instead of $T = 1$. This difference in performance is relatively small in our experiments when $K = 2$. It should be noted however, that it is not expected to be generally true that this configuration is efficient at processing searches.

Based on the arguments above, we choose to use the configuration with $B = 16$KB, $K = 2$ and $T = 4$ when we compare the overall approaches in Section 7.5.

7.4 Naive B-tree index

This section presents the results from the experiments performed with the naive B-tree index. As will be clear from the following subsections, we were not able to run all planned experiments for the naive B-tree index. There were no errors, but the structure became very slow when running with $N = 1$ million documents. When testing the first configuration, the experiment had slower and slower progress as the number of indexed documents exceeded 500000. After more than 100 hours, it had passed 800000 documents. The progress was very at this point, and the process was seldom able to use more than 1% user time on the CPU.

Because we have 4 different configurations for the naive B-tree index, we could expect running this experiment for all of them would take almost a month. This was clearly infeasible within the time frame of this project. We therefore chose to not run any more of them. We will however, provide some of the results from the aborted experiment.

This section has a similar structure as the presentations for the other methods in previous sections. We first present results from the experiments, with focus on update speed, update latency and search performance. We then compare the obtained results with our estimates from Chapter 6. At the end of the section, we discuss the results and try to explain the results we find surprising, before we choose which configuration of a naive B-tree index we will compare with the other methods.

7.4.1 Update speed

As for the other methods, we test update speed in the naive B-tree index by constructing indexes with various sizes from scratch, and recording the time spent constructing the index. The average construction time for the different configurations of the naive B-tree index is shown in Figure 7.21. The sample standard deviation, calculated as shown in Equation 7.1, is plotted in

Figure 7.22. To make it easier to read exact values, the average construction times are repeated in Table 7.14.

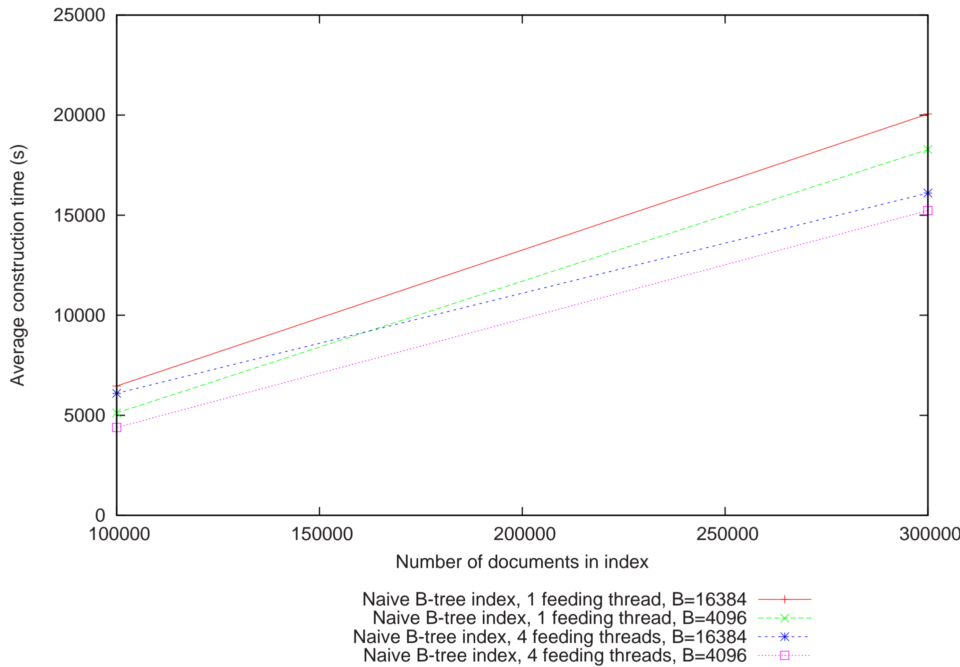

Figure 7.21: Average construction times for naive B-tree index with various index sizes

Documents	**1 feeding thread**		**4 feeding threads**	
	$B = 16$KB	$B = 4$KB	$B = 16$KB	$B = 4$KB
100000.0	1h 47m 41s	1h 25m 19s	1h 41m 38s	1h 13m 10s
300000.0	5h 34m 18s	5h 4m 41s	4h 28m 24s	4h 13m 51s

Table 7.14: Average time spent constructing naive B-tree indexes

There are two main observations from Figure 7.21. It is obvious that it is generally faster to construct an index with 4 feeding threads than with 1, as expected. It also seems to generally be faster to construct a naive B-tree index when $B = 4$KB than when $B = 16$KB. We will discuss possible causes of this behaviour in Section 7.4.5.

From Figure 7.22, we see that the sample standard deviation changes with varying index size. When $N = 100000$, the configurations with $B = 4$KB have lower standard deviation than the ones with $B = 16$KB. When $N = 300000$, it is the other way around. We will not be able to buffer a complete naive B-tree index for 300000 documents. Disk accesses will thus be required when we access nodes that are not currently buffered. Even if we only have one feeding thread, there are two other threads that access the B-tree. The first of these is the thread updating the document lengths of all documents, and the other is the thread that is responsible for updating nodes at higher levels in the B-tree when new entries are inserted. Slight differences in the scheduling may thus cause significant differences in the results. This explains the significant observed variability for the time spent constructing the index when $N = 300000$. At first glance, it does not explain the increased variability when $B = 4$KB however.

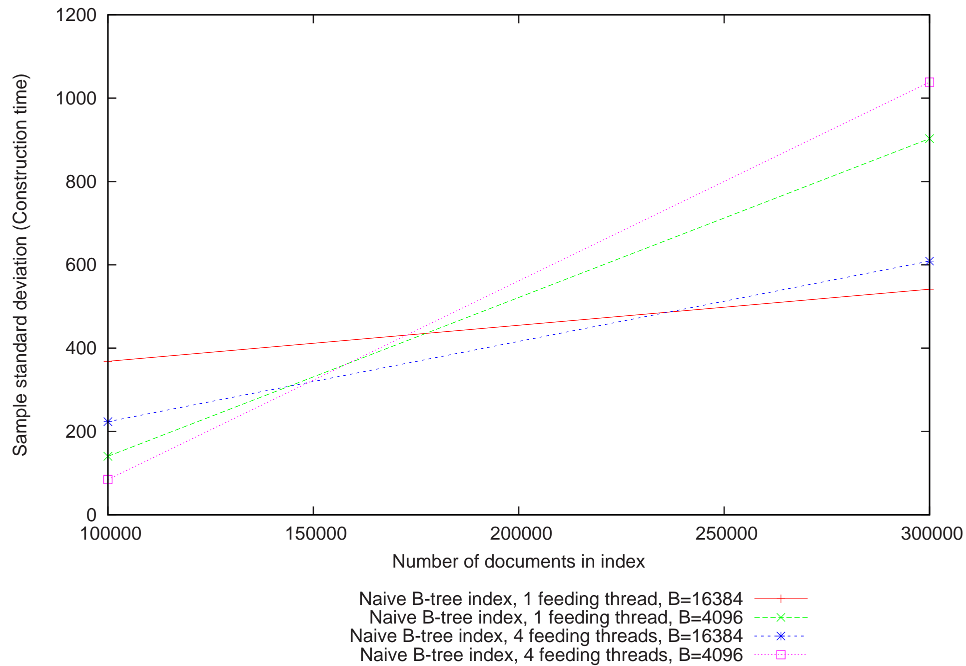

Figure 7.22: Sample standard deviation for construction of a naive B-tree index with various size

The calculations in Section 6.6.5 suggest that the expected construction time for the naive B-tree index changes more with varying hit ratio when the B-tree is high. Even if the resulting index is expected to have $he = 3$ both when using small and large buffers, it is obvious that this height will be reached later in the construction phase when using large buffers. The hit ratio may of course be affected by differences in the scheduling. Because the B-tree is higher on average during construction when using $B = 4$KB, this may explain the increased variability in this case.

The logs from the experiments suggest that the variability is typically introduced in later phases of construction. This supports the theory that differences in hit ratio can cause the variability, because the hit ratio is only relevant when we are not able to buffer the complete index in the buffer pool.

It is not a surprise that the sample standard deviation is higher when using 4 feeding threads than 1. As there are more threads running, it is more likely with differences in the scheduling in different runs.

7.4.2 Update latency

We visualize the update latency for the naive B-tree index as we did for the other tested methods in previous sections. The construction in the naive B-tree index does not go through phases like the other methods however. We therefore need to find another way to measure when documents

are read and when they are searchable. It would clearly generate excessive amounts of output to record when each document is read and searchable. We therefore choose to define 10 phases for each constructed index. We thus record when we have read all of the first 10% of the documents in the collection to index. We also record when the first 10% of the documents are searchable. This process will be repeated for the next 10%, and so on. By assuming that both reading documents and indexing them proceed at steady rates within each phase, we are able to create similar plots as we did for the other methods.

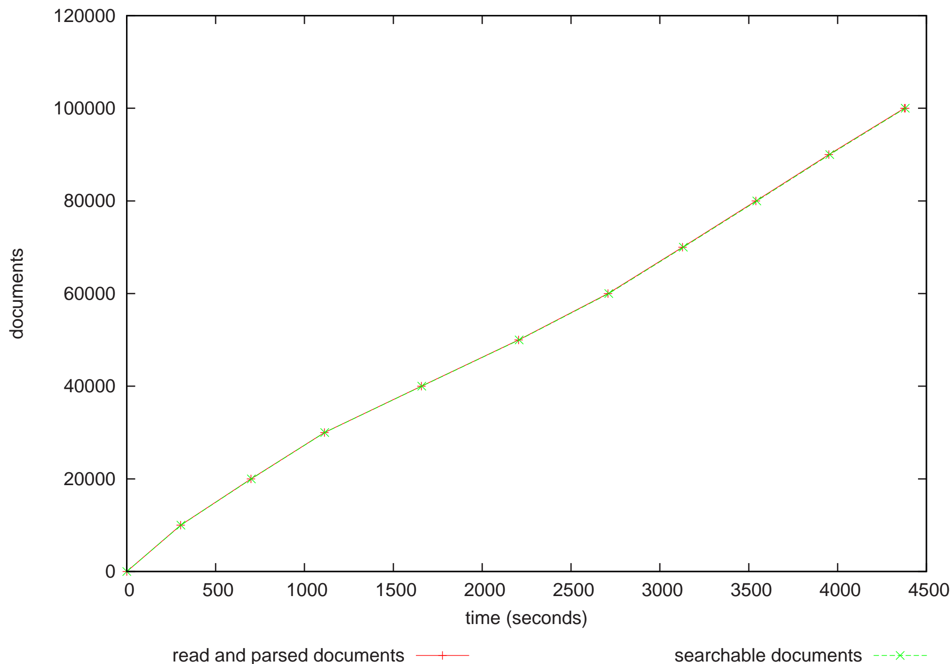

Figure 7.23: Read and searchable documents over time using naive B-tree index with 4 feeding threads, 4kB buffers and 100000 documents

Figure 7.23 shows a plot describing the update latency in a naive B-tree index with $N = 100000$ documents, constructed by 4 feeding threads. The buffer size in this case was 4KB. Figure 7.24 shows a similar plot for an index for $N = 300000$ documents. This index was constructed with 1 feeding thread and has buffers with size 16KB.

These figures clearly show that the update latency is very small in the naive B-tree index, regardless of configuration. It is actually hard to see that there are two different graphs in any of the figures. The rate at which new documents can be indexed is quite modest however, as shown in the previous section.

7.4.3 Search performance

The search performance in naive B-tree indexes is tested by searching for groups of terms with varying frequency, just like for the other tested methods. Figures 7.25, 7.26 and 7.27 show the time spent searching for 100 terms with low, medium and high frequency, respectively.

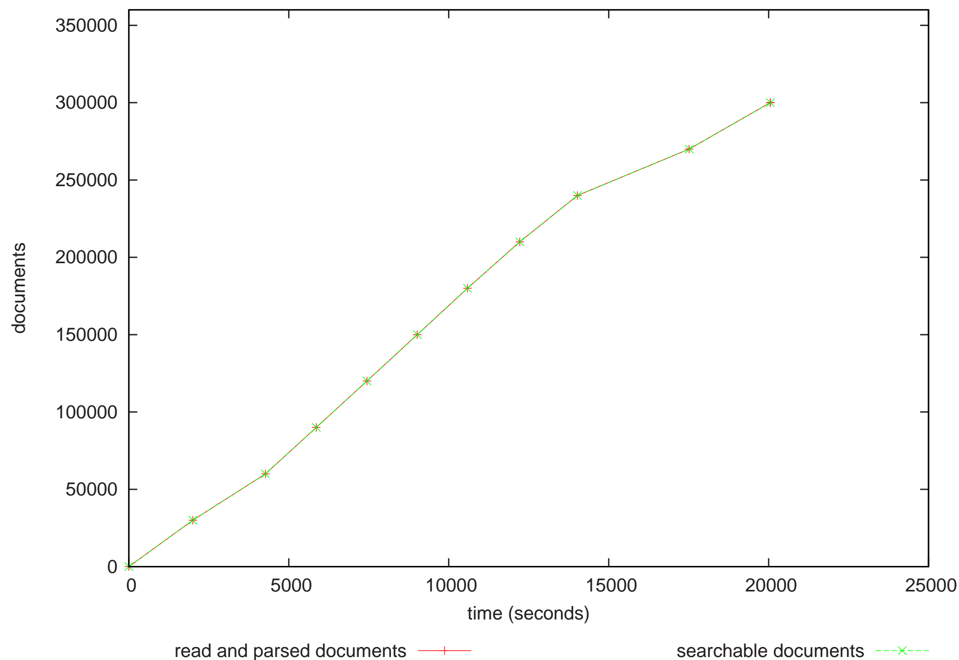

Figure 7.24: Read and searchable documents over time using naive B-tree index with 1 feeding thread, 16kB buffers and 300000 documents

The results from all groups of terms seem to more or less follow the same pattern. Regardless of the frequency of the terms searched for, it is more efficient to use $B = 16$KB than 4KB. The configuration with $B = 4$KB and one feeding thread is slowest in most cases.

It was expected that using $B = 16$KB would be more efficient than $B = 4$KB. We did not expect significant differences between configurations with varying number of feeding threads however. When searching for frequent terms, the 100 searches are performed approximately 900 seconds faster in the largest index when it is constructed with 4 feeding threads than with 1 when using small buffers. The difference is much smaller when using large buffers.

When using 4 feeding threads instead of 1, we may permute the order in which index entries are inserted into the B-tree. This may obviously affect which nodes that are cached when construction ends. If the buffered nodes are used in the searches, we will obviously have much more efficient searches. This can explain that there is differences between using 1 and 4 feeding threads. The difference observed when searching for frequent terms in the largest index with small buffers seems quite large however. By looking at the logs, we notice that there are typically more writes while searching for long terms when the index was constructed with 1 feeding thread instead of 4. This is unexpected, because it is likely that 4 feeding threads will generate more dirty buffers at the end of the construction phase. It is possible that the number of writes generated in total is lower when using 4 feeding threads however, because several buffers are updated many times before they are flushed. This will typically generate fewer writes overall, which in turn may cause fewer dirty buffers at the end. Such an explanation does not seem very likely, but it is not necessarily impossible that this happens in one case in our experiments.

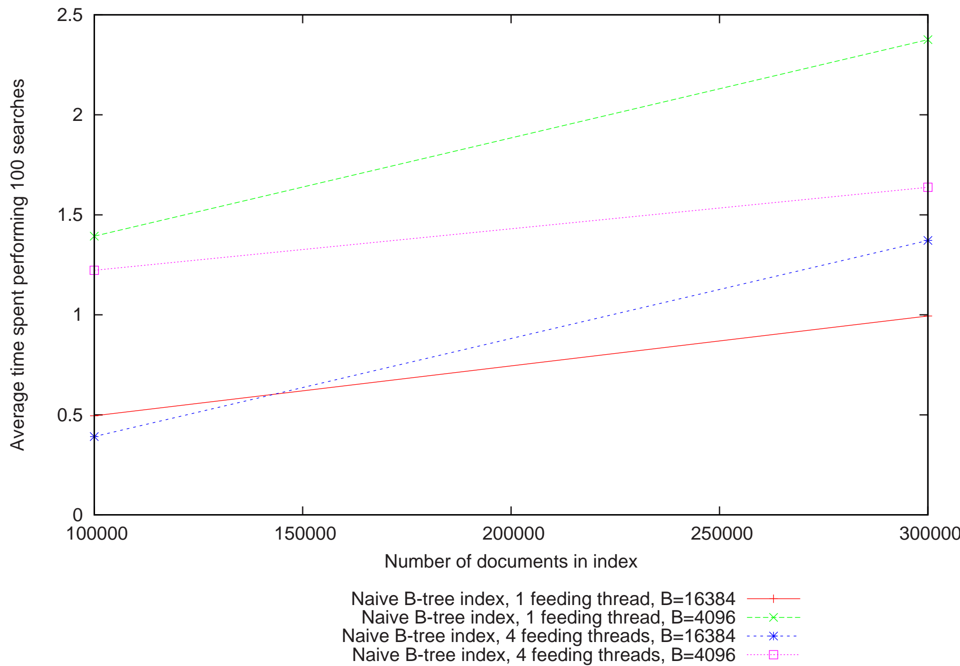

Figure 7.25: Average time spent performing 100 searches for terms with low frequency in a naive B-tree index

7.4.4 Actual results versus estimates

This section will compare the obtained results with the estimates given in sections 6.6 and 6.7.3. We will consider update speed in Section 7.4.4.1, and search performance in Section 7.4.4.2.

7.4.4.1 Update speed

When we estimated the time spent constructing a naive B-tree index, we did not end up with an exact number. We found it difficult to estimate the hit ratio, and we therefore chose to provide the estimate as a function of the average hit ratio. We will therefore do things the other way around for this structure. Based on the construction time, we will calculate the average hit ratio based on our estimates. We will then try to consider to which extent this hit ratio seems reasonable.

We estimated the time spent constructing an index for $N = 300000$ documents with 1 feeding thread in Section 6.6. The actual results from the experiments are repeated in Table 7.15

$B = 16$KB	$B = 4$KB
5h 34m 18s	5h 4m 41s

Table 7.15: Average time spent constructing naive B-tree indexes

The calculations in Section 6.6.5 suggested that the time spent inserting index entries in the

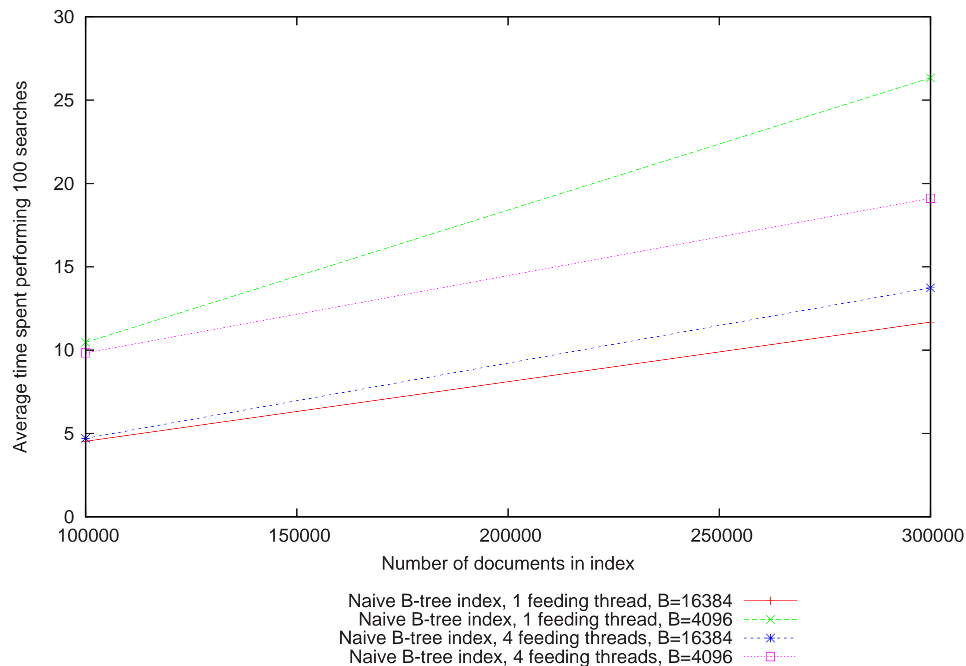

Figure 7.26: Average time spent performing 100 searches for terms with medium frequency in a naive B-tree index

B-tree and the possible reads required to read unbuffered nodes, was likely to be the bottleneck during construction of the naive B-tree index for 300000 documents. Based on equations 6.53 and 6.54, we calculate the assumed hit ratio. When using small buffers, our calculations suggest that the average hit ratio is 0.9996. For large buffers, the estimated hit ratio is 0.9994.

We mentioned in Section 6.6.5 that this calculation will give an upper bound on the actual hit ratio, because we assume that the B-tree has the same size throughout its lifecycle, while it in practice starts out empty. Even so, these calculations suggest that the hit ratio is very attractive. We will try to evaluate to which extent the calculated hit ratio is reasonable, by estimating the size of the complete B-tree. We estimated in Section 6.6.5 that the naive B-tree index would require 986412 nodes when using small buffers, and 245519 nodes when using large buffers. It is thus straight-forward to calculate the expected sizes for these files. The expected size of the index when using small buffers is 3.76GB and it is 3.75GB when using large buffers. The complete size of the buffers in our buffer pool is 750MB. Some of these buffers will be used to buffer parts of the B-tree in the document manager. If we assume that no parts of the document manager are buffered, we are able to buffer approximately 19.5% of the complete index files. For such a setting, the hit ratio depicted above seems impressive.

We argued in Section 6.6.5 that the hit ratio is likely to be quite high, due to the Zipf distribution assumed for the terms in the document collection. When we also keep in mind that the calculation actually finds an upper bound on the hit ratio, we have no reason to conclude that our estimates are definitely unreasonable.

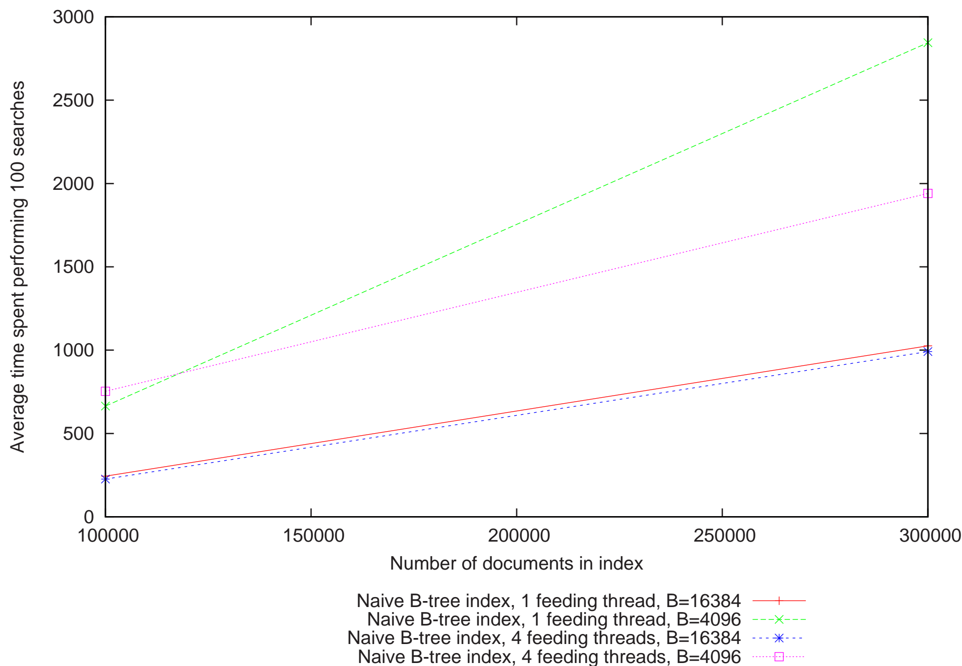

Figure 7.27: Average time spent performing 100 searches for terms with high frequency in a naive B-tree index

7.4.4.2 Search performance

In Section 6.7.3, we estimated the time spent performing searches for terms with varying frequency. As for the other methods, we did not consider the time spent accessing the B-tree in the document manager to retrieve the URIs for the documents that matches the query. We did however; assume worst-case behaviour in the actual search. We will now compare these estimates with the actual results. Table 7.16 repeats the estimates from Section 6.7.3, and Table 7.17 shows the actual results in a readable format.

Term frequency	**Naive B-tree index**	
	$B = 4$KB	$B = 16$KB
high	23.08	7.51
medium	0.092	0.066
low	0.066	0.066

Table 7.16: Expected average time for searches for terms with various frequency ($N = 300000$)

Tables 7.16 and 7.17 shows that for the terms with low frequency, the estimates are higher than the actual time spent performing the searches for all configurations. These results suggest that searches in the B-tree will typically not have worst-case performance. With the calculations of the estimated hit ratio from the previous section in mind, this seems reasonable. We also note that we have no guarantee that the searches for terms with low frequency will have any matches in the index we test with. The reason is that $N = 300000$ for this case, while the terms were

Term frequency	1 **feeding thread**		4 **feeding threads**	
	$B = 4$KB	$B = 16$KB	$B = 4$KB	$B = 16$KB
high	28.445	10.256	19.404	9.911
medium	0.263	0.117	0.191	0.137
low	0.024	0.010	0.016	0.014

Table 7.17: Search performance in naive B-tree indexes ($N = 300000$)

chosen from an index with $N = 1$ million documents. We thus do not necessarily need to access the document manager for each search, which we probably will for the other groups of terms.

For terms with medium and high frequency, the actual results are actually slower than the estimates suggest in most configurations. These searches will typically access the document manager to retrieve the URIs for the documents matching the search. Even so, based on the calculated hit ratio from the previous section, these results are slightly worse than expected. We will therefore analyze the reasons for this behaviour by looking at the log from an experiment with the naive B-tree index in Appendix B.4. The log is from an experiment with 1 feeding thread and $B = 16$KB.

In the phases where we search for terms with medium and high frequency, the disk statistics show that there is a very high number of disk accesses. We would expect that there are some reads, but there are actually a lot of writes as well in this phase. The number of reads is also higher than we expected. The significant number of writes suggests that there are still a lot of dirty buffers in the buffer pool when construction ends. These will be written to the *FileChannel* by the flushing thread, and they will eventually be written to disk. It may also be updates pending in higher levels of the B-tree when the construction ends. The thread responsible for processing these updates may thus actually generate writes after the construction is done. We believe it is unlikely that the number of updates is large enough to have a significant impact however. Most of the observed disk writes are thus probably caused by the flushing thread which writes out the content of dirty buffers. The fact that there are disk writes will typically slow down the read operations required in the search.

We noted above that there were more disk reads than we expected in these phases as well. It is definitely likely that several of these reads are caused by accesses to the B-tree in the document manager. We would expect more of these reads when searching for terms with medium than high frequency, because larger parts of the B-tree in the document manager is buffered when the searches for terms with high frequency are processed. There is also another thread in our application that may cause a lot of reads. The thread updating the tf-idf lengths of all documents will, at regular intervals, iterate over all nodes in the leaf level of the B-tree. The leaf nodes in the B-tree are typically the ones that are accessed least frequently. The buffers storing the content of these nodes will thus typically be replaced when using an LRU replacement policy in the buffer pool. The iteration over the leaf level in the B-tree will thus typically generate a significant number of disk accesses. We have unfortunately not recorded when this process starts and stops in our output from the experiments. We are therefore not able to decide whether this may be the cause of some of the disk accesses observed in the mentioned search phases.

The discussion above shows that there are several processes that are likely to access the disk while the searches are processed. When the searches require disk accesses, it is thus likely that their performance will suffer significantly because of the load on the disk. This explains the results observed when searching for the terms with medium and high frequency.

7.4.5 Discussion of results

We have presented the results from the experiments performed with the naive B-tree index above. We have also commented on the results to explain unexpected behaviour. There are still a few aspects however, that have not been treated in detail. The first is that the configurations with $B = 16$KB proved to have worse performance than the ones with $B = 4$KB, both for $N = 300000$ and for $N = 100000$. When $N = 100000$, we are able to buffer the complete index. The fact that the configurations with $B = 16$KB are slower in this case, suggests that more processing time is required to insert entries in the B-tree when using larger buffers. We will explain why below. In addition, we will present the partial results from the run were we started to index 1 million documents, but aborted the run after slightly over 800000.

When we use larger buffers in the buffer pool, there are larger nodes in the B-tree. We will thus typically have fewer levels in the B-tree, as several estimates in Chapter 6 suggest. The fact that it is more efficient to construct a naive B-tree index with 100000 documents when using small buffers thus seems unexpected. The binary searches are in larger lists when using larger buffers, but we also require fewer binary searches to reach the leaf level on average. We have no reason to believe that it is relevant for this phase whether one uses small or large buffers. The only explanation we can come up with to explain these results, is thus that the insert in the actual node at the leaf level is more efficient with smaller buffers. Because we should enable binary searches in the nodes, the entries must be stored sequentially. We do actually not store the entries sequentially however, but rather store the pointers to the entries in sorted order on the entries they point to. Even so, when a new entry is inserted, we have to move the pointers to all entries within the node that are larger than the inserted entry. This is typically assumed to be an efficient operation because of cache effects, but it might be the cause of the slower insertions when using larger nodes.

We will now present the partial results from the experiment where we started indexing 1 million documents in a naive B-tree index. A plot showing when documents are read and indexed in this case is shown in Figure 7.28.

Even though the update latency is always low for the naive B-tree index, the update speed becomes very slow after a while, as shown in Figure 7.28. This happens when approximately 500000 documents have been added to the index. When there are 700000 documents in the index, it takes more than 100000 seconds to add 100000 more documents. We thus index less than 1 document per second during this phase. We will not provide further comments on this experiment, but note that it shows clearly that our implementation of a naive B-tree index does not scale well to reasonable index sizes.

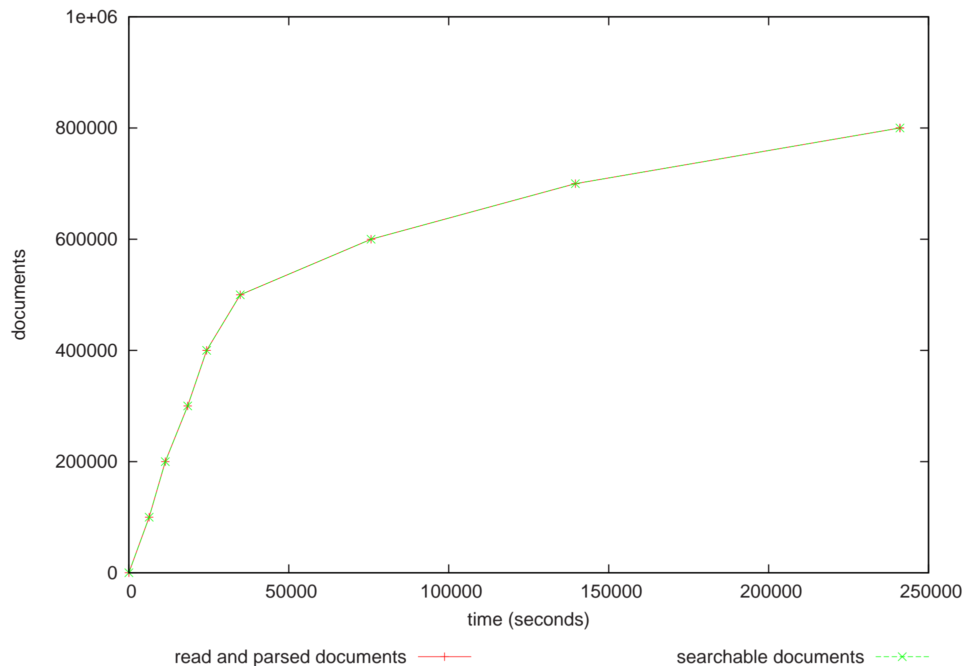

Figure 7.28: Read and indexed documents over time in a naive B-tree index

7.4.6 Chosen configuration

All configurations of the naive B-tree index have a very attractive update latency. On other aspects however, the performance is questionable. The results from testing update speed clearly showed that it was beneficial to use 4 feeding threads instead of 1. Our chosen configuration for this method will thus clearly have 4 feeding threads.

A choice between using 4KB or 16KB buffers is basically a choice of whether we prefer update speed or search speed. It seems to be larger differences between the two when considering search speed than update speed however, and we thus choose the configuration with $B = 16$KB and 4 feeding threads.

7.5 Comparison of the chosen configurations

This section compares the overall methods tested in this report. We have chosen the best configuration for each method after presenting the results. We will now compare these chosen configurations based on updatability and search performance in the following subsections, before we conclude in Section 7.5.3.

7.5.1 Updatability

Both the rate at which updates can be processed and the update latency, are important aspects of updatability. We will compare the chosen methods according to both of these aspects. Figure 7.29 shows the average construction time for the different indexes, which describes the rate at which updates can be processed.

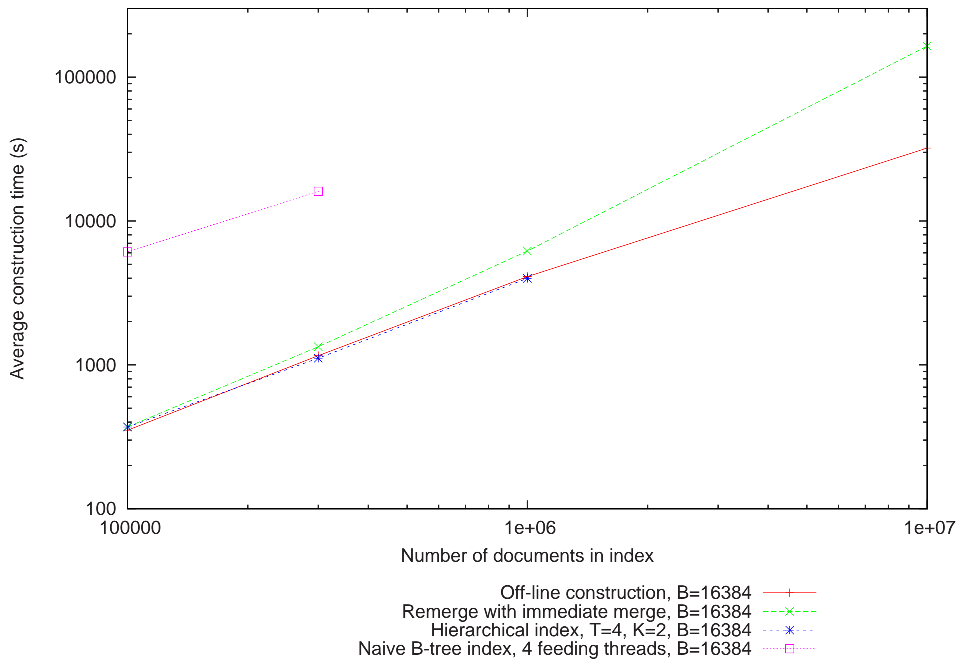

Figure 7.29: Average time spent constructing indexes with the chosen configurations

We do not have results for all index sizes for the hierarchical index and the naive B-tree index, as explained above. The plots for these methods thus stop after 1 million and 300000 documents respectively. Nonetheless, Figure 7.29 shows some general trends. The naive B-tree index is slower at constructing an index with 100000 documents than the two fastest methods are at constructing an index with 1 million documents. As explained in Section 7.4.5, the feeding rate for this method drops significantly after 500000 documents have been added to the index as well. Its update speed is thus far from attractive.

The two configurations for remerge perform quite similarly for the smallest index, but off-line construction is significantly faster at constructing the larger indexes. Even if off-line construction seems to be quite fast, the hierarchical index is actually slightly faster at constructing the indexes with 300000 and 1 million documents. This might seem to be a surprising result because we know that the hierarchical method will perform more merges. We have concluded in the above sections however, that the construction of partial indexes is CPU bound, while the merges are disk bound. The best case possible for a hierarchical index with $T = 4$ mentioned at the end of section 6.5.2, is thus very close to the actual case. It is unfortunate that we do not have results from constructing the hierarchical index with 10 million documents, because a comparison with off-line construction in this case would have been interesting.

We conclude that off-line construction and the hierarchical index are the best when it comes to update speed in our implementation. Figure 7.30 shows the plots describing update latency for all methods. We have chosen to use $N = 300000$ in all graphs, to make them easier to compare. The range on the x-axis is the same for the hierarchical index and the two configurations for remerge. Using the range from the experiment with the naive B-tree index on the other methods, would have made the plots nearly invisible. The x-range in the plot for the naive B-tree index is thus larger than the others.

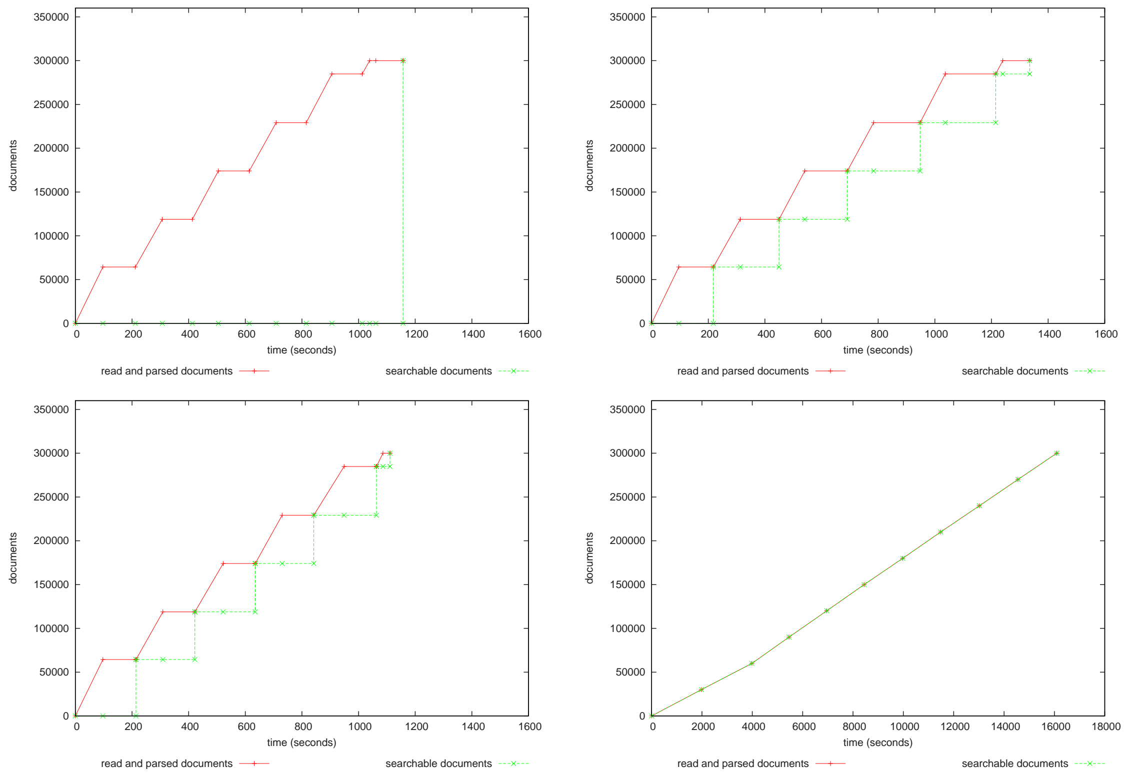

Figure 7.30: Plot showing the update latency for the chosen configurations. The top left figure shows remerge as off-line construction. The top right figure shows remerge with immediate merge, while the bottom left shows the hierarchical index. All these first three figures have the same range on the x-axis. The bottom right figure shows the plot for the naive B-tree index

Figure 7.30 shows that the naive B-tree index has the lowest update latency of the tested methods, even though it is slow at constructing the index. Remerge as off-line construction is obviously the method with the worst update latency. As explained in previous sections in this chapter, the update latency in remerge with immediate merge is dependent on the size of the complete index. We see from the plot that the update latency clearly increases as new batches are added. This is not true to the same extent for the hierarchical index however, and its update latency is more attractive.

7.5.2 Search performance

This section compares the search performance for the chosen configurations. As we have done when comparing all configurations for a given method, we consider searching for terms with different frequencies. The results are shown in Figure 7.31.

All methods seem to show quite similar performance when searching for terms with low frequency, except that remerge with immediate merge is significantly slower than off-line when $N = 10$ million documents. The same effect is shown when searching for terms with medium frequency. When searching for terms with high frequency however, the index constructed with immediate merges shows similar performance as the one constructed with immediate merge.

For terms with medium frequency, it is clear that the naive B-tree index has worse search performance than the others. This is a tendency that is even clearer when searching for common terms. In these two cases, the hierarchic index also provides slightly slower searches than the two configurations of remerge.

In conclusion, it is clear that the naive B-tree index provides slower searches than the other methods, especially for terms that are reasonably common. The remerge configuration with immediate merge provides slower searches in the largest index according to our results. We keep in mind though that these experiments were only run once. Otherwise, the configurations for remerge and hierarchical indexes have comparable search performance even though the hierarchical methods are slightly slower in some cases.

7.5.3 Conclusion

The results provided above show that even though the naive B-tree index has very attractive update latency, both its search performance and update speed make this method unattractive. We note however, that the fact that this method is capable of processing updates at a quite steady rate is a beneficial feature. The major problem is however that this rate is quite slow even when the complete index can be buffered. In addition, the rate decreases to less than 1 document per second when there is a significant number of documents in the index. We thus conclude that despite some beneficial features, our implementation of a naive B-tree index is generally not an attractive index for a large document collection with updates.

The configuration of remerge that works as off-line construction is fast at handling updates, and provides efficient searches. Its update latency is prohibitive however, because no documents are searchable until all documents are indexed.

The two last methods, remerge with immediate merge and the hierarchical index, both have an update latency in between the naive B-tree index and remerge as off-line construction. We have argued that the latency in our implementation of the hierarchical index is less dependent on the size of the complete index. It is possible to modify remerge with immediate merge slightly, such that it will have the same characteristics as the hierarchical index on this aspect. We will concentrate on comparing the methods the way we have implemented them however, and thus

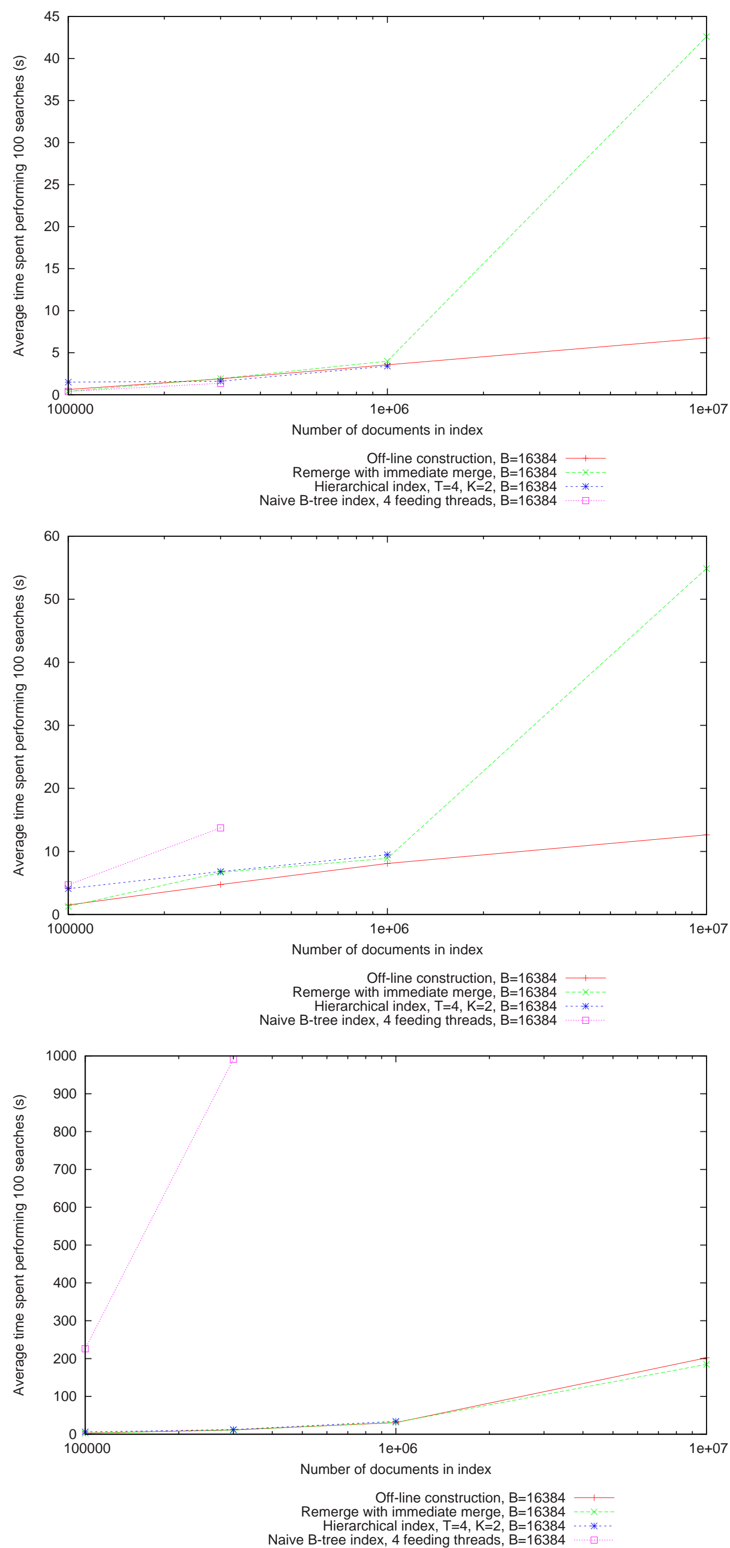

Figure 7.31: Average time spent performing 100 searches for terms with low (top figure), medium (middle figure) and high (bottom figure) frequency in the compared structures.

conclude that our implementation of the hierarchical index has a more attractive update latency than remerge with immediate merge.

The search efficiency is comparable in remerge with immediate merge and the hierarchical index, according to the presentation above. We should note however, that we have only tested searching in indexes with given sizes. When searching in the index with 1 million documents, there are only 3 non-empty indexes in the hierarchy. The fact that the number of disk accesses in a search in a hierarchical index varies based on the size of the index is a disadvantage.

The update speed in remerge with immediate merge decreases as the size of the complete index grows. This is caused by the larger and larger merges required to make the partial index become part of the main index. The update speed for the hierarchical index however, is very attractive in our experiments. Because the construction of partial indexes is CPU bound and the merges are disk bound, they can go on in parallel. It is actually faster to construct a hierarchical index with 1 million documents than to construct the index with off-line construction. We will discuss the fact that construction of partial indexes is CPU bound in Section 7.7. It is still clear however, that in our current implementation, the update speed in the hierarchical index is very attractive. We should also note that the fact that both of these methods indexes documents in batches, makes the rate at which documents are read vary significantly. It is generally more attractive to be able to index a given number of documents per second during the whole lifecycle of the index. We will consider possible modifications to achieve such a goal in section 7.7.

In conclusion, we believe that the hierarchical index is the most attractive method in our implementation. The impressive update speed is considered more attractive than the lower variability in search performance found in remerge with immediate merge. The modification we made to the hierarchical index, namely to allow several searchable partial indexes, has proven to have a positive effect on the update speed in our implementation.

7.6 Intrusiveness of output

To obtain the results provided above and be able to analyze them, our application has generated output which was written to a file. This will obviously generate disk writes, which will again have an effect on the overall performance. We have tried to limit the amount of output so that it should have limited effect on the end result. In order to test whether we have succeeded, we introduced some additional experiments in Section 5.3.4. These experiments construct an index with 300000 documents with one configuration for each overall method. We will compare the results from these runs with the average time spent constructing such an index in our experiments, and the sample standard deviation.

Table 7.18 shows the results from the experiments run without output, together with the average construction time and the sample standard deviation from the actual experiments with the same configurations. We note that in all of the tests, the time spent constructing the index without generating output is lower than the average from the actual experiments. This implies that the output does affect the end result. The difference is always within one sample standard deviation

	Remerge	Hierarchic index	Naive B-tree index
Average construction time with output	1157.72	1111.49	16104.05
Sample standard deviation	100.41	84.03	609.07
Construction time without output	1074	1047	15666

Table 7.18: Comparison of construction times for indexes with and without output

however. This allows us to conclude that even though the generated output probably affects the end result, the effect is small enough to ensure that our obtained results are reasonably trustworthy.

7.7 Possible future improvements of implementation

We have now presented the results from all experiments in this project. Together with the efficiency model developed in Chapter 6, this has made us more aware of the advantages and disadvantages of some of our implementation choices. We will provide an overview of the most obvious future improvements of our implementation in this section.

The discussion in Section 7.2.5 pointed out that the fact that we never force a *FileChannel* to flush the changes it contains to disk may have a negative effect on performance. We do not know for sure if the theory presented there is correct, but the fact that we do not know exactly when changes are written to disk also makes it harder to provide accurate estimates on the time spent accessing disk. We can for example implement a scheme where we do not run the thread writing buffers to their *FileChannel*s at regular time intervals, but rather start it when a given fraction of the buffers in the buffer pool are dirty. We could then write out all the dirty buffers, and force each *FileChannel* to flush the changes to disk. When processing a number of dirty buffers in one batch, we may also sort them. We would thus be able to write out the buffers belonging to the same file immediately after one another. We could also write them out in the order in which they occur in the file. This may help the *FileChannel* to get a sequential access pattern.

If our theory introduced in Section 7.2.5 is correct, the scheme introduced above would avoid the significant differences in construction time for large indexes based on whether we use large or small buffers. In addition, it would probably be easier to analyze the expected number of disk writes during a merge for example.

When using remerge as off-line construction, the results above show that we are able to index approximately 311 documents per second when creating an index with the first 10 million documents in GOV2. To get an idea of the quality of this result, we can compare it with related work from the literature. [BC06] reports results from experiments where they are able to make partial indexes for the complete GOV2 collection in 4.66 hours. We are able to construct partial indexes for 10 million out of the 25.2 million documents in the collection in approximately 7.57 hours. If we assume that the rest of the documents have the same average size, we would thus use approximately 18.93 hours constructing partial indexes for the complete collection. There

are obviously room for significant improvements in our implementation.

Without the knowledge we now possess, it would have been tempting to suggest that the reason for their improved performance is that they compress the index. Our efficiency model suggested that both phases during construction of partial indexes were CPU bound however, and the actual results confirmed this theory. It is therefore obvious that compressing the index can not have a positive effect on the performance in this phase. It thus seems like the performance of the modification of in-memory accumulation we have proposed, is questionable. We will compare our method to the one suggested in [BC05c] to analyze this aspect.

When parsing a document in our implementation, we create a *HashMap* that contains entries for all unique terms in the document. Each entry is a list of occurrences of the term within the document. The contents of this *HashMap* is then added to the in-memory index. The next phase sorts the terms from all documents in a multi-way merge. By using the method suggested in [BC05c] instead, we would only parse the document, and perform a look-up in a *HashMap* for each term. The occurrence of the term is then stored in the associated entry in the *HashMap*. Because the expected search time in a hashing structure is typically $O(1)$, it does not matter how many entries it contains. It is therefore likely that the accumulation phase with this method is just as efficient as the one we suggested. The main advantage with the method however, is that we do not need to sort the entries in the second phase. We just write out the inverted lists in sorted term order. It is therefore likely that this phase is disk bound, as opposed to our implementation. Even if our method has benefits in that it only uses constant memory apart from the buffers, the poorer performance makes it unattractive. Unfortunately, we were not aware of the method presented in [BC05c] when this implementation choice was made. We will obviously change this in the future. If we then end up with more phases that are disk bound, we will also implement compression.

The naive B-tree index is an example of an eager update scheme. Its major benefit compared to the other methods we test, is that the inserted documents are almost immediately searchable. This is not true for the methods we test that batch documents. It is claimed in [LZW06] that the documents inserted in their index are immediately searchable, even though they batch updates. We should thus be able to do the same. There are two reasons why we do not support immediate searches in the current implementation. The most important reason is that we have decided to support the tf-idf ranking scheme. To be able to calculate reasonably correct document lengths according to this scheme, we need information about the number of occurrences of each term in a document in the complete collection. To obtain such information we would have to access other indexes. To do this for every term would probably require a lot of disk accesses while parsing the documents. The performance would thus typically suffer.

The other reason why documents are not immediately searchable is that our in-memory index does not support efficient searches. When we chose this structure, we had already decided to use tf-idf, and allowing searches while accumulating the index was thus not a very attractive feature. By changing to another ranking scheme than tf-idf, and switching the structure of the memory resident index, we would be able to make a document searchable immediately after it has been added.

Using tf-idf also has several other negative effects on performance. To keep the document

lengths reasonably correct, we access the largest dictionary in the current hierarchy each time we perform a merge with smaller indexes in a hierarchical index. In addition, the dictionary in the largest index of the hierarchy is accessed when we make partial indexes searchable. In the naive B-tree index, there is a thread that recalculates all document lengths at regular intervals. All of these aspects have negative effects on performance, as the results and discussions above have shown. Several authors claim that the extra computation costs in tf-idf are not justified by significantly improved ranking, as mentioned in Chapter 2. It thus seems like an obvious choice to switch to another ranking scheme in the future.

In all of the methods we test that batch updates, the construction of partial indexes ensures that we read and parse documents in one phase, and write out the resulting partial index in another. While we are writing out a partial index, we are not able to process updates. We would prefer being able to support a steady rate of updates. To do so, we suggest partitioning the buffers set aside for in-memory accumulation into two subsets. We will start by filling the first. When it is full, we will start writing it to disk. At the same time, we initiate accumulation in the other subset of buffers. In our experiments, the phase parsing the documents is currently slightly faster on average than the phase sorting the entries. As noted in Section 7.3.4.1, the complexity of the sorting phase is not linear. It is thus likely that reducing the size of a batch will make the time spent in both of these phases similar. If we adjust the sizes of these partitions, it is thus likely that we are able to support a steady feeding rate. This ends our discussion of possible future improvements.

Chapter 8

Conclusion

This report has compared three overall approaches for reflecting newly added documents in a document collection in inverted indexes. In addition to performing experiments, we have developed estimates for the expected performance of the different structures.

The results from experiments show that our efficiency model is capable of estimating the time spent during most phases of index construction with reasonable accuracy. The estimates are not correct for merges of indexes however.

We have introduced two possible refinements in the implemented methods. The first is a scheme for accumulating an index in memory during sort-based inversion. Although this modification has attractive memory characteristics and asymptotic complexity, its performance is modest compared to the method introduced in [BC05c]. The problem with the modification is that it makes both phases of construction of partial indexes CPU bound.

The other modification is for hierarchical indexes, where we define a variable representing the maximum number of partial indexes we allow searches in. Allowing searches in more than one partial index has a positive effect on the update speed in our implementation. This is partly due to the fact that construction of partial indexes is CPU bound.

This modification makes the search performance slightly poorer on average, but we conclude from our experiments that the hierarchical index which allows searches in 4 partial indexes performs best of all methods in our implementation. Remerge with immediate merge is the best of the other tested methods. It is slower on both update speed and latency than the hierarchical method. We also test a naive B-tree index. Even though this method has impressive update latency, its update speed and search performance are inhibitive according to our experiments.

All tested methods in our implementation use the tf-idf ranking scheme. We have proposed solutions for how this ranking scheme can be incorporated into updatable indexes. Even if these solutions make the overhead involved in supporting this ranking scheme smaller, there are still problems that make us conclude that tf-idf is not well fit for use in updatable indexes. The most important problem is that it is hard to support immediate searches in memory resident indexes

and still have a decent update speed. We will therefore not use tf-idf in future implementations of updatable indexes.

8.1 Further work

In further work we will improve our current solution, and possibly bring in new aspects. The possible improvements of our current solution are presented first.

Having an accurate efficiency model both makes it easier to understand the advantages and disadvantages of our implementation, and helps us consider which possible future improvements are worth implementing. To make the model estimate the time spent during merges more accurately, we will estimate the variables for CPU performance outside the application instead of in a baseline run. We will also perform the experiments suggested in Section 7.2.4.1 to achieve better estimates for the time spent accessing disk.

We will also try to improve the implementation. We have indicated several possible improvements in Section 7.7. Of possible new features it would be interesting to incorporate into our solution, we mention a distinction between long and short inverted lists. Employing such a strategy with hierarchical indexes is shown to have impressive performance in [BCL06]. Their search performance is not that impressive however, and this is an obvious area for further work.

Bibliography

[ADHN06] Lars Arge, Andrew Danner, Herman Haverkort, and Zeh Norbert. I/o efficient hierarchical watershed decomposition of grid terrain models. In *Proceedings of International Symposium on Spatial Data Handling*, 2006.

[BC05a] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 317–318, New York, NY, USA, 2005. ACM Press.

[BC05b] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. Technical report, Waterloo, Canada, 2005.

[BC05c] Stefan Büttcher and Charles L. A. Clarke. Memory management strategies for single-pass index construction in text retrieval systems. Technical Report CS-2005-32, University of Waterloo, Waterloo, Canada, 2005.

[BC06] Stefan Büttcher and Charles L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems. In *ECIR '06: Proceedings of the 14th ACM Conference on Information and Knowledge Management*, New York, NY, USA, 2006. ACM Press.

[BCL06] Stefan Büttcher, Charles L. A. Clarke, and Brad Lushman. Hybrid index maintenance for growing text collections. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 356–363, New York, NY, USA, 2006. ACM Press.

[BHG] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*.

[Bjø06] Truls A. Bjørklund. A survey of inverted indexes for dynamic document collections. Project Report in TDT4715 Construction of algorithms and visualization, Specialisation, June 2006. -.

[Bra] Kjell Bratbergsengen. *TDT4225 Lagring og behandling av store datamengder*.

[BYRN99] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999. BAE r2 99:1 1.Ex.

[CC95] C. Clarke and G. Cormack. Dynamic inverted indexes for a distributed full-text retrieval system, 1995.

[CCB94] C. Clarke, G. Cormack, and F. Burkowski. Fast inverted indexes with on-line update, 1994.

[CH98] T. Chiueh and L. Huang. Efficient real-time index updates in text retrieval systems. Technical report, SUNY at Stony Brook, NY, USA, 1998.

[CP90] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the 13th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[Eli75] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[FJ92] Christos Faloutsos and H. V. Jagadish. On b-tree indices for skewed distributions. In Li-Yan Yuan, editor, *VLDB*, pages 363–374. Morgan Kaufmann, 1992.

[Fre02] James C. French. Modeling web data. In *JCDL '02: Proceedings of the 2nd ACM/IEEE-CS joint conference on Digital libraries*, pages 320–321, New York, NY, USA, 2002. ACM Press.

[GF04] David A. Grossman and Ophir Frieder. *Information Retrieval: Algorithms and Heuristics*. Springer, 2004. GRO d3 04:1 1.Ex.

[Gol66] S W Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT- 12(3):399–401, 1966.

[Gri05] Nils Grimsmo. Dynamic indexes vs. static hierarchies for substring search. Master's thesis, 2005.

[GWC04] Steven Garcia, Hugh E. Williams, and Adam Cannane. Access-ordered indexes. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 7–14, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[HCG+06] Lene Hallen, Joacim Lunewski Christiansen, Nils Grimsmo, Truls A. Bjørklund, and Per Kristian Helland. Brille search engine, 2006.

[HZ03] Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.*, 54(8):713–729, 2003.

[LL07] Robert W.P. Luk and Wai Lam. Efficient in-memory extensible inverted file. *Information Systems*, 32(5):733–754, 2007.

[LMZ05] Nicholas Lester, Alistair Moffat, and Justin Zobel. Fast on-line index construction by geometric partitioning. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 776–783, New York, NY, USA, 2005. ACM Press.

[LWP+01] Lipyeow Lim, Min Wang, Sriram Padmanabhan, Jeffrey Scott Vitter, and Ramesh Agarwal". Characterizing Web document change. *Lecture Notes in Computer Science*, 2118:133–??, 2001.

[LWP+03] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Agarwal. Dynamic maintenance of web indexes using landmarks, 2003.

[LZW04] Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *CRPIT '26: Proceedings of the 27th conference on Australasian computer science*, pages 15–23, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[LZW06] Nicholas Lester, Justin Zobel, and Hugh Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4):916–933, 2006.

[OvL80] Marc Overmars and Jan van Leeuwen. Some principles for dynamizing decomposable searching problems. Technical Report RUU-CS-80-1, Rijksuniversiteit Utrecht, 1980.

[PZSD96] Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10):749–764, 1996.

[RE04] Knut Magne Risvik and Tor Egge. The fms search kernel and its performance characteristics, 2004.

[RWB99] S. Robertson, S. Walker, and M. Beaulieu. Okapi at trec-7: automatic ad hoc, filtering, vlc, and interactive track. In *Proceedings of the Seventh Text Retrieval Conference*, pages 253–264, 1999.

[SC05] Wann-Yun Shieh and Chung-Ping Chung. A statistics-based approach to incrementally update inverted files. *Inf. Process. Manage.*, 41(2):275–288, 2005.

[SWYZ02] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 222–229, New York, NY, USA, 2002. ACM Press.

[TGMS94] Anthony Tomasic, Hector Garcia-Molina, and Kurt A. Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD Conference*, pages 289–300, 1994.

[WMB99] Ian Witten, Alistar Moffat, and Timothy C. Bell. *Managing Gigabytes*. Academic Press, 1999. WIT i 99:1 1.Ex.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least-Effort*. Addison-Wesley, Cambridge, MA, 1949.

Appendix A

Terms searched for in experiments

A.1 Frequent terms

```
research
3
10
national
health
2002
02
will
18
15
as
c
public
9
n
30
4
in
been
page
state
from
or
have
5
may
an
```

```
e
on
would
8
2
0
u
to
00
a
was
has
m
department
12
data
16
these
03
if
can
with
home
not
de
of
01
11
for
is
by
other
one
html
also
that
it
this
1
we
be
services
6
program
were
```

their
14
i
gov
13
new
its
information
which
they
any
you
are
at
the
b
2003
and
d
use
all
no
time
20
7
more
about
s

A.2 Terms with medium frequency

mailto
europeans
22z
1181
91109
hvn
1056
postcard
6112
antispam
campylobacter
asociaci

jam
phy
shawn
kerry
rk
frawley
infantil
meteorologists
tutor
seis
3042
mindspring
snack
clemente
guthrie
underscores
crowded
infantil
manipulating
eli
relativity
04z
remuneration
hugo
empleo
psychotherapy
voa
o3
concentrates
uty
rectal
qt
1134
dove
osage
1355
2121
fbo
stacked
invocation
sponge
listeriosis
cty
wc3
negotiators

deliberative
organizer
underscores
24hr
uphold
dsc
debated
rck
civilization
2048
1545
banco
causation
aasa
aasa
manatee
22314
diminishing
refrigerators
rp2
crafted
escort
chung
dewey
03e
ninr
1431
waller
ep5
peabody
sponge
posterior
6112
10z
2048
vest
foci
2310
vlc
negotiators
2350
cryo
5800

A.3 Terms with one occurrence

```
8985741
3456645
5a90
316778
x4262
sc760e
affectedthe
d037968
190048660
164654
nicololas
ibbmec
m11n12e
hh227pc5
rec18301
129x1
3002894512
mar345remote
fluoderm
57217907
p5162345
2047593
bragenski
genequantification04
t1654
880937
meji
geophyics
frutex
laszek
fd011148
smallparticles
210807
sonorit
daciformis
blki
tetraurelia
bodyteen
q030
stipulatation
hamelberg
shozari
0614543
```

miller99
279199
5176920
10076473
hurger
h2105
nj04114
632088
weblookup
1240616
1940218bg
858890
vollbert
pursuingbusiness
subtartive
ms9000e
kichijoji
803194
790580
4588448
30490944
nwwg4000200049
2581152
4127670
otocalm
pnu020
17166380
206400
19992833
8873154
v90596
mo00061
73ay404
mdat40
0922e
14bs229
lp00043725
wcir023311800
r0001
regionallabor
a786gent
5r01ca095662
1061002
2089079
nrmap000100673

```
1802144lg
9649858
2000hnwx0005
tn10231
yfmz99
nouve
812080100
wdbda24c1ff092237
monkeywrench
90et0170
12604751
cedell
```

Appendix B

Logs from selected experiments

This chapter contains the output from selected experiments. The purpose of including them is to use them while analyzing to which extent the different phases are CPU or I/O bound when we present the results in Chapter 7.

We have chosen not to include the output from all experiments, because it would require a lot of pages. The output from all experiments may be obtained by contacting the author.

B.1 Remerge as off-line construction

This log is from an experiment where an index with $N = 1$ million documents was built with off-line construction. The experiment was run with the largest buffer size in our experiments, $B = 16$KB.

```
Buildfile: build.xml

clean:
   [delete] Deleting directory /usr/brille/bin

build:
    [mkdir] Created dir: /usr/brille/bin
    [javac] Compiling 86 source files to /usr/brille/bin
     [copy] Copying 1 file to /usr/brille/bin

deploy:
      [jar] Building jar: /usr/brille/brille.jar

run:
     [java] The operating system is Linux
```

```
[java] adding file /data/gov2-corpus
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 9762
[java] Reads merged: 3956
[java] Sectors read: 2310168
[java] Nr of milliseconds spent reading: 16836
[java] Writes completed: 36
[java] Writes merged: 22
[java] Sectors written: 464
[java] Nr of milliseconds spent writing: 188
[java] Nr of milliseconds spent doing I/O: 16660
[java] Start time: 1181903284777337000
[java] Time spent: 101017302000
[java] Device sdb
[java] Reads completed: 12
[java] Reads merged: 0
[java] Sectors read: 192
[java] Nr of milliseconds spent reading: 188
[java] Writes completed: 266
[java] Writes merged: 1108
[java] Sectors written: 11032
[java] Nr of milliseconds spent writing: 1340
[java] Nr of milliseconds spent doing I/O: 196
[java] Start time: 1181903284777337000
[java] Time spent: 101017313000
[java] cpu0 %user: 88 %nice: 0 %system: 1 %iowait: 0 %idle: 9
[java] cpu1 %user: 25 %nice: 0 %system: 0 %iowait: 0 %idle: 73
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1181903385798792000
[java] Time spent: 133970412000
[java] Device sdb
[java] Reads completed: 5
[java] Reads merged: 0
[java] Sectors read: 40
```

```
[java] Nr of milliseconds spent reading: 240
[java] Writes completed: 998
[java] Writes merged: 81171
[java] Sectors written: 657392
[java] Nr of milliseconds spent writing: 39528
[java] Nr of milliseconds spent doing I/O: 2580
[java] Start time: 1181903385798792000
[java] Time spent: 133970411000
[java] cpu0 %user: 5 %nice: 0 %system: 0 %iowait: 2 %idle: 92
[java] cpu1 %user: 95 %nice: 0 %system: 0 %iowait: 0 %idle: 4
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 8628
[java] Reads merged: 3590
[java] Sectors read: 2051528
[java] Nr of milliseconds spent reading: 17860
[java] Writes completed: 37
[java] Writes merged: 19
[java] Sectors written: 448
[java] Nr of milliseconds spent writing: 144
[java] Nr of milliseconds spent doing I/O: 17600
[java] Start time: 1181903519781480000
[java] Time spent: 98331354000
[java] Device sdb
[java] Reads completed: 10
[java] Reads merged: 0
[java] Sectors read: 96
[java] Nr of milliseconds spent reading: 144
[java] Writes completed: 177
[java] Writes merged: 2497
[java] Sectors written: 21400
[java] Nr of milliseconds spent writing: 840
[java] Nr of milliseconds spent doing I/O: 248
[java] Start time: 1181903519781480000
[java] Time spent: 98331329000
[java] cpu0 %user: 84 %nice: 0 %system: 1 %iowait: 2 %idle: 10
[java] cpu1 %user: 26 %nice: 0 %system: 0 %iowait: 0 %idle: 72
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 4
[java] Writes merged: 2
```

```
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 8
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1181903618115834000
[java] Time spent: 125761172000
[java] Device sdb
[java] Reads completed: 114
[java] Reads merged: 56
[java] Sectors read: 2376
[java] Nr of milliseconds spent reading: 1032
[java] Writes completed: 1133
[java] Writes merged: 82205
[java] Sectors written: 666744
[java] Nr of milliseconds spent writing: 40876
[java] Nr of milliseconds spent doing I/O: 3212
[java] Start time: 1181903618115834000
[java] Time spent: 125761176000
[java] cpu0 %user: 63 %nice: 0 %system: 0 %iowait: 1 %idle: 34
[java] cpu1 %user: 36 %nice: 0 %system: 0 %iowait: 1 %idle: 61
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 8281
[java] Reads merged: 3344
[java] Sectors read: 1972664
[java] Nr of milliseconds spent reading: 17896
[java] Writes completed: 36
[java] Writes merged: 19
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 92
[java] Nr of milliseconds spent doing I/O: 17752
[java] Start time: 1181903743880416000
[java] Time spent: 96344741000
[java] Device sdb
[java] Reads completed: 554
[java] Reads merged: 1415
[java] Sectors read: 17536
[java] Nr of milliseconds spent reading: 1960
[java] Writes completed: 838
[java] Writes merged: 47598
[java] Sectors written: 387528
[java] Nr of milliseconds spent writing: 8024
[java] Nr of milliseconds spent doing I/O: 3912
[java] Start time: 1181903743880416000
[java] Time spent: 96344744000
[java] cpu0 %user: 85 %nice: 0 %system: 1 %iowait: 1 %idle: 10
```

```
[java] cpu1 %user: 26 %nice: 0 %system: 0 %iowait: 0 %idle: 72
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1181903840227858000
[java] Time spent: 127402109000
[java] Device sdb
[java] Reads completed: 734
[java] Reads merged: 2572
[java] Sectors read: 26624
[java] Nr of milliseconds spent reading: 5452
[java] Writes completed: 975
[java] Writes merged: 81474
[java] Sectors written: 659632
[java] Nr of milliseconds spent writing: 41108
[java] Nr of milliseconds spent doing I/O: 5116
[java] Start time: 1181903840227858000
[java] Time spent: 127402121000
[java] cpu0 %user: 90 %nice: 0 %system: 0 %iowait: 2 %idle: 7
[java] cpu1 %user: 9 %nice: 0 %system: 0 %iowait: 1 %idle: 89
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 8578
[java] Reads merged: 3512
[java] Sectors read: 2047072
[java] Nr of milliseconds spent reading: 18620
[java] Writes completed: 34
[java] Writes merged: 19
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 84
[java] Nr of milliseconds spent doing I/O: 18476
[java] Start time: 1181903967632802000
[java] Time spent: 98363780000
[java] Device sdb
[java] Reads completed: 175
[java] Reads merged: 538
[java] Sectors read: 6304
```

```
[java] Nr of milliseconds spent reading: 948
[java] Writes completed: 187
[java] Writes merged: 2720
[java] Sectors written: 23280
[java] Nr of milliseconds spent writing: 892
[java] Nr of milliseconds spent doing I/O: 680
[java] Start time: 1181903967632802000
[java] Time spent: 98363782000
[java] cpu0 %user: 65 %nice: 0 %system: 1 %iowait: 2 %idle: 30
[java] cpu1 %user: 45 %nice: 0 %system: 0 %iowait: 2 %idle: 50
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 5
[java] Writes merged: 1
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 40
[java] Nr of milliseconds spent doing I/O: 24
[java] Start time: 1181904065999194000
[java] Time spent: 126290648000
[java] Device sdb
[java] Reads completed: 1208
[java] Reads merged: 3159
[java] Sectors read: 34960
[java] Nr of milliseconds spent reading: 5128
[java] Writes completed: 1029
[java] Writes merged: 80734
[java] Sectors written: 654168
[java] Nr of milliseconds spent writing: 39764
[java] Nr of milliseconds spent doing I/O: 5004
[java] Start time: 1181904065999194000
[java] Time spent: 126290649000
[java] cpu0 %user: 95 %nice: 0 %system: 0 %iowait: 1 %idle: 3
[java] cpu1 %user: 5 %nice: 0 %system: 0 %iowait: 2 %idle: 92
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7728
[java] Reads merged: 3175
[java] Sectors read: 1841896
[java] Nr of milliseconds spent reading: 22808
[java] Writes completed: 36
[java] Writes merged: 20
```

```
[java] Sectors written: 448
[java] Nr of milliseconds spent writing: 148
[java] Nr of milliseconds spent doing I/O: 22608
[java] Start time: 1181904192292938000
[java] Time spent: 94804494000
[java] Device sdb
[java] Reads completed: 354
[java] Reads merged: 679
[java] Sectors read: 11320
[java] Nr of milliseconds spent reading: 2568
[java] Writes completed: 209
[java] Writes merged: 3396
[java] Sectors written: 28848
[java] Nr of milliseconds spent writing: 2092
[java] Nr of milliseconds spent doing I/O: 1292
[java] Start time: 1181904192292938000
[java] Time spent: 94804488000
[java] cpu0 %user: 75 %nice: 0 %system: 1 %iowait: 3 %idle: 19
[java] cpu1 %user: 36 %nice: 0 %system: 0 %iowait: 2 %idle: 60
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 8
[java] Writes completed: 5
[java] Writes merged: 1
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 28
[java] Nr of milliseconds spent doing I/O: 28
[java] Start time: 1181904287099127000
[java] Time spent: 127047171000
[java] Device sdb
[java] Reads completed: 166
[java] Reads merged: 147
[java] Sectors read: 3576
[java] Nr of milliseconds spent reading: 1540
[java] Writes completed: 1061
[java] Writes merged: 83367
[java] Sectors written: 675472
[java] Nr of milliseconds spent writing: 39752
[java] Nr of milliseconds spent doing I/O: 3628
[java] Start time: 1181904287099127000
[java] Time spent: 127047174000
[java] cpu0 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
```

```
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7570
[java] Reads merged: 3110
[java] Sectors read: 1802696
[java] Nr of milliseconds spent reading: 22748
[java] Writes completed: 34
[java] Writes merged: 18
[java] Sectors written: 416
[java] Nr of milliseconds spent writing: 168
[java] Nr of milliseconds spent doing I/O: 22580
[java] Start time: 1181904414148340000
[java] Time spent: 93364940000
[java] Device sdb
[java] Reads completed: 51
[java] Reads merged: 106
[java] Sectors read: 1344
[java] Nr of milliseconds spent reading: 484
[java] Writes completed: 159
[java] Writes merged: 1246
[java] Sectors written: 11248
[java] Nr of milliseconds spent writing: 592
[java] Nr of milliseconds spent doing I/O: 392
[java] Start time: 1181904414148340000
[java] Time spent: 93364932000
[java] cpu0 %user: 78 %nice: 0 %system: 1 %iowait: 3 %idle: 17
[java] cpu1 %user: 33 %nice: 0 %system: 0 %iowait: 3 %idle: 63
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 2
[java] Reads merged: 0
[java] Sectors read: 512
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1181904507514900000
[java] Time spent: 127342079000
[java] Device sdb
[java] Reads completed: 61
[java] Reads merged: 93
[java] Sectors read: 1232
```

```
[java] Nr of milliseconds spent reading: 320
[java] Writes completed: 1055
[java] Writes merged: 81829
[java] Sectors written: 663136
[java] Nr of milliseconds spent writing: 37552
[java] Nr of milliseconds spent doing I/O: 3000
[java] Start time: 1181904507514900000
[java] Time spent: 127342082000
[java] cpu0 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 1 %idle: 97
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7672
[java] Reads merged: 3117
[java] Sectors read: 1822208
[java] Nr of milliseconds spent reading: 22556
[java] Writes completed: 36
[java] Writes merged: 18
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 192
[java] Nr of milliseconds spent doing I/O: 22324
[java] Start time: 1181904634858743000
[java] Time spent: 93369155000
[java] Device sdb
[java] Reads completed: 317
[java] Reads merged: 655
[java] Sectors read: 7840
[java] Nr of milliseconds spent reading: 1684
[java] Writes completed: 157
[java] Writes merged: 2516
[java] Sectors written: 21384
[java] Nr of milliseconds spent writing: 1264
[java] Nr of milliseconds spent doing I/O: 1008
[java] Start time: 1181904634858743000
[java] Time spent: 93369161000
[java] cpu0 %user: 89 %nice: 0 %system: 1 %iowait: 3 %idle: 5
[java] cpu1 %user: 22 %nice: 0 %system: 0 %iowait: 0 %idle: 76
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
```

```
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1181904728229756000
[java] Time spent: 127546040000
[java] Device sdb
[java] Reads completed: 233
[java] Reads merged: 452
[java] Sectors read: 6640
[java] Nr of milliseconds spent reading: 1560
[java] Writes completed: 1046
[java] Writes merged: 80498
[java] Sectors written: 647904
[java] Nr of milliseconds spent writing: 36952
[java] Nr of milliseconds spent doing I/O: 3624
[java] Start time: 1181904728229756000
[java] Time spent: 127546044000
[java] cpu0 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 97
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7471
[java] Reads merged: 3186
[java] Sectors read: 1798064
[java] Nr of milliseconds spent reading: 20520
[java] Writes completed: 35
[java] Writes merged: 19
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 136
[java] Nr of milliseconds spent doing I/O: 20276
[java] Start time: 1181904855777881000
[java] Time spent: 93440024000
[java] Device sdb
[java] Reads completed: 135
[java] Reads merged: 159
[java] Sectors read: 3416
[java] Nr of milliseconds spent reading: 1080
[java] Writes completed: 198
[java] Writes merged: 1980
[java] Sectors written: 37712
[java] Nr of milliseconds spent writing: 5184
[java] Nr of milliseconds spent doing I/O: 780
[java] Start time: 1181904855777881000
[java] Time spent: 93440029000
[java] cpu0 %user: 88 %nice: 0 %system: 1 %iowait: 3 %idle: 6
```

```
[java] cpu1 %user: 22 %nice: 0 %system: 0 %iowait: 0 %idle: 75
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1181904949219614000
[java] Time spent: 126079713000
[java] Device sdb
[java] Reads completed: 65
[java] Reads merged: 115
[java] Sectors read: 1448
[java] Nr of milliseconds spent reading: 368
[java] Writes completed: 1059
[java] Writes merged: 81764
[java] Sectors written: 662656
[java] Nr of milliseconds spent writing: 38236
[java] Nr of milliseconds spent doing I/O: 3060
[java] Start time: 1181904949219614000
[java] Time spent: 126079715000
[java] cpu0 %user: 32 %nice: 0 %system: 0 %iowait: 1 %idle: 66
[java] cpu1 %user: 68 %nice: 0 %system: 0 %iowait: 0 %idle: 31
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7633
[java] Reads merged: 3214
[java] Sectors read: 1834304
[java] Nr of milliseconds spent reading: 19700
[java] Writes completed: 35
[java] Writes merged: 19
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 120
[java] Nr of milliseconds spent doing I/O: 19456
[java] Start time: 1181905075301257000
[java] Time spent: 93880135000
[java] Device sdb
[java] Reads completed: 53
[java] Reads merged: 139
[java] Sectors read: 1568
```

```
[java] Nr of milliseconds spent reading: 328
[java] Writes completed: 147
[java] Writes merged: 2782
[java] Sectors written: 23456
[java] Nr of milliseconds spent writing: 1280
[java] Nr of milliseconds spent doing I/O: 320
[java] Start time: 1181905075301257000
[java] Time spent: 93880140000
[java] cpu0 %user: 80 %nice: 0 %system: 1 %iowait: 3 %idle: 14
[java] cpu1 %user: 31 %nice: 0 %system: 0 %iowait: 2 %idle: 66
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 8
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1181905169183275000
[java] Time spent: 126537498000
[java] Device sdb
[java] Reads completed: 50
[java] Reads merged: 96
[java] Sectors read: 1168
[java] Nr of milliseconds spent reading: 232
[java] Writes completed: 1058
[java] Writes merged: 83092
[java] Sectors written: 673248
[java] Nr of milliseconds spent writing: 38568
[java] Nr of milliseconds spent doing I/O: 3036
[java] Start time: 1181905169183275000
[java] Time spent: 126537500000
[java] cpu0 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 1 %idle: 97
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7651
[java] Reads merged: 3238
[java] Sectors read: 1833840
[java] Nr of milliseconds spent reading: 19984
[java] Writes completed: 35
[java] Writes merged: 19
```

```
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 132
[java] Nr of milliseconds spent doing I/O: 19632
[java] Start time: 1181905295722691000
[java] Time spent: 94684349000
[java] Device sdb
[java] Reads completed: 41
[java] Reads merged: 95
[java] Sectors read: 1144
[java] Nr of milliseconds spent reading: 164
[java] Writes completed: 131
[java] Writes merged: 1497
[java] Sectors written: 13024
[java] Nr of milliseconds spent writing: 628
[java] Nr of milliseconds spent doing I/O: 236
[java] Start time: 1181905295722691000
[java] Time spent: 94684348000
[java] cpu0 %user: 93 %nice: 0 %system: 1 %iowait: 3 %idle: 1
[java] cpu1 %user: 17 %nice: 0 %system: 0 %iowait: 2 %idle: 78
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 6
[java] Writes merged: 2
[java] Sectors written: 64
[java] Nr of milliseconds spent writing: 32
[java] Nr of milliseconds spent doing I/O: 24
[java] Start time: 1181905390408920000
[java] Time spent: 122327582000
[java] Device sdb
[java] Reads completed: 153
[java] Reads merged: 150
[java] Sectors read: 3528
[java] Nr of milliseconds spent reading: 1656
[java] Writes completed: 1055
[java] Writes merged: 80363
[java] Sectors written: 651400
[java] Nr of milliseconds spent writing: 50800
[java] Nr of milliseconds spent doing I/O: 3872
[java] Start time: 1181905390408920000
[java] Time spent: 122327578000
[java] cpu0 %user: 92 %nice: 0 %system: 0 %iowait: 0 %idle: 7
```

```
[java] cpu1 %user: 8 %nice: 0 %system: 0 %iowait: 2 %idle: 89
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7547
[java] Reads merged: 3224
[java] Sectors read: 1816368
[java] Nr of milliseconds spent reading: 20252
[java] Writes completed: 35
[java] Writes merged: 20
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 116
[java] Nr of milliseconds spent doing I/O: 19888
[java] Start time: 1181905512738224000
[java] Time spent: 95568366000
[java] Device sdb
[java] Reads completed: 39
[java] Reads merged: 102
[java] Sectors read: 1152
[java] Nr of milliseconds spent reading: 248
[java] Writes completed: 150
[java] Writes merged: 4551
[java] Sectors written: 37616
[java] Nr of milliseconds spent writing: 7596
[java] Nr of milliseconds spent doing I/O: 376
[java] Start time: 1181905512738224000
[java] Time spent: 95568373000
[java] cpu0 %user: 77 %nice: 0 %system: 1 %iowait: 4 %idle: 17
[java] cpu1 %user: 33 %nice: 0 %system: 0 %iowait: 2 %idle: 63
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 2
[java] Reads merged: 0
[java] Sectors read: 512
[java] Nr of milliseconds spent reading: 8
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1181905608308416000
[java] Time spent: 120812279000
[java] Device sdb
[java] Reads completed: 52
[java] Reads merged: 94
[java] Sectors read: 1224
```

```
[java] Nr of milliseconds spent reading: 240
[java] Writes completed: 1016
[java] Writes merged: 81331
[java] Sectors written: 658808
[java] Nr of milliseconds spent writing: 39680
[java] Nr of milliseconds spent doing I/O: 2808
[java] Start time: 1181905608308416000
[java] Time spent: 120812331000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] cpu1 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7664
[java] Reads merged: 3204
[java] Sectors read: 1831984
[java] Nr of milliseconds spent reading: 20260
[java] Writes completed: 37
[java] Writes merged: 20
[java] Sectors written: 456
[java] Nr of milliseconds spent writing: 84
[java] Nr of milliseconds spent doing I/O: 19980
[java] Start time: 1181905729122475000
[java] Time spent: 95819514000
[java] Device sdb
[java] Reads completed: 40
[java] Reads merged: 82
[java] Sectors read: 1032
[java] Nr of milliseconds spent reading: 440
[java] Writes completed: 176
[java] Writes merged: 3665
[java] Sectors written: 30752
[java] Nr of milliseconds spent writing: 1720
[java] Nr of milliseconds spent doing I/O: 544
[java] Start time: 1181905729122475000
[java] Time spent: 95819520000
[java] cpu0 %user: 53 %nice: 0 %system: 0 %iowait: 3 %idle: 42
[java] cpu1 %user: 56 %nice: 0 %system: 1 %iowait: 4 %idle: 37
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 3
[java] Writes merged: 1
```

```
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 8
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1181905824943792000
[java] Time spent: 121218065000
[java] Device sdb
[java] Reads completed: 249
[java] Reads merged: 176
[java] Sectors read: 8144
[java] Nr of milliseconds spent reading: 1316
[java] Writes completed: 1045
[java] Writes merged: 82509
[java] Sectors written: 668480
[java] Nr of milliseconds spent writing: 41344
[java] Nr of milliseconds spent doing I/O: 3768
[java] Start time: 1181905824943792000
[java] Time spent: 121218068000
[java] cpu0 %user: 85 %nice: 0 %system: 0 %iowait: 0 %idle: 13
[java] cpu1 %user: 14 %nice: 0 %system: 0 %iowait: 2 %idle: 83
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7538
[java] Reads merged: 3230
[java] Sectors read: 1806624
[java] Nr of milliseconds spent reading: 20436
[java] Writes completed: 37
[java] Writes merged: 21
[java] Sectors written: 464
[java] Nr of milliseconds spent writing: 188
[java] Nr of milliseconds spent doing I/O: 20092
[java] Start time: 1181905946163602000
[java] Time spent: 94968336000
[java] Device sdb
[java] Reads completed: 36
[java] Reads merged: 84
[java] Sectors read: 1016
[java] Nr of milliseconds spent reading: 116
[java] Writes completed: 153
[java] Writes merged: 2611
[java] Sectors written: 22112
[java] Nr of milliseconds spent writing: 988
[java] Nr of milliseconds spent doing I/O: 252
[java] Start time: 1181905946163602000
[java] Time spent: 94968341000
[java] cpu0 %user: 72 %nice: 0 %system: 1 %iowait: 3 %idle: 23
```

```
[java] cpu1 %user: 38 %nice: 0 %system: 0 %iowait: 0 %idle: 60
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 24
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1181906041133754000
[java] Time spent: 121273101000
[java] Device sdb
[java] Reads completed: 162
[java] Reads merged: 155
[java] Sectors read: 3368
[java] Nr of milliseconds spent reading: 1336
[java] Writes completed: 1047
[java] Writes merged: 80960
[java] Sectors written: 656104
[java] Nr of milliseconds spent writing: 44212
[java] Nr of milliseconds spent doing I/O: 3524
[java] Start time: 1181906041133754000
[java] Time spent: 121273104000
[java] cpu0 %user: 6 %nice: 0 %system: 0 %iowait: 3 %idle: 90
[java] cpu1 %user: 94 %nice: 0 %system: 0 %iowait: 0 %idle: 5
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7356
[java] Reads merged: 3063
[java] Sectors read: 1765536
[java] Nr of milliseconds spent reading: 20888
[java] Writes completed: 40
[java] Writes merged: 22
[java] Sectors written: 496
[java] Nr of milliseconds spent writing: 152
[java] Nr of milliseconds spent doing I/O: 20656
[java] Start time: 1181906162409207000
[java] Time spent: 95115147000
[java] Device sdb
[java] Reads completed: 86
[java] Reads merged: 145
[java] Sectors read: 2384
```

```
[java] Nr of milliseconds spent reading: 592
[java] Writes completed: 168
[java] Writes merged: 4220
[java] Sectors written: 35128
[java] Nr of milliseconds spent writing: 2084
[java] Nr of milliseconds spent doing I/O: 616
[java] Start time: 1181906162409207000
[java] Time spent: 95115155000
[java] cpu0 %user: 35 %nice: 0 %system: 0 %iowait: 4 %idle: 59
[java] cpu1 %user: 74 %nice: 0 %system: 1 %iowait: 4 %idle: 19
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 2
[java] Writes merged: 0
[java] Sectors written: 16
[java] Nr of milliseconds spent writing: 28
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1181906257526270000
[java] Time spent: 119667058000
[java] Device sdb
[java] Reads completed: 1853
[java] Reads merged: 4659
[java] Sectors read: 52840
[java] Nr of milliseconds spent reading: 8652
[java] Writes completed: 997
[java] Writes merged: 81532
[java] Sectors written: 660272
[java] Nr of milliseconds spent writing: 43488
[java] Nr of milliseconds spent doing I/O: 5888
[java] Start time: 1181906257526270000
[java] Time spent: 119667061000
[java] cpu0 %user: 29 %nice: 0 %system: 0 %iowait: 1 %idle: 68
[java] cpu1 %user: 70 %nice: 0 %system: 0 %iowait: 2 %idle: 26
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7143
[java] Reads merged: 3052
[java] Sectors read: 1717968
[java] Nr of milliseconds spent reading: 20828
[java] Writes completed: 33
[java] Writes merged: 20
```

```
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 128
[java] Nr of milliseconds spent doing I/O: 20556
[java] Start time: 1181906377195140000
[java] Time spent: 93673472000
[java] Device sdb
[java] Reads completed: 6
[java] Reads merged: 7
[java] Sectors read: 104
[java] Nr of milliseconds spent reading: 88
[java] Writes completed: 166
[java] Writes merged: 4028
[java] Sectors written: 33576
[java] Nr of milliseconds spent writing: 2000
[java] Nr of milliseconds spent doing I/O: 268
[java] Start time: 1181906377195140000
[java] Time spent: 93673470000
[java] cpu0 %user: 29 %nice: 0 %system: 0 %iowait: 3 %idle: 67
[java] cpu1 %user: 81 %nice: 0 %system: 1 %iowait: 2 %idle: 13
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 8
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1181906470870516000
[java] Time spent: 116213914000
[java] Device sdb
[java] Reads completed: 88
[java] Reads merged: 82
[java] Sectors read: 1376
[java] Nr of milliseconds spent reading: 344
[java] Writes completed: 1015
[java] Writes merged: 83651
[java] Sectors written: 677360
[java] Nr of milliseconds spent writing: 47040
[java] Nr of milliseconds spent doing I/O: 3104
[java] Start time: 1181906470870516000
[java] Time spent: 116213807000
[java] cpu0 %user: 93 %nice: 0 %system: 0 %iowait: 0 %idle: 6
```

```
[java] cpu1 %user: 7 %nice: 0 %system: 0 %iowait: 1 %idle: 91
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7235
[java] Reads merged: 3070
[java] Sectors read: 1741560
[java] Nr of milliseconds spent reading: 20496
[java] Writes completed: 33
[java] Writes merged: 20
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 72
[java] Nr of milliseconds spent doing I/O: 20240
[java] Start time: 1181906587085926000
[java] Time spent: 94314953000
[java] Device sdb
[java] Reads completed: 58
[java] Reads merged: 79
[java] Sectors read: 1104
[java] Nr of milliseconds spent reading: 256
[java] Writes completed: 158
[java] Writes merged: 1905
[java] Sectors written: 16536
[java] Nr of milliseconds spent writing: 568
[java] Nr of milliseconds spent doing I/O: 256
[java] Start time: 1181906587085926000
[java] Time spent: 94314949000
[java] cpu0 %user: 57 %nice: 0 %system: 1 %iowait: 3 %idle: 37
[java] cpu1 %user: 52 %nice: 0 %system: 0 %iowait: 2 %idle: 44
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 7
[java] Writes merged: 1
[java] Sectors written: 64
[java] Nr of milliseconds spent writing: 28
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1181906681402559000
[java] Time spent: 117494892000
[java] Device sdb
[java] Reads completed: 204
[java] Reads merged: 161
[java] Sectors read: 3984
```

```
[java] Nr of milliseconds spent reading: 1356
[java] Writes completed: 1065
[java] Writes merged: 84280
[java] Sectors written: 682784
[java] Nr of milliseconds spent writing: 46428
[java] Nr of milliseconds spent doing I/O: 3600
[java] Start time: 1181906681402559000
[java] Time spent: 117494887000
[java] cpu0 %user: 88 %nice: 0 %system: 0 %iowait: 0 %idle: 10
[java] cpu1 %user: 11 %nice: 0 %system: 0 %iowait: 1 %idle: 86
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7624
[java] Reads merged: 3177
[java] Sectors read: 1826640
[java] Nr of milliseconds spent reading: 19720
[java] Writes completed: 35
[java] Writes merged: 19
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 108
[java] Nr of milliseconds spent doing I/O: 19468
[java] Start time: 1181906798899030000
[java] Time spent: 94970417000
[java] Device sdb
[java] Reads completed: 63
[java] Reads merged: 80
[java] Sectors read: 1184
[java] Nr of milliseconds spent reading: 268
[java] Writes completed: 151
[java] Writes merged: 1412
[java] Sectors written: 12560
[java] Nr of milliseconds spent writing: 484
[java] Nr of milliseconds spent doing I/O: 272
[java] Start time: 1181906798899030000
[java] Time spent: 94970424000
[java] cpu0 %user: 92 %nice: 0 %system: 1 %iowait: 3 %idle: 1
[java] cpu1 %user: 18 %nice: 0 %system: 0 %iowait: 0 %idle: 81
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 2
```

```
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1181906893871169000
[java] Time spent: 122713249000
[java] Device sdb
[java] Reads completed: 92
[java] Reads merged: 87
[java] Sectors read: 1440
[java] Nr of milliseconds spent reading: 584
[java] Writes completed: 954
[java] Writes merged: 80702
[java] Sectors written: 653304
[java] Nr of milliseconds spent writing: 42048
[java] Nr of milliseconds spent doing I/O: 3104
[java] Start time: 1181906893871169000
[java] Time spent: 122713250000
[java] cpu0 %user: 28 %nice: 0 %system: 0 %iowait: 2 %idle: 70
[java] cpu1 %user: 72 %nice: 0 %system: 0 %iowait: 0 %idle: 26
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7877
[java] Reads merged: 3217
[java] Sectors read: 1884160
[java] Nr of milliseconds spent reading: 19092
[java] Writes completed: 36
[java] Writes merged: 20
[java] Sectors written: 448
[java] Nr of milliseconds spent writing: 88
[java] Nr of milliseconds spent doing I/O: 18792
[java] Start time: 1181907016586003000
[java] Time spent: 95721653000
[java] Device sdb
[java] Reads completed: 60
[java] Reads merged: 84
[java] Sectors read: 1208
[java] Nr of milliseconds spent reading: 264
[java] Writes completed: 173
[java] Writes merged: 4491
[java] Sectors written: 37336
[java] Nr of milliseconds spent writing: 2168
[java] Nr of milliseconds spent doing I/O: 352
[java] Start time: 1181907016586003000
[java] Time spent: 95721659000
[java] cpu0 %user: 87 %nice: 0 %system: 1 %iowait: 3 %idle: 6
```

```
[java] cpu1 %user: 21 %nice: 0 %system: 0 %iowait: 0 %idle: 77
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1181907112309336000
[java] Time spent: 128085095000
[java] Device sdb
[java] Reads completed: 91
[java] Reads merged: 89
[java] Sectors read: 1440
[java] Nr of milliseconds spent reading: 444
[java] Writes completed: 1001
[java] Writes merged: 80259
[java] Sectors written: 650128
[java] Nr of milliseconds spent writing: 36844
[java] Nr of milliseconds spent doing I/O: 3016
[java] Start time: 1181907112309336000
[java] Time spent: 128085104000
[java] cpu0 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 1 %idle: 97
[java] All files added
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 1718
[java] Reads merged: 719
[java] Sectors read: 408376
[java] Nr of milliseconds spent reading: 4092
[java] Writes completed: 9
[java] Writes merged: 5
[java] Sectors written: 112
[java] Nr of milliseconds spent writing: 52
[java] Nr of milliseconds spent doing I/O: 4056
[java] Start time: 1181907240396178000
[java] Time spent: 22059235000
[java] Device sdb
[java] Reads completed: 25
[java] Reads merged: 24
```

```
[java] Sectors read: 472
[java] Nr of milliseconds spent reading: 328
[java] Writes completed: 88
[java] Writes merged: 3437
[java] Sectors written: 28208
[java] Nr of milliseconds spent writing: 2000
[java] Nr of milliseconds spent doing I/O: 304
[java] Start time: 1181907240396178000
[java] Time spent: 22059241000
[java] cpu0 %user: 86 %nice: 0 %system: 1 %iowait: 10 %idle: 1
[java] cpu1 %user: 16 %nice: 0 %system: 0 %iowait: 0 %idle: 82
[java] done flushing last partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 2
[java] Writes merged: 1
[java] Sectors written: 24
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1181907262457083000
[java] Time spent: 20300905000
[java] Device sdb
[java] Reads completed: 4
[java] Reads merged: 0
[java] Sectors read: 32
[java] Nr of milliseconds spent reading: 112
[java] Writes completed: 217
[java] Writes merged: 17859
[java] Sectors written: 144624
[java] Nr of milliseconds spent writing: 10904
[java] Nr of milliseconds spent doing I/O: 684
[java] Start time: 1181907262457083000
[java] Time spent: 20300908000
[java] cpu0 %user: 70 %nice: 0 %system: 0 %iowait: 1 %idle: 24
[java] cpu1 %user: 27 %nice: 0 %system: 0 %iowait: 7 %idle: 66
[java] done merging all files
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 1
```

```
[java] Writes merged: 0
[java] Sectors written: 8
[java] Nr of milliseconds spent writing: 4
[java] Nr of milliseconds spent doing I/O: 4
[java] Start time: 1181907282763230000
[java] Time spent: 407938053000
[java] Device sdb
[java] Reads completed: 58831
[java] Reads merged: 26530
[java] Sectors read: 12372176
[java] Nr of milliseconds spent reading: 1835888
[java] Writes completed: 20210
[java] Writes merged: 1472074
[java] Sectors written: 11938512
[java] Nr of milliseconds spent writing: 10294668
[java] Nr of milliseconds spent doing I/O: 377344
[java] Start time: 1181907282763230000
[java] Time spent: 407938054000
[java] cpu0 %user: 35 %nice: 0 %system: 4 %iowait: 58 %idle: 1
[java] cpu1 %user: 2 %nice: 0 %system: 1 %iowait: 28 %idle: 67
[java] Device sda
[java] Reads completed: 142691
[java] Reads merged: 59398
[java] Sectors read: 34117496
[java] Nr of milliseconds spent reading: 365640
[java] Writes completed: 720
[java] Writes merged: 379
[java] Sectors written: 8792
[java] Nr of milliseconds spent writing: 2712
[java] Nr of milliseconds spent doing I/O: 361368
[java] Start time: 1181903283661596000
[java] Time spent: 4407202792000
[java] Device sdb
[java] Reads completed: 66553
[java] Reads merged: 43454
[java] Sectors read: 12588536
[java] Nr of milliseconds spent reading: 1880396
[java] Writes completed: 42909
[java] Writes merged: 3061424
[java] Sectors written: 24836128
[java] Nr of milliseconds spent writing: 11091700
[java] Nr of milliseconds spent doing I/O: 455500
[java] Start time: 1181903283661596000
[java] Time spent: 4407202519000
[java] cpu0 %user: 67 %nice: 0 %system: 1 %iowait: 7 %idle: 24
```

```
[java] cpu1 %user: 31 %nice: 0 %system: 0 %iowait: 4 %idle: 63
[java] Created the index in 73 minutes and 26 seconds.
[java] starting searches for small terms
[java] Searching for 8985741
[java] Searching for 3456645
[java] Searching for 5a90
[java] Searching for 316778
[java] Searching for x4262
[java] Searching for sc760e
[java] Searching for affectedthe
[java] Searching for d037968
[java] Searching for 190048660
[java] Searching for 164654
[java] Searching for nicololas
[java] Searching for ibbmec
[java] Searching for m11n12e
[java] Searching for hh227pc5
[java] Searching for rec18301
[java] Searching for 129x1
[java] Searching for 3002894512
[java] Searching for mar345remote
[java] Searching for fluoderm
[java] Searching for 57217907
[java] Searching for p5162345
[java] Searching for 2047593
[java] Searching for bragenski
[java] Searching for genequantification04
[java] Searching for t1654
[java] Searching for 880937
[java] Searching for meji
[java] Searching for geophyics
[java] Searching for frutex
[java] Searching for laszek
[java] Searching for fd011148
[java] Searching for smallparticles
[java] Searching for 210807
[java] Searching for sonorit
[java] Searching for daciformis
[java] Searching for blki
[java] Searching for tetraurelia
[java] Searching for bodyteen
[java] Searching for q030
[java] Searching for stipulatation
[java] Searching for hamelberg
[java] Searching for shozari
```

```
[java] Searching for 0614543
[java] Searching for miller99
[java] Searching for 279199
[java] Searching for 5176920
[java] Searching for 10076473
[java] Searching for hurger
[java] Searching for h2105
[java] Searching for nj04114
[java] Searching for 632088
[java] Searching for weblookup
[java] Searching for 1240616
[java] Searching for 1940218bg
[java] Searching for 858890
[java] Searching for vollbert
[java] Searching for pursuingbusiness
[java] Searching for subtartive
[java] Searching for ms9000e
[java] Searching for kichijoji
[java] Searching for 803194
[java] Searching for 790580
[java] Searching for 4588448
[java] Searching for 30490944
[java] Searching for nwwg4000200049
[java] Searching for 2581152
[java] Searching for 4127670
[java] Searching for otocalm
[java] Searching for pnu020
[java] Searching for 17166380
[java] Searching for 206400
[java] Searching for 19992833
[java] Searching for 8873154
[java] Searching for v90596
[java] Searching for mo00061
[java] Searching for 73ay404
[java] Searching for mdat40
[java] Searching for 0922e
[java] Searching for 14bs229
[java] Searching for lp00043725
[java] Searching for wcir023311800
[java] Searching for r0001
[java] Searching for regionallabor
[java] Searching for a786gent
[java] Searching for 5r01ca095662
[java] Searching for 1061002
[java] Searching for 2089079
```

```
[java] Searching for nrmap000100673
[java] Searching for 1802144lg
[java] Searching for 9649858
[java] Searching for 2000hnwx0005
[java] Searching for tn10231
[java] Searching for yfmz99
[java] Searching for nouve
[java] Searching for 812080100
[java] Searching for wdbda24c1ff092237
[java] Searching for monkeywrench
[java] Searching for 90et0170
[java] Searching for 12604751
[java] Searching for cedell
[java] Finished with small term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1181907710904669000
[java] Time spent: 3844100000
[java] Device sdb
[java] Reads completed: 548
[java] Reads merged: 173
[java] Sectors read: 12648
[java] Nr of milliseconds spent reading: 6184
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 3816
[java] Start time: 1181907710904669000
[java] Time spent: 3844100000
[java] cpu0 %user: 1 %nice: 0 %system: 0 %iowait: 94 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 15 %idle: 112
[java] starting searches for medium terms
[java] Searching for mailto
[java] Searching for europeans
[java] Searching for 22z
[java] Searching for 1181
```

```
[java] Searching for 91109
[java] Searching for hvn
[java] Searching for 1056
[java] Searching for postcard
[java] Searching for 6112
[java] Searching for antispam
[java] Searching for campylobacter
[java] Searching for asociaci
[java] Searching for jam
[java] Searching for phy
[java] Searching for shawn
[java] Searching for kerry
[java] Searching for rk
[java] Searching for frawley
[java] Searching for infantil
[java] Searching for meteorologists
[java] Searching for tutor
[java] Searching for seis
[java] Searching for 3042
[java] Searching for mindspring
[java] Searching for snack
[java] Searching for clemente
[java] Searching for guthrie
[java] Searching for underscores
[java] Searching for crowded
[java] Searching for infantil
[java] Searching for manipulating
[java] Searching for eli
[java] Searching for relativity
[java] Searching for 04z
[java] Searching for remuneration
[java] Searching for hugo
[java] Searching for empleo
[java] Searching for psychotherapy
[java] Searching for voa
[java] Searching for o3
[java] Searching for concentrates
[java] Searching for uty
[java] Searching for rectal
[java] Searching for qt
[java] Searching for 1134
[java] Searching for dove
[java] Searching for osage
[java] Searching for 1355
[java] Searching for 2121
```

```
[java] Searching for fbo
[java] Searching for stacked
[java] Searching for invocation
[java] Searching for sponge
[java] Searching for listeriosis
[java] Searching for cty
[java] Searching for wc3
[java] Searching for negotiators
[java] Searching for deliberative
[java] Searching for organizer
[java] Searching for underscores
[java] Searching for 24hr
[java] Searching for uphold
[java] Searching for dsc
[java] Searching for debated
[java] Searching for rck
[java] Searching for civilization
[java] Searching for 2048
[java] Searching for 1545
[java] Searching for banco
[java] Searching for causation
[java] Searching for aasa
[java] Searching for aasa
[java] Searching for manatee
[java] Searching for 22314
[java] Searching for diminishing
[java] Searching for refrigerators
[java] Searching for rp2
[java] Searching for crafted
[java] Searching for escort
[java] Searching for chung
[java] Searching for dewey
[java] Searching for 03e
[java] Searching for ninr
[java] Searching for 1431
[java] Searching for waller
[java] Searching for ep5
[java] Searching for peabody
[java] Searching for sponge
[java] Searching for posterior
[java] Searching for 6112
[java] Searching for 10z
[java] Searching for 2048
[java] Searching for vest
[java] Searching for foci
```

```
[java] Searching for 2310
[java] Searching for vlc
[java] Searching for negotiators
[java] Searching for 2350
[java] Searching for cryo
[java] Searching for 5800
[java] Finished with medium term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1181907714756951000
[java] Time spent: 7942245000
[java] Device sdb
[java] Reads completed: 1075
[java] Reads merged: 11
[java] Sectors read: 34952
[java] Nr of milliseconds spent reading: 7932
[java] Writes completed: 9
[java] Writes merged: 19
[java] Sectors written: 224
[java] Nr of milliseconds spent writing: 144
[java] Nr of milliseconds spent doing I/O: 7804
[java] Start time: 1181907714756951000
[java] Time spent: 7942247000
[java] cpu0 %user: 1 %nice: 0 %system: 0 %iowait: 96 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 0 %idle: 98
[java] starting searches for long terms
[java] Searching for research
[java] Searching for 3
[java] Searching for 10
[java] Searching for national
[java] Searching for health
[java] Searching for 2002
[java] Searching for 02
[java] Searching for will
[java] Searching for 18
[java] Searching for 15
[java] Searching for as
```

```
[java] Searching for c
[java] Searching for public
[java] Searching for 9
[java] Searching for n
[java] Searching for 30
[java] Searching for 4
[java] Searching for in
[java] Searching for been
[java] Searching for page
[java] Searching for state
[java] Searching for from
[java] Searching for or
[java] Searching for have
[java] Searching for 5
[java] Searching for may
[java] Searching for an
[java] Searching for e
[java] Searching for on
[java] Searching for would
[java] Searching for 8
[java] Searching for 2
[java] Searching for 0
[java] Searching for u
[java] Searching for to
[java] Searching for 00
[java] Searching for a
[java] Searching for was
[java] Searching for has
[java] Searching for m
[java] Searching for department
[java] Searching for 12
[java] Searching for data
[java] Searching for 16
[java] Searching for these
[java] Searching for 03
[java] Searching for if
[java] Searching for can
[java] Searching for with
[java] Searching for home
[java] Searching for not
[java] Searching for de
[java] Searching for of
[java] Searching for 01
[java] Searching for 11
[java] Searching for for
```

```
[java] Searching for is
[java] Searching for by
[java] Searching for other
[java] Searching for one
[java] Searching for html
[java] Searching for also
[java] Searching for that
[java] Searching for it
[java] Searching for this
[java] Searching for 1
[java] Searching for we
[java] Searching for be
[java] Searching for services
[java] Searching for 6
[java] Searching for program
[java] Searching for were
[java] Searching for their
[java] Searching for 14
[java] Searching for i
[java] Searching for gov
[java] Searching for 13
[java] Searching for new
[java] Searching for its
[java] Searching for information
[java] Searching for which
[java] Searching for they
[java] Searching for any
[java] Searching for you
[java] Searching for are
[java] Searching for at
[java] Searching for the
[java] Searching for b
[java] Searching for 2003
[java] Searching for and
[java] Searching for d
[java] Searching for use
[java] Searching for all
[java] Searching for no
[java] Searching for time
[java] Searching for 20
[java] Searching for 7
[java] Searching for more
[java] Searching for about
[java] Searching for s
[java] Finished with long term searches
```

```
     [java] Device sda
     [java] Reads completed: 0
     [java] Reads merged: 0
     [java] Sectors read: 0
     [java] Nr of milliseconds spent reading: 0
     [java] Writes completed: 0
     [java] Writes merged: 0
     [java] Sectors written: 0
     [java] Nr of milliseconds spent writing: 0
     [java] Nr of milliseconds spent doing I/O: 0
     [java] Start time: 1181907722712154000
     [java] Time spent: 31098478000
     [java] Device sdb
     [java] Reads completed: 14180
     [java] Reads merged: 6534
     [java] Sectors read: 3197760
     [java] Nr of milliseconds spent reading: 30136
     [java] Writes completed: 73
     [java] Writes merged: 49
     [java] Sectors written: 976
     [java] Nr of milliseconds spent writing: 2348
     [java] Nr of milliseconds spent doing I/O: 25476
     [java] Start time: 1181907722712154000
     [java] Time spent: 31098483000
     [java] cpu0 %user: 45 %nice: 0 %system: 8 %iowait: 47 %idle: 0
     [java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 5 %idle: 94

BUILD SUCCESSFUL
Total time: 74 minutes 51 seconds
```

B.2 Remerge with immediate merge

This log is from an experiment with similar characteristics as the previous section, except that we use remerge with immediate merge here.

```
Buildfile: build.xml

clean:
   [delete] Deleting directory /usr/brille/bin

build:
    [mkdir] Created dir: /usr/brille/bin
    [javac] Compiling 91 source files to /usr/brille/bin
```

```
     [copy] Copying 1 file to /usr/brille/bin

deploy:
      [jar] Building jar: /usr/brille/brille.jar

run:
     [java] The operating system is Linux
     [java] adding file /data/gov2-corpus
     [java] done with accumulation in memory
     [java] Device sda
     [java] Reads completed: 9800
     [java] Reads merged: 3919
     [java] Sectors read: 2310176
     [java] Nr of milliseconds spent reading: 15132
     [java] Writes completed: 38
     [java] Writes merged: 22
     [java] Sectors written: 480
     [java] Nr of milliseconds spent writing: 192
     [java] Nr of milliseconds spent doing I/O: 14872
     [java] Start time: 1185208602099209000
     [java] Time spent: 95382157000
     [java] Device sdb
     [java] Reads completed: 159
     [java] Reads merged: 194
     [java] Sectors read: 7704
     [java] Nr of milliseconds spent reading: 856
     [java] Writes completed: 331
     [java] Writes merged: 1246
     [java] Sectors written: 12648
     [java] Nr of milliseconds spent writing: 2384
     [java] Nr of milliseconds spent doing I/O: 704
     [java] Start time: 1185208602099209000
     [java] Time spent: 95382141000
     [java] cpu0 %user: 75 %nice: 0 %system: 1 %iowait: 1 %idle: 21
     [java] cpu1 %user: 39 %nice: 0 %system: 0 %iowait: 0 %idle: 60
     [java] done flushing partial dict
     [java] Device sda
     [java] Reads completed: 0
     [java] Reads merged: 0
     [java] Sectors read: 0
     [java] Nr of milliseconds spent reading: 0
     [java] Writes completed: 4
     [java] Writes merged: 1
     [java] Sectors written: 40
     [java] Nr of milliseconds spent writing: 16
```

```
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1185208697485429000
[java] Time spent: 132067311000
[java] Device sdb
[java] Reads completed: 10
[java] Reads merged: 0
[java] Sectors read: 80
[java] Nr of milliseconds spent reading: 588
[java] Writes completed: 1661
[java] Writes merged: 82597
[java] Sectors written: 674096
[java] Nr of milliseconds spent writing: 22028
[java] Nr of milliseconds spent doing I/O: 2624
[java] Start time: 1185208697485429000
[java] Time spent: 132067308000
[java] cpu0 %user: 3 %nice: 0 %system: 0 %iowait: 2 %idle: 93
[java] cpu1 %user: 97 %nice: 0 %system: 0 %iowait: 0 %idle: 2
[java] done merging one file. 64346 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185208829563237000
[java] Time spent: 10629309000
[java] Device sdb
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 62
[java] Writes merged: 1369
[java] Sectors written: 11448
[java] Nr of milliseconds spent writing: 244
[java] Nr of milliseconds spent doing I/O: 44
[java] Start time: 1185208829563237000
[java] Time spent: 10629227000
[java] cpu0 %user: 3 %nice: 0 %system: 0 %iowait: 0 %idle: 92
[java] cpu1 %user: 96 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done with accumulation in memory
```

```
[java] Device sda
[java] Reads completed: 8655
[java] Reads merged: 3576
[java] Sectors read: 2051600
[java] Nr of milliseconds spent reading: 15300
[java] Writes completed: 36
[java] Writes merged: 19
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 84
[java] Nr of milliseconds spent doing I/O: 15112
[java] Start time: 1185208840192546000
[java] Time spent: 93629252000
[java] Device sdb
[java] Reads completed: 48
[java] Reads merged: 0
[java] Sectors read: 496
[java] Nr of milliseconds spent reading: 488
[java] Writes completed: 618
[java] Writes merged: 2141
[java] Sectors written: 22072
[java] Nr of milliseconds spent writing: 648
[java] Nr of milliseconds spent doing I/O: 424
[java] Start time: 1185208840192464000
[java] Time spent: 93629213000
[java] cpu0 %user: 97 %nice: 0 %system: 1 %iowait: 0 %idle: 2
[java] cpu1 %user: 3 %nice: 0 %system: 0 %iowait: 0 %idle: 97
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 2
[java] Reads merged: 0
[java] Sectors read: 512
[java] Nr of milliseconds spent reading: 8
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1185208933823044000
[java] Time spent: 122945790000
[java] Device sdb
[java] Reads completed: 53
[java] Reads merged: 41
[java] Sectors read: 1040
[java] Nr of milliseconds spent reading: 1072
[java] Writes completed: 1919
```

```
[java] Writes merged: 94975
[java] Sectors written: 775208
[java] Nr of milliseconds spent writing: 34176
[java] Nr of milliseconds spent doing I/O: 4444
[java] Start time: 1185208933823044000
[java] Time spent: 122945794000
[java] cpu0 %user: 62 %nice: 0 %system: 0 %iowait: 1 %idle: 35
[java] cpu1 %user: 37 %nice: 0 %system: 0 %iowait: 0 %idle: 61
[java] done merging all files. 118820 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185209056772024000
[java] Time spent: 34414311000
[java] Device sdb
[java] Reads completed: 5361
[java] Reads merged: 2635
[java] Sectors read: 1279032
[java] Nr of milliseconds spent reading: 52808
[java] Writes completed: 2511
[java] Writes merged: 194156
[java] Sectors written: 1573416
[java] Nr of milliseconds spent writing: 647608
[java] Nr of milliseconds spent doing I/O: 29176
[java] Start time: 1185209056772024000
[java] Time spent: 34414309000
[java] cpu0 %user: 43 %nice: 0 %system: 5 %iowait: 48 %idle: 2
[java] cpu1 %user: 2 %nice: 0 %system: 3 %iowait: 24 %idle: 67
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 8353
[java] Reads merged: 3277
[java] Sectors read: 1972704
[java] Nr of milliseconds spent reading: 19156
[java] Writes completed: 34
[java] Writes merged: 19
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 228
```

```
[java] Nr of milliseconds spent doing I/O: 18996
[java] Start time: 1185209091186335000
[java] Time spent: 90532556000
[java] Device sdb
[java] Reads completed: 161
[java] Reads merged: 200
[java] Sectors read: 3680
[java] Nr of milliseconds spent reading: 2312
[java] Writes completed: 673
[java] Writes merged: 4269
[java] Sectors written: 39536
[java] Nr of milliseconds spent writing: 4020
[java] Nr of milliseconds spent doing I/O: 1296
[java] Start time: 1185209091186333000
[java] Time spent: 90532537000
[java] cpu0 %user: 95 %nice: 0 %system: 0 %iowait: 1 %idle: 4
[java] cpu1 %user: 1 %nice: 0 %system: 0 %iowait: 0 %idle: 99
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1185209181721066000
[java] Time spent: 123471423000
[java] Device sdb
[java] Reads completed: 48
[java] Reads merged: 107
[java] Sectors read: 1240
[java] Nr of milliseconds spent reading: 732
[java] Writes completed: 1518
[java] Writes merged: 83626
[java] Sectors written: 681192
[java] Nr of milliseconds spent writing: 23952
[java] Nr of milliseconds spent doing I/O: 3172
[java] Start time: 1185209181721066000
[java] Time spent: 123471439000
[java] cpu0 %user: 5 %nice: 0 %system: 0 %iowait: 2 %idle: 91
[java] cpu1 %user: 95 %nice: 0 %system: 0 %iowait: 0 %idle: 4
[java] done merging all files. 174050 docs searchable
```

```
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185209305195125000
[java] Time spent: 46264382000
[java] Device sdb
[java] Reads completed: 7726
[java] Reads merged: 5672
[java] Sectors read: 1932224
[java] Nr of milliseconds spent reading: 58552
[java] Writes completed: 3310
[java] Writes merged: 242177
[java] Sectors written: 1963920
[java] Nr of milliseconds spent writing: 714960
[java] Nr of milliseconds spent doing I/O: 36784
[java] Start time: 1185209305195125000
[java] Time spent: 46264386000
[java] cpu0 %user: 45 %nice: 0 %system: 7 %iowait: 46 %idle: 2
[java] cpu1 %user: 3 %nice: 0 %system: 2 %iowait: 28 %idle: 66
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 8670
[java] Reads merged: 3491
[java] Sectors read: 2047592
[java] Nr of milliseconds spent reading: 18908
[java] Writes completed: 33
[java] Writes merged: 20
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 184
[java] Nr of milliseconds spent doing I/O: 18696
[java] Start time: 1185209351459507000
[java] Time spent: 91350652000
[java] Device sdb
[java] Reads completed: 81
[java] Reads merged: 132
[java] Sectors read: 1856
[java] Nr of milliseconds spent reading: 1120
[java] Writes completed: 556
```

```
[java] Writes merged: 6385
[java] Sectors written: 55536
[java] Nr of milliseconds spent writing: 4008
[java] Nr of milliseconds spent doing I/O: 812
[java] Start time: 1185209351459511000
[java] Time spent: 91350647000
[java] cpu0 %user: 75 %nice: 0 %system: 0 %iowait: 0 %idle: 25
[java] cpu1 %user: 4 %nice: 0 %system: 0 %iowait: 0 %idle: 96
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 24
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1185209442812171000
[java] Time spent: 123293204000
[java] Device sdb
[java] Reads completed: 65
[java] Reads merged: 137
[java] Sectors read: 1616
[java] Nr of milliseconds spent reading: 632
[java] Writes completed: 1035
[java] Writes merged: 81819
[java] Sectors written: 662912
[java] Nr of milliseconds spent writing: 41712
[java] Nr of milliseconds spent doing I/O: 3304
[java] Start time: 1185209442812171000
[java] Time spent: 123293208000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] cpu1 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done merging all files. 229182 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
```

```
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185209566107333000
[java] Time spent: 59706216000
[java] Device sdb
[java] Reads completed: 10380
[java] Reads merged: 7871
[java] Sectors read: 2608816
[java] Nr of milliseconds spent reading: 64256
[java] Writes completed: 3913
[java] Writes merged: 306729
[java] Sectors written: 2485176
[java] Nr of milliseconds spent writing: 1989216
[java] Nr of milliseconds spent doing I/O: 44496
[java] Start time: 1185209566107333000
[java] Time spent: 59706221000
[java] cpu0 %user: 45 %nice: 0 %system: 7 %iowait: 45 %idle: 1
[java] cpu1 %user: 3 %nice: 0 %system: 2 %iowait: 38 %idle: 57
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7810
[java] Reads merged: 3143
[java] Sectors read: 1842296
[java] Nr of milliseconds spent reading: 19452
[java] Writes completed: 34
[java] Writes merged: 19
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 180
[java] Nr of milliseconds spent doing I/O: 19268
[java] Start time: 1185209625813549000
[java] Time spent: 88783449000
[java] Device sdb
[java] Reads completed: 625
[java] Reads merged: 1654
[java] Sectors read: 21152
[java] Nr of milliseconds spent reading: 6152
[java] Writes completed: 452
[java] Writes merged: 21850
[java] Sectors written: 178424
[java] Nr of milliseconds spent writing: 199648
[java] Nr of milliseconds spent doing I/O: 4188
[java] Start time: 1185209625813554000
[java] Time spent: 88783463000
[java] cpu0 %user: 97 %nice: 0 %system: 0 %iowait: 1 %idle: 2
[java] cpu1 %user: 27 %nice: 0 %system: 0 %iowait: 0 %idle: 73
[java] done flushing partial dict
```

```
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 2
[java] Writes merged: 0
[java] Sectors written: 16
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1185209714598997000
[java] Time spent: 124281380000
[java] Device sdb
[java] Reads completed: 595
[java] Reads merged: 1207
[java] Sectors read: 18472
[java] Nr of milliseconds spent reading: 3972
[java] Writes completed: 999
[java] Writes merged: 83006
[java] Sectors written: 672096
[java] Nr of milliseconds spent writing: 40784
[java] Nr of milliseconds spent doing I/O: 4784
[java] Start time: 1185209714598997000
[java] Time spent: 124281384000
[java] cpu0 %user: 49 %nice: 0 %system: 0 %iowait: 2 %idle: 47
[java] cpu1 %user: 50 %nice: 0 %system: 0 %iowait: 0 %idle: 49
[java] done merging all files. 284815 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185209838882779000
[java] Time spent: 71882772000
[java] Device sdb
[java] Reads completed: 13200
[java] Reads merged: 8856
[java] Sectors read: 3285784
[java] Nr of milliseconds spent reading: 72224
[java] Writes completed: 5155
```

```
[java] Writes merged: 401177
[java] Sectors written: 3250688
[java] Nr of milliseconds spent writing: 2728744
[java] Nr of milliseconds spent doing I/O: 53940
[java] Start time: 1185209838882779000
[java] Time spent: 71882798000
[java] cpu0 %user: 46 %nice: 0 %system: 7 %iowait: 43 %idle: 2
[java] cpu1 %user: 3 %nice: 0 %system: 2 %iowait: 38 %idle: 55
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7582
[java] Reads merged: 3162
[java] Sectors read: 1803208
[java] Nr of milliseconds spent reading: 17940
[java] Writes completed: 30
[java] Writes merged: 15
[java] Sectors written: 360
[java] Nr of milliseconds spent writing: 84
[java] Nr of milliseconds spent doing I/O: 17588
[java] Start time: 1185209910765551000
[java] Time spent: 87801577000
[java] Device sdb
[java] Reads completed: 49
[java] Reads merged: 92
[java] Sectors read: 1176
[java] Nr of milliseconds spent reading: 552
[java] Writes completed: 266
[java] Writes merged: 8148
[java] Sectors written: 67328
[java] Nr of milliseconds spent writing: 11916
[java] Nr of milliseconds spent doing I/O: 632
[java] Start time: 1185209910765577000
[java] Time spent: 87801556000
[java] cpu0 %user: 73 %nice: 0 %system: 0 %iowait: 0 %idle: 27
[java] cpu1 %user: 15 %nice: 0 %system: 0 %iowait: 0 %idle: 85
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 2
[java] Reads merged: 0
[java] Sectors read: 512
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 1
[java] Writes merged: 0
[java] Sectors written: 8
[java] Nr of milliseconds spent writing: 4
```

```
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1185209998569067000
[java] Time spent: 123737521000
[java] Device sdb
[java] Reads completed: 55
[java] Reads merged: 101
[java] Sectors read: 1280
[java] Nr of milliseconds spent reading: 244
[java] Writes completed: 976
[java] Writes merged: 82267
[java] Sectors written: 665968
[java] Nr of milliseconds spent writing: 43136
[java] Nr of milliseconds spent doing I/O: 2992
[java] Start time: 1185209998569067000
[java] Time spent: 123737516000
[java] cpu0 %user: 9 %nice: 0 %system: 0 %iowait: 2 %idle: 87
[java] cpu1 %user: 90 %nice: 0 %system: 0 %iowait: 0 %idle: 8
[java] done merging all files. 340196 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185210122308402000
[java] Time spent: 88786130000
[java] Device sdb
[java] Reads completed: 15660
[java] Reads merged: 10203
[java] Sectors read: 3939048
[java] Nr of milliseconds spent reading: 81444
[java] Writes completed: 5617
[java] Writes merged: 481040
[java] Sectors written: 3893272
[java] Nr of milliseconds spent writing: 3240296
[java] Nr of milliseconds spent doing I/O: 61492
[java] Start time: 1185210122308402000
[java] Time spent: 88786133000
[java] cpu0 %user: 43 %nice: 0 %system: 7 %iowait: 46 %idle: 2
[java] cpu1 %user: 4 %nice: 0 %system: 3 %iowait: 42 %idle: 49
[java] done with accumulation in memory
```

```
[java] Device sda
[java] Reads completed: 7689
[java] Reads merged: 3164
[java] Sectors read: 1822736
[java] Nr of milliseconds spent reading: 17360
[java] Writes completed: 33
[java] Writes merged: 18
[java] Sectors written: 408
[java] Nr of milliseconds spent writing: 176
[java] Nr of milliseconds spent doing I/O: 17088
[java] Start time: 1185210211094532000
[java] Time spent: 87684366000
[java] Device sdb
[java] Reads completed: 62
[java] Reads merged: 114
[java] Sectors read: 1456
[java] Nr of milliseconds spent reading: 616
[java] Writes completed: 205
[java] Writes merged: 10457
[java] Sectors written: 85304
[java] Nr of milliseconds spent writing: 15640
[java] Nr of milliseconds spent doing I/O: 816
[java] Start time: 1185210211094535000
[java] Time spent: 87684370000
[java] cpu0 %user: 70 %nice: 0 %system: 0 %iowait: 1 %idle: 29
[java] cpu1 %user: 21 %nice: 0 %system: 0 %iowait: 0 %idle: 79
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1185210298780757000
[java] Time spent: 124271753000
[java] Device sdb
[java] Reads completed: 155
[java] Reads merged: 165
[java] Sectors read: 3600
[java] Nr of milliseconds spent reading: 1384
[java] Writes completed: 1053
```

```
[java] Writes merged: 80211
[java] Sectors written: 650160
[java] Nr of milliseconds spent writing: 40172
[java] Nr of milliseconds spent doing I/O: 3636
[java] Start time: 1185210298780757000
[java] Time spent: 124271755000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] cpu1 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done merging all files. 396473 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185210423054178000
[java] Time spent: 99973649000
[java] Device sdb
[java] Reads completed: 18362
[java] Reads merged: 11355
[java] Sectors read: 4598272
[java] Nr of milliseconds spent reading: 93364
[java] Writes completed: 7152
[java] Writes merged: 567520
[java] Sectors written: 4597424
[java] Nr of milliseconds spent writing: 3881660
[java] Nr of milliseconds spent doing I/O: 72460
[java] Start time: 1185210423054178000
[java] Time spent: 99973651000
[java] cpu0 %user: 46 %nice: 0 %system: 7 %iowait: 43 %idle: 1
[java] cpu1 %user: 3 %nice: 0 %system: 3 %iowait: 48 %idle: 44
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7550
[java] Reads merged: 3171
[java] Sectors read: 1798432
[java] Nr of milliseconds spent reading: 17584
[java] Writes completed: 32
[java] Writes merged: 16
[java] Sectors written: 384
[java] Nr of milliseconds spent writing: 136
```

```
[java] Nr of milliseconds spent doing I/O: 17288
[java] Start time: 1185210523027827000
[java] Time spent: 87108375000
[java] Device sdb
[java] Reads completed: 50
[java] Reads merged: 95
[java] Sectors read: 1200
[java] Nr of milliseconds spent reading: 476
[java] Writes completed: 222
[java] Writes merged: 6681
[java] Sectors written: 55240
[java] Nr of milliseconds spent writing: 7796
[java] Nr of milliseconds spent doing I/O: 600
[java] Start time: 1185210523027829000
[java] Time spent: 87108381000
[java] cpu0 %user: 71 %nice: 0 %system: 0 %iowait: 1 %idle: 28
[java] cpu1 %user: 1 %nice: 0 %system: 0 %iowait: 0 %idle: 99
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1185210610138061000
[java] Time spent: 123543199000
[java] Device sdb
[java] Reads completed: 1734
[java] Reads merged: 4871
[java] Sectors read: 53264
[java] Nr of milliseconds spent reading: 8404
[java] Writes completed: 1023
[java] Writes merged: 83004
[java] Sectors written: 672256
[java] Nr of milliseconds spent writing: 44332
[java] Nr of milliseconds spent doing I/O: 6200
[java] Start time: 1185210610138061000
[java] Time spent: 123543201000
[java] cpu0 %user: 1 %nice: 0 %system: 0 %iowait: 3 %idle: 94
[java] cpu1 %user: 98 %nice: 0 %system: 0 %iowait: 1 %idle: 0
[java] done merging all files. 451485 docs searchable
```

```
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185210733682963000
[java] Time spent: 115693702000
[java] Device sdb
[java] Reads completed: 21012
[java] Reads merged: 14075
[java] Sectors read: 5259808
[java] Nr of milliseconds spent reading: 100228
[java] Writes completed: 7666
[java] Writes merged: 647440
[java] Sectors written: 5210264
[java] Nr of milliseconds spent writing: 4225936
[java] Nr of milliseconds spent doing I/O: 79736
[java] Start time: 1185210733682963000
[java] Time spent: 115693705000
[java] cpu0 %user: 44 %nice: 0 %system: 8 %iowait: 45 %idle: 2
[java] cpu1 %user: 4 %nice: 0 %system: 2 %iowait: 47 %idle: 45
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7697
[java] Reads merged: 3200
[java] Sectors read: 1834688
[java] Nr of milliseconds spent reading: 16844
[java] Writes completed: 34
[java] Writes merged: 17
[java] Sectors written: 408
[java] Nr of milliseconds spent writing: 152
[java] Nr of milliseconds spent doing I/O: 16592
[java] Start time: 1185210849376665000
[java] Time spent: 87732838000
[java] Device sdb
[java] Reads completed: 165
[java] Reads merged: 190
[java] Sectors read: 3912
[java] Nr of milliseconds spent reading: 1716
[java] Writes completed: 335
```

```
[java] Writes merged: 5395
[java] Sectors written: 76472
[java] Nr of milliseconds spent writing: 48404
[java] Nr of milliseconds spent doing I/O: 1292
[java] Start time: 1185210849376668000
[java] Time spent: 87732878000
[java] cpu0 %user: 73 %nice: 0 %system: 0 %iowait: 1 %idle: 26
[java] cpu1 %user: 25 %nice: 0 %system: 0 %iowait: 0 %idle: 75
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1185210937111355000
[java] Time spent: 124243372000
[java] Device sdb
[java] Reads completed: 84
[java] Reads merged: 25
[java] Sectors read: 1352
[java] Nr of milliseconds spent reading: 1012
[java] Writes completed: 993
[java] Writes merged: 81325
[java] Sectors written: 658568
[java] Nr of milliseconds spent writing: 40716
[java] Nr of milliseconds spent doing I/O: 3272
[java] Start time: 1185210937111355000
[java] Time spent: 124243429000
[java] cpu0 %user: 89 %nice: 0 %system: 0 %iowait: 0 %idle: 10
[java] cpu1 %user: 11 %nice: 0 %system: 0 %iowait: 1 %idle: 87
[java] done merging all files. 507239 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
```

```
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185211061356455000
[java] Time spent: 127648328000
[java] Device sdb
[java] Reads completed: 23747
[java] Reads merged: 16936
[java] Sectors read: 5915800
[java] Nr of milliseconds spent reading: 114344
[java] Writes completed: 9021
[java] Writes merged: 722846
[java] Sectors written: 5855016
[java] Nr of milliseconds spent writing: 4792040
[java] Nr of milliseconds spent doing I/O: 92152
[java] Start time: 1185211061356455000
[java] Time spent: 127648330000
[java] cpu0 %user: 45 %nice: 0 %system: 8 %iowait: 44 %idle: 2
[java] cpu1 %user: 5 %nice: 0 %system: 2 %iowait: 49 %idle: 42
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7750
[java] Reads merged: 3220
[java] Sectors read: 1834608
[java] Nr of milliseconds spent reading: 17384
[java] Writes completed: 33
[java] Writes merged: 19
[java] Sectors written: 416
[java] Nr of milliseconds spent writing: 116
[java] Nr of milliseconds spent doing I/O: 17088
[java] Start time: 1185211189004783000
[java] Time spent: 88771772000
[java] Device sdb
[java] Reads completed: 91
[java] Reads merged: 81
[java] Sectors read: 1448
[java] Nr of milliseconds spent reading: 700
[java] Writes completed: 341
[java] Writes merged: 12493
[java] Sectors written: 102688
[java] Nr of milliseconds spent writing: 23116
[java] Nr of milliseconds spent doing I/O: 912
[java] Start time: 1185211189004785000
[java] Time spent: 88771779000
[java] cpu0 %user: 73 %nice: 0 %system: 0 %iowait: 0 %idle: 27
[java] cpu1 %user: 25 %nice: 0 %system: 0 %iowait: 0 %idle: 75
[java] done flushing partial dict
```

```
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 2
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1185211277778434000
[java] Time spent: 119998957000
[java] Device sdb
[java] Reads completed: 108
[java] Reads merged: 68
[java] Sectors read: 1432
[java] Nr of milliseconds spent reading: 788
[java] Writes completed: 1027
[java] Writes merged: 81484
[java] Sectors written: 660128
[java] Nr of milliseconds spent writing: 43748
[java] Nr of milliseconds spent doing I/O: 3300
[java] Start time: 1185211277778434000
[java] Time spent: 119998959000
[java] cpu0 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 3 %idle: 95
[java] done merging all files. 560270 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185211397779082000
[java] Time spent: 141453946000
[java] Device sdb
[java] Reads completed: 26573
[java] Reads merged: 14848
[java] Sectors read: 6575744
[java] Nr of milliseconds spent reading: 123244
[java] Writes completed: 9445
```

```
[java] Writes merged: 785864
[java] Sectors written: 6362544
[java] Nr of milliseconds spent writing: 5357048
[java] Nr of milliseconds spent doing I/O: 100228
[java] Start time: 1185211397779082000
[java] Time spent: 141453949000
[java] cpu0 %user: 45 %nice: 0 %system: 8 %iowait: 44 %idle: 1
[java] cpu1 %user: 3 %nice: 0 %system: 3 %iowait: 49 %idle: 43
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7652
[java] Reads merged: 3181
[java] Sectors read: 1816768
[java] Nr of milliseconds spent reading: 17520
[java] Writes completed: 33
[java] Writes merged: 19
[java] Sectors written: 416
[java] Nr of milliseconds spent writing: 140
[java] Nr of milliseconds spent doing I/O: 17224
[java] Start time: 1185211539233028000
[java] Time spent: 89543329000
[java] Device sdb
[java] Reads completed: 66
[java] Reads merged: 77
[java] Sectors read: 1216
[java] Nr of milliseconds spent reading: 268
[java] Writes completed: 491
[java] Writes merged: 31421
[java] Sectors written: 255328
[java] Nr of milliseconds spent writing: 146264
[java] Nr of milliseconds spent doing I/O: 1756
[java] Start time: 1185211539233031000
[java] Time spent: 89543334000
[java] cpu0 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 1
[java] cpu1 %user: 16 %nice: 0 %system: 0 %iowait: 0 %idle: 84
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 2
[java] Reads merged: 0
[java] Sectors read: 512
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 2
[java] Writes merged: 0
[java] Sectors written: 16
[java] Nr of milliseconds spent writing: 8
```

```
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1185211628778153000
[java] Time spent: 118569464000
[java] Device sdb
[java] Reads completed: 88
[java] Reads merged: 100
[java] Sectors read: 1504
[java] Nr of milliseconds spent reading: 548
[java] Writes completed: 1079
[java] Writes merged: 83707
[java] Sectors written: 678336
[java] Nr of milliseconds spent writing: 44028
[java] Nr of milliseconds spent doing I/O: 3372
[java] Start time: 1185211628778153000
[java] Time spent: 118569469000
[java] cpu0 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] done merging all files. 612522 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185211747349286000
[java] Time spent: 149847411000
[java] Device sdb
[java] Reads completed: 29242
[java] Reads merged: 16272
[java] Sectors read: 7231440
[java] Nr of milliseconds spent reading: 128592
[java] Writes completed: 11653
[java] Writes merged: 890829
[java] Sectors written: 7193120
[java] Nr of milliseconds spent writing: 6348524
[java] Nr of milliseconds spent doing I/O: 112568
[java] Start time: 1185211747349286000
[java] Time spent: 149847416000
[java] cpu0 %user: 46 %nice: 0 %system: 8 %iowait: 42 %idle: 2
[java] cpu1 %user: 4 %nice: 0 %system: 2 %iowait: 43 %idle: 49
[java] done with accumulation in memory
```

```
[java] Device sda
[java] Reads completed: 7743
[java] Reads merged: 3198
[java] Sectors read: 1832192
[java] Nr of milliseconds spent reading: 17480
[java] Writes completed: 36
[java] Writes merged: 19
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 128
[java] Nr of milliseconds spent doing I/O: 17220
[java] Start time: 1185211897196697000
[java] Time spent: 88771077000
[java] Device sdb
[java] Reads completed: 68
[java] Reads merged: 109
[java] Sectors read: 1488
[java] Nr of milliseconds spent reading: 1140
[java] Writes completed: 283
[java] Writes merged: 6416
[java] Sectors written: 80424
[java] Nr of milliseconds spent writing: 46700
[java] Nr of milliseconds spent doing I/O: 912
[java] Start time: 1185211897196702000
[java] Time spent: 88771079000
[java] cpu0 %user: 86 %nice: 0 %system: 1 %iowait: 0 %idle: 13
[java] cpu1 %user: 16 %nice: 0 %system: 0 %iowait: 0 %idle: 84
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 4
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1185211985969809000
[java] Time spent: 119078061000
[java] Device sdb
[java] Reads completed: 189
[java] Reads merged: 196
[java] Sectors read: 4120
[java] Nr of milliseconds spent reading: 1972
[java] Writes completed: 1073
```

```
[java] Writes merged: 82053
[java] Sectors written: 665056
[java] Nr of milliseconds spent writing: 40708
[java] Nr of milliseconds spent doing I/O: 3680
[java] Start time: 1185211985969809000
[java] Time spent: 119078057000
[java] cpu0 %user: 5 %nice: 0 %system: 0 %iowait: 3 %idle: 91
[java] cpu1 %user: 95 %nice: 0 %system: 0 %iowait: 0 %idle: 4
[java] done merging all files. 665057 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185212105049579000
[java] Time spent: 167244881000
[java] Device sdb
[java] Reads completed: 32098
[java] Reads merged: 21561
[java] Sectors read: 7892728
[java] Nr of milliseconds spent reading: 145320
[java] Writes completed: 12829
[java] Writes merged: 967683
[java] Sectors written: 7844264
[java] Nr of milliseconds spent writing: 6863212
[java] Nr of milliseconds spent doing I/O: 126372
[java] Start time: 1185212105049579000
[java] Time spent: 167244884000
[java] cpu0 %user: 43 %nice: 0 %system: 8 %iowait: 44 %idle: 2
[java] cpu1 %user: 5 %nice: 0 %system: 2 %iowait: 47 %idle: 44
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7648
[java] Reads merged: 3204
[java] Sectors read: 1807200
[java] Nr of milliseconds spent reading: 17348
[java] Writes completed: 35
[java] Writes merged: 19
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 144
```

```
[java] Nr of milliseconds spent doing I/O: 17004
[java] Start time: 1185212272294460000
[java] Time spent: 87822168000
[java] Device sdb
[java] Reads completed: 29
[java] Reads merged: 25
[java] Sectors read: 504
[java] Nr of milliseconds spent reading: 520
[java] Writes completed: 259
[java] Writes merged: 11241
[java] Sectors written: 92032
[java] Nr of milliseconds spent writing: 19548
[java] Nr of milliseconds spent doing I/O: 788
[java] Start time: 1185212272294463000
[java] Time spent: 87822172000
[java] cpu0 %user: 70 %nice: 0 %system: 0 %iowait: 1 %idle: 29
[java] cpu1 %user: 2 %nice: 0 %system: 0 %iowait: 0 %idle: 98
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1185212360118342000
[java] Time spent: 119308315000
[java] Device sdb
[java] Reads completed: 46
[java] Reads merged: 0
[java] Sectors read: 368
[java] Nr of milliseconds spent reading: 556
[java] Writes completed: 1076
[java] Writes merged: 82200
[java] Sectors written: 666248
[java] Nr of milliseconds spent writing: 42712
[java] Nr of milliseconds spent doing I/O: 3196
[java] Start time: 1185212360118342000
[java] Time spent: 119308319000
[java] cpu0 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 1 %idle: 98
[java] done merging all files. 717628 docs searchable
```

```
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185212479428907000
[java] Time spent: 184797743000
[java] Device sdb
[java] Reads completed: 34784
[java] Reads merged: 23343
[java] Sectors read: 8549872
[java] Nr of milliseconds spent reading: 162008
[java] Writes completed: 13570
[java] Writes merged: 1050893
[java] Sectors written: 8515856
[java] Nr of milliseconds spent writing: 7863744
[java] Nr of milliseconds spent doing I/O: 139064
[java] Start time: 1185212479428907000
[java] Time spent: 184797747000
[java] cpu0 %user: 44 %nice: 0 %system: 8 %iowait: 45 %idle: 1
[java] cpu1 %user: 4 %nice: 0 %system: 3 %iowait: 46 %idle: 45
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7455
[java] Reads merged: 3030
[java] Sectors read: 1766200
[java] Nr of milliseconds spent reading: 18532
[java] Writes completed: 34
[java] Writes merged: 17
[java] Sectors written: 408
[java] Nr of milliseconds spent writing: 200
[java] Nr of milliseconds spent doing I/O: 17932
[java] Start time: 1185212664226650000
[java] Time spent: 87400698000
[java] Device sdb
[java] Reads completed: 179
[java] Reads merged: 154
[java] Sectors read: 3736
[java] Nr of milliseconds spent reading: 1748
[java] Writes completed: 286
```

```
[java] Writes merged: 9431
[java] Sectors written: 77760
[java] Nr of milliseconds spent writing: 16004
[java] Nr of milliseconds spent doing I/O: 1304
[java] Start time: 1185212664226654000
[java] Time spent: 87400700000
[java] cpu0 %user: 65 %nice: 0 %system: 0 %iowait: 1 %idle: 34
[java] cpu1 %user: 3 %nice: 0 %system: 0 %iowait: 0 %idle: 97
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1185212751629044000
[java] Time spent: 117703725000
[java] Device sdb
[java] Reads completed: 49
[java] Reads merged: 0
[java] Sectors read: 392
[java] Nr of milliseconds spent reading: 484
[java] Writes completed: 1052
[java] Writes merged: 84412
[java] Sectors written: 683736
[java] Nr of milliseconds spent writing: 43824
[java] Nr of milliseconds spent doing I/O: 3136
[java] Start time: 1185212751629044000
[java] Time spent: 117703728000
[java] cpu0 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] done merging all files. 769762 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
```

```
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185212869335419000
[java] Time spent: 193367674000
[java] Device sdb
[java] Reads completed: 37434
[java] Reads merged: 23138
[java] Sectors read: 9205784
[java] Nr of milliseconds spent reading: 172560
[java] Writes completed: 14464
[java] Writes merged: 1110921
[java] Sectors written: 9003192
[java] Nr of milliseconds spent writing: 8199580
[java] Nr of milliseconds spent doing I/O: 147696
[java] Start time: 1185212869335419000
[java] Time spent: 193367672000
[java] cpu0 %user: 45 %nice: 0 %system: 8 %iowait: 43 %idle: 2
[java] cpu1 %user: 4 %nice: 0 %system: 2 %iowait: 51 %idle: 41
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7244
[java] Reads merged: 3043
[java] Sectors read: 1718496
[java] Nr of milliseconds spent reading: 17500
[java] Writes completed: 34
[java] Writes merged: 17
[java] Sectors written: 408
[java] Nr of milliseconds spent writing: 112
[java] Nr of milliseconds spent doing I/O: 17316
[java] Start time: 1185213062703093000
[java] Time spent: 87862250000
[java] Device sdb
[java] Reads completed: 122
[java] Reads merged: 138
[java] Sectors read: 2152
[java] Nr of milliseconds spent reading: 3776
[java] Writes completed: 453
[java] Writes merged: 29466
[java] Sectors written: 239360
[java] Nr of milliseconds spent writing: 242000
[java] Nr of milliseconds spent doing I/O: 2572
[java] Start time: 1185213062703091000
[java] Time spent: 87862254000
[java] cpu0 %user: 83 %nice: 0 %system: 1 %iowait: 0 %idle: 16
[java] cpu1 %user: 21 %nice: 0 %system: 0 %iowait: 0 %idle: 79
[java] done flushing partial dict
```

```
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 4
[java] Start time: 1185213150567105000
[java] Time spent: 114631694000
[java] Device sdb
[java] Reads completed: 53
[java] Reads merged: 0
[java] Sectors read: 424
[java] Nr of milliseconds spent reading: 628
[java] Writes completed: 1039
[java] Writes merged: 81475
[java] Sectors written: 660128
[java] Nr of milliseconds spent writing: 43832
[java] Nr of milliseconds spent doing I/O: 3192
[java] Start time: 1185213150567105000
[java] Time spent: 114631696000
[java] cpu0 %user: 44 %nice: 0 %system: 0 %iowait: 1 %idle: 54
[java] cpu1 %user: 56 %nice: 0 %system: 0 %iowait: 0 %idle: 42
[java] done merging all files. 819919 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185213265200375000
[java] Time spent: 210693430000
[java] Device sdb
[java] Reads completed: 40327
[java] Reads merged: 18910
[java] Sectors read: 9871408
[java] Nr of milliseconds spent reading: 191668
[java] Writes completed: 16021
```

```
[java] Writes merged: 1205828
[java] Sectors written: 9774928
[java] Nr of milliseconds spent writing: 8990124
[java] Nr of milliseconds spent doing I/O: 159932
[java] Start time: 1185213265200375000
[java] Time spent: 210693433000
[java] cpu0 %user: 44 %nice: 0 %system: 8 %iowait: 44 %idle: 2
[java] cpu1 %user: 5 %nice: 0 %system: 3 %iowait: 42 %idle: 48
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7347
[java] Reads merged: 3037
[java] Sectors read: 1742096
[java] Nr of milliseconds spent reading: 16988
[java] Writes completed: 35
[java] Writes merged: 19
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 104
[java] Nr of milliseconds spent doing I/O: 16880
[java] Start time: 1185213475893805000
[java] Time spent: 86095548000
[java] Device sdb
[java] Reads completed: 21
[java] Reads merged: 0
[java] Sectors read: 240
[java] Nr of milliseconds spent reading: 252
[java] Writes completed: 409
[java] Writes merged: 18949
[java] Sectors written: 154864
[java] Nr of milliseconds spent writing: 63608
[java] Nr of milliseconds spent doing I/O: 1096
[java] Start time: 1185213475893808000
[java] Time spent: 86095552000
[java] cpu0 %user: 83 %nice: 0 %system: 0 %iowait: 0 %idle: 17
[java] cpu1 %user: 23 %nice: 0 %system: 0 %iowait: 0 %idle: 77
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
```

```
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1185213561991057000
[java] Time spent: 116786413000
[java] Device sdb
[java] Reads completed: 56
[java] Reads merged: 0
[java] Sectors read: 448
[java] Nr of milliseconds spent reading: 608
[java] Writes completed: 1033
[java] Writes merged: 81902
[java] Sectors written: 663520
[java] Nr of milliseconds spent writing: 43164
[java] Nr of milliseconds spent doing I/O: 3300
[java] Start time: 1185213561991057000
[java] Time spent: 116786417000
[java] cpu0 %user: 90 %nice: 0 %system: 0 %iowait: 0 %idle: 8
[java] cpu1 %user: 9 %nice: 0 %system: 0 %iowait: 1 %idle: 89
[java] done merging all files. 870991 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185213678780309000
[java] Time spent: 221501462000
[java] Device sdb
[java] Reads completed: 43072
[java] Reads merged: 21706
[java] Sectors read: 10532576
[java] Nr of milliseconds spent reading: 191640
[java] Writes completed: 16978
[java] Writes merged: 1289091
[java] Sectors written: 10448784
[java] Nr of milliseconds spent writing: 9518948
[java] Nr of milliseconds spent doing I/O: 168308
[java] Start time: 1185213678780309000
[java] Time spent: 221501465000
[java] cpu0 %user: 45 %nice: 0 %system: 8 %iowait: 44 %idle: 2
[java] cpu1 %user: 5 %nice: 0 %system: 3 %iowait: 45 %idle: 46
[java] done with accumulation in memory
```

```
[java] Device sda
[java] Reads completed: 7714
[java] Reads merged: 3155
[java] Sectors read: 1827288
[java] Nr of milliseconds spent reading: 15908
[java] Writes completed: 33
[java] Writes merged: 16
[java] Sectors written: 392
[java] Nr of milliseconds spent writing: 96
[java] Nr of milliseconds spent doing I/O: 15784
[java] Start time: 1185213900281771000
[java] Time spent: 86954272000
[java] Device sdb
[java] Reads completed: 107
[java] Reads merged: 71
[java] Sectors read: 2496
[java] Nr of milliseconds spent reading: 1044
[java] Writes completed: 421
[java] Writes merged: 18303
[java] Sectors written: 149800
[java] Nr of milliseconds spent writing: 56020
[java] Nr of milliseconds spent doing I/O: 1560
[java] Start time: 1185213900281774000
[java] Time spent: 86954276000
[java] cpu0 %user: 70 %nice: 0 %system: 0 %iowait: 0 %idle: 30
[java] cpu1 %user: 15 %nice: 0 %system: 0 %iowait: 0 %idle: 85
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 2
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1185213987237810000
[java] Time spent: 122313184000
[java] Device sdb
[java] Reads completed: 62
[java] Reads merged: 0
[java] Sectors read: 496
[java] Nr of milliseconds spent reading: 596
[java] Writes completed: 1054
```

```
[java] Writes merged: 81183
[java] Sectors written: 657928
[java] Nr of milliseconds spent writing: 39620
[java] Nr of milliseconds spent doing I/O: 3304
[java] Start time: 1185213987237810000
[java] Time spent: 122313187000
[java] cpu0 %user: 85 %nice: 0 %system: 0 %iowait: 0 %idle: 14
[java] cpu1 %user: 15 %nice: 0 %system: 0 %iowait: 1 %idle: 83
[java] done merging all files. 927243 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185214109552680000
[java] Time spent: 240190988000
[java] Device sdb
[java] Reads completed: 45102
[java] Reads merged: 27935
[java] Sectors read: 11191048
[java] Nr of milliseconds spent reading: 206044
[java] Writes completed: 17745
[java] Writes merged: 1384679
[java] Sectors written: 11110088
[java] Nr of milliseconds spent writing: 10707560
[java] Nr of milliseconds spent doing I/O: 187244
[java] Start time: 1185214109552680000
[java] Time spent: 240190991000
[java] cpu0 %user: 44 %nice: 0 %system: 8 %iowait: 45 %idle: 2
[java] cpu1 %user: 4 %nice: 0 %system: 2 %iowait: 48 %idle: 43
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7957
[java] Reads merged: 3202
[java] Sectors read: 1884584
[java] Nr of milliseconds spent reading: 18872
[java] Writes completed: 34
[java] Writes merged: 19
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 264
```

```
[java] Nr of milliseconds spent doing I/O: 18736
[java] Start time: 1185214349743668000
[java] Time spent: 87784612000
[java] Device sdb
[java] Reads completed: 20
[java] Reads merged: 0
[java] Sectors read: 232
[java] Nr of milliseconds spent reading: 948
[java] Writes completed: 340
[java] Writes merged: 3027
[java] Sectors written: 136432
[java] Nr of milliseconds spent writing: 136356
[java] Nr of milliseconds spent doing I/O: 1084
[java] Start time: 1185214349743671000
[java] Time spent: 87784616000
[java] cpu0 %user: 63 %nice: 0 %system: 0 %iowait: 1 %idle: 36
[java] cpu1 %user: 28 %nice: 0 %system: 0 %iowait: 0 %idle: 72
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 36
[java] Nr of milliseconds spent doing I/O: 36
[java] Start time: 1185214437529922000
[java] Time spent: 125787073000
[java] Device sdb
[java] Reads completed: 163
[java] Reads merged: 72
[java] Sectors read: 2880
[java] Nr of milliseconds spent reading: 1768
[java] Writes completed: 1014
[java] Writes merged: 82918
[java] Sectors written: 671480
[java] Nr of milliseconds spent writing: 38596
[java] Nr of milliseconds spent doing I/O: 3892
[java] Start time: 1185214437529922000
[java] Time spent: 125787076000
[java] cpu0 %user: 71 %nice: 0 %system: 0 %iowait: 0 %idle: 27
[java] cpu1 %user: 28 %nice: 0 %system: 0 %iowait: 1 %idle: 69
[java] done merging all files. 987164 docs searchable
```

```
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185214563318658000
[java] Time spent: 253761325000
[java] Device sdb
[java] Reads completed: 47709
[java] Reads merged: 21966
[java] Sectors read: 11844832
[java] Nr of milliseconds spent reading: 217192
[java] Writes completed: 16814
[java] Writes merged: 1426281
[java] Sectors written: 11544800
[java] Nr of milliseconds spent writing: 10864048
[java] Nr of milliseconds spent doing I/O: 187524
[java] Start time: 1185214563318658000
[java] Time spent: 253761330000
[java] cpu0 %user: 44 %nice: 0 %system: 8 %iowait: 45 %idle: 1
[java] cpu1 %user: 4 %nice: 0 %system: 2 %iowait: 52 %idle: 39
[java] All files added
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 1755
[java] Reads merged: 752
[java] Sectors read: 408760
[java] Nr of milliseconds spent reading: 4148
[java] Writes completed: 7
[java] Writes merged: 3
[java] Sectors written: 80
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 3980
[java] Start time: 1185214817079983000
[java] Time spent: 20606730000
[java] Device sdb
[java] Reads completed: 21
[java] Reads merged: 10
[java] Sectors read: 264
[java] Nr of milliseconds spent reading: 884
```

```
[java] Writes completed: 525
[java] Writes merged: 40839
[java] Sectors written: 330912
[java] Nr of milliseconds spent writing: 250688
[java] Nr of milliseconds spent doing I/O: 2496
[java] Start time: 1185214817079988000
[java] Time spent: 20606730000
[java] cpu0 %user: 65 %nice: 0 %system: 0 %iowait: 0 %idle: 35
[java] cpu1 %user: 5 %nice: 0 %system: 0 %iowait: 0 %idle: 95
[java] done flushing partial dict
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1185214837688387000
[java] Time spent: 20347050000
[java] Device sdb
[java] Reads completed: 70
[java] Reads merged: 18
[java] Sectors read: 760
[java] Nr of milliseconds spent reading: 696
[java] Writes completed: 208
[java] Writes merged: 17819
[java] Sectors written: 144256
[java] Nr of milliseconds spent writing: 10944
[java] Nr of milliseconds spent doing I/O: 1188
[java] Start time: 1185214837688387000
[java] Time spent: 20347052000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 15 %idle: 85
[java] cpu1 %user: 96 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done merging all files. 1000000 docs searchable
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 1
[java] Writes merged: 0
[java] Sectors written: 8
```

```
[java] Nr of milliseconds spent writing: 8
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1185214858036977000
[java] Time spent: 259273132000
[java] Device sdb
[java] Reads completed: 48916
[java] Reads merged: 34645
[java] Sectors read: 11973160
[java] Nr of milliseconds spent reading: 224664
[java] Writes completed: 18760
[java] Writes merged: 1466355
[java] Sectors written: 11881672
[java] Nr of milliseconds spent writing: 11871440
[java] Nr of milliseconds spent doing I/O: 200424
[java] Start time: 1185214858036977000
[java] Time spent: 259273137000
[java] cpu0 %user: 42 %nice: 0 %system: 8 %iowait: 46 %idle: 2
[java] cpu1 %user: 5 %nice: 0 %system: 2 %iowait: 47 %idle: 44
[java] Device sda
[java] Reads completed: 144084
[java] Reads merged: 59125
[java] Sectors read: 34124952
[java] Nr of milliseconds spent reading: 319388
[java] Writes completed: 675
[java] Writes merged: 349
[java] Sectors written: 8192
[java] Nr of milliseconds spent writing: 3020
[java] Nr of milliseconds spent doing I/O: 314952
[java] Start time: 1185208600946941000
[java] Time spent: 6516375102000
[java] Device sdb
[java] Reads completed: 506515
[java] Reads merged: 312456
[java] Sectors read: 123838608
[java] Nr of milliseconds spent reading: 2452448
[java] Writes completed: 220984
[java] Writes merged: 16903040
[java] Sectors written: 136995304
[java] Nr of milliseconds spent writing: 110821884
[java] Nr of milliseconds spent doing I/O: 2090924
[java] Start time: 1185208600946941000
[java] Time spent: 6516374817000
[java] cpu0 %user: 53 %nice: 0 %system: 3 %iowait: 19 %idle: 23
[java] cpu1 %user: 29 %nice: 0 %system: 1 %iowait: 19 %idle: 49
[java] Created the index in 108 minutes and 35 seconds.
```

```
[java] starting searches for small terms
[java] Searching for 8985741
[java] Searching for 3456645
[java] Searching for 5a90
[java] Searching for 316778
[java] Searching for x4262
[java] Searching for sc760e
[java] Searching for affectedthe
[java] Searching for d037968
[java] Searching for 190048660
[java] Searching for 164654
[java] Searching for nicololas
[java] Searching for ibbmec
[java] Searching for m11n12e
[java] Searching for hh227pc5
[java] Searching for rec18301
[java] Searching for 129x1
[java] Searching for 3002894512
[java] Searching for mar345remote
[java] Searching for fluoderm
[java] Searching for 57217907
[java] Searching for p5162345
[java] Searching for 2047593
[java] Searching for bragenski
[java] Searching for genequantification04
[java] Searching for t1654
[java] Searching for 880937
[java] Searching for meji
[java] Searching for geophyics
[java] Searching for frutex
[java] Searching for laszek
[java] Searching for fd011148
[java] Searching for smallparticles
[java] Searching for 210807
[java] Searching for sonorit
[java] Searching for daciformis
[java] Searching for blki
[java] Searching for tetraurelia
[java] Searching for bodyteen
[java] Searching for q030
[java] Searching for stipulatation
[java] Searching for hamelberg
[java] Searching for shozari
[java] Searching for 0614543
[java] Searching for miller99
```

```
[java] Searching for 279199
[java] Searching for 5176920
[java] Searching for 10076473
[java] Searching for hurger
[java] Searching for h2105
[java] Searching for nj04114
[java] Searching for 632088
[java] Searching for weblookup
[java] Searching for 1240616
[java] Searching for 1940218bg
[java] Searching for 858890
[java] Searching for vollbert
[java] Searching for pursuingbusiness
[java] Searching for subtartive
[java] Searching for ms9000e
[java] Searching for kichijoji
[java] Searching for 803194
[java] Searching for 790580
[java] Searching for 4588448
[java] Searching for 30490944
[java] Searching for nwwg4000200049
[java] Searching for 2581152
[java] Searching for 4127670
[java] Searching for otocalm
[java] Searching for pnu020
[java] Searching for 17166380
[java] Searching for 206400
[java] Searching for 19992833
[java] Searching for 8873154
[java] Searching for v90596
[java] Searching for mo00061
[java] Searching for 73ay404
[java] Searching for mdat40
[java] Searching for 0922e
[java] Searching for 14bs229
[java] Searching for lp00043725
[java] Searching for wcir023311800
[java] Searching for r0001
[java] Searching for regionallabor
[java] Searching for a786gent
[java] Searching for 5r01ca095662
[java] Searching for 1061002
[java] Searching for 2089079
[java] Searching for nrmap000100673
[java] Searching for 1802144lg
```

```
[java] Searching for 9649858
[java] Searching for 2000hnwx0005
[java] Searching for tn10231
[java] Searching for yfmz99
[java] Searching for nouve
[java] Searching for 812080100
[java] Searching for wdbda24c1ff092237
[java] Searching for monkeywrench
[java] Searching for 90et0170
[java] Searching for 12604751
[java] Searching for cedell
[java] Finished with small term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185215137352000000
[java] Time spent: 3892635000
[java] Device sdb
[java] Reads completed: 475
[java] Reads merged: 113
[java] Sectors read: 11744
[java] Nr of milliseconds spent reading: 4896
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 3868
[java] Start time: 1185215137352000000
[java] Time spent: 3892634000
[java] cpu0 %user: 0 %nice: 0 %system: 1 %iowait: 95 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 0 %idle: 97
[java] starting searches for medium terms
[java] Searching for mailto
[java] Searching for europeans
[java] Searching for 22z
[java] Searching for 1181
[java] Searching for 91109
[java] Searching for hvn
```

```
[java] Searching for 1056
[java] Searching for postcard
[java] Searching for 6112
[java] Searching for antispam
[java] Searching for campylobacter
[java] Searching for asociaci
[java] Searching for jam
[java] Searching for phy
[java] Searching for shawn
[java] Searching for kerry
[java] Searching for rk
[java] Searching for frawley
[java] Searching for infantil
[java] Searching for meteorologists
[java] Searching for tutor
[java] Searching for seis
[java] Searching for 3042
[java] Searching for mindspring
[java] Searching for snack
[java] Searching for clemente
[java] Searching for guthrie
[java] Searching for underscores
[java] Searching for crowded
[java] Searching for infantil
[java] Searching for manipulating
[java] Searching for eli
[java] Searching for relativity
[java] Searching for 04z
[java] Searching for remuneration
[java] Searching for hugo
[java] Searching for empleo
[java] Searching for psychotherapy
[java] Searching for voa
[java] Searching for o3
[java] Searching for concentrates
[java] Searching for uty
[java] Searching for rectal
[java] Searching for qt
[java] Searching for 1134
[java] Searching for dove
[java] Searching for osage
[java] Searching for 1355
[java] Searching for 2121
[java] Searching for fbo
[java] Searching for stacked
```

```
[java] Searching for invocation
[java] Searching for sponge
[java] Searching for listeriosis
[java] Searching for cty
[java] Searching for wc3
[java] Searching for negotiators
[java] Searching for deliberative
[java] Searching for organizer
[java] Searching for underscores
[java] Searching for 24hr
[java] Searching for uphold
[java] Searching for dsc
[java] Searching for debated
[java] Searching for rck
[java] Searching for civilization
[java] Searching for 2048
[java] Searching for 1545
[java] Searching for banco
[java] Searching for causation
[java] Searching for aasa
[java] Searching for aasa
[java] Searching for manatee
[java] Searching for 22314
[java] Searching for diminishing
[java] Searching for refrigerators
[java] Searching for rp2
[java] Searching for crafted
[java] Searching for escort
[java] Searching for chung
[java] Searching for dewey
[java] Searching for 03e
[java] Searching for ninr
[java] Searching for 1431
[java] Searching for waller
[java] Searching for ep5
[java] Searching for peabody
[java] Searching for sponge
[java] Searching for posterior
[java] Searching for 6112
[java] Searching for 10z
[java] Searching for 2048
[java] Searching for vest
[java] Searching for foci
[java] Searching for 2310
[java] Searching for vlc
```

```
[java] Searching for negotiators
[java] Searching for 2350
[java] Searching for cryo
[java] Searching for 5800
[java] Finished with medium term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1185215141253804000
[java] Time spent: 9021487000
[java] Device sdb
[java] Reads completed: 1158
[java] Reads merged: 124
[java] Sectors read: 36632
[java] Nr of milliseconds spent reading: 10372
[java] Writes completed: 94
[java] Writes merged: 15
[java] Sectors written: 872
[java] Nr of milliseconds spent writing: 26064
[java] Nr of milliseconds spent doing I/O: 8880
[java] Start time: 1185215141253804000
[java] Time spent: 9021490000
[java] cpu0 %user: 1 %nice: 0 %system: 0 %iowait: 97 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 0 %idle: 99
[java] starting searches for long terms
[java] Searching for research
[java] Searching for 3
[java] Searching for 10
[java] Searching for national
[java] Searching for health
[java] Searching for 2002
[java] Searching for 02
[java] Searching for will
[java] Searching for 18
[java] Searching for 15
[java] Searching for as
[java] Searching for c
[java] Searching for public
```

```
[java] Searching for 9
[java] Searching for n
[java] Searching for 30
[java] Searching for 4
[java] Searching for in
[java] Searching for been
[java] Searching for page
[java] Searching for state
[java] Searching for from
[java] Searching for or
[java] Searching for have
[java] Searching for 5
[java] Searching for may
[java] Searching for an
[java] Searching for e
[java] Searching for on
[java] Searching for would
[java] Searching for 8
[java] Searching for 2
[java] Searching for 0
[java] Searching for u
[java] Searching for to
[java] Searching for 00
[java] Searching for a
[java] Searching for was
[java] Searching for has
[java] Searching for m
[java] Searching for department
[java] Searching for 12
[java] Searching for data
[java] Searching for 16
[java] Searching for these
[java] Searching for 03
[java] Searching for if
[java] Searching for can
[java] Searching for with
[java] Searching for home
[java] Searching for not
[java] Searching for de
[java] Searching for of
[java] Searching for 01
[java] Searching for 11
[java] Searching for for
[java] Searching for is
[java] Searching for by
```

```
[java] Searching for other
[java] Searching for one
[java] Searching for html
[java] Searching for also
[java] Searching for that
[java] Searching for it
[java] Searching for this
[java] Searching for 1
[java] Searching for we
[java] Searching for be
[java] Searching for services
[java] Searching for 6
[java] Searching for program
[java] Searching for were
[java] Searching for their
[java] Searching for 14
[java] Searching for i
[java] Searching for gov
[java] Searching for 13
[java] Searching for new
[java] Searching for its
[java] Searching for information
[java] Searching for which
[java] Searching for they
[java] Searching for any
[java] Searching for you
[java] Searching for are
[java] Searching for at
[java] Searching for the
[java] Searching for b
[java] Searching for 2003
[java] Searching for and
[java] Searching for d
[java] Searching for use
[java] Searching for all
[java] Searching for no
[java] Searching for time
[java] Searching for 20
[java] Searching for 7
[java] Searching for more
[java] Searching for about
[java] Searching for s
[java] Finished with long term searches
[java] Device sda
[java] Reads completed: 0
```

```
     [java] Reads merged: 0
     [java] Sectors read: 0
     [java] Nr of milliseconds spent reading: 0
     [java] Writes completed: 0
     [java] Writes merged: 0
     [java] Sectors written: 0
     [java] Nr of milliseconds spent writing: 0
     [java] Nr of milliseconds spent doing I/O: 0
     [java] Start time: 1185215150287230000
     [java] Time spent: 31480387000
     [java] Device sdb
     [java] Reads completed: 14207
     [java] Reads merged: 6537
     [java] Sectors read: 3198032
     [java] Nr of milliseconds spent reading: 33572
     [java] Writes completed: 75
     [java] Writes merged: 54
     [java] Sectors written: 1032
     [java] Nr of milliseconds spent writing: 1516
     [java] Nr of milliseconds spent doing I/O: 27520
     [java] Start time: 1185215150287230000
     [java] Time spent: 31480391000
     [java] cpu0 %user: 39 %nice: 0 %system: 8 %iowait: 52 %idle: 0
     [java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 1 %idle: 99

BUILD SUCCESSFUL
Total time: 111 minutes 51 seconds
```

B.3 Hierarchical index

This section contains a log from an experiment with a hierarchical index. The index constructed contains $N = 1$ million documents, and the variables in the hierarchical method is set to $T = 1$, $K = 2$ and $B = 16$KB.

```
Buildfile: build.xml

clean:
   [delete] Deleting directory /usr/brille/bin

build:
    [mkdir] Created dir: /usr/brille/bin
    [javac] Compiling 86 source files to /usr/brille/bin
     [copy] Copying 1 file to /usr/brille/bin
```

```
deploy:
      [jar] Building jar: /usr/brille/brille.jar

run:
     [java] The operating system is Linux
     [java] adding file /data/gov2-corpus
     [java] done with accumulation in memory
     [java] Device sda
     [java] Reads completed: 9807
     [java] Reads merged: 3911
     [java] Sectors read: 2310168
     [java] Nr of milliseconds spent reading: 15660
     [java] Writes completed: 36
     [java] Writes merged: 22
     [java] Sectors written: 464
     [java] Nr of milliseconds spent writing: 156
     [java] Nr of milliseconds spent doing I/O: 15460
     [java] Start time: 1183329584076922000
     [java] Time spent: 97556493000
     [java] Device sdb
     [java] Reads completed: 0
     [java] Reads merged: 0
     [java] Sectors read: 0
     [java] Nr of milliseconds spent reading: 0
     [java] Writes completed: 249
     [java] Writes merged: 1045
     [java] Sectors written: 10368
     [java] Nr of milliseconds spent writing: 1032
     [java] Nr of milliseconds spent doing I/O: 104
     [java] Start time: 1183329584076922000
     [java] Time spent: 97556487000
     [java] cpu0 %user: 56 %nice: 0 %system: 1 %iowait: 0 %idle: 41
     [java] cpu1 %user: 57 %nice: 0 %system: 1 %iowait: 0 %idle: 42
     [java] done with flushing partial file
     [java] Device sda
     [java] Reads completed: 0
     [java] Reads merged: 0
     [java] Sectors read: 0
     [java] Nr of milliseconds spent reading: 0
     [java] Writes completed: 4
     [java] Writes merged: 1
     [java] Sectors written: 40
     [java] Nr of milliseconds spent writing: 16
     [java] Nr of milliseconds spent doing I/O: 12
```

```
[java] Start time: 1183329681637740000
[java] Time spent: 131451088000
[java] Device sdb
[java] Reads completed: 32
[java] Reads merged: 1
[java] Sectors read: 296
[java] Nr of milliseconds spent reading: 456
[java] Writes completed: 1011
[java] Writes merged: 80505
[java] Sectors written: 652160
[java] Nr of milliseconds spent writing: 37056
[java] Nr of milliseconds spent doing I/O: 2596
[java] Start time: 1183329681637740000
[java] Time spent: 131451087000
[java] cpu0 %user: 27 %nice: 0 %system: 0 %iowait: 1 %idle: 71
[java] cpu1 %user: 73 %nice: 0 %system: 0 %iowait: 0 %idle: 25
[java] done fixing rank for new partial file, 64346 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183329813096518000
[java] Time spent: 10714708000
[java] Device sdb
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 28
[java] Writes merged: 2276
[java] Sectors written: 18432
[java] Nr of milliseconds spent writing: 2584
[java] Nr of milliseconds spent doing I/O: 132
[java] Start time: 1183329813096518000
[java] Time spent: 10714625000
[java] cpu0 %user: 3 %nice: 0 %system: 0 %iowait: 1 %idle: 92
[java] cpu1 %user: 106 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] Starting waiting because of too many small indexes at
         time 1183329823812777000
```

```
[java] Starting merge at time 1183329823882210000
[java] Done waiting for merging. Waited for 69618000 nanoseconds
[java] Done with the merge starting at time 1183329823882210000
         after 180000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 8626
[java] Reads merged: 3592
[java] Sectors read: 2051528
[java] Nr of milliseconds spent reading: 19748
[java] Writes completed: 36
[java] Writes merged: 19
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 192
[java] Nr of milliseconds spent doing I/O: 19604
[java] Start time: 1183329823812745000
[java] Time spent: 95873121000
[java] Device sdb
[java] Reads completed: 109
[java] Reads merged: 58
[java] Sectors read: 2528
[java] Nr of milliseconds spent reading: 1028
[java] Writes completed: 268
[java] Writes merged: 9412
[java] Sectors written: 77464
[java] Nr of milliseconds spent writing: 1844
[java] Nr of milliseconds spent doing I/O: 1200
[java] Start time: 1183329823812745000
[java] Time spent: 95873126000
[java] cpu0 %user: 60 %nice: 0 %system: 1 %iowait: 3 %idle: 34
[java] cpu1 %user: 48 %nice: 0 %system: 1 %iowait: 4 %idle: 45
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 2
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1183329919688765000
[java] Time spent: 123746117000
[java] Device sdb
```

```
[java] Reads completed: 162
[java] Reads merged: 449
[java] Sectors read: 4976
[java] Nr of milliseconds spent reading: 1892
[java] Writes completed: 1102
[java] Writes merged: 81222
[java] Sectors written: 658624
[java] Nr of milliseconds spent writing: 45648
[java] Nr of milliseconds spent doing I/O: 4184
[java] Start time: 1183329919688765000
[java] Time spent: 123746118000
[java] cpu0 %user: 79 %nice: 0 %system: 0 %iowait: 1 %idle: 18
[java] cpu1 %user: 20 %nice: 0 %system: 0 %iowait: 0 %idle: 79
[java] done fixing rank for new partial file, 118820 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183330043452594000
[java] Time spent: 13270200000
[java] Device sdb
[java] Reads completed: 172
[java] Reads merged: 204
[java] Sectors read: 26832
[java] Nr of milliseconds spent reading: 1652
[java] Writes completed: 51
[java] Writes merged: 2268
[java] Sectors written: 18552
[java] Nr of milliseconds spent writing: 1292
[java] Nr of milliseconds spent doing I/O: 1308
[java] Start time: 1183330043452594000
[java] Time spent: 13270201000
[java] cpu0 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 2 %nice: 0 %system: 0 %iowait: 0 %idle: 98
[java] Starting waiting because of too many small indexes at
         time 1183330056725827000
[java] Starting merge at time 1183330056739291000
[java] Done with the merge starting at time 1183330056739291000
         after 25345513000 nanoseconds
```

```
[java] Done waiting for merging. Waited for 25358989000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 8371
[java] Reads merged: 3259
[java] Sectors read: 1972704
[java] Nr of milliseconds spent reading: 19704
[java] Writes completed: 35
[java] Writes merged: 18
[java] Sectors written: 424
[java] Nr of milliseconds spent writing: 148
[java] Nr of milliseconds spent doing I/O: 19496
[java] Start time: 1183330056725821000
[java] Time spent: 116953331000
[java] Device sdb
[java] Reads completed: 4115
[java] Reads merged: 2545
[java] Sectors read: 1020488
[java] Nr of milliseconds spent reading: 23780
[java] Writes completed: 2160
[java] Writes merged: 165228
[java] Sectors written: 1339144
[java] Nr of milliseconds spent writing: 772676
[java] Nr of milliseconds spent doing I/O: 19376
[java] Start time: 1183330056725821000
[java] Time spent: 116957371000
[java] cpu0 %user: 63 %nice: 0 %system: 2 %iowait: 9 %idle: 24
[java] cpu1 %user: 38 %nice: 0 %system: 1 %iowait: 4 %idle: 55
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 8
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 24
[java] Start time: 1183330173685748000
[java] Time spent: 124152791000
[java] Device sdb
[java] Reads completed: 53
[java] Reads merged: 6
[java] Sectors read: 680
```

```
[java] Nr of milliseconds spent reading: 752
[java] Writes completed: 1151
[java] Writes merged: 82258
[java] Sectors written: 667320
[java] Nr of milliseconds spent writing: 46920
[java] Nr of milliseconds spent doing I/O: 3508
[java] Start time: 1183330173685748000
[java] Time spent: 124152796000
[java] cpu0 %user: 4 %nice: 0 %system: 0 %iowait: 4 %idle: 90
[java] cpu1 %user: 95 %nice: 0 %system: 0 %iowait: 0 %idle: 4
[java] done fixing rank for new partial file, 174050 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183330297841489000
[java] Time spent: 13330693000
[java] Device sdb
[java] Reads completed: 185
[java] Reads merged: 59
[java] Sectors read: 41368
[java] Nr of milliseconds spent reading: 1804
[java] Writes completed: 21
[java] Writes merged: 1064
[java] Sectors written: 8680
[java] Nr of milliseconds spent writing: 208
[java] Nr of milliseconds spent doing I/O: 1744
[java] Start time: 1183330297841489000
[java] Time spent: 13330660000
[java] cpu0 %user: 91 %nice: 0 %system: 0 %iowait: 0 %idle: 10
[java] cpu1 %user: 11 %nice: 0 %system: 0 %iowait: 0 %idle: 83
[java] Starting waiting because of too many small indexes at
         time 1183330311173882000
[java] Starting merge at time 1183330311269595000
[java] Done with the merge starting at time 1183330311269595000
         after 125000 nanoseconds
[java] Done waiting for merging. Waited for 95844000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
```

```
[java] Reads completed: 8679
[java] Reads merged: 3478
[java] Sectors read: 2047560
[java] Nr of milliseconds spent reading: 19756
[java] Writes completed: 36
[java] Writes merged: 18
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 320
[java] Nr of milliseconds spent doing I/O: 19700
[java] Start time: 1183330311173876000
[java] Time spent: 94492581000
[java] Device sdb
[java] Reads completed: 17
[java] Reads merged: 32
[java] Sectors read: 392
[java] Nr of milliseconds spent reading: 236
[java] Writes completed: 667
[java] Writes merged: 40528
[java] Sectors written: 329576
[java] Nr of milliseconds spent writing: 9632
[java] Nr of milliseconds spent doing I/O: 3996
[java] Start time: 1183330311173876000
[java] Time spent: 94492586000
[java] cpu0 %user: 89 %nice: 0 %system: 2 %iowait: 2 %idle: 5
[java] cpu1 %user: 22 %nice: 0 %system: 0 %iowait: 0 %idle: 77
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 20
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1183330405668610000
[java] Time spent: 122960995000
[java] Device sdb
[java] Reads completed: 1012
[java] Reads merged: 2690
[java] Sectors read: 30704
[java] Nr of milliseconds spent reading: 4324
[java] Writes completed: 1014
[java] Writes merged: 82310
```

```
[java] Sectors written: 666672
[java] Nr of milliseconds spent writing: 39352
[java] Nr of milliseconds spent doing I/O: 5104
[java] Start time: 1183330405668610000
[java] Time spent: 122960983000
[java] cpu0 %user: 4 %nice: 0 %system: 0 %iowait: 2 %idle: 92
[java] cpu1 %user: 96 %nice: 0 %system: 0 %iowait: 1 %idle: 2
[java] done fixing rank for new partial file, 229182 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183330528631351000
[java] Time spent: 13106425000
[java] Device sdb
[java] Reads completed: 284
[java] Reads merged: 864
[java] Sectors read: 9264
[java] Nr of milliseconds spent reading: 1596
[java] Writes completed: 1
[java] Writes merged: 0
[java] Sectors written: 8
[java] Nr of milliseconds spent writing: 104
[java] Nr of milliseconds spent doing I/O: 896
[java] Start time: 1183330528631351000
[java] Time spent: 13106428000
[java] cpu0 %user: 5 %nice: 0 %system: 0 %iowait: 3 %idle: 91
[java] cpu1 %user: 96 %nice: 0 %system: 0 %iowait: 0 %idle: 3
[java] Starting waiting because of too many small indexes at
         time 1183330541739533000
[java] Starting merge at time 1183330541842560000
[java] Done with the merge starting at time 1183330541842560000
         after 63040198000 nanoseconds
[java] Done waiting for merging. Waited for 63143247000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7729
[java] Reads merged: 3230
[java] Sectors read: 1842032
```

```
[java] Nr of milliseconds spent reading: 18832
[java] Writes completed: 36
[java] Writes merged: 18
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 132
[java] Nr of milliseconds spent doing I/O: 18536
[java] Start time: 1183330541739528000
[java] Time spent: 152716348000
[java] Device sdb
[java] Reads completed: 10103
[java] Reads merged: 6536
[java] Sectors read: 2510824
[java] Nr of milliseconds spent reading: 97072
[java] Writes completed: 4054
[java] Writes merged: 328461
[java] Sectors written: 2660128
[java] Nr of milliseconds spent writing: 2064536
[java] Nr of milliseconds spent doing I/O: 51116
[java] Start time: 1183330541739528000
[java] Time spent: 152716353000
[java] cpu0 %user: 66 %nice: 0 %system: 3 %iowait: 19 %idle: 10
[java] cpu1 %user: 21 %nice: 0 %system: 1 %iowait: 18 %idle: 58
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1183330694457810000
[java] Time spent: 123211458000
[java] Device sdb
[java] Reads completed: 49
[java] Reads merged: 96
[java] Sectors read: 1192
[java] Nr of milliseconds spent reading: 440
[java] Writes completed: 1035
[java] Writes merged: 82926
[java] Sectors written: 671728
[java] Nr of milliseconds spent writing: 47988
[java] Nr of milliseconds spent doing I/O: 3768
```

```
[java] Start time: 1183330694457810000
[java] Time spent: 123211453000
[java] cpu0 %user: 47 %nice: 0 %system: 0 %iowait: 1 %idle: 50
[java] cpu1 %user: 52 %nice: 0 %system: 0 %iowait: 0 %idle: 47
[java] done fixing rank for new partial file, 284815 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183330817671230000
[java] Time spent: 14260335000
[java] Device sdb
[java] Reads completed: 321
[java] Reads merged: 179
[java] Sectors read: 62912
[java] Nr of milliseconds spent reading: 2128
[java] Writes completed: 20
[java] Writes merged: 788
[java] Sectors written: 6464
[java] Nr of milliseconds spent writing: 112
[java] Nr of milliseconds spent doing I/O: 1900
[java] Start time: 1183330817671230000
[java] Time spent: 14260344000
[java] cpu0 %user: 90 %nice: 0 %system: 1 %iowait: 1 %idle: 1
[java] cpu1 %user: 3 %nice: 0 %system: 0 %iowait: 1 %idle: 97
[java] Starting waiting because of too many small indexes at
         time 1183330831934944000
[java] Starting merge at time 1183330831958851000
[java] Done with the merge starting at time 1183330831958851000
         after 127000 nanoseconds
[java] Done waiting for merging. Waited for 24040000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7510
[java] Reads merged: 3170
[java] Sectors read: 1802696
[java] Nr of milliseconds spent reading: 18164
[java] Writes completed: 31
[java] Writes merged: 17
```

```
[java] Sectors written: 384
[java] Nr of milliseconds spent writing: 72
[java] Nr of milliseconds spent doing I/O: 17852
[java] Start time: 1183330831934939000
[java] Time spent: 87845494000
[java] Device sdb
[java] Reads completed: 44
[java] Reads merged: 13
[java] Sectors read: 664
[java] Nr of milliseconds spent reading: 684
[java] Writes completed: 158
[java] Writes merged: 869
[java] Sectors written: 8240
[java] Nr of milliseconds spent writing: 1284
[java] Nr of milliseconds spent doing I/O: 496
[java] Start time: 1183330831934939000
[java] Time spent: 87845500000
[java] cpu0 %user: 91 %nice: 0 %system: 1 %iowait: 0 %idle: 6
[java] cpu1 %user: 24 %nice: 0 %system: 0 %iowait: 0 %idle: 74
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 2
[java] Reads merged: 0
[java] Sectors read: 512
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 2
[java] Writes merged: 0
[java] Sectors written: 16
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1183330919782301000
[java] Time spent: 123316828000
[java] Device sdb
[java] Reads completed: 18
[java] Reads merged: 0
[java] Sectors read: 144
[java] Nr of milliseconds spent reading: 216
[java] Writes completed: 1022
[java] Writes merged: 81575
[java] Sectors written: 660840
[java] Nr of milliseconds spent writing: 42372
[java] Nr of milliseconds spent doing I/O: 3000
[java] Start time: 1183330919782301000
[java] Time spent: 123316832000
[java] cpu0 %user: 84 %nice: 0 %system: 0 %iowait: 0 %idle: 15
```

```
[java] cpu1 %user: 15 %nice: 0 %system: 0 %iowait: 0 %idle: 83
[java] done fixing rank for new partial file, 340196 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183331043100984000
[java] Time spent: 14540843000
[java] Device sdb
[java] Reads completed: 40
[java] Reads merged: 7
[java] Sectors read: 472
[java] Nr of milliseconds spent reading: 744
[java] Writes completed: 40
[java] Writes merged: 1995
[java] Sectors written: 16280
[java] Nr of milliseconds spent writing: 628
[java] Nr of milliseconds spent doing I/O: 800
[java] Start time: 1183331043100984000
[java] Time spent: 14540878000
[java] cpu0 %user: 89 %nice: 0 %system: 2 %iowait: 4 %idle: 0
[java] cpu1 %user: 2 %nice: 0 %system: 0 %iowait: 0 %idle: 101
[java] Starting waiting because of too many small indexes at
         time 1183331057644156000
[java] Starting merge at time 1183331057707536000
[java] Done with the merge starting at time 1183331057707536000
         after 29537461000 nanoseconds
[java] Done waiting for merging. Waited for 29601004000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7673
[java] Reads merged: 3184
[java] Sectors read: 1822736
[java] Nr of milliseconds spent reading: 18236
[java] Writes completed: 33
[java] Writes merged: 19
[java] Sectors written: 416
[java] Nr of milliseconds spent writing: 144
[java] Nr of milliseconds spent doing I/O: 17960
```

```
[java] Start time: 1183331057644151000
[java] Time spent: 118834439000
[java] Device sdb
[java] Reads completed: 4456
[java] Reads merged: 2148
[java] Sectors read: 1060448
[java] Nr of milliseconds spent reading: 24428
[java] Writes completed: 2013
[java] Writes merged: 166398
[java] Sectors written: 1347312
[java] Nr of milliseconds spent writing: 744064
[java] Nr of milliseconds spent doing I/O: 18960
[java] Start time: 1183331057644151000
[java] Time spent: 118834431000
[java] cpu0 %user: 80 %nice: 0 %system: 2 %iowait: 9 %idle: 8
[java] cpu1 %user: 22 %nice: 0 %system: 1 %iowait: 10 %idle: 65
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1183331176480077000
[java] Time spent: 124146370000
[java] Device sdb
[java] Reads completed: 24
[java] Reads merged: 11
[java] Sectors read: 280
[java] Nr of milliseconds spent reading: 376
[java] Writes completed: 992
[java] Writes merged: 80922
[java] Sectors written: 655360
[java] Nr of milliseconds spent writing: 39640
[java] Nr of milliseconds spent doing I/O: 2932
[java] Start time: 1183331176480077000
[java] Time spent: 124146372000
[java] cpu0 %user: 15 %nice: 0 %system: 0 %iowait: 2 %idle: 82
[java] cpu1 %user: 85 %nice: 0 %system: 0 %iowait: 0 %idle: 14
[java] done fixing rank for new partial file, 396473 docs searchable.
[java] Device sda
```

```
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183331300627974000
[java] Time spent: 14712194000
[java] Device sdb
[java] Reads completed: 30
[java] Reads merged: 0
[java] Sectors read: 240
[java] Nr of milliseconds spent reading: 692
[java] Writes completed: 33
[java] Writes merged: 2594
[java] Sectors written: 21024
[java] Nr of milliseconds spent writing: 1128
[java] Nr of milliseconds spent doing I/O: 764
[java] Start time: 1183331300627974000
[java] Time spent: 14712196000
[java] cpu0 %user: 87 %nice: 0 %system: 1 %iowait: 4 %idle: 4
[java] cpu1 %user: 6 %nice: 0 %system: 0 %iowait: 0 %idle: 91
[java] Starting waiting because of too many small indexes at
         time 1183331315342712000
[java] Starting merge at time 1183331315430013000
[java] Done with the merge starting at time 1183331315430013000
         after 129000 nanoseconds
[java] Done waiting for merging. Waited for 87437000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7501
[java] Reads merged: 3153
[java] Sectors read: 1798056
[java] Nr of milliseconds spent reading: 17988
[java] Writes completed: 31
[java] Writes merged: 16
[java] Sectors written: 376
[java] Nr of milliseconds spent writing: 104
[java] Nr of milliseconds spent doing I/O: 17772
[java] Start time: 1183331315342706000
[java] Time spent: 87967251000
[java] Device sdb
```

```
[java] Reads completed: 12
[java] Reads merged: 0
[java] Sectors read: 136
[java] Nr of milliseconds spent reading: 228
[java] Writes completed: 143
[java] Writes merged: 923
[java] Sectors written: 8528
[java] Nr of milliseconds spent writing: 408
[java] Nr of milliseconds spent doing I/O: 292
[java] Start time: 1183331315342706000
[java] Time spent: 87967310000
[java] cpu0 %user: 84 %nice: 0 %system: 1 %iowait: 0 %idle: 12
[java] cpu1 %user: 29 %nice: 0 %system: 0 %iowait: 0 %idle: 69
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 24
[java] Nr of milliseconds spent doing I/O: 24
[java] Start time: 1183331403311644000
[java] Time spent: 122161816000
[java] Device sdb
[java] Reads completed: 54
[java] Reads merged: 1
[java] Sectors read: 648
[java] Nr of milliseconds spent reading: 640
[java] Writes completed: 1029
[java] Writes merged: 81436
[java] Sectors written: 659776
[java] Nr of milliseconds spent writing: 40844
[java] Nr of milliseconds spent doing I/O: 3332
[java] Start time: 1183331403311644000
[java] Time spent: 122161819000
[java] cpu0 %user: 40 %nice: 0 %system: 0 %iowait: 1 %idle: 58
[java] cpu1 %user: 60 %nice: 0 %system: 0 %iowait: 0 %idle: 39
[java] done fixing rank for new partial file, 451485 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
```

```
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183331525476603000
[java] Time spent: 14642338000
[java] Device sdb
[java] Reads completed: 27
[java] Reads merged: 7
[java] Sectors read: 216
[java] Nr of milliseconds spent reading: 656
[java] Writes completed: 44
[java] Writes merged: 2266
[java] Sectors written: 18480
[java] Nr of milliseconds spent writing: 896
[java] Nr of milliseconds spent doing I/O: 748
[java] Start time: 1183331525476603000
[java] Time spent: 14642340000
[java] cpu0 %user: 93 %nice: 0 %system: 2 %iowait: 4 %idle: 4
[java] cpu1 %user: 6 %nice: 0 %system: 0 %iowait: 0 %idle: 90
[java] Starting waiting because of too many small indexes at
         time 1183331540122001000
[java] Starting merge at time 1183331540186641000
[java] Done waiting for merging. Waited for 128825571000 nanoseconds
[java] Done with the merge starting at time 1183331540186641000
         after 128760912000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7718
[java] Reads merged: 3178
[java] Sectors read: 1834696
[java] Nr of milliseconds spent reading: 17708
[java] Writes completed: 34
[java] Writes merged: 17
[java] Sectors written: 408
[java] Nr of milliseconds spent writing: 172
[java] Nr of milliseconds spent doing I/O: 17456
[java] Start time: 1183331540121996000
[java] Time spent: 217754034000
[java] Device sdb
[java] Reads completed: 20818
[java] Reads merged: 14215
[java] Sectors read: 5203976
```

```
[java] Nr of milliseconds spent reading: 192852
[java] Writes completed: 8001
[java] Writes merged: 652372
[java] Sectors written: 5283008
[java] Nr of milliseconds spent writing: 4175832
[java] Nr of milliseconds spent doing I/O: 102288
[java] Start time: 1183331540121996000
[java] Time spent: 217754039000
[java] cpu0 %user: 58 %nice: 0 %system: 4 %iowait: 28 %idle: 8
[java] cpu1 %user: 17 %nice: 0 %system: 2 %iowait: 23 %idle: 57
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1183331757877697000
[java] Time spent: 123073594000
[java] Device sdb
[java] Reads completed: 1004
[java] Reads merged: 2211
[java] Sectors read: 26480
[java] Nr of milliseconds spent reading: 5944
[java] Writes completed: 996
[java] Writes merged: 80945
[java] Sectors written: 655552
[java] Nr of milliseconds spent writing: 39960
[java] Nr of milliseconds spent doing I/O: 4764
[java] Start time: 1183331757877697000
[java] Time spent: 123073597000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 3 %idle: 95
[java] cpu1 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done fixing rank for new partial file, 507239 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
```

```
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183331880952782000
[java] Time spent: 14623196000
[java] Device sdb
[java] Reads completed: 453
[java] Reads merged: 234
[java] Sectors read: 102328
[java] Nr of milliseconds spent reading: 2720
[java] Writes completed: 46
[java] Writes merged: 2860
[java] Sectors written: 23248
[java] Nr of milliseconds spent writing: 2000
[java] Nr of milliseconds spent doing I/O: 2352
[java] Start time: 1183331880952782000
[java] Time spent: 14623199000
[java] cpu0 %user: 92 %nice: 0 %system: 2 %iowait: 1 %idle: 0
[java] cpu1 %user: 1 %nice: 0 %system: 0 %iowait: 0 %idle: 95
[java] Starting waiting because of too many small indexes at
         time 1183331895578752000
[java] Starting merge at time 1183331895642614000
[java] Done with the merge starting at time 1183331895642614000
         after 132000 nanoseconds
[java] Done waiting for merging. Waited for 64985000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7671
[java] Reads merged: 3217
[java] Sectors read: 1833832
[java] Nr of milliseconds spent reading: 18604
[java] Writes completed: 35
[java] Writes merged: 19
[java] Sectors written: 432
[java] Nr of milliseconds spent writing: 84
[java] Nr of milliseconds spent doing I/O: 18316
[java] Start time: 1183331895578747000
[java] Time spent: 89779051000
[java] Device sdb
[java] Reads completed: 64
[java] Reads merged: 30
[java] Sectors read: 1040
[java] Nr of milliseconds spent reading: 412
[java] Writes completed: 117
[java] Writes merged: 904
```

```
[java] Sectors written: 8176
[java] Nr of milliseconds spent writing: 684
[java] Nr of milliseconds spent doing I/O: 484
[java] Start time: 1183331895578747000
[java] Time spent: 89779057000
[java] cpu0 %user: 62 %nice: 0 %system: 1 %iowait: 0 %idle: 36
[java] cpu1 %user: 51 %nice: 0 %system: 0 %iowait: 0 %idle: 46
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 4
[java] Writes merged: 2
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1183331985359419000
[java] Time spent: 120377609000
[java] Device sdb
[java] Reads completed: 227
[java] Reads merged: 225
[java] Sectors read: 9104
[java] Nr of milliseconds spent reading: 1672
[java] Writes completed: 1044
[java] Writes merged: 81414
[java] Sectors written: 659720
[java] Nr of milliseconds spent writing: 43616
[java] Nr of milliseconds spent doing I/O: 3860
[java] Start time: 1183331985359419000
[java] Time spent: 120377605000
[java] cpu0 %user: 87 %nice: 0 %system: 0 %iowait: 0 %idle: 12
[java] cpu1 %user: 13 %nice: 0 %system: 0 %iowait: 1 %idle: 84
[java] done fixing rank for new partial file, 560270 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
```

```
[java] Start time: 1183332105738612000
[java] Time spent: 14411579000
[java] Device sdb
[java] Reads completed: 124
[java] Reads merged: 66
[java] Sectors read: 1904
[java] Nr of milliseconds spent reading: 800
[java] Writes completed: 47
[java] Writes merged: 2621
[java] Sectors written: 21344
[java] Nr of milliseconds spent writing: 1280
[java] Nr of milliseconds spent doing I/O: 868
[java] Start time: 1183332105738612000
[java] Time spent: 14411582000
[java] cpu0 %user: 89 %nice: 0 %system: 1 %iowait: 4 %idle: 0
[java] cpu1 %user: 1 %nice: 0 %system: 0 %iowait: 0 %idle: 93
[java] Starting waiting because of too many small indexes at
         time 1183332120151720000
[java] Starting merge at time 1183332120195240000
[java] Done with the merge starting at time 1183332120195240000
         after 31999976000 nanoseconds
[java] Done waiting for merging. Waited for 32043518000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7635
[java] Reads merged: 3198
[java] Sectors read: 1816768
[java] Nr of milliseconds spent reading: 18100
[java] Writes completed: 34
[java] Writes merged: 18
[java] Sectors written: 416
[java] Nr of milliseconds spent writing: 152
[java] Nr of milliseconds spent doing I/O: 17872
[java] Start time: 1183332120151715000
[java] Time spent: 121907589000
[java] Device sdb
[java] Reads completed: 4491
[java] Reads merged: 2830
[java] Sectors read: 1078560
[java] Nr of milliseconds spent reading: 27440
[java] Writes completed: 2127
[java] Writes merged: 166870
[java] Sectors written: 1352016
[java] Nr of milliseconds spent writing: 797188
[java] Nr of milliseconds spent doing I/O: 19588
```

```
[java] Start time: 1183332120151715000
[java] Time spent: 121907596000
[java] cpu0 %user: 76 %nice: 0 %system: 2 %iowait: 11 %idle: 9
[java] cpu1 %user: 25 %nice: 0 %system: 1 %iowait: 6 %idle: 66
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 2
[java] Reads merged: 0
[java] Sectors read: 512
[java] Nr of milliseconds spent reading: 8
[java] Writes completed: 1
[java] Writes merged: 0
[java] Sectors written: 8
[java] Nr of milliseconds spent writing: 4
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1183332242060856000
[java] Time spent: 118606921000
[java] Device sdb
[java] Reads completed: 215
[java] Reads merged: 145
[java] Sectors read: 3976
[java] Nr of milliseconds spent reading: 1984
[java] Writes completed: 1030
[java] Writes merged: 81681
[java] Sectors written: 661720
[java] Nr of milliseconds spent writing: 43264
[java] Nr of milliseconds spent doing I/O: 3636
[java] Start time: 1183332242060856000
[java] Time spent: 118606925000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] cpu1 %user: 100 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done fixing rank for new partial file, 612522 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183332360669215000
[java] Time spent: 14181357000
[java] Device sdb
```

```
[java] Reads completed: 83
[java] Reads merged: 87
[java] Sectors read: 1440
[java] Nr of milliseconds spent reading: 784
[java] Writes completed: 64
[java] Writes merged: 2464
[java] Sectors written: 20224
[java] Nr of milliseconds spent writing: 1376
[java] Nr of milliseconds spent doing I/O: 724
[java] Start time: 1183332360669215000
[java] Time spent: 14181361000
[java] cpu0 %user: 88 %nice: 0 %system: 2 %iowait: 3 %idle: 6
[java] cpu1 %user: 8 %nice: 0 %system: 0 %iowait: 0 %idle: 91
[java] Starting waiting because of too many small indexes at
         time 1183332374852022000
[java] Starting merge at time 1183332374901548000
[java] Done with the merge starting at time 1183332374901548000
         after 142000 nanoseconds
[java] Done waiting for merging. Waited for 49732000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7669
[java] Reads merged: 3198
[java] Sectors read: 1831728
[java] Nr of milliseconds spent reading: 18108
[java] Writes completed: 37
[java] Writes merged: 21
[java] Sectors written: 464
[java] Nr of milliseconds spent writing: 164
[java] Nr of milliseconds spent doing I/O: 17920
[java] Start time: 1183332374852017000
[java] Time spent: 89519299000
[java] Device sdb
[java] Reads completed: 76
[java] Reads merged: 101
[java] Sectors read: 1464
[java] Nr of milliseconds spent reading: 220
[java] Writes completed: 117
[java] Writes merged: 867
[java] Sectors written: 7904
[java] Nr of milliseconds spent writing: 312
[java] Nr of milliseconds spent doing I/O: 292
[java] Start time: 1183332374852017000
[java] Time spent: 89519309000
[java] cpu0 %user: 72 %nice: 0 %system: 1 %iowait: 0 %idle: 24
```

```
[java] cpu1 %user: 40 %nice: 0 %system: 0 %iowait: 0 %idle: 57
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 4
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1183332464372814000
[java] Time spent: 118494149000
[java] Device sdb
[java] Reads completed: 109
[java] Reads merged: 52
[java] Sectors read: 1528
[java] Nr of milliseconds spent reading: 1120
[java] Writes completed: 1020
[java] Writes merged: 83876
[java] Sectors written: 679200
[java] Nr of milliseconds spent writing: 42412
[java] Nr of milliseconds spent doing I/O: 3376
[java] Start time: 1183332464372814000
[java] Time spent: 118494153000
[java] cpu0 %user: 68 %nice: 0 %system: 0 %iowait: 0 %idle: 31
[java] cpu1 %user: 32 %nice: 0 %system: 0 %iowait: 1 %idle: 66
[java] done fixing rank for new partial file, 665057 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183332582868386000
[java] Time spent: 14313172000
[java] Device sdb
[java] Reads completed: 41
[java] Reads merged: 0
[java] Sectors read: 376
```

```
[java] Nr of milliseconds spent reading: 516
[java] Writes completed: 50
[java] Writes merged: 389
[java] Sectors written: 3512
[java] Nr of milliseconds spent writing: 196
[java] Nr of milliseconds spent doing I/O: 520
[java] Start time: 1183332582868386000
[java] Time spent: 14313176000
[java] cpu0 %user: 93 %nice: 0 %system: 1 %iowait: 3 %idle: 3
[java] cpu1 %user: 4 %nice: 0 %system: 0 %iowait: 0 %idle: 90
[java] Starting waiting because of too many small indexes at
         time 1183332597184366000
[java] Starting merge at time 1183332597202036000
[java] Done with the merge starting at time 1183332597202036000
         after 67333273000 nanoseconds
[java] Done waiting for merging. Waited for 67351065000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7638
[java] Reads merged: 3219
[java] Sectors read: 1807160
[java] Nr of milliseconds spent reading: 18308
[java] Writes completed: 35
[java] Writes merged: 20
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 128
[java] Nr of milliseconds spent doing I/O: 17992
[java] Start time: 1183332597184360000
[java] Time spent: 156542871000
[java] Device sdb
[java] Reads completed: 10449
[java] Reads merged: 5495
[java] Sectors read: 2595664
[java] Nr of milliseconds spent reading: 87480
[java] Writes completed: 4227
[java] Writes merged: 330871
[java] Sectors written: 2680816
[java] Nr of milliseconds spent writing: 1894020
[java] Nr of milliseconds spent doing I/O: 49880
[java] Start time: 1183332597184360000
[java] Time spent: 156542877000
[java] cpu0 %user: 69 %nice: 0 %system: 3 %iowait: 16 %idle: 9
[java] cpu1 %user: 20 %nice: 0 %system: 2 %iowait: 13 %idle: 64
[java] done with flushing partial file
[java] Device sda
```

```
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1183332753728765000
[java] Time spent: 117875041000
[java] Device sdb
[java] Reads completed: 74
[java] Reads merged: 1
[java] Sectors read: 808
[java] Nr of milliseconds spent reading: 864
[java] Writes completed: 1011
[java] Writes merged: 82144
[java] Sectors written: 665288
[java] Nr of milliseconds spent writing: 45932
[java] Nr of milliseconds spent doing I/O: 3400
[java] Start time: 1183332753728765000
[java] Time spent: 117875044000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 3 %idle: 95
[java] cpu1 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done fixing rank for new partial file, 717628 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183332871605173000
[java] Time spent: 14717889000
[java] Device sdb
[java] Reads completed: 89
[java] Reads merged: 28
[java] Sectors read: 960
[java] Nr of milliseconds spent reading: 988
[java] Writes completed: 70
[java] Writes merged: 2114
```

```
[java] Sectors written: 17472
[java] Nr of milliseconds spent writing: 1132
[java] Nr of milliseconds spent doing I/O: 908
[java] Start time: 1183332871605173000
[java] Time spent: 14717893000
[java] cpu0 %user: 90 %nice: 0 %system: 2 %iowait: 4 %idle: 0
[java] cpu1 %user: 2 %nice: 0 %system: 0 %iowait: 0 %idle: 95
[java] Starting waiting because of too many small indexes at
         time 1183332886326175000
[java] Starting merge at time 1183332886410285000
[java] Done with the merge starting at time 1183332886410285000
         after 123000 nanoseconds
[java] Done waiting for merging. Waited for 84405000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7380
[java] Reads merged: 3036
[java] Sectors read: 1765544
[java] Nr of milliseconds spent reading: 18300
[java] Writes completed: 34
[java] Writes merged: 21
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 164
[java] Nr of milliseconds spent doing I/O: 18196
[java] Start time: 1183332886326170000
[java] Time spent: 88358683000
[java] Device sdb
[java] Reads completed: 115
[java] Reads merged: 76
[java] Sectors read: 2512
[java] Nr of milliseconds spent reading: 1140
[java] Writes completed: 140
[java] Writes merged: 888
[java] Sectors written: 8248
[java] Nr of milliseconds spent writing: 1036
[java] Nr of milliseconds spent doing I/O: 664
[java] Start time: 1183332886326170000
[java] Time spent: 88358687000
[java] cpu0 %user: 78 %nice: 0 %system: 1 %iowait: 0 %idle: 19
[java] cpu1 %user: 36 %nice: 0 %system: 0 %iowait: 0 %idle: 62
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
```

```
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 5
[java] Writes merged: 1
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 24
[java] Nr of milliseconds spent doing I/O: 20
[java] Start time: 1183332974686338000
[java] Time spent: 116814405000
[java] Device sdb
[java] Reads completed: 54
[java] Reads merged: 22
[java] Sectors read: 608
[java] Nr of milliseconds spent reading: 640
[java] Writes completed: 993
[java] Writes merged: 83282
[java] Sectors written: 674224
[java] Nr of milliseconds spent writing: 45196
[java] Nr of milliseconds spent doing I/O: 3176
[java] Start time: 1183332974686338000
[java] Time spent: 116814407000
[java] cpu0 %user: 15 %nice: 0 %system: 0 %iowait: 1 %idle: 82
[java] cpu1 %user: 84 %nice: 0 %system: 0 %iowait: 0 %idle: 14
[java] done fixing rank for new partial file, 769762 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183333091502200000
[java] Time spent: 14495151000
[java] Device sdb
[java] Reads completed: 36
[java] Reads merged: 1
[java] Sectors read: 1128
[java] Nr of milliseconds spent reading: 344
[java] Writes completed: 31
[java] Writes merged: 1445
[java] Sectors written: 11808
[java] Nr of milliseconds spent writing: 444
[java] Nr of milliseconds spent doing I/O: 420
```

```
[java] Start time: 1183333091502200000
[java] Time spent: 14495152000
[java] cpu0 %user: 97 %nice: 0 %system: 2 %iowait: 2 %idle: 1
[java] cpu1 %user: 3 %nice: 0 %system: 0 %iowait: 0 %idle: 100
[java] Starting waiting because of too many small indexes at
         time 1183333105999299000
[java] Starting merge at time 1183333106026631000
[java] Done with the merge starting at time 1183333106026631000
         after 34174262000 nanoseconds
[java] Done waiting for merging. Waited for 34201616000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7219
[java] Reads merged: 3058
[java] Sectors read: 1718480
[java] Nr of milliseconds spent reading: 18924
[java] Writes completed: 32
[java] Writes merged: 17
[java] Sectors written: 392
[java] Nr of milliseconds spent writing: 88
[java] Nr of milliseconds spent doing I/O: 18676
[java] Start time: 1183333105999293000
[java] Time spent: 122311972000
[java] Device sdb
[java] Reads completed: 4657
[java] Reads merged: 2236
[java] Sectors read: 1145304
[java] Nr of milliseconds spent reading: 30844
[java] Writes completed: 2233
[java] Writes merged: 167787
[java] Sectors written: 1360192
[java] Nr of milliseconds spent writing: 806720
[java] Nr of milliseconds spent doing I/O: 20640
[java] Start time: 1183333105999293000
[java] Time spent: 122311981000
[java] cpu0 %user: 51 %nice: 0 %system: 1 %iowait: 9 %idle: 36
[java] cpu1 %user: 47 %nice: 0 %system: 2 %iowait: 9 %idle: 39
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 1
[java] Reads merged: 0
[java] Sectors read: 256
[java] Nr of milliseconds spent reading: 4
[java] Writes completed: 5
[java] Writes merged: 1
```

```
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 8
[java] Start time: 1183333228312787000
[java] Time spent: 114014488000
[java] Device sdb
[java] Reads completed: 34
[java] Reads merged: 0
[java] Sectors read: 272
[java] Nr of milliseconds spent reading: 420
[java] Writes completed: 995
[java] Writes merged: 82708
[java] Sectors written: 669648
[java] Nr of milliseconds spent writing: 48564
[java] Nr of milliseconds spent doing I/O: 2972
[java] Start time: 1183333228312787000
[java] Time spent: 114014491000
[java] cpu0 %user: 9 %nice: 0 %system: 0 %iowait: 2 %idle: 88
[java] cpu1 %user: 91 %nice: 0 %system: 0 %iowait: 0 %idle: 8
[java] done fixing rank for new partial file, 819919 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183333342330191000
[java] Time spent: 13558961000
[java] Device sdb
[java] Reads completed: 24
[java] Reads merged: 0
[java] Sectors read: 360
[java] Nr of milliseconds spent reading: 244
[java] Writes completed: 41
[java] Writes merged: 2057
[java] Sectors written: 16784
[java] Nr of milliseconds spent writing: 884
[java] Nr of milliseconds spent doing I/O: 328
[java] Start time: 1183333342330191000
[java] Time spent: 13558963000
[java] cpu0 %user: 93 %nice: 0 %system: 2 %iowait: 1 %idle: 7
```

```
[java] cpu1 %user: 7 %nice: 0 %system: 0 %iowait: 0 %idle: 95
[java] Starting waiting because of too many small indexes at
         time 1183333355890577000
[java] Starting merge at time 1183333355952670000
[java] Done with the merge starting at time 1183333355952670000
         after 119000 nanoseconds
[java] Done waiting for merging. Waited for 62269000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7260
[java] Reads merged: 3047
[java] Sectors read: 1741576
[java] Nr of milliseconds spent reading: 18644
[java] Writes completed: 34
[java] Writes merged: 18
[java] Sectors written: 416
[java] Nr of milliseconds spent writing: 84
[java] Nr of milliseconds spent doing I/O: 18516
[java] Start time: 1183333355890571000
[java] Time spent: 88448888000
[java] Device sdb
[java] Reads completed: 55
[java] Reads merged: 1
[java] Sectors read: 680
[java] Nr of milliseconds spent reading: 548
[java] Writes completed: 123
[java] Writes merged: 868
[java] Sectors written: 7936
[java] Nr of milliseconds spent writing: 572
[java] Nr of milliseconds spent doing I/O: 588
[java] Start time: 1183333355890571000
[java] Time spent: 88448893000
[java] cpu0 %user: 34 %nice: 0 %system: 0 %iowait: 0 %idle: 65
[java] cpu1 %user: 79 %nice: 0 %system: 1 %iowait: 0 %idle: 17
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 1
[java] Sectors written: 40
[java] Nr of milliseconds spent writing: 8
[java] Nr of milliseconds spent doing I/O: 4
```

```
[java] Start time: 1183333444340955000
[java] Time spent: 115338302000
[java] Device sdb
[java] Reads completed: 132
[java] Reads merged: 56
[java] Sectors read: 2392
[java] Nr of milliseconds spent reading: 1376
[java] Writes completed: 1020
[java] Writes merged: 83074
[java] Sectors written: 672768
[java] Nr of milliseconds spent writing: 44788
[java] Nr of milliseconds spent doing I/O: 3480
[java] Start time: 1183333444340955000
[java] Time spent: 115338304000
[java] cpu0 %user: 92 %nice: 0 %system: 0 %iowait: 0 %idle: 6
[java] cpu1 %user: 7 %nice: 0 %system: 0 %iowait: 2 %idle: 90
[java] done fixing rank for new partial file, 870991 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183333559682078000
[java] Time spent: 14062384000
[java] Device sdb
[java] Reads completed: 36
[java] Reads merged: 0
[java] Sectors read: 920
[java] Nr of milliseconds spent reading: 240
[java] Writes completed: 34
[java] Writes merged: 1726
[java] Sectors written: 14080
[java] Nr of milliseconds spent writing: 672
[java] Nr of milliseconds spent doing I/O: 308
[java] Start time: 1183333559682078000
[java] Time spent: 14062388000
[java] cpu0 %user: 89 %nice: 0 %system: 1 %iowait: 1 %idle: 1
[java] cpu1 %user: 2 %nice: 0 %system: 0 %iowait: 0 %idle: 97
[java] Starting waiting because of too many small indexes at
         time 1183333573747212000
```

```
[java] Starting merge at time 1183333573816913000
[java] Done waiting for merging. Waited for 259179050000 nanoseconds
[java] Done with the merge starting at time 1183333573816913000
         after 259109815000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7694
[java] Reads merged: 3160
[java] Sectors read: 1827264
[java] Nr of milliseconds spent reading: 17464
[java] Writes completed: 34
[java] Writes merged: 16
[java] Sectors written: 400
[java] Nr of milliseconds spent writing: 156
[java] Nr of milliseconds spent doing I/O: 17416
[java] Start time: 1183333573747207000
[java] Time spent: 348023535000
[java] Device sdb
[java] Reads completed: 42905
[java] Reads merged: 24126
[java] Sectors read: 10505040
[java] Nr of milliseconds spent reading: 397656
[java] Writes completed: 17170
[java] Writes merged: 1304127
[java] Sectors written: 10570664
[java] Nr of milliseconds spent writing: 8665076
[java] Nr of milliseconds spent doing I/O: 207188
[java] Start time: 1183333573747207000
[java] Time spent: 348023542000
[java] cpu0 %user: 53 %nice: 0 %system: 4 %iowait: 36 %idle: 5
[java] cpu1 %user: 10 %nice: 0 %system: 2 %iowait: 25 %idle: 61
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 4
[java] Writes merged: 2
[java] Sectors written: 48
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 16
[java] Start time: 1183333921772269000
[java] Time spent: 120489947000
[java] Device sdb
```

```
[java] Reads completed: 90
[java] Reads merged: 1
[java] Sectors read: 936
[java] Nr of milliseconds spent reading: 1284
[java] Writes completed: 1003
[java] Writes merged: 82818
[java] Sectors written: 670624
[java] Nr of milliseconds spent writing: 49232
[java] Nr of milliseconds spent doing I/O: 3764
[java] Start time: 1183333921772269000
[java] Time spent: 120489950000
[java] cpu0 %user: 0 %nice: 0 %system: 0 %iowait: 3 %idle: 95
[java] cpu1 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] done fixing rank for new partial file, 927243 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183334042263638000
[java] Time spent: 14548729000
[java] Device sdb
[java] Reads completed: 830
[java] Reads merged: 374
[java] Sectors read: 170336
[java] Nr of milliseconds spent reading: 3472
[java] Writes completed: 21
[java] Writes merged: 1377
[java] Sectors written: 11184
[java] Nr of milliseconds spent writing: 264
[java] Nr of milliseconds spent doing I/O: 2984
[java] Start time: 1183334042263638000
[java] Time spent: 14548731000
[java] cpu0 %user: 86 %nice: 0 %system: 3 %iowait: 2 %idle: 4
[java] cpu1 %user: 5 %nice: 0 %system: 0 %iowait: 0 %idle: 97
[java] Starting waiting because of too many small indexes at
         time 1183334056815095000
[java] Starting merge at time 1183334056904053000
[java] Done with the merge starting at time 1183334056904053000
         after 131000 nanoseconds
```

```
[java] Done waiting for merging. Waited for 89096000 nanoseconds
[java] done with accumulation in memory
[java] Device sda
[java] Reads completed: 7932
[java] Reads merged: 3186
[java] Sectors read: 1884336
[java] Nr of milliseconds spent reading: 16560
[java] Writes completed: 36
[java] Writes merged: 19
[java] Sectors written: 440
[java] Nr of milliseconds spent writing: 64
[java] Nr of milliseconds spent doing I/O: 16400
[java] Start time: 1183334056815089000
[java] Time spent: 89070897000
[java] Device sdb
[java] Reads completed: 105
[java] Reads merged: 56
[java] Sectors read: 2208
[java] Nr of milliseconds spent reading: 852
[java] Writes completed: 144
[java] Writes merged: 979
[java] Sectors written: 9016
[java] Nr of milliseconds spent writing: 520
[java] Nr of milliseconds spent doing I/O: 544
[java] Start time: 1183334056815089000
[java] Time spent: 89070903000
[java] cpu0 %user: 84 %nice: 0 %system: 2 %iowait: 0 %idle: 12
[java] cpu1 %user: 30 %nice: 0 %system: 0 %iowait: 0 %idle: 69
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 12
[java] Nr of milliseconds spent doing I/O: 12
[java] Start time: 1183334145887537000
[java] Time spent: 125407212000
[java] Device sdb
[java] Reads completed: 48
[java] Reads merged: 0
[java] Sectors read: 384
```

```
[java] Nr of milliseconds spent reading: 540
[java] Writes completed: 965
[java] Writes merged: 80261
[java] Sectors written: 649848
[java] Nr of milliseconds spent writing: 37140
[java] Nr of milliseconds spent doing I/O: 2932
[java] Start time: 1183334145887537000
[java] Time spent: 125407214000
[java] cpu0 %user: 99 %nice: 0 %system: 0 %iowait: 0 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 96
[java] done fixing rank for new partial file, 987164 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183334271297547000
[java] Time spent: 14421581000
[java] Device sdb
[java] Reads completed: 90
[java] Reads merged: 1
[java] Sectors read: 2944
[java] Nr of milliseconds spent reading: 992
[java] Writes completed: 64
[java] Writes merged: 3282
[java] Sectors written: 26768
[java] Nr of milliseconds spent writing: 2116
[java] Nr of milliseconds spent doing I/O: 948
[java] Start time: 1183334271297547000
[java] Time spent: 14421584000
[java] cpu0 %user: 95 %nice: 0 %system: 2 %iowait: 3 %idle: 1
[java] cpu1 %user: 2 %nice: 0 %system: 0 %iowait: 2 %idle: 97
[java] Starting waiting because of too many small indexes at
         time 1183334285721919000
[java] Starting merge at time 1183334285780918000
[java] Done with the merge starting at time 1183334285780918000
         after 31243513000 nanoseconds
[java] Done waiting for merging. Waited for 31302887000 nanoseconds
[java] All files added
[java] done with accumulation in memory
```

```
[java] Device sda
[java] Reads completed: 1773
[java] Reads merged: 735
[java] Sectors read: 408768
[java] Nr of milliseconds spent reading: 3608
[java] Writes completed: 6
[java] Writes merged: 3
[java] Sectors written: 72
[java] Nr of milliseconds spent writing: 16
[java] Nr of milliseconds spent doing I/O: 3540
[java] Start time: 1183334285721913000
[java] Time spent: 50728305000
[java] Device sdb
[java] Reads completed: 4673
[java] Reads merged: 3227
[java] Sectors read: 1148240
[java] Nr of milliseconds spent reading: 26648
[java] Writes completed: 1906
[java] Writes merged: 166064
[java] Sectors written: 1343784
[java] Nr of milliseconds spent writing: 716796
[java] Nr of milliseconds spent doing I/O: 18912
[java] Start time: 1183334285721913000
[java] Time spent: 50728310000
[java] cpu0 %user: 69 %nice: 0 %system: 4 %iowait: 19 %idle: 5
[java] cpu1 %user: 14 %nice: 0 %system: 2 %iowait: 13 %idle: 69
[java] done with flushing partial file
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 3
[java] Writes merged: 1
[java] Sectors written: 32
[java] Nr of milliseconds spent writing: 4
[java] Nr of milliseconds spent doing I/O: 4
[java] Start time: 1183334336451711000
[java] Time spent: 20142814000
[java] Device sdb
[java] Reads completed: 45
[java] Reads merged: 0
[java] Sectors read: 360
[java] Nr of milliseconds spent reading: 404
[java] Writes completed: 278
```

```
[java] Writes merged: 15282
[java] Sectors written: 124496
[java] Nr of milliseconds spent writing: 7532
[java] Nr of milliseconds spent doing I/O: 944
[java] Start time: 1183334336451711000
[java] Time spent: 20142816000
[java] cpu0 %user: 72 %nice: 0 %system: 0 %iowait: 3 %idle: 23
[java] cpu1 %user: 27 %nice: 0 %system: 0 %iowait: 0 %idle: 73
[java] done fixing rank for new partial file, 1000000 docs searchable.
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183334356596452000
[java] Time spent: 3189714000
[java] Device sdb
[java] Reads completed: 5
[java] Reads merged: 0
[java] Sectors read: 96
[java] Nr of milliseconds spent reading: 108
[java] Writes completed: 34
[java] Writes merged: 3161
[java] Sectors written: 25560
[java] Nr of milliseconds spent writing: 1404
[java] Nr of milliseconds spent doing I/O: 164
[java] Start time: 1183334356596452000
[java] Time spent: 3189718000
[java] cpu0 %user: 104 %nice: 0 %system: 0 %iowait: 2 %idle: 0
[java] cpu1 %user: 1 %nice: 0 %system: 0 %iowait: 0 %idle: 105
[java] Starting waiting because of too many small indexes at
         time 1183334359788985000
[java] Starting merge at time 1183334359877079000
[java] Done with the merge starting at time 1183334359877079000
         after 123000 nanoseconds
[java] Done waiting for merging. Waited for 88385000 nanoseconds
[java] Device sda
[java] Reads completed: 143501
[java] Reads merged: 59209
[java] Sectors read: 34121728
```

```
[java] Nr of milliseconds spent reading: 332472
[java] Writes completed: 689
[java] Writes merged: 356
[java] Sectors written: 8360
[java] Nr of milliseconds spent writing: 2808
[java] Nr of milliseconds spent doing I/O: 328968
[java] Start time: 1183329582964992000
[java] Time spent: 4776918319000
[java] Device sdb
[java] Reads completed: 113573
[java] Reads merged: 71897
[java] Sectors read: 26790152
[java] Nr of milliseconds spent reading: 959480
[java] Writes completed: 65493
[java] Writes merged: 5033058
[java] Sectors written: 40789880
[java] Nr of milliseconds spent writing: 21460656
[java] Nr of milliseconds spent doing I/O: 600288
[java] Start time: 1183329582964992000
[java] Time spent: 4776918034000
[java] cpu0 %user: 54 %nice: 0 %system: 1 %iowait: 7 %idle: 36
[java] cpu1 %user: 43 %nice: 0 %system: 0 %iowait: 5 %idle: 49
[java] Created the index in 79 minutes and 35 seconds.
[java] starting searches for small terms
[java] Searching for 8985741
[java] Searching for 3456645
[java] Searching for 5a90
[java] Searching for 316778
[java] Searching for x4262
[java] Searching for sc760e
[java] Searching for affectedthe
[java] Searching for d037968
[java] Searching for 190048660
[java] Searching for 164654
[java] Searching for nicololas
[java] Searching for ibbmec
[java] Searching for m11n12e
[java] Searching for hh227pc5
[java] Searching for rec18301
[java] Searching for 129x1
[java] Searching for 3002894512
[java] Searching for mar345remote
[java] Searching for fluoderm
[java] Searching for 57217907
[java] Searching for p5162345
```

```
[java] Searching for 2047593
[java] Searching for bragenski
[java] Searching for genequantification04
[java] Searching for t1654
[java] Searching for 880937
[java] Searching for meji
[java] Searching for geophyics
[java] Searching for frutex
[java] Searching for laszek
[java] Searching for fd011148
[java] Searching for smallparticles
[java] Searching for 210807
[java] Searching for sonorit
[java] Searching for daciformis
[java] Searching for blki
[java] Searching for tetraurelia
[java] Searching for bodyteen
[java] Searching for q030
[java] Searching for stipulatation
[java] Searching for hamelberg
[java] Searching for shozari
[java] Searching for 0614543
[java] Searching for miller99
[java] Searching for 279199
[java] Searching for 5176920
[java] Searching for 10076473
[java] Searching for hurger
[java] Searching for h2105
[java] Searching for nj04114
[java] Searching for 632088
[java] Searching for weblookup
[java] Searching for 1240616
[java] Searching for 1940218bg
[java] Searching for 858890
[java] Searching for vollbert
[java] Searching for pursuingbusiness
[java] Searching for subtartive
[java] Searching for ms9000e
[java] Searching for kichijoji
[java] Searching for 803194
[java] Searching for 790580
[java] Searching for 4588448
[java] Searching for 30490944
[java] Searching for nwwg4000200049
[java] Searching for 2581152
```

```
[java] Searching for 4127670
[java] Searching for otocalm
[java] Searching for pnu020
[java] Searching for 17166380
[java] Searching for 206400
[java] Searching for 19992833
[java] Searching for 8873154
[java] Searching for v90596
[java] Searching for mo00061
[java] Searching for 73ay404
[java] Searching for mdat40
[java] Searching for 0922e
[java] Searching for 14bs229
[java] Searching for lp00043725
[java] Searching for wcir023311800
[java] Searching for r0001
[java] Searching for regionallabor
[java] Searching for a786gent
[java] Searching for 5r01ca095662
[java] Searching for 1061002
[java] Searching for 2089079
[java] Searching for nrmap000100673
[java] Searching for 1802144lg
[java] Searching for 9649858
[java] Searching for 2000hnwx0005
[java] Searching for tn10231
[java] Searching for yfmz99
[java] Searching for nouve
[java] Searching for 812080100
[java] Searching for wdbda24c1ff092237
[java] Searching for monkeywrench
[java] Searching for 90et0170
[java] Searching for 12604751
[java] Searching for cedell
[java] Finished with small term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
```

```
[java] Start time: 1183334379924086000
[java] Time spent: 3196732000
[java] Device sdb
[java] Reads completed: 378
[java] Reads merged: 0
[java] Sectors read: 9512
[java] Nr of milliseconds spent reading: 3184
[java] Writes completed: 3
[java] Writes merged: 7
[java] Sectors written: 80
[java] Nr of milliseconds spent writing: 292
[java] Nr of milliseconds spent doing I/O: 3168
[java] Start time: 1183334379924086000
[java] Time spent: 3196731000
[java] cpu0 %user: 0 %nice: 0 %system: 1 %iowait: 105 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 0 %idle: 79
[java] starting searches for medium terms
[java] Searching for mailto
[java] Searching for europeans
[java] Searching for 22z
[java] Searching for 1181
[java] Searching for 91109
[java] Searching for hvn
[java] Searching for 1056
[java] Searching for postcard
[java] Searching for 6112
[java] Searching for antispam
[java] Searching for campylobacter
[java] Searching for asociaci
[java] Searching for jam
[java] Searching for phy
[java] Searching for shawn
[java] Searching for kerry
[java] Searching for rk
[java] Searching for frawley
[java] Searching for infantil
[java] Searching for meteorologists
[java] Searching for tutor
[java] Searching for seis
[java] Searching for 3042
[java] Searching for mindspring
[java] Searching for snack
[java] Searching for clemente
[java] Searching for guthrie
[java] Searching for underscores
```

```
[java] Searching for crowded
[java] Searching for infantil
[java] Searching for manipulating
[java] Searching for eli
[java] Searching for relativity
[java] Searching for 04z
[java] Searching for remuneration
[java] Searching for hugo
[java] Searching for empleo
[java] Searching for psychotherapy
[java] Searching for voa
[java] Searching for o3
[java] Searching for concentrates
[java] Searching for uty
[java] Searching for rectal
[java] Searching for qt
[java] Searching for 1134
[java] Searching for dove
[java] Searching for osage
[java] Searching for 1355
[java] Searching for 2121
[java] Searching for fbo
[java] Searching for stacked
[java] Searching for invocation
[java] Searching for sponge
[java] Searching for listeriosis
[java] Searching for cty
[java] Searching for wc3
[java] Searching for negotiators
[java] Searching for deliberative
[java] Searching for organizer
[java] Searching for underscores
[java] Searching for 24hr
[java] Searching for uphold
[java] Searching for dsc
[java] Searching for debated
[java] Searching for rck
[java] Searching for civilization
[java] Searching for 2048
[java] Searching for 1545
[java] Searching for banco
[java] Searching for causation
[java] Searching for aasa
[java] Searching for aasa
[java] Searching for manatee
```

```
[java] Searching for 22314
[java] Searching for diminishing
[java] Searching for refrigerators
[java] Searching for rp2
[java] Searching for crafted
[java] Searching for escort
[java] Searching for chung
[java] Searching for dewey
[java] Searching for 03e
[java] Searching for ninr
[java] Searching for 1431
[java] Searching for waller
[java] Searching for ep5
[java] Searching for peabody
[java] Searching for sponge
[java] Searching for posterior
[java] Searching for 6112
[java] Searching for 10z
[java] Searching for 2048
[java] Searching for vest
[java] Searching for foci
[java] Searching for 2310
[java] Searching for vlc
[java] Searching for negotiators
[java] Searching for 2350
[java] Searching for cryo
[java] Searching for 5800
[java] Finished with medium term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183334383129267000
[java] Time spent: 9167677000
[java] Device sdb
[java] Reads completed: 1175
[java] Reads merged: 54
[java] Sectors read: 37048
[java] Nr of milliseconds spent reading: 10340
```

```
[java] Writes completed: 12
[java] Writes merged: 23
[java] Sectors written: 280
[java] Nr of milliseconds spent writing: 604
[java] Nr of milliseconds spent doing I/O: 9064
[java] Start time: 1183334383129267000
[java] Time spent: 9167679000
[java] cpu0 %user: 2 %nice: 0 %system: 1 %iowait: 97 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 98
[java] starting searches for long terms
[java] Searching for research
[java] Searching for 3
[java] Searching for 10
[java] Searching for national
[java] Searching for health
[java] Searching for 2002
[java] Searching for 02
[java] Searching for will
[java] Searching for 18
[java] Searching for 15
[java] Searching for as
[java] Searching for c
[java] Searching for public
[java] Searching for 9
[java] Searching for n
[java] Searching for 30
[java] Searching for 4
[java] Searching for in
[java] Searching for been
[java] Searching for page
[java] Searching for state
[java] Searching for from
[java] Searching for or
[java] Searching for have
[java] Searching for 5
[java] Searching for may
[java] Searching for an
[java] Searching for e
[java] Searching for on
[java] Searching for would
[java] Searching for 8
[java] Searching for 2
[java] Searching for 0
[java] Searching for u
[java] Searching for to
```

```
[java] Searching for 00
[java] Searching for a
[java] Searching for was
[java] Searching for has
[java] Searching for m
[java] Searching for department
[java] Searching for 12
[java] Searching for data
[java] Searching for 16
[java] Searching for these
[java] Searching for 03
[java] Searching for if
[java] Searching for can
[java] Searching for with
[java] Searching for home
[java] Searching for not
[java] Searching for de
[java] Searching for of
[java] Searching for 01
[java] Searching for 11
[java] Searching for for
[java] Searching for is
[java] Searching for by
[java] Searching for other
[java] Searching for one
[java] Searching for html
[java] Searching for also
[java] Searching for that
[java] Searching for it
[java] Searching for this
[java] Searching for 1
[java] Searching for we
[java] Searching for be
[java] Searching for services
[java] Searching for 6
[java] Searching for program
[java] Searching for were
[java] Searching for their
[java] Searching for 14
[java] Searching for i
[java] Searching for gov
[java] Searching for 13
[java] Searching for new
[java] Searching for its
[java] Searching for information
```

```
[java] Searching for which
[java] Searching for they
[java] Searching for any
[java] Searching for you
[java] Searching for are
[java] Searching for at
[java] Searching for the
[java] Searching for b
[java] Searching for 2003
[java] Searching for and
[java] Searching for d
[java] Searching for use
[java] Searching for all
[java] Searching for no
[java] Searching for time
[java] Searching for 20
[java] Searching for 7
[java] Searching for more
[java] Searching for about
[java] Searching for s
[java] Finished with long term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1183334392312009000
[java] Time spent: 32737065000
[java] Device sdb
[java] Reads completed: 14882
[java] Reads merged: 7062
[java] Sectors read: 3243464
[java] Nr of milliseconds spent reading: 34972
[java] Writes completed: 29
[java] Writes merged: 35
[java] Sectors written: 512
[java] Nr of milliseconds spent writing: 1084
[java] Nr of milliseconds spent doing I/O: 28972
[java] Start time: 1183334392312009000
[java] Time spent: 32737068000
```

```
     [java] cpu0 %user: 36 %nice: 0 %system: 7 %iowait: 53 %idle: 0
     [java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 2 %idle: 95

BUILD SUCCESSFUL
Total time: 81 minutes 18 seconds
```

B.4 Naive B-tree index

This section contains a log from an experiment with a naive B-tree index. The experiment is run with the largest buffer size, and constructs an index with $N = 300000$ documents.

```
Buildfile: build.xml

clean:
   [delete] Deleting directory /usr/brille/bin

build:
    [mkdir] Created dir: /usr/brille/bin
    [javac] Compiling 91 source files to /usr/brille/bin
     [copy] Copying 1 file to /usr/brille/bin

deploy:
      [jar] Building jar: /usr/brille/brille.jar

run:
     [java] The operating system is Linux
     [java] Adding file /data/gov2-corpus
     [java] Have read 30000 after 1184409403055371000 nanoseconds
     [java] 30000 documents added to index
     [java] Device sda
     [java] Reads completed: 4143
     [java] Reads merged: 1842
     [java] Sectors read: 1007032
     [java] Nr of milliseconds spent reading: 29424
     [java] Writes completed: 616
     [java] Writes merged: 289
     [java] Sectors written: 7240
     [java] Nr of milliseconds spent writing: 560
     [java] Nr of milliseconds spent doing I/O: 29140
     [java] Start time: 1184407403617004000
     [java] Time spent: 1999618440000
     [java] Device sdb
     [java] Reads completed: 45
```

```
[java] Reads merged: 0
[java] Sectors read: 576
[java] Nr of milliseconds spent reading: 944
[java] Writes completed: 652216
[java] Writes merged: 1999177
[java] Sectors written: 21373656
[java] Nr of milliseconds spent writing: 251132012
[java] Nr of milliseconds spent doing I/O: 1794280
[java] Start time: 1184407403617004000
[java] Time spent: 1999618487000
[java] cpu0 %user: 31 %nice: 0 %system: 1 %iowait: 61 %idle: 4
[java] cpu1 %user: 32 %nice: 0 %system: 1 %iowait: 26 %idle: 39
[java] Have read 60000 after 1184411649131201000 nanoseconds
[java] 60000 documents added to index
[java] Device sda
[java] Reads completed: 4563
[java] Reads merged: 2178
[java] Sectors read: 1135384
[java] Nr of milliseconds spent reading: 32956
[java] Writes completed: 914
[java] Writes merged: 327
[java] Sectors written: 9928
[java] Nr of milliseconds spent writing: 1704
[java] Nr of milliseconds spent doing I/O: 33784
[java] Start time: 1184409403237018000
[java] Time spent: 2247128810000
[java] Device sdb
[java] Reads completed: 7922
[java] Reads merged: 16491
[java] Sectors read: 277608
[java] Nr of milliseconds spent reading: 234596
[java] Writes completed: 609899
[java] Writes merged: 1720037
[java] Sectors written: 18725952
[java] Nr of milliseconds spent writing: 317433064
[java] Nr of milliseconds spent doing I/O: 2244196
[java] Start time: 1184409403237018000
[java] Time spent: 2247128744000
[java] cpu0 %user: 46 %nice: 0 %system: 1 %iowait: 51 %idle: 0
[java] cpu1 %user: 42 %nice: 0 %system: 1 %iowait: 36 %idle: 19
[java] Have read 90000 after 1184413202186700000 nanoseconds
[java] 90000 documents added to index
[java] Device sda
[java] Reads completed: 4682
[java] Reads merged: 2187
```

```
[java] Sectors read: 1151928
[java] Nr of milliseconds spent reading: 29616
[java] Writes completed: 747
[java] Writes merged: 253
[java] Sectors written: 8000
[java] Nr of milliseconds spent writing: 1348
[java] Nr of milliseconds spent doing I/O: 29380
[java] Start time: 1184411650370478000
[java] Time spent: 1551974911000
[java] Device sdb
[java] Reads completed: 97642
[java] Reads merged: 15168
[java] Sectors read: 2732288
[java] Nr of milliseconds spent reading: 1618340
[java] Writes completed: 218177
[java] Writes merged: 599671
[java] Sectors written: 6573400
[java] Nr of milliseconds spent writing: 219388468
[java] Nr of milliseconds spent doing I/O: 1551880
[java] Start time: 1184411650370478000
[java] Time spent: 1551974852000
[java] cpu0 %user: 34 %nice: 0 %system: 1 %iowait: 64 %idle: 0
[java] cpu1 %user: 60 %nice: 0 %system: 0 %iowait: 22 %idle: 16
[java] Have read 120000 after 1184414759259276000 nanoseconds
[java] 120000 documents added to index
[java] Device sda
[java] Reads completed: 4550
[java] Reads merged: 2076
[java] Sectors read: 1110048
[java] Nr of milliseconds spent reading: 34904
[java] Writes completed: 746
[java] Writes merged: 251
[java] Sectors written: 7976
[java] Nr of milliseconds spent writing: 1532
[java] Nr of milliseconds spent doing I/O: 35788
[java] Start time: 1184413202347877000
[java] Time spent: 1557060332000
[java] Device sdb
[java] Reads completed: 99751
[java] Reads merged: 2067
[java] Sectors read: 2766528
[java] Nr of milliseconds spent reading: 1461436
[java] Writes completed: 220294
[java] Writes merged: 591154
[java] Sectors written: 6511392
```

```
[java] Nr of milliseconds spent writing: 220546304
[java] Nr of milliseconds spent doing I/O: 1556960
[java] Start time: 1184413202347877000
[java] Time spent: 1557060337000
[java] cpu0 %user: 33 %nice: 0 %system: 1 %iowait: 65 %idle: 0
[java] cpu1 %user: 61 %nice: 0 %system: 0 %iowait: 24 %idle: 13
[java] Have read 150000 after 1184416329542205000 nanoseconds
[java] 150000 documents added to index
[java] Device sda
[java] Reads completed: 4374
[java] Reads merged: 1911
[java] Sectors read: 1069952
[java] Nr of milliseconds spent reading: 30568
[java] Writes completed: 739
[java] Writes merged: 247
[java] Sectors written: 7888
[java] Nr of milliseconds spent writing: 1416
[java] Nr of milliseconds spent doing I/O: 31484
[java] Start time: 1184414759416000000
[java] Time spent: 1570491852000
[java] Device sdb
[java] Reads completed: 115762
[java] Reads merged: 1121
[java] Sectors read: 3189448
[java] Nr of milliseconds spent reading: 1474952
[java] Writes completed: 201452
[java] Writes merged: 542911
[java] Sectors written: 5975064
[java] Nr of milliseconds spent writing: 221655604
[java] Nr of milliseconds spent doing I/O: 1570368
[java] Start time: 1184414759416000000
[java] Time spent: 1570491853000
[java] cpu0 %user: 35 %nice: 0 %system: 1 %iowait: 63 %idle: 0
[java] cpu1 %user: 58 %nice: 0 %system: 0 %iowait: 18 %idle: 22
[java] Have read 180000 after 1184417850334368000 nanoseconds
[java] 180000 documents added to index
[java] Device sda
[java] Reads completed: 4476
[java] Reads merged: 1953
[java] Sectors read: 1093056
[java] Nr of milliseconds spent reading: 27404
[java] Writes completed: 742
[java] Writes merged: 249
[java] Sectors written: 7928
[java] Nr of milliseconds spent writing: 1232
```

```
[java] Nr of milliseconds spent doing I/O: 27568
[java] Start time: 1184416329906675000
[java] Time spent: 1520700200000
[java] Device sdb
[java] Reads completed: 112998
[java] Reads merged: 1008
[java] Sectors read: 3105720
[java] Nr of milliseconds spent reading: 1533808
[java] Writes completed: 181636
[java] Writes merged: 487851
[java] Sectors written: 5370248
[java] Nr of milliseconds spent writing: 216030564
[java] Nr of milliseconds spent doing I/O: 1520612
[java] Start time: 1184416329906675000
[java] Time spent: 1520700179000
[java] cpu0 %user: 35 %nice: 0 %system: 1 %iowait: 63 %idle: 0
[java] cpu1 %user: 61 %nice: 0 %system: 0 %iowait: 20 %idle: 17
[java] Have read 210000 after 1184419487202249000 nanoseconds
[java] 210000 documents added to index
[java] Device sda
[java] Reads completed: 4497
[java] Reads merged: 2030
[java] Sectors read: 1106712
[java] Nr of milliseconds spent reading: 31564
[java] Writes completed: 763
[java] Writes merged: 255
[java] Sectors written: 8144
[java] Nr of milliseconds spent writing: 1588
[java] Nr of milliseconds spent doing I/O: 32304
[java] Start time: 1184417850610119000
[java] Time spent: 1636660445000
[java] Device sdb
[java] Reads completed: 111932
[java] Reads merged: 4602
[java] Sectors read: 3070304
[java] Nr of milliseconds spent reading: 1747104
[java] Writes completed: 194062
[java] Writes merged: 521770
[java] Sectors written: 5739024
[java] Nr of milliseconds spent writing: 232342384
[java] Nr of milliseconds spent doing I/O: 1636560
[java] Start time: 1184417850610119000
[java] Time spent: 1636660447000
[java] cpu0 %user: 31 %nice: 0 %system: 1 %iowait: 66 %idle: 0
[java] cpu1 %user: 57 %nice: 0 %system: 0 %iowait: 21 %idle: 21
```

```
[java] Have read 240000 after 1184421158555644000 nanoseconds
[java] 240000 documents added to index
[java] Device sda
[java] Reads completed: 4450
[java] Reads merged: 2144
[java] Sectors read: 1102048
[java] Nr of milliseconds spent reading: 33480
[java] Writes completed: 805
[java] Writes merged: 270
[java] Sectors written: 8600
[java] Nr of milliseconds spent writing: 1592
[java] Nr of milliseconds spent doing I/O: 34260
[java] Start time: 1184419487278732000
[java] Time spent: 1671473727000
[java] Device sdb
[java] Reads completed: 127156
[java] Reads merged: 1541
[java] Sectors read: 3486280
[java] Nr of milliseconds spent reading: 1867620
[java] Writes completed: 177050
[java] Writes merged: 478415
[java] Sectors written: 5256528
[java] Nr of milliseconds spent writing: 236911312
[java] Nr of milliseconds spent doing I/O: 1671380
[java] Start time: 1184419487278732000
[java] Time spent: 1671473735000
[java] cpu0 %user: 36 %nice: 0 %system: 1 %iowait: 62 %idle: 0
[java] cpu1 %user: 51 %nice: 0 %system: 0 %iowait: 27 %idle: 20
[java] Have read 270000 after 1184423201726690000 nanoseconds
[java] 270000 documents added to index
[java] Device sda
[java] Reads completed: 3994
[java] Reads merged: 1769
[java] Sectors read: 975104
[java] Nr of milliseconds spent reading: 22492
[java] Writes completed: 977
[java] Writes merged: 326
[java] Sectors written: 10424
[java] Nr of milliseconds spent writing: 1760
[java] Nr of milliseconds spent doing I/O: 23752
[java] Start time: 1184421158755688000
[java] Time spent: 2043508532000
[java] Device sdb
[java] Reads completed: 154674
[java] Reads merged: 1448
```

```
[java] Sectors read: 4241904
[java] Nr of milliseconds spent reading: 2678152
[java] Writes completed: 213302
[java] Writes merged: 565534
[java] Sectors written: 6244320
[java] Nr of milliseconds spent writing: 290638500
[java] Nr of milliseconds spent doing I/O: 2043392
[java] Start time: 1184421158755688000
[java] Time spent: 2043508535000
[java] cpu0 %user: 35 %nice: 0 %system: 1 %iowait: 62 %idle: 0
[java] cpu1 %user: 35 %nice: 0 %system: 0 %iowait: 43 %idle: 20
[java] Have read 300000 after 1184427829878282000 nanoseconds
[java] 300000 documents added to index
[java] Device sda
[java] Reads completed: 4062
[java] Reads merged: 1753
[java] Sectors read: 965336
[java] Nr of milliseconds spent reading: 26628
[java] Writes completed: 1934
[java] Writes merged: 644
[java] Sectors written: 20624
[java] Nr of milliseconds spent writing: 3972
[java] Nr of milliseconds spent doing I/O: 29000
[java] Start time: 1184423202267466000
[java] Time spent: 4628150541000
[java] Device sdb
[java] Reads completed: 359780
[java] Reads merged: 2643
[java] Sectors read: 9857088
[java] Nr of milliseconds spent reading: 8351460
[java] Writes completed: 415987
[java] Writes merged: 1071796
[java] Sectors written: 11916872
[java] Nr of milliseconds spent writing: 661021175
[java] Nr of milliseconds spent doing I/O: 4627880
[java] Start time: 1184423202267466000
[java] Time spent: 4628150550000
[java] cpu0 %user: 21 %nice: 0 %system: 0 %iowait: 77 %idle: 0
[java] cpu1 %user: 11 %nice: 0 %system: 0 %iowait: 76 %idle: 12
[java] Index constructed
[java] Device sda
[java] Reads completed: 43791
[java] Reads merged: 19843
[java] Sectors read: 10716600
[java] Nr of milliseconds spent reading: 299036
```

```
[java] Writes completed: 8986
[java] Writes merged: 3112
[java] Sectors written: 96784
[java] Nr of milliseconds spent writing: 16708
[java] Nr of milliseconds spent doing I/O: 306464
[java] Start time: 1184407402586799000
[java] Time spent: 20427841016000
[java] Device sdb
[java] Reads completed: 1187766
[java] Reads merged: 46400
[java] Sectors read: 32731192
[java] Nr of milliseconds spent reading: 20971480
[java] Writes completed: 3084110
[java] Writes merged: 8578404
[java] Sectors written: 93687392
[java] Nr of milliseconds spent writing: 2867152843
[java] Nr of milliseconds spent doing I/O: 20218512
[java] Start time: 1184407402586799000
[java] Time spent: 20427840804000
[java] cpu0 %user: 32 %nice: 0 %system: 1 %iowait: 65 %idle: 0
[java] cpu1 %user: 41 %nice: 0 %system: 0 %iowait: 38 %idle: 19
[java] Created the index in 340 minutes and 26 seconds.
[java] starting searches for small terms
[java] Searching for 8985741
[java] Searching for 3456645
[java] Searching for 5a90
[java] Searching for 316778
[java] Searching for x4262
[java] Searching for sc760e
[java] Searching for affectedthe
[java] Searching for d037968
[java] Searching for 190048660
[java] Searching for 164654
[java] Searching for nicololas
[java] Searching for ibbmec
[java] Searching for m11n12e
[java] Searching for hh227pc5
[java] Searching for rec18301
[java] Searching for 129x1
[java] Searching for 3002894512
[java] Searching for mar345remote
[java] Searching for fluoderm
[java] Searching for 57217907
[java] Searching for p5162345
[java] Searching for 2047593
```

```
[java] Searching for bragenski
[java] Searching for genequantification04
[java] Searching for t1654
[java] Searching for 880937
[java] Searching for meji
[java] Searching for geophyics
[java] Searching for frutex
[java] Searching for laszek
[java] Searching for fd011148
[java] Searching for smallparticles
[java] Searching for 210807
[java] Searching for sonorit
[java] Searching for daciformis
[java] Searching for blki
[java] Searching for tetraurelia
[java] Searching for bodyteen
[java] Searching for q030
[java] Searching for stipulatation
[java] Searching for hamelberg
[java] Searching for shozari
[java] Searching for 0614543
[java] Searching for miller99
[java] Searching for 279199
[java] Searching for 5176920
[java] Searching for 10076473
[java] Searching for hurger
[java] Searching for h2105
[java] Searching for nj04114
[java] Searching for 632088
[java] Searching for weblookup
[java] Searching for 1240616
[java] Searching for 1940218bg
[java] Searching for 858890
[java] Searching for vollbert
[java] Searching for pursuingbusiness
[java] Searching for subtartive
[java] Searching for ms9000e
[java] Searching for kichijoji
[java] Searching for 803194
[java] Searching for 790580
[java] Searching for 4588448
[java] Searching for 30490944
[java] Searching for nwwg4000200049
[java] Searching for 2581152
[java] Searching for 4127670
```

```
[java] Searching for otocalm
[java] Searching for pnu020
[java] Searching for 17166380
[java] Searching for 206400
[java] Searching for 19992833
[java] Searching for 8873154
[java] Searching for v90596
[java] Searching for mo00061
[java] Searching for 73ay404
[java] Searching for mdat40
[java] Searching for 0922e
[java] Searching for 14bs229
[java] Searching for lp00043725
[java] Searching for wcir023311800
[java] Searching for r0001
[java] Searching for regionallabor
[java] Searching for a786gent
[java] Searching for 5r01ca095662
[java] Searching for 1061002
[java] Searching for 2089079
[java] Searching for nrmap000100673
[java] Searching for 1802144lg
[java] Searching for 9649858
[java] Searching for 2000hnwx0005
[java] Searching for tn10231
[java] Searching for yfmz99
[java] Searching for nouve
[java] Searching for 812080100
[java] Searching for wdbda24c1ff092237
[java] Searching for monkeywrench
[java] Searching for 90et0170
[java] Searching for 12604751
[java] Searching for cedell
[java] Finished with small term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1184427850569283000
```

```
[java] Time spent: 13846288000
[java] Device sdb
[java] Reads completed: 120
[java] Reads merged: 127
[java] Sectors read: 4084
[java] Nr of milliseconds spent reading: 55060
[java] Writes completed: 78
[java] Writes merged: 197
[java] Sectors written: 2032
[java] Nr of milliseconds spent writing: 15828
[java] Nr of milliseconds spent doing I/O: 13844
[java] Start time: 1184427850569283000
[java] Time spent: 1386287000
[java] cpu0 %user: 1 %nice: 0 %system: 0 %iowait: 95 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 73 %idle: 25
[java] starting searches for medium terms
[java] Searching for mailto
[java] Searching for europeans
[java] Searching for 22z
[java] Searching for 1181
[java] Searching for 91109
[java] Searching for hvn
[java] Searching for 1056
[java] Searching for postcard
[java] Searching for 6112
[java] Searching for antispam
[java] Searching for campylobacter
[java] Searching for asociaci
[java] Searching for jam
[java] Searching for phy
[java] Searching for shawn
[java] Searching for kerry
[java] Searching for rk
[java] Searching for frawley
[java] Searching for infantil
[java] Searching for meteorologists
[java] Searching for tutor
[java] Searching for seis
[java] Searching for 3042
[java] Searching for mindspring
[java] Searching for snack
[java] Searching for clemente
[java] Searching for guthrie
[java] Searching for underscores
[java] Searching for crowded
```

```
[java] Searching for infantil
[java] Searching for manipulating
[java] Searching for eli
[java] Searching for relativity
[java] Searching for 04z
[java] Searching for remuneration
[java] Searching for hugo
[java] Searching for empleo
[java] Searching for psychotherapy
[java] Searching for voa
[java] Searching for o3
[java] Searching for concentrates
[java] Searching for uty
[java] Searching for rectal
[java] Searching for qt
[java] Searching for 1134
[java] Searching for dove
[java] Searching for osage
[java] Searching for 1355
[java] Searching for 2121
[java] Searching for fbo
[java] Searching for stacked
[java] Searching for invocation
[java] Searching for sponge
[java] Searching for listeriosis
[java] Searching for cty
[java] Searching for wc3
[java] Searching for negotiators
[java] Searching for deliberative
[java] Searching for organizer
[java] Searching for underscores
[java] Searching for 24hr
[java] Searching for uphold
[java] Searching for dsc
[java] Searching for debated
[java] Searching for rck
[java] Searching for civilization
[java] Searching for 2048
[java] Searching for 1545
[java] Searching for banco
[java] Searching for causation
[java] Searching for aasa
[java] Searching for aasa
[java] Searching for manatee
[java] Searching for 22314
```

```
[java] Searching for diminishing
[java] Searching for refrigerators
[java] Searching for rp2
[java] Searching for crafted
[java] Searching for escort
[java] Searching for chung
[java] Searching for dewey
[java] Searching for 03e
[java] Searching for ninr
[java] Searching for 1431
[java] Searching for waller
[java] Searching for ep5
[java] Searching for peabody
[java] Searching for sponge
[java] Searching for posterior
[java] Searching for 6112
[java] Searching for 10z
[java] Searching for 2048
[java] Searching for vest
[java] Searching for foci
[java] Searching for 2310
[java] Searching for vlc
[java] Searching for negotiators
[java] Searching for 2350
[java] Searching for cryo
[java] Searching for 5800
[java] Finished with medium term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1184427864423304000
[java] Time spent: 13158866000
[java] Device sdb
[java] Reads completed: 1739
[java] Reads merged: 1044
[java] Sectors read: 48864
[java] Nr of milliseconds spent reading: 37320
[java] Writes completed: 648
```

```
[java] Writes merged: 1836
[java] Sectors written: 20136
[java] Nr of milliseconds spent writing: 1736364
[java] Nr of milliseconds spent doing I/O: 13156
[java] Start time: 1184427864423304000
[java] Time spent: 13158854000
[java] cpu0 %user: 4 %nice: 0 %system: 0 %iowait: 95 %idle: 0
[java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 101 %idle: 0
[java] starting searches for long terms
[java] Searching for research
[java] Searching for 3
[java] Searching for 10
[java] Searching for national
[java] Searching for health
[java] Searching for 2002
[java] Searching for 02
[java] Searching for will
[java] Searching for 18
[java] Searching for 15
[java] Searching for as
[java] Searching for c
[java] Searching for public
[java] Searching for 9
[java] Searching for n
[java] Searching for 30
[java] Searching for 4
[java] Searching for in
[java] Searching for been
[java] Searching for page
[java] Searching for state
[java] Searching for from
[java] Searching for or
[java] Searching for have
[java] Searching for 5
[java] Searching for may
[java] Searching for an
[java] Searching for e
[java] Searching for on
[java] Searching for would
[java] Searching for 8
[java] Searching for 2
[java] Searching for 0
[java] Searching for u
[java] Searching for to
[java] Searching for 00
```

```
[java] Searching for a
[java] Searching for was
[java] Searching for has
[java] Searching for m
[java] Searching for department
[java] Searching for 12
[java] Searching for data
[java] Searching for 16
[java] Searching for these
[java] Searching for 03
[java] Searching for if
[java] Searching for can
[java] Searching for with
[java] Searching for home
[java] Searching for not
[java] Searching for de
[java] Searching for of
[java] Searching for 01
[java] Searching for 11
[java] Searching for for
[java] Searching for is
[java] Searching for by
[java] Searching for other
[java] Searching for one
[java] Searching for html
[java] Searching for also
[java] Searching for that
[java] Searching for it
[java] Searching for this
[java] Searching for 1
[java] Searching for we
[java] Searching for be
[java] Searching for services
[java] Searching for 6
[java] Searching for program
[java] Searching for were
[java] Searching for their
[java] Searching for 14
[java] Searching for i
[java] Searching for gov
[java] Searching for 13
[java] Searching for new
[java] Searching for its
[java] Searching for information
[java] Searching for which
```

```
[java] Searching for they
[java] Searching for any
[java] Searching for you
[java] Searching for are
[java] Searching for at
[java] Searching for the
[java] Searching for b
[java] Searching for 2003
[java] Searching for and
[java] Searching for d
[java] Searching for use
[java] Searching for all
[java] Searching for no
[java] Searching for time
[java] Searching for 20
[java] Searching for 7
[java] Searching for more
[java] Searching for about
[java] Searching for s
[java] Finished with long term searches
[java] Device sda
[java] Reads completed: 0
[java] Reads merged: 0
[java] Sectors read: 0
[java] Nr of milliseconds spent reading: 0
[java] Writes completed: 0
[java] Writes merged: 0
[java] Sectors written: 0
[java] Nr of milliseconds spent writing: 0
[java] Nr of milliseconds spent doing I/O: 0
[java] Start time: 1184427877843552000
[java] Time spent: 1035562051000
[java] Device sdb
[java] Reads completed: 126911
[java] Reads merged: 5509
[java] Sectors read: 3521960
[java] Nr of milliseconds spent reading: 2326096
[java] Writes completed: 44363
[java] Writes merged: 126145
[java] Sectors written: 1375488
[java] Nr of milliseconds spent writing: 113118132
[java] Nr of milliseconds spent doing I/O: 1035496
[java] Start time: 1184427877843552000
[java] Time spent: 1035562051000
[java] cpu0 %user: 5 %nice: 0 %system: 0 %iowait: 93 %idle: 0
```

```
     [java] cpu1 %user: 0 %nice: 0 %system: 0 %iowait: 63 %idle: 36

BUILD SUCCESSFUL
Total time: 383 minutes 46 seconds
```

Appendix C

Code

All the code included here is in use in a normal run of Brille. The unit tests of all classes and various other testing scripts are excluded to limit the amount of code slightly. This code is also available from `http://www.idi.ntnu.no/~trulsamu/brille.tar.gz` to make it easier to get an overview in the editor of your choice.

C.1 brille

C.1.1 brille.BrilleDefinitions

```
package brille;

/**
 * This file contains all definitions in brille. The only other file that
 * needs to be modyfied is the properties file for log4j. All variables
 * here are explained.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public interface BrilleDefinitions {
  //The following ones should never be modyfied unless unexplained errors
  //are wanted.
  public static final int INT_SIZE = 4;
  public static final int LONG_SIZE = 8;
  public static final int SHORT_SIZE = 2;

  //Here starts the configuration of the search engine.
  //The first parts are just general stuff about where
  //to save the index and such
  public static final String INDEX_PATH = "/usr/brille/index";
  public static final String FILE_WITH_SMALL_SEARCH_TERMS =
    "/usr/brille/configs/searchlist_short.txt";
  public static final String FILE_WITH_MEDIUM_SEARCH_TERMS =
    "/usr/brille/configs/searchlist_medium.txt";
  public static final String FILE_WITH_LARGE_SEARCH_TERMS =
    "/usr/brille/configs/searchlist_long.txt";
  public static final String FILES_TO_INDEX = "/data/gov2-corpus";

  //Variable to configure the amount of documents to be indexed.
```

```
//It will go through the FILES_TO_INDEX in lexicographical order until
//this number of documents is added.
public static final int STOP_AFTER = 10000000;

//These three should not be modified
public static int BTREE_INDEX = 1;
public static int REBUILD_INDEX = 2;
public static int HIERARCHIC_INDEX = 3;

//Choose the index-type. Must be one of the three above
public static final int INDEX_TYPE = HIERARCHIC_INDEX;

//only applicable when remerge is used. Immediate merge
//set to true implies that all batches should immediately
//be merged with the main index.
public static final boolean IMMEDIATE_MERGE = false;

//Whether we are testing with the GOV2-collection, in which case the
//parsing of files is special
public static final boolean GOV2 = true;

//The size of a buffer
public static final int BUFFER_SIZE = 16384;

//The total number of bytes used for buffers
public static final int TOT_BUFFER_SIZE = 786432000;

//File naming
public static final String DOCMAN_FILE_NAME="docman.dat";

//It is possible to configure brille to use two different bufferpools,
//one for the index and one for the document manager, allthough this is not
//tested in my diploma.
public static final boolean SPLITTED_BUFFERPOOLS = false;
//this one is only applied when the above is set to true.
public static final double PART_FOR_DOCMAN = 0.2;
//This one only applies to rebuild and hierarchic
public static final double PART_FOR_BUILDING = 0.5;
//Brille may have descriptions on each occurrence of a term in a document.
//This value says how many bits that should be used.
public static final int NR_OF_BITS_FROM_DESCR=0;

//The amount of time the flushingthread in the bufferpool should wait between
//each round of flushing all dirty buffers.
public static final int SLEEP_TIME_FOR_BUFFERPOOL_FLUSHINGTHREAD=100;

//Things specific for BTREE_INDEX:
public static final int SLEEP_TIME_FOR_INDEX_UPDATE_THREAD = 50;
public static final int SLEEP_TIME_FOR_RANKINGS_UPDATE_THREAD = 64346;

//some sleeptimes
public static final int SLEEP_TIME_FOR_DOCMAN_UPDATE_THREAD = 500;
public static final int SLEEP_TIME_FOR_DELFROMBP_THREAD = 10000;

//B-tree configuration:
public static final int MAX_KEY_LENGTH = 128;
//WARNING: Do not modify the calculation here
//(It will cause strange errors (typically in FixRankingsThread))
public static final int MAX_ENTRY_LENGTH =
  ((int)BUFFER_SIZE/2)-7-MAX_KEY_LENGTH;
public static final String TREE_DRAWING = "/var/log/tree.dat";

//SortedListDictionary configuration
public static final int MAX_TERM_LENGTH = Math.min(
  (int)BUFFER_SIZE-4*INT_SIZE-LONG_SIZE,MAX_KEY_LENGTH-6);

//For the hierarchic index:
public static final int K = 2;
public static final int MAX_NR_OF_INDEXES_OF_SMALLEST_SIZE = 4;
```

```
  public static final int SLEEP_TIME_FOR_HIERARCHY_THREAD = 100;
  public static final int SLEEP_TIME_FOR_HIERARCHIC_DELETER = 10000;

  //For naive B-tree index
  public static final int NR_OF_FEEDING_THREADS = 1;
  public static final int OUTPUT_INTERVAL = STOP_AFTER/10;

  //I/O-statistics options  for Linux:
  public static final String PROPER_DISK_DEVICE_PREFIXES = "sd";
  //this implies that things like /dev/sda, /dev/sdb and so on
  //will be gathered statistics for. It will not gather info for the different
  //partitions allthough this is a very simple modification...

  //Debugging options
  public static final boolean DEBUG = false;
  public static final boolean UNPIN_BUG_DEBUG = false;
  public static final boolean HIERARCHIC_DEBUG = false;
  public static final boolean BTREE_READING_DEBUG = false;

  //When the B-tree is tested, this can be set to a positive value to enable
  //some extra testing of the amount of updates/inserts the b-tree can handle
  //each second. When most of the tree fits in the buffers, it is tested to use
  //approximately 77 microseconds on each insert/update.
  public static final int SLEEP_TIME_FOR_STATISTICSCOLLECTOR = -1;

  public static final boolean BASELINERUN = false;
  public static final boolean OUTPUT_DISK_STATISTICS = true;
}
```

C.1.2 brille.BTreeFeedingThread

```
package brille;

import java.util.concurrent.CyclicBarrier;

import org.apache.log4j.Logger;

import brille.dict.FullBTreeIndex;
import brille.docman.DocManager;
import brille.inv.IndexEntry;
import brille.utils.AlternativeGOV2FileParser;
import brille.utils.ByteArr;
import brille.utils.Document;
import brille.utils.IOStats;
import brille.utils.IOStatsGatherer;

/**
 * This class represents a thread that is capable of inserting already
 * parsed documents into a naive B-tree index.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class BTreeFeedingThread extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private AlternativeGOV2FileParser fp;
  private DocManager docMan;
  private boolean running;
  private IOStatsGatherer gath;
  private FullBTreeIndex index;
  private CyclicBarrier barrier;

  /**
   * Default constructor
   *
   * @param barrier - barrier to wait for when there are no more documents
```

```
 *  to add to the index.
 * @param index - the index to insert documents into
 * @param docMan - the document manager
 * @param fp - the file parser to retrieve parsed documents from
 * @param gath - object used to end and start gathering of I/O-statistics.
 */
public BTreeFeedingThread(CyclicBarrier barrier, FullBTreeIndex index,
    DocManager docMan, AlternativeGOV2FileParser fp, IOStatsGatherer gath){
  this.barrier = barrier;
  this.index = index;
  this.fp = fp;
  this.docMan = docMan;
  this.gath = gath;
  running = true;
}


public void run(){
  while(running){
    Document currdoc = fp.getNextDocument();
    while(currdoc!=null){
      if(BrilleDefinitions.BTREE_READING_DEBUG)
        logger.info("got a document to insert");
      int docNr = docMan.addDoc(currdoc.getUri());
      if(BrilleDefinitions.DEBUG)logger.info("got docnr "+docNr);
      docMan.setLengthForDoc(currdoc.getNumterms(),docNr);
      double numdocs=(double)(docMan.getNumActiveDocs()+1);

      double rankval = 0.0;
      for(ByteArr term : currdoc.getTermList()){
        int[] locsforThisTerm = currdoc.getTerms().get(term).getLocs().getIntArray();
        if(BrilleDefinitions.DEBUG)
          logger.info("length of the array is "+locsforThisTerm.length);
        if(term.getL()>index.getMaxBytesInTerm()){
          logger.warn("must skip term "+term.decodeString()+
              " because it is too long... " +
              "A search for this term must be done with brute force");
          continue;
        }
        double nr =
          (double)index.insert(term,new IndexEntry(docNr,locsforThisTerm,null));
        rankval+=
          Math.pow((((double)locsforThisTerm.length)*Math.log(numdocs/nr)),2.0);
      }
      if(BrilleDefinitions.DEBUG)
        logger.info("Have added all terms");
      rankval = Math.sqrt(rankval);
      if(BrilleDefinitions.DEBUG)
        logger.info("setting rankval to "+rankval+" for doc "+docNr);
      docMan.setRankingValue(docNr,rankval);
      docMan.unBlackList(docNr);
      if(BrilleDefinitions.OUTPUT_DISK_STATISTICS && (docNr%BrilleDefinitions.
          OUTPUT_INTERVAL)==0){
        if(gath!=null){
          IOStats[] res = gath.endGatheringAndReturnIOStats();
          System.out.println(docNr+" documents added to index");
          for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
          gath.startGatheringIOStats();
        }
      }
      if(BrilleDefinitions.BTREE_READING_DEBUG)
        logger.info("Done inserting document");
      currdoc = fp.getNextDocument();
    }
    if(BrilleDefinitions.BTREE_READING_DEBUG)logger.info("got a null doc");
    running = !fp.isStopped();
    if(BrilleDefinitions.BTREE_READING_DEBUG){
      if(running)logger.info("but the thread is still running");
      else logger.info("the thread is not running anymore");
```

```
      }
    }
    if(BrilleDefinitions.DEBUG)logger.info("is now done running");
    try{
      barrier.await();
    }catch(Exception e){
      e.printStackTrace();
    }
  }
}
```

C.1.3 brille.BTreeIndexMaster

```
package brille;

import java.io.BufferedReader;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.concurrent.CyclicBarrier;

import org.apache.log4j.Logger;

import brille.buffering.BuffPool;
import brille.buffering.NewBufferPool;
import brille.dict.FullBTreeIndex;
import brille.docman.DocManager;
import brille.inv.IndexEntry;
import brille.utils.AlternativeGOV2FileParser;
import brille.utils.ByteArr;
import brille.utils.FreeBSDIOStatsGatherer;
import brille.utils.IOStatsGatherer;
import brille.utils.IntArrayList;
import brille.utils.LinuxIOStatsGatherer;
import brille.utils.NoMoreDocsException;

/**
 * This class is the master of the incremental B-tree indexing method.
 * It uses a FullBTreeIndex for the index, and a DocManager for managing
 * the documents.
 *
 * Most methods here are declared in the implemented interfaces, and thus
 * commented here. Notice that the finishedBatch()-method here does not do
 * anything, because this is a fully incremental index.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class BTreeIndexMaster implements Searcher,Indexer{

  protected final Logger logger = Logger.getLogger(getClass());

  private DocManager docMan;
  private FullBTreeIndex index;
  private BuffPool docmanbp;
  private BuffPool indexbp;
  private AlternativeGOV2FileParser fp;
  private IOStatsGatherer gath;
  private BTreeFeedingThread[] threads;

  private long numTermsAdded;
  private BTreeStatisticsCollector coll;

  /**
   * Default constructor
```

```
 *
 */
public BTreeIndexMaster(CyclicBarrier barrier){
  configure();
  if(BrilleDefinitions.GOV2){
    fp = new AlternativeGOV2FileParser();
    if(BrilleDefinitions.OUTPUT_DISK_STATISTICS){
      if(PerformanceTestingOfBrille.os.equals("linux"))
        gath = new LinuxIOStatsGatherer();
      else if(PerformanceTestingOfBrille.os.equals("freebsd"))
        gath = new FreeBSDIOStatsGatherer();
      else gath = null;
      if(gath!=null) gath.startGatheringIOStats();
    }
    threads =
      new BTreeFeedingThread[BrilleDefinitions.NR_OF_FEEDING_THREADS];
    for(int i=0;i<BrilleDefinitions.NR_OF_FEEDING_THREADS;i++){
      threads[i]= new BTreeFeedingThread(barrier,index,docMan,fp,gath);
      threads[i].start();
    }
    if(gath!=null) gath.startGatheringIOStats();
    fp.start();
  }
}

private void configure(){
  numTermsAdded = 0;
  docMan = new DocManager();
  if(BrilleDefinitions.SPLITTED_BUFFERPOOLS){
    int totBuffers =
      BrilleDefinitions.TOT_BUFFER_SIZE/BrilleDefinitions.BUFFER_SIZE;
    int nrBuffersForDocMan =
      (int)(totBuffers*BrilleDefinitions.PART_FOR_DOCMAN);
    docmanbp = new NewBufferPool(nrBuffersForDocMan);
    indexbp = new NewBufferPool(totBuffers-nrBuffersForDocMan);
  }
  else{
    int totBuffers =
      BrilleDefinitions.TOT_BUFFER_SIZE/BrilleDefinitions.BUFFER_SIZE;
    docmanbp = new NewBufferPool(totBuffers);
    indexbp = docmanbp;
  }
  docMan.setBufferPool(docmanbp);
  index = new FullBTreeIndex(0,indexbp);
  docMan.setIndexForUpdateThread(index);
  if(BrilleDefinitions.SLEEP_TIME_FOR_STATISTICSCOLLECTOR>0){
    coll = new BTreeStatisticsCollector(index.getBTree(),this);
    coll.start();
  }
}

/**
 * Method for statistics measuring the number of terms added to the index.
 * Each unique term in a document is counted (but they do not need to be
 * unique over the whole document collection).
 *
 * @return the number of terms added
 */
public long getNumTermsAdded(){
  return numTermsAdded;
}


public List<Result> search(List<ByteArr> searchList, int resfrom, int resto){
  if(BrilleDefinitions.DEBUG)
    logger.info("searching for the following terms...");
  ArrayList<SearchResultHandle> l = new ArrayList<SearchResultHandle>();
  for(int i=0;i<searchList.size();i++){
    l.add(index.search(searchList.get(i)));
```

```
    if(BrilleDefinitions.DEBUG)logger.info(searchList.get(i).toString());
  }
  SearchResultMerger srm = new SearchResultMerger(l,docMan);
  if(BrilleDefinitions.DEBUG)logger.info("has made searchresultmerger");
  ArrayList<Result> ret = new ArrayList<Result>();
  if(resto==-1){
    if(BrilleDefinitions.DEBUG)logger.info("should return all results");
    while(srm.hasMoreResults()){
      Result res = srm.getNextResult();
      if(res!=null)ret.add(res);
    }
    Collections.sort(ret);
    for(int i=0;i<ret.size();i++){
      URI ins=docMan.getUriForDocNr(ret.get(i).getDocNr());
      if(ins==null)
        ret.remove(i--);
      else
        ret.get(i).setURI(ins);
    }
    ArrayList<Result> newret = new ArrayList<Result>();
    for(int i=ret.size()-1;i>=0;i--) newret.add(ret.get(i));
    ret=newret;
  }
  else{
    if(BrilleDefinitions.DEBUG)logger.info("making resultheap");
    ResultHeap rh = new ResultHeap(resto);
    while(srm.hasMoreResults()) rh.insertIfGudd(srm.getNextResult());
    for(int i=0;i<resfrom && rh.getCurrNr()>0;i++)rh.extractMin();
    for(int i=resfrom;i<resto && rh.getCurrNr()>0;i++)
      ret.add(rh.extractMin());
    for(int i=0;i<ret.size();i++){
      URI ins=docMan.getUriForDocNr(ret.get(i).getDocNr());
      if(ins==null)
        ret.remove(i--);
      else
        ret.get(i).setURI(ins);
    }
    ArrayList<Result> newret = new ArrayList<Result>();
    for(int i=ret.size()-1;i>=0;i--) newret.add(ret.get(i));
    ret=newret;
  }
  if(BrilleDefinitions.DEBUG)logger.info("returning results froms search");
  return ret;
}


public void deleteIndex() {
  if(BrilleDefinitions.DEBUG)logger.info("should delete index");
  configure();
  runGC();
  if(BrilleDefinitions.DEBUG)logger.info("have deleted index");
}


private void runGC(){
  //gc needs to be run several times...
  for(int i=0;i<6;i++) System.gc();
}


public long memUsage(){
  runGC();
  return Runtime.getRuntime().totalMemory()-
    Runtime.getRuntime().freeMemory();
}


public long diskSize() {
  return docMan.diskSize()+index.diskSize();
```

```
}

public void addDocument(URI uri, BufferedReader in)
    throws IOException, NoMoreDocsException {
  if(BrilleDefinitions.GOV2){
    if(BrilleDefinitions.BTREE_READING_DEBUG)
      logger.debug("Adding file to read "+uri.toString());
    fp.setNewFile(in);
    if(BrilleDefinitions.BTREE_READING_DEBUG)
      logger.debug("waiting for done reading...");
    fp.waitForDoneReading();
    if(BrilleDefinitions.BTREE_READING_DEBUG)
      logger.debug("done waiting. Checking if stopped...");
    if(fp.isStopped()){
      if(BrilleDefinitions.BTREE_READING_DEBUG)
        logger.debug("it is stopped");
      throw new NoMoreDocsException();
    }
    else if(BrilleDefinitions.BTREE_READING_DEBUG)
      logger.debug("it is not stopped");
  }
  else{
    if(BrilleDefinitions.DEBUG)
      logger.info("adding document with URI "+uri.toString());
    int docNr = docMan.addDoc(uri);
    if(BrilleDefinitions.DEBUG)
      logger.info(uri.toString()+" got docnr "+docNr+
          " and is added to docMan");
    String inp = in.readLine();
    HashMap<String,IntArrayList> tokensfromDoc =
      new HashMap<String,IntArrayList>();
    int currloc=0;
    while(inp!=null){
      inp = inp.toLowerCase();
      String[] ins = inp.split("\\W+");
      for(int i=0;i<ins.length;i++){
        if(ins[i].equals(""))continue;
        if(!tokensfromDoc.containsKey(ins[i]))
          tokensfromDoc.put(ins[i],new IntArrayList());
        tokensfromDoc.get(ins[i]).add(currloc++);
      }
      inp=in.readLine();
    }
    if(BrilleDefinitions.DEBUG)logger.info("finished reading in document ");
    docMan.setLengthForDoc(currloc+1,docNr);
    double numdocs=(double)(docMan.getNumActiveDocs()+1);
    double rankval = 0.0;
    for(String s:tokensfromDoc.keySet()){
      if(BrilleDefinitions.DEBUG)
        logger.info("adding entries for term "+s+" from docnr "+docNr);
      int[] locsforThisTerm = tokensfromDoc.get(s).getIntArray();
      if(BrilleDefinitions.DEBUG)
        logger.info("length of the array is "+locsforThisTerm.length);
      ByteArr toIns = new ByteArr(s);
      if(toIns.getL()>index.getMaxBytesInTerm()){
        logger.warn("must skip term "+toIns.decodeString()+
            " because it is too long... " +
            "A search for this term must be done with brute force");
        continue;
      }
      if(BrilleDefinitions.SLEEP_TIME_FOR_STATISTICSCOLLECTOR>0)
        numTermsAdded++;
      double nr =
        (double)index.insert(toIns,new IndexEntry(docNr,locsforThisTerm,null));
      rankval+=
        Math.pow((((double)locsforThisTerm.length)*Math.log(numdocs/nr)),2.0);
    }
    rankval = Math.sqrt(rankval);
```

```
      if(BrilleDefinitions.DEBUG)
        logger.info("setting rankval to "+rankval+" for doc "+docNr);
      docMan.setRankingValue(docNr,rankval);
      docMan.unBlackList(docNr);
    }
  }


  public void deleteDoc(int docNr){
    logger.error("tried to delete doc with docnr "+docNr+
        " but deletes are not supported");
    throw new Error("delete is not implemented");
  }


  public void deleteDoc(URI uri) {
    int docNr=docMan.getDocNrForURI(uri);
    if(docNr==-1)
      throw new
        IllegalArgumentException("doc that should be deleted does not exist "+
            uri.toString());
    deleteDoc(docNr);
  }


  public void shutDown(){
    if(BrilleDefinitions.DEBUG)logger.info("shutting down");
    docMan.shutDown();
    index.shutDown();
    indexbp.shutDown();
    docmanbp.shutDown();
    if(BrilleDefinitions.SLEEP_TIME_FOR_STATISTICSCOLLECTOR>0)coll.stopRunning();
    if(BrilleDefinitions.DEBUG)logger.info("done shutting down");
  }


  public void finishedBatch() { /* no-op */}
}
```

C.1.4 brille.DelFromBPThread

```
package brille;

import java.util.ArrayList;

import org.apache.log4j.Logger;

import brille.buffering.BuffPool;
import brille.dict.SortedListDictionary;

/**
 * This thread takes care of safely deleting files from its associated bufferpool.
 * When a file should be deleted, it is added here. This thread checks regularily
 * whether there are no more threads accessing the file. When there isn't, all
 * dirty buffers containing parts of this file is set to not be dirty, and the
 * file is deleted.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class DelFromBPThread extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private BuffPool bp;
  private ArrayList<SortedListDictionary> dictsToDel;
  private boolean running;
```

```
/**
 * Constructor
 *
 * @param bp - the bufferpool this should faciliate safe deletions from
 */
public DelFromBPThread(BuffPool bp){
  dictsToDel = new ArrayList<SortedListDictionary>();
  running=true;
  this.bp=bp;
}

/**
 * Method for adding a new dictionary that should be deleted.
 *
 * @param toAdd - the file number of the dictionary to be deleted from this
 *  threads associated buffer pool
 */
public synchronized void addNrToDel(SortedListDictionary toAdd){
  if(BrilleDefinitions.DEBUG)
    logger.info("adding to delthread "+dictsToDel.size());
  if(toAdd==null && BrilleDefinitions.DEBUG){
    logger.info("toAdd is NULL");
    throw new Error("toAdd is NULL");
  }
  dictsToDel.add(toAdd);
}

/**
 * Find size of list of dicts to del.
 *
 * @return size of list of dicts to del
 */
public synchronized int size(){
  return dictsToDel.size();
}

/**
 * Method for retrieving the Dictionary with the given index in the list
 * of ones to delete.
 *
 * @param i - the index of the dictionary to get
 * @return the given dictionary
 */
public synchronized SortedListDictionary get(int i){
  return dictsToDel.get(i);
}

/**
 * Method for removing a dictionary from the list of ones to be deleted.
 *
 * @param i - the index of the dictionary from the list to delete
 */
public synchronized void remove(int i){
  dictsToDel.remove(i);
}

/**
 * Method for stopping this thread.
 *
 */
public void stopRunning(){
  running=false;
}

public void run(){
  while(running){
    try{
      Thread.sleep(BrilleDefinitions.SLEEP_TIME_FOR_DELFROMBP_THREAD);
    }
```

```
      catch(Exception e){
        logger.error("could not sleep",e);
        running=false;
      }
      for(int i=0;i<size();i++){
        SortedListDictionary curr = get(i);
        if(BrilleDefinitions.DEBUG) logger.info("getting i");
        if(curr.isZeroAccessingAndDelete()){
          bp.unDirtyAllWithAndDelete(curr.getDictNr());
          bp.unDirtyAllWithAndDelete(curr.getInvNr());
          remove(i--);
        }
      }
    }
  }
}
```

C.1.5 brille.HierarchicIndex

```
package brille;

import java.util.ArrayList;

import org.apache.log4j.Logger;

import brille.dict.SortedListDictionary;
import brille.utils.IntArrayList;

/**
 * This class is made to faciliate atomic switches when merges go on in a
 * hierarchic index. It has two arraylists, one with the search dictionaries
 * (the hierarchy), and one with the new small dictionaries flushed by the
 * thread adding new documents.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class HierarchicIndex extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private boolean deleted;
  private ArrayList<SortedListDictionary> toSearch;
  private ArrayList<SortedListDictionary> small;
  private IntArrayList numBlocks;
  private IntArrayList smallBlocks;
  private int nrReading;

  /**
   * Constructor, taking all important member variables as arguments.
   *
   * @param toSearch - the current hierarchy of indexes
   * @param small - the current small indexes
   * @param numBlocks - the number of blocks in the indexes in the hierarchy
   * @param smallBlocks - the number of blocks in the small indexes
   */
  public HierarchicIndex(ArrayList<SortedListDictionary> toSearch,
      ArrayList<SortedListDictionary> small, IntArrayList numBlocks,
      IntArrayList smallBlocks){
    this.toSearch = toSearch;
    this.small = small;
    this.numBlocks = numBlocks;
    this.smallBlocks = smallBlocks;
    for(int i=0;i<toSearch.size();i++)
      if(toSearch.get(i)!=null && !toSearch.get(i).increads())
        throw new Error("could not increads for new search-index");
    for(int i=0;i<small.size();i++)
      if(!small.get(i).increads())
```

```
      throw new Error("could not increads for new small-index");
}

/**
 * Getter for the hierarchy of search dictionaries.
 *
 * @return the list of search dictionaries
 */
public ArrayList<SortedListDictionary> getSearch(){
  return toSearch;
}

/**
 * Getting the list of block sizes for the hierarchy.
 *
 * @return the list of block sizes
 */
public IntArrayList getNumBlocks(){
  return numBlocks;
}

/**
 * Getter for the list of small dictionaries.
 *
 * @return - the list of small dictionaries
 */
public synchronized ArrayList<SortedListDictionary> getSmall(){
  return small;
}

/**
 * Getter fro the number of blocks in the small dictionaries.
 *
 * @return num blocks used for small dictionaries
 */
public synchronized IntArrayList getSmallBlocks(){
  return smallBlocks;
}

/**
 * Testing whether there are any indexes in the hierarchy.
 *
 * @return whether or not there are any indexes in the hierarchy
 */
public synchronized boolean allEmpty(){
  return toSearch.size()==0;
}

/**
 * Method for adding a new small index that has been flushed.
 *
 * @param dict - the new small dictionary
 * @param blocks - the number of blocks used to flush that dictionary
 */
public void addNewSmall(SortedListDictionary dict, int blocks){
  if(!dict.increads()) throw new Error("Could not inc reads for dict");
  small.add(dict);
  smallBlocks.add(blocks);
}

/**
 * Method for incrementing the number of threads accessing this index.
 * If the index is not deleted, the request will be granted, and this
 * method will returned true. Otherwise, the thread must find another
 * index to access.
 *
 * @return whether or not the thread is allowed to access this index
 */
public synchronized boolean incNrReading(){
```

```
    if(deleted) return false;
    nrReading++;
    return true;
  }

  /**
   * Method for decrementing the number of threads accessing this index.
   *
   */
  public synchronized void decNrReading(){
    nrReading--;
  }


  private synchronized boolean isZeroReading(){
    return nrReading==0;
  }

  /**
   * Method used to delete this index. The method starts this thread,
   * which notices when no more threads reads this index, in which case it
   * will decrement the number of threads reading the different indexes. The
   * ones of them having zero readers after that will then be deleted.
   *
   */
  public synchronized void delete(){
    deleted = true;
    start();
  }

  public void run(){
    while(true){
      try{
        Thread.sleep(BrilleDefinitions.SLEEP_TIME_FOR_HIERARCHIC_DELETER);
      }
      catch(InterruptedException ie){
        logger.error("could not sleep",ie);
      }
      if(BrilleDefinitions.HIERARCHIC_DEBUG)
        logger.debug("the deletion shit is running");
      if(isZeroReading()){
        for(int i=0;i<toSearch.size();i++)
          if(toSearch.get(i)!=null)toSearch.get(i).decreads();
        for(int i=0;i<small.size();i++) small.get(i).decreads();
        break;
      }
    }
  }
}
```

C.1.6 brille.HierarchicIndexMaster

```
package brille;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.URI;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;

import org.apache.log4j.Logger;

import brille.buffering.NewBufferPool;
import brille.dict.ExtendableDocTermEntry;
```

```
import brille.dict.InMemPartialIndex;
import brille.dict.MergerThread;
import brille.dict.SortedListDictionary;
import brille.docman.StaticDocManager;
import brille.utils.ByteArr;
import brille.utils.Document;
import brille.utils.FreeBSDIOStatsGatherer;
import brille.utils.GOV2FileParser;
import brille.utils.IOStats;
import brille.utils.IOStatsGatherer;
import brille.utils.IntArrayList;
import brille.utils.LinuxIOStatsGatherer;
import brille.utils.NoMoreDocsException;

/**
 * This class is responsible for controlling the hierarchic index method in
 * Brille. The most special thing about this one is how the ranking is
 * handled. This is described in my Masters thesis, but the length of
 * a document is generally not calculated on the basis of the complete
 * document collection, but rather the part represented in the largest
 * index and the index the given document is in (if it is not the largest).
 * This provides a more efficient way of handling ranking values than
 * recalculating them for the whole collection often. As ranking is not an
 * exact science, but rather a heuristic, it is not considered a problem to
 * change the heuristic slightly.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.dict.MergerThread
 */
public class HierarchicIndexMaster implements Indexer, Searcher{

  protected final Logger logger = Logger.getLogger(getClass());

  private StaticDocManager docMan;
  private DelFromBPThread dft;
  private NewBufferPool docmanbp;
  private NewBufferPool indexbp;
  private int maxNrBuffersForBuilding;
  private int currNr;
  private InMemPartialIndex currPartialDict;
  private MergerThread mt;
  private HierarchicIndex h;
  private GOV2FileParser fp;
  private IOStatsGatherer gath;
  private Object waitingObject;

  /**
   * Default constructor
   *
   */
  public HierarchicIndexMaster(){
    h = new HierarchicIndex(new ArrayList<SortedListDictionary>(),
        new ArrayList<SortedListDictionary>(), new IntArrayList(),
        new IntArrayList());
    configure();
    dft = new DelFromBPThread(indexbp);
    dft.start();
    mt = new MergerThread(docMan,this,dft);
    mt.start();
    waitingObject = new Object();
    if(BrilleDefinitions.GOV2){
      fp = new GOV2FileParser();
      if(BrilleDefinitions.OUTPUT_DISK_STATISTICS){
        if(PerformanceTestingOfBrille.os.equals("linux"))
          gath = new LinuxIOStatsGatherer();
        else if(PerformanceTestingOfBrille.os.equals("freebsd"))
          gath = new FreeBSDIOStatsGatherer();
        else gath = null;
        if(gath!=null) gath.startGatheringIOStats();
```

```
    }
  }
}

/**
 * Method for returning the next valid suffix for a dict/invfile.
 *
 * @return the next valid suffix
 */
public synchronized int getNextNr(){
  return currNr++;
}

/**
 * Method for retrieving the maximum number of buffers pinned in memory to
 * accumulate a new index in memory. Note that this number is also equal
 * to the maximum size/buffersize of the smallest index in the hierarchy.
 *
 * @return the maximum nr of buffers used for accumulating a new index
 */
public int getMaxNrBuffersForBuilding(){
  return maxNrBuffersForBuilding;
}

/**
 * Getter for the buffer pool used by the index.
 *
 * @return the buffer pool used by the index
 */
public NewBufferPool getIndexBP(){
  return indexbp;
}

/**
 * Getter for the index. This is important, because it is not sufficient to
 * just return the current index. We must check that we are allowed to
 * access it by calling increads on the index. When that returns true, we
 * can return. Remember to use the release method afterwards.
 *
 * @return a reference to a hierarchic index we can access.
 */
public HierarchicIndex getIndex(){
  HierarchicIndex myH = h;
  while(!myH.incNrReading()) myH=h;
  return myH;
}

/**
 * Release the "lock" mentioned in the above method.
 *
 * @param myH - the index to release the lock for
 */
public void releaseIndex(HierarchicIndex myH){
  myH.decNrReading();
}

private void waitFor(){
  synchronized(waitingObject){
    try {
      if(h.getSmall().size()>=
        BrilleDefinitions.MAX_NR_OF_INDEXES_OF_SMALLEST_SIZE)
        waitingObject.wait();
    } catch (InterruptedException e) {
      logger.error("could not wait",e);
      e.printStackTrace();
      //System.exit(1);
    }
  }
}
```

```
/**
 * This method is used by the MergingThread to notify the thread adding
 * small dictionaries that the list of small dictionaries is no longer
 * empty.
 *
 */
private void notifyWaitingThreads(){
  synchronized(waitingObject){
    HierarchicIndex myH = getIndex();
    if(myH.getSmall().size()<
        BrilleDefinitions.MAX_NR_OF_INDEXES_OF_SMALLEST_SIZE)
      waitingObject.notify();
    releaseIndex(myH);
  }
}


private void configure(){
  currNr=0;
  docMan = new StaticDocManager();
  if(BrilleDefinitions.SPLITTED_BUFFERPOOLS){
    int totBuffers =
      BrilleDefinitions.TOT_BUFFER_SIZE/BrilleDefinitions.BUFFER_SIZE;
    int nrBuffersForDocMan =
      (int)(totBuffers*BrilleDefinitions.PART_FOR_DOCMAN);
    maxNrBuffersForBuilding =
      (int)((totBuffers-nrBuffersForDocMan)*
          BrilleDefinitions.PART_FOR_BUILDING);
    docmanbp = new NewBufferPool(nrBuffersForDocMan);
    indexbp = new NewBufferPool(totBuffers-nrBuffersForDocMan);
  }
  else{
    int totBuffers =
      BrilleDefinitions.TOT_BUFFER_SIZE/BrilleDefinitions.BUFFER_SIZE;
    maxNrBuffersForBuilding =
      (int)(totBuffers*BrilleDefinitions.PART_FOR_BUILDING);
    docmanbp = new NewBufferPool(totBuffers);
    indexbp = docmanbp;
  }
  docMan.setBufferPool(docmanbp);
  createNewInMemDict();
}

/**
 * Method for making a path for a new dictionary or invfile.
 *
 * @param suffix - the suffix (which is the name of the file)
 * @return a String representatin of a path
 */
public String getPath(String suffix){
  return BrilleDefinitions.INDEX_PATH+"/"+suffix;
}

/**
 * Important method used to switch indexes by the mergerthread
 *
 * @param removed - the indexes removed from the current one
 * @param removedSmall - the small indexes removed
 * @param newOne - the new index the removed ones has been merged into
 * @param numblocks - the number of blocks in the new index
 * @param position - the position the new index should have in the hierarchy
 */
public synchronized void switchIndexes(IntArrayList removed,
    IntArrayList removedSmall, SortedListDictionary newOne,
    int numblocks, int position){
  ArrayList<SortedListDictionary> small = h.getSmall();
  ArrayList<SortedListDictionary> oldOne = h.getSearch();
  IntArrayList smallblocks = h.getSmallBlocks();
```

```
    IntArrayList realBlocks = h.getNumBlocks();
    IntArrayList newSmallBlocks = new IntArrayList();
    ArrayList<SortedListDictionary> newSmall =
      new ArrayList<SortedListDictionary>();
    for(int i=0;i<small.size();i++){
      boolean in = true;
      for(int j = 0 ;j<removedSmall.size();j++)
        if(removedSmall.get(j)==small.get(i).getDictNr()){in = false; break;}
      if(in){
        newSmall.add(small.get(i));
        newSmallBlocks.add(smallblocks.get(i));
      }
    }
    IntArrayList newBlocks = new IntArrayList();
    ArrayList<SortedListDictionary> newSearch =
      new ArrayList<SortedListDictionary>();
    for(int i=0;i<=position;i++){
      newSearch.add(null);
      newBlocks.add(0);
    }
    int correctPos = 0;
    int nr = maxNrBuffersForBuilding;
    if(BrilleDefinitions.HIERARCHIC_DEBUG)
      logger.debug("numblocks is "+numblocks+" and maxBuffers is "+
          nr+" and k is "+BrilleDefinitions.K);
    while(numblocks>nr){
      correctPos++;
      nr*=BrilleDefinitions.K;
    }
    if(BrilleDefinitions.HIERARCHIC_DEBUG)
      logger.debug("found correct index to be "+correctPos);
    newSearch.set(correctPos,newOne);
    newBlocks.replace(numblocks,correctPos);
    for(int i=position+1;i<oldOne.size();i++){
      newSearch.add(oldOne.get(i));
      newBlocks.add(realBlocks.get(i));
    }
    HierarchicIndex newH =
      new HierarchicIndex(newSearch,newSmall,newBlocks,newSmallBlocks);
    if(BrilleDefinitions.HIERARCHIC_DEBUG){
      logger.debug("now ready to switch. The newH has "+newSearch.size()+
          " in search and "+newSmall.size()+" in small");
      logger.debug("The blocksizes are now in search:");
      for(int i=0;i<newSearch.size();i++)
        logger.debug(i+": "+newBlocks.get(i));
    }
    HierarchicIndex oldH = h;
    oldH.delete();
    h = newH;
    notifyWaitingThreads();
  }


  private synchronized void addNewSmall(SortedListDictionary dict,
      int numBlocks){
    HierarchicIndex myH = getIndex();
    myH.addNewSmall(dict,numBlocks);
    releaseIndex(myH);
  }


  private void createNewInMemDict(){
    if(BrilleDefinitions.HIERARCHIC_DEBUG) logger.debug("new in dict");
    if(currPartialDict!=null){
      if(BrilleDefinitions.HIERARCHIC_DEBUG) logger.debug("should flush");
      if(gath!=null){
        IOStats[] res = gath.endGatheringAndReturnIOStats();
        System.out.println("done with accumulation in memory");
        for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
```

```
      gath.startGatheringIOStats();
    }
    int numblocks = currPartialDict.flush();
    if(gath!=null){
      IOStats[] res = gath.endGatheringAndReturnIOStats();
      System.out.println("done with flushing partial file");
      for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
      gath.startGatheringIOStats();
    }
    if(BrilleDefinitions.HIERARCHIC_DEBUG) logger.debug("done flushing");
    SortedListDictionary dict =
      new SortedListDictionary(currPartialDict.getDictNr(),
          currPartialDict.getInvNr(),indexbp,docMan,h.allEmpty()?
              null:this,numblocks);
    if(gath!=null){
      IOStats[] res = gath.endGatheringAndReturnIOStats();
      System.out.println("done fixing rank for new partial file, "+
          docMan.getNumActiveDocs()+" docs searchable.");
      for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
      gath.startGatheringIOStats();
    }
    if(BrilleDefinitions.HIERARCHIC_DEBUG)
      logger.debug("done fixing ranks for simple");
    addNewSmall(dict,numblocks);
    if(BrilleDefinitions.HIERARCHIC_DEBUG)
      logger.debug("added new small dict");
  }
  HierarchicIndex myH = getIndex();
  while(myH.getSmall().size()>=
    BrilleDefinitions.MAX_NR_OF_INDEXES_OF_SMALLEST_SIZE){
    releaseIndex(myH);
    if(BrilleDefinitions.HIERARCHIC_DEBUG) logger.debug("need to wait");
    long startWait = System.nanoTime();
    System.out.println("Starting waiting because of too " +
        "many small indexes at time "+startWait);
    waitFor();
    long waitedFor = System.nanoTime()-startWait;
    System.out.println("Done waiting for merging. Waited for "+
        waitedFor+" nanoseconds");
    if(BrilleDefinitions.HIERARCHIC_DEBUG) logger.debug("done waiting");
    myH = getIndex();
  }
  releaseIndex(myH);
  int nnr=getNextNr();
  String ipath = getPath("inv"+nnr+".dat");
  String dpath = getPath("dict"+nnr+".dat");
  currPartialDict =
    new InMemPartialIndex(indexbp,maxNrBuffersForBuilding,ipath,dpath);
}


public void addDocument(URI uri,BufferedReader in)
throws FileNotFoundException, IOException, NoMoreDocsException {
  if(BrilleDefinitions.GOV2){
    if(docMan.getNumActiveDocs()>4000000){
      System.out.println("Adding new file "+uri.toString());
    }
    fp.setNewFile(in);
    Document currdoc = fp.getNextDocument();
    while(currdoc!=null){
      int docNr = docMan.addDoc(currdoc.getUri());

      docMan.setLengthForDoc(currdoc.getNumterms(),docNr);
      if(BrilleDefinitions.DEBUG) logger.debug("done reading in doc");
      while(!currPartialDict.addDocument(
          currdoc.getTerms(),currdoc.getTermList(),docNr,
          currdoc.getRoomInInMemIndex()))
        createNewInMemDict();
      if(BrilleDefinitions.DEBUG) logger.debug("done inserting");
```

```
        docMan.unBlackList(docNr);
        if(BrilleDefinitions.DEBUG) logger.debug("adddocument finished");
        currdoc = fp.getNextDocument();
      }
      if(fp.isStopped()){
        throw new NoMoreDocsException();
      }
    }
    else{
      if(BrilleDefinitions.DEBUG)
        logger.debug("should add document "+uri.toString());
      int docNr = docMan.addDoc(uri);
      if(BrilleDefinitions.DEBUG)
        logger.debug("done adding to docma, and got docnr "+docNr);
      ArrayList<ByteArr> terms = new ArrayList<ByteArr>();
      int roomInInMemIndex =0;

      String inp = in.readLine();
      HashMap<String,ExtendableDocTermEntry> tokensfromDoc =
        new HashMap<String,ExtendableDocTermEntry>();
      int currloc=1;
      while(inp!=null){
        inp = inp.toLowerCase();
        String[] ins = inp.split("\\W+");
        for(int i=0;i<ins.length;i++){
          if(ins[i].equals(""))continue;
          if(!tokensfromDoc.containsKey(ins[i])){
            ByteArr toAdd = new ByteArr(ins[i]);
            if(toAdd.getL()>BrilleDefinitions.MAX_TERM_LENGTH){
              logger.warn("skipping term "+toAdd.decodeString());
              continue;
            }
            tokensfromDoc.put(ins[i],new ExtendableDocTermEntry());
            if(toAdd.getL()<1)
              throw new Error("have a term with less than 1 in length!");
            terms.add(toAdd);
            roomInInMemIndex+=(6+toAdd.getL());
          }
          tokensfromDoc.get(ins[i]).addLoc(currloc++);
          roomInInMemIndex+=4;
        }
        inp=in.readLine();
      }
      docMan.setLengthForDoc(currloc,docNr);
      if(BrilleDefinitions.DEBUG) logger.debug("done reading in doc");
      while(!currPartialDict.addDoc(
          tokensfromDoc,terms,docNr,roomInInMemIndex)){
        createNewInMemDict();
      }
      if(BrilleDefinitions.DEBUG) logger.debug("done inserting");
      docMan.unBlackList(docNr);
      if(BrilleDefinitions.DEBUG) logger.debug("adddocument finished");
    }
  }


  public void finishedBatch(){
    createNewInMemDict();
  }


  public long diskSize() {
    return docMan.diskSize()+indexbp.totalDiskSize();
  }

  /**
   * Method for retrieving the number of docs in the largest index containing
   * the argument term. This is used extensively in ranking.
   *
```

```
 * @param term - the term to get the doc count for
 * @param myH - the hierarchic index (incread done)
 * @return - the doc cnt in the largest index
 */
public int getCntInLargest(ByteArr term, HierarchicIndex myH){
  ArrayList<SortedListDictionary> indexes = myH.getSearch();
  return indexes.get(indexes.size()-1).getNumDocsContainingTerm(term);
}


private void runGC(){
  //gc needs to be run several times...
  for(int i=0;i<6;i++) System.gc();
}


public long memUsage() {
  runGC();
  return Runtime.getRuntime().totalMemory()-
    Runtime.getRuntime().freeMemory();
}


public void deleteDoc(int docNr) {
  throw new Error("deletDoc is not implemented for hierarchic");
}


public void deleteDoc(URI uri) {
  deleteDoc(docMan.getDocNrForURI(uri));
}


public List<Result> search(List<ByteArr> searchList, int resfrom, int resto){
  HierarchicIndex myH = h;
  while(!myH.incNrReading())myH = h;
  ArrayList<SortedListDictionary> currIndexes = myH.getSearch();
  ArrayList<SortedListDictionary> currSmallIndexes = myH.getSmall();
  if(BrilleDefinitions.HIERARCHIC_DEBUG)
    logger.debug("in search. Length of search is "+currIndexes.size()+
        ", and length of small is "+currSmallIndexes.size());
  IntArrayList num = new IntArrayList();
  if(currIndexes.size()>0) for(int i=0;i<searchList.size();i++)
    num.add(getCntInLargest(searchList.get(i),myH));
  ArrayList<Result> ret = new ArrayList<Result>();
  if(resto==-1){
    for(int j=0;j<currIndexes.size();j++){
      if(currIndexes.get(j)==null) continue;
      ArrayList<SearchResultHandle> l =
        new ArrayList<SearchResultHandle>(searchList.size());
      for(int i=0;i<searchList.size();i++){
        if(j==currIndexes.size()-1)
          l.add(currIndexes.get(j).search(searchList.get(i)));
        else l.add(currIndexes.get(j).search(searchList.get(i),num.get(i)));
      }
      SearchResultMerger srm = new SearchResultMerger(l,docMan);
      while(srm.hasMoreResults()) ret.add(srm.getNextResult());
    }
    for(int j=0;j<currSmallIndexes.size();j++){
      if(currSmallIndexes.get(j)==null) continue;
      ArrayList<SearchResultHandle> l =
        new ArrayList<SearchResultHandle>(searchList.size());
      if(currIndexes.size()>0) for(int i=0;i<searchList.size();i++)
        l.add(currSmallIndexes.get(j).search(searchList.get(i),num.get(i)));
      else for(int i=0;i<searchList.size();i++)
        l.add(currSmallIndexes.get(j).search(searchList.get(i)));
      SearchResultMerger srm = new SearchResultMerger(l,docMan);
      while(srm.hasMoreResults()) ret.add(srm.getNextResult());
    }
```

```
      Collections.sort(ret);
      ArrayList<Result> newret = new ArrayList<Result>();
      for(int i=ret.size()-1;i>=0;i--){
        URI ins=docMan.getUriForDocNr(ret.get(i).getDocNr());
        if(ins!=null){
          ret.get(i).setURI(ins);
          newret.add(ret.get(i));
        }
      }
      ret = newret;
    }
    else{
      ResultHeap rh = new ResultHeap(resto);
      for(int j=0;j<currIndexes.size();j++){
        if(currIndexes.get(j)==null) continue;
        ArrayList<SearchResultHandle> l =
          new ArrayList<SearchResultHandle>(searchList.size());
        for(int i=0;i<searchList.size();i++){
          if(j==currIndexes.size()-1)
            l.add(currIndexes.get(j).search(searchList.get(i)));
          else
            l.add(currIndexes.get(j).search(searchList.get(i),num.get(i)));
        }
        SearchResultMerger srm = new SearchResultMerger(l,docMan);
        while(srm.hasMoreResults()) rh.insertIfGudd(srm.getNextResult());
      }
      for(int j=0;j<currSmallIndexes.size();j++){
        if(currSmallIndexes.get(j)==null) continue;
        ArrayList<SearchResultHandle> l =
          new ArrayList<SearchResultHandle>(searchList.size());
        if(currIndexes.size()>0) for(int i=0;i<searchList.size();i++)
          l.add(currSmallIndexes.get(j).search(searchList.get(i),num.get(i)));
        else for(int i=0;i<searchList.size();i++)
          l.add(currSmallIndexes.get(j).search(searchList.get(i)));
        SearchResultMerger srm = new SearchResultMerger(l,docMan);
        while(srm.hasMoreResults()) rh.insertIfGudd(srm.getNextResult());
      }
      ArrayList<Result> newret = new ArrayList<Result>();
      for(int i=0;i<resfrom && rh.getCurrNr()>0;i++)rh.extractMin();
      for(int i=resfrom;i<resto && rh.getCurrNr()>0;i++)
        ret.add(rh.extractMin());
      for(int i=ret.size()-1;i>=0;i--){
        URI ins=docMan.getUriForDocNr(ret.get(i).getDocNr());
        if(ins!=null){
          ret.get(i).setURI(ins);
          newret.add(ret.get(i));
        }
      }
      ret = newret;
    }
    releaseIndex(myH);
    return ret;
  }


  public void deleteIndex() {
    throw new IllegalArgumentException("delete not implemented");
  }


  public void shutDown() {
    docMan.shutDown();
    dft.stopRunning();
    mt.stopRunning();
    if(BrilleDefinitions.GOV2) fp.stopRunning();
  }
}
```

C.1.7 brille.Indexer

```
package brille;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.net.URI;

import brille.utils.NoMoreDocsException;

/**
 * This interface defines what a class responsible for indexing should
 * provide of methods.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public interface Indexer{

  /**
   * Method for adding a new document.
   *
   * @param uri - an uri describing a path to the document
   * @param in - a BufferedReader capable of reading the document
   * @throws FileNotFoundException
   * @throws IOException
   * @throws NoMoreDocsException
   */
  public void addDocument(URI uri,BufferedReader in)
    throws FileNotFoundException, IOException, NoMoreDocsException;


  /**
   * Method to be called when all documents in the current batch
   * are added. This will make sure the indexer builds the complete index.
   * This method is a no-op for incremental indexes.
   *
   */
  public void finishedBatch();

  /**
   * Method for testing diskusage
   *
   * @return disksize used by the index
   */
  public long diskSize();

  /**
   * Method for testing the amount of memory currently used.
   *
   * @return the number of bytes of memory used
   */
  public long memUsage();

  /**
   * Delete the document with the given document number.
   *
   * @param docNr -the document number of the document to delete
   */
  public void deleteDoc(int docNr);

  /**
   * Delete the document with the given uri.
   *
   * @param uri - the uri of the document to delete
   */
  public void deleteDoc(URI uri);

  /**
```

```
   * Method called when the application shuts down to make sure that all
   * running threads are stopped
   *
   */
  public void shutDown();
}
```

C.1.8 brille.NotImplementedException

```
package brille;

/**
 * The old classic Exception.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class NotImplementedException extends Exception {

  private static final long serialVersionUID = -4178112624313311797L;

  public NotImplementedException(String cause){
    super(cause);
  }

}
```

C.1.9 brille.PerformanceTestingOfBrille

```
package brille;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.File;
import java.io.InputStreamReader;
import java.net.URI;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CyclicBarrier;
import java.util.zip.ZipEntry;
import java.util.zip.ZipInputStream;

import org.apache.log4j.Logger;

import brille.docman.StaticDocManager;
import brille.utils.ByteArr;
import brille.utils.FreeBSDIOStatsGatherer;
import brille.utils.IOStats;
import brille.utils.IOStatsGatherer;
import brille.utils.LinuxIOStatsGatherer;
import brille.utils.NoMoreDocsException;

/**
 * This class is used to start running performance tests of brille.
 * It only provides methods to add files to the index and search.
 * It starts by creating an index of the kind defined in
 * brille.BrilleDefinitions.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class PerformanceTestingOfBrille {

  protected static final Logger logger =
    Logger.getLogger("PerformanceTesting");
```

```
public static String os;

public static void addAllFilesInDirToMaster(File dir, Indexer master)
throws Exception{
  File[] fs = dir.listFiles();
  Arrays.sort(fs);
  for(int i=0;i<fs.length;i++){
    if(fs[i].isDirectory()){
      addAllFilesInDirToMaster(fs[i],master);
    }
    else{
      if(BrilleDefinitions.DEBUG)
        System.out.println("Trying to add document "+fs[i].toString());
      BufferedReader f = new BufferedReader(new FileReader(fs[i]));
      master.addDocument(fs[i].getAbsoluteFile().toURI(), f);
      f.close();
      if(BrilleDefinitions.DEBUG)
        System.out.println("finished adding that doc");
    }
  }
}


public static void addAllFilesInFileToMaster(ZipInputStream z,
    Indexer master) throws Exception{
  BufferedReader in = new BufferedReader(new InputStreamReader(z),65536);
  ZipEntry ze = z.getNextEntry();
  while(ze!=null){
    if(!ze.isDirectory()) master.addDocument(new URI(ze.getName()),in);
    z.closeEntry();
    ze = z.getNextEntry();
  }
  in.close();
}

private static void performAllSearches(ArrayList<ArrayList<ByteArr>> l,
    Searcher s){
  for(int i=0;i<l.size();i++){
    if(BrilleDefinitions.OUTPUT_DISK_STATISTICS)
      System.out.println("Searching for "+l.get(i).get(0).decodeString());
    List<Result> res = s.search(l.get(i),0,10);
    //this is just done to do something with the results, to ensure that
    //no optimizations can remove the search.
    for(int j=0;j<res.size()-1;j++)
      if(res.get(j).getRelevance()<res.get(j).getRelevance())
        throw new Error("bad ranking");
  }
}

private static ArrayList<ArrayList<ByteArr>>
    getAllSearches(BufferedReader in) throws Exception{
  ArrayList<ArrayList<ByteArr>> ret = new ArrayList<ArrayList<ByteArr>>();
  int num = Integer.parseInt(in.readLine());
  for(int k=0;k<num;k++){
    String searchFor = in.readLine().toLowerCase();
    ByteArr se = new ByteArr(searchFor);
    ArrayList<ByteArr> sea = new ArrayList<ByteArr>();
    sea.add(se);
    ret.add(sea);
  }
  return ret;
}

public static void main(String[] args) throws Exception{
  try{
    IOStatsGatherer gath=null;
    if(BrilleDefinitions.OUTPUT_DISK_STATISTICS){
      os = System.getProperty("os.name").toLowerCase();
```

```
    if(os.equals("freebsd")){
      System.out.println("The operating system is FreeBSD");
      gath = new FreeBSDIOStatsGatherer();
      gath.startGatheringIOStats();
    }
    else if(os.equals("linux")){
      System.out.println("The operating system is Linux");
      gath = new LinuxIOStatsGatherer();
      gath.startGatheringIOStats();
    }
    else{
      System.out.println("Found your os to be "+os);
      System.out.println("Gathering of I/O statistics is not supported"+
          " on this operating system.");
      System.out.println("Feel free to implement it yourself. Have a look"+
          " at the solutions for other operating systems in brille.utils,"+
          " and send me an email if you want to contribute " +
          "(trulsamu@idi.ntnu.no). I'll be grateful:)");
    }
  }
  String[] inp = BrilleDefinitions.FILES_TO_INDEX.split("\\|");
  File[] dirs = new File[inp.length];
  for(int i=0;i<dirs.length;i++)dirs[i]=new File(inp[i]);

  if(BrilleDefinitions.DEBUG)System.out.println("Starting run..");
  if(BrilleDefinitions.INDEX_TYPE==BrilleDefinitions.BTREE_INDEX){
    if(BrilleDefinitions.DEBUG)
      System.out.println("This run is a btree-run");
    CyclicBarrier barrier =
      new CyclicBarrier(BrilleDefinitions.NR_OF_FEEDING_THREADS+1);
    BTreeIndexMaster master = new BTreeIndexMaster(barrier);
    if(BrilleDefinitions.DEBUG)
      System.out.println("Finished creating master");
    long startTime = System.currentTimeMillis();
    try{
      for(int j=0;j<dirs.length;j++){
        if(BrilleDefinitions.OUTPUT_DISK_STATISTICS)
          System.out.println("Adding file "+dirs[j].getAbsolutePath());
        if(dirs[j].getName().endsWith("zip"))
          addAllFilesInFileToMaster(
              new ZipInputStream(new FileInputStream(dirs[j])),master);
        else addAllFilesInDirToMaster(dirs[j],master);
      }
    }catch(NoMoreDocsException nmde){}
    barrier.await();
    if(gath!=null){
      IOStats[] vals = gath.endGatheringAndReturnIOStats();
      System.out.println("Index constructed");
      for(int i=0;i<vals.length;i++)
        System.out.println(vals[i].toString());
    }
    long elapsedTime = System.currentTimeMillis()-startTime;
    long mins = elapsedTime / 60000;
    long secs = elapsedTime - mins*60000;
    secs/=1000;
    System.out.println("Created the index in "+mins+
        " minutes and "+secs+" seconds.");

    if(BrilleDefinitions.DEBUG)
      System.out.println("Finished adding files. Sleeping....");
    Thread.sleep(20000);
    if(BrilleDefinitions.DEBUG)System.out.println("Done sleeping");

    BufferedReader in = new BufferedReader(
        new FileReader(BrilleDefinitions.FILE_WITH_SMALL_SEARCH_TERMS));
    ArrayList<ArrayList<ByteArr>> searchfor = getAllSearches(in);
    if(gath!=null){
      System.out.println("starting searches for small terms");
      gath.startGatheringIOStats();
```

```
    }
    performAllSearches(searchfor,master);
    if(gath!=null){
      IOStats[] vals = gath.endGatheringAndReturnIOStats();
      System.out.println("Finished with small term searches");
      for(int i=0;i<vals.length;i++)
        System.out.println(vals[i].toString());
    }
    in = new BufferedReader(
        new FileReader(BrilleDefinitions.FILE_WITH_MEDIUM_SEARCH_TERMS));
    searchfor = getAllSearches(in);
    if(gath!=null){
      System.out.println("starting searches for medium terms");
      gath.startGatheringIOStats();
    }
    performAllSearches(searchfor,master);
    if(gath!=null){
      IOStats[] vals = gath.endGatheringAndReturnIOStats();
      System.out.println("Finished with medium term searches");
      for(int i=0;i<vals.length;i++)
        System.out.println(vals[i].toString());
    }
    in = new BufferedReader(
        new FileReader(BrilleDefinitions.FILE_WITH_LARGE_SEARCH_TERMS));
    searchfor = getAllSearches(in);
    if(gath!=null){
      System.out.println("starting searches for long terms");
      gath.startGatheringIOStats();
    }
    performAllSearches(searchfor,master);
    if(gath!=null){
      IOStats[] vals = gath.endGatheringAndReturnIOStats();
      System.out.println("Finished with long term searches");
      for(int i=0;i<vals.length;i++)
        System.out.println(vals[i].toString());
    }
    master.shutDown();
    if(BrilleDefinitions.DEBUG)
      System.out.println("master is now shut down.");
  }
  else{
    Indexer master=null;
    if(BrilleDefinitions.INDEX_TYPE==BrilleDefinitions.REBUILD_INDEX)
      master = new RemergeIndexMaster();
    else master = new HierarchicIndexMaster();
    if(BrilleDefinitions.DEBUG)
      System.out.println("Finished creating master");
    long startTime = System.currentTimeMillis();
    try{
      for(int j=0;j<dirs.length;j++){
        if(BrilleDefinitions.OUTPUT_DISK_STATISTICS)
          System.out.println("adding file "+dirs[j].getAbsolutePath());
        if(dirs[j].getName().endsWith("zip"))
          addAllFilesInFileToMaster(
              new ZipInputStream(new FileInputStream(dirs[j])),master);
        else addAllFilesInDirToMaster(dirs[j],master);
      }
    }catch(NoMoreDocsException nmde){}
    if(BrilleDefinitions.OUTPUT_DISK_STATISTICS)
      System.out.println("All files added");
    master.finishedBatch();
    if(gath!=null){
      IOStats[] vals = gath.endGatheringAndReturnIOStats();
      for(int i=0;i<vals.length;i++)
        System.out.println(vals[i].toString());
    }
    long elapsedTime = System.currentTimeMillis()-startTime;
    long mins = elapsedTime / 60000;
    long secs = elapsedTime - mins*60000;
```

```
secs/=1000;
System.out.println("Created the index in "+mins+
    " minutes and "+secs+" seconds.");

if(BrilleDefinitions.DEBUG)
  System.out.println("Finished adding files. Sleeping....");
Thread.sleep(20000);
if(BrilleDefinitions.DEBUG)System.out.println("Done sleeping");
int cnt = 0;
do{
  BufferedReader in = new BufferedReader(
      new FileReader(BrilleDefinitions.FILE_WITH_SMALL_SEARCH_TERMS));
  ArrayList<ArrayList<ByteArr>> searchfor = getAllSearches(in);
  if(gath!=null){
    System.out.println("starting searches for small terms");
    gath.startGatheringIOStats();
  }
  performAllSearches(searchfor,(Searcher)master);
  if(gath!=null){
    IOStats[] vals = gath.endGatheringAndReturnIOStats();
    System.out.println("Finished with small term searches");
    for(int i=0;i<vals.length;i++)
      System.out.println(vals[i].toString());
  }
  in = new BufferedReader(
      new FileReader(BrilleDefinitions.FILE_WITH_MEDIUM_SEARCH_TERMS));
  searchfor = getAllSearches(in);
  if(gath!=null){
    System.out.println("starting searches for medium terms");
    gath.startGatheringIOStats();
  }
  performAllSearches(searchfor,(Searcher)master);
  if(gath!=null){
    IOStats[] vals = gath.endGatheringAndReturnIOStats();
    System.out.println("Finished with medium term searches");
    for(int i=0;i<vals.length;i++)
      System.out.println(vals[i].toString());
  }
  in = new BufferedReader(
      new FileReader(BrilleDefinitions.FILE_WITH_LARGE_SEARCH_TERMS));
  searchfor = getAllSearches(in);
  if(gath!=null){
    System.out.println("starting searches for long terms");
    gath.startGatheringIOStats();
  }
  performAllSearches(searchfor,(Searcher)master);
  if(gath!=null){
    IOStats[] vals = gath.endGatheringAndReturnIOStats();
    System.out.println("Finished with long term searches");
    for(int i=0;i<vals.length;i++)
      System.out.println(vals[i].toString());
  }
  cnt++;
}while(BrilleDefinitions.BASELINERUN && cnt<1000);

if(BrilleDefinitions.BASELINERUN){
  RemergeIndexMaster m = ((RemergeIndexMaster)master);
  System.out.println("Used "+m.getSearchDictSearchTime()+
      " nanoseconds searching for 300000 entries in the dictionary");

  StaticDocManager access = m.getDocMan();
  long insertTime = m.getTimeSpentInsertingInBTree();
  long starttime = 0;
  int numToIns = 1000000-access.getNumActiveDocs();
  for(int i=0;i<numToIns;i++){
    URI test = new URI("testing"+i);
    starttime = System.nanoTime();
    int docNr = access.addDoc(test);
    insertTime += (System.nanoTime()-starttime);
```

```
          access.unBlackList(docNr);
        }
        System.out.println("Used "+insertTime+
            " nanoseconds adding 1000000 entries to B-tree");

        long startTimeAccess = 0;
        long elapsedTimeAccess = 0;
        int numDocs = m.getDocMan().getNumActiveDocs();
        for(int j=0;j<100000;j++){
          int val = 1+(int)(Math.random()*numDocs);
          startTimeAccess = System.nanoTime();
          URI uri= access.getUriForDocNr(val);
          elapsedTimeAccess += (System.nanoTime()-startTimeAccess);
          logger.debug("It is "+uri.toString().length()+" chars in uri");
        }
        System.out.println("Used "+elapsedTimeAccess+
            " nanosecond searching for 100000 entries in B-tree");
      }

      if(BrilleDefinitions.INDEX_TYPE==BrilleDefinitions.REBUILD_INDEX)
        ((RemergeIndexMaster)master).shutDown();
      else ((HierarchicIndexMaster)master).shutDown();
      if(BrilleDefinitions.DEBUG)
        System.out.println("master is now shut down.");
    }
  }catch(Exception e){
    System.out.println("Exception in main thread!!!");
    e.printStackTrace();
  }catch(Error e){
    System.out.println("Error in main thread!!");
    e.printStackTrace();
  }
 }
}
```

C.1.10 brille.RemergeIndexMaster

```
package brille;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.URI;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;

import org.apache.log4j.Logger;

import brille.buffering.BuffPool;
import brille.buffering.NewBufferPool;
import brille.buffering.UndirtyThread;
import brille.dict.ExtendableDocTermEntry;
import brille.dict.InMemPartialIndex;
import brille.dict.SortedListDictionary;
import brille.docman.StaticDocManager;
import brille.utils.ByteArr;
import brille.utils.Document;
import brille.utils.FreeBSDIOStatsGatherer;
import brille.utils.GOV2FileParser;
import brille.utils.IOStats;
import brille.utils.IOStatsGatherer;
import brille.utils.IntArrayList;
import brille.utils.LinuxIOStatsGatherer;
import brille.utils.NoMoreDocsException;
```

```
/**
 * This class is the master of the remerge method. When new documents are
 * added, they are accumulated in memory until either a complete batch is
 * added or the memory is full. When all documents in a batch is added,
 * all partial indexes (which are now flushed to disk) are merged together
 * to form a single inverted index.
 *
 * During updates, the batch is added, and merged into the current search
 * index when the complete batch is added.
 *
 * Most methods here are defined in the implemented interfaces, and are thus
 * commented there.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class RemergeIndexMaster implements Indexer,Searcher{

  protected final Logger logger = Logger.getLogger(getClass());

  private int currNr;
  private InMemPartialIndex currPartialDict;
  private IntArrayList mergedicts;
  private IntArrayList mergeinvs;
  private SortedListDictionary searchDict;
  private StaticDocManager docMan;
  private BuffPool docmanbp;
  private BuffPool indexbp;
  private int maxNrBuffersForBuilding;
  private DelFromBPThread dft;
  private GOV2FileParser fp;
  //private SweetParser fp;
  private IOStatsGatherer gath;

  //variables for baseline run
  private long timeSpent;
  private long startTime;
  private long timeSpentWithAddingDocsInMem;
  private long numBytesAddedToMemIndex;
  /**
   * Default constructor
   *
   */
  public RemergeIndexMaster(){
    if(BrilleDefinitions.BASELINERUN){
      timeSpent=0;
      timeSpentWithAddingDocsInMem=0;
      numBytesAddedToMemIndex=0;
    }
    configure();
    dft = new DelFromBPThread(indexbp);
    dft.start();
    mergedicts = new IntArrayList();
    mergeinvs = new IntArrayList();
    if(BrilleDefinitions.GOV2){
      fp = new GOV2FileParser();
      //fp = new SweetParser(this);
      if(BrilleDefinitions.OUTPUT_DISK_STATISTICS){
        if(PerformanceTestingOfBrille.os.equals("linux"))
          gath = new LinuxIOStatsGatherer();
        else if(PerformanceTestingOfBrille.os.equals("freebsd"))
          gath = new FreeBSDIOStatsGatherer();
        else gath = null;
        if(gath!=null) gath.startGatheringIOStats();
      }
    }
  }
```

```java
public void shutDown(){
  docMan.shutDown();
  dft.stopRunning();
  indexbp.shutDown();
  docmanbp.shutDown();
  if(BrilleDefinitions.GOV2) fp.stopRunning();
}

public StaticDocManager getDocMan(){
  return docMan;
}

public long getSearchDictSearchTime(){
  return searchDict.getSearchTime();
}


private int getNextNr(){
  return currNr++;
}

private void configure(){
  currNr=0;
  mergedicts = new IntArrayList();
  mergeinvs = new IntArrayList();
  searchDict=null;
  docMan = new StaticDocManager();
  if(BrilleDefinitions.SPLITTED_BUFFERPOOLS){
    int totBuffers =
      BrilleDefinitions.TOT_BUFFER_SIZE/BrilleDefinitions.BUFFER_SIZE;
    int nrBuffersForDocMan =
      (int)(totBuffers*BrilleDefinitions.PART_FOR_DOCMAN);
    maxNrBuffersForBuilding =
      (int)((totBuffers-nrBuffersForDocMan)*
          BrilleDefinitions.PART_FOR_BUILDING);
    docmanbp = new NewBufferPool(nrBuffersForDocMan);
    indexbp = new NewBufferPool(totBuffers-nrBuffersForDocMan);
  }
  else{
    int totBuffers =
      BrilleDefinitions.TOT_BUFFER_SIZE/BrilleDefinitions.BUFFER_SIZE;
    if(BrilleDefinitions.DEBUG) logger.info("numbuffers is "+totBuffers);
    maxNrBuffersForBuilding =
      (int)(totBuffers*BrilleDefinitions.PART_FOR_BUILDING);
    if(BrilleDefinitions.DEBUG)
      logger.info("maxnrBuffersForBuilding is "+maxNrBuffersForBuilding);
    docmanbp = new NewBufferPool(totBuffers);
    indexbp = docmanbp;
  }
  docMan.setBufferPool(docmanbp);
  createNewInMemDict();
}

/**
 * Method for creating a new path with the argument suffix
 *
 * @param suffix - the suffix of the path to create(typically the file name)
 * @return a String representation of the path
 */
public String getPath(String suffix){
  return BrilleDefinitions.INDEX_PATH+"/"+suffix;
}

public long getTimeSpentInsertingInBTree(){
  return timeSpent;
}


private void createNewInMemDict(){
```

```
  if(BrilleDefinitions.DEBUG) logger.debug("creating new inMemDict");
  if(currPartialDict!=null){
    if(BrilleDefinitions.IMMEDIATE_MERGE) finishedBatch();
    else{
      mergedicts.add(currPartialDict.getDictNr());
      mergeinvs.add(currPartialDict.getInvNr());
      if(gath!=null){
        IOStats[] res = gath.endGatheringAndReturnIOStats();
        System.out.println("done with accumulation in memory");
        for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
        gath.startGatheringIOStats();
      }
      if(BrilleDefinitions.BASELINERUN){
        timeSpentWithAddingDocsInMem+=
          (currPartialDict.getTimeSpentAddingDocs());
        numBytesAddedToMemIndex+=currPartialDict.getUsedBytes();
      }
      currPartialDict.flush();
      if(BrilleDefinitions.BASELINERUN){
        long flushing = currPartialDict.getTimeSpentFlushing();
        System.out.println("Used "+flushing+
            " nanoseconds to flush the new docs in inmemindex. "+
            "There are now "+docMan.getNumActiveDocs()+" docs flushed.");
      }
      if(gath!=null){
        IOStats[] res = gath.endGatheringAndReturnIOStats();
        System.out.println("done with flushing partial file");
        for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
        gath.startGatheringIOStats();
      }
    }
  }
  int nnr=getNextNr();
  String ipath = getPath("inv"+nnr+".dat");
  String dpath = getPath("dict"+nnr+".dat");
  currPartialDict =
    new InMemPartialIndex(indexbp,maxNrBuffersForBuilding,ipath,dpath);
}


public void addDocument(URI uri, BufferedReader in)
    throws FileNotFoundException, IOException, NoMoreDocsException {
  if(BrilleDefinitions.GOV2){
    fp.setNewFile(in);
    Document currdoc = fp.getNextDocument();
    while(currdoc!=null){
      if(BrilleDefinitions.BASELINERUN){

        startTime = System.nanoTime();
      }
      int docNr = docMan.addDoc(currdoc.getUri());
      if(BrilleDefinitions.BASELINERUN){
        timeSpent += (System.nanoTime()-startTime);
      }
      docMan.setLengthForDoc(currdoc.getNumterms(),docNr);
      if(BrilleDefinitions.DEBUG) logger.debug("done reading in doc");
      while(!currPartialDict.addDocument(currdoc.getTerms(),
          currdoc.getTermList(),docNr,currdoc.getRoomInInMemIndex()))
        createNewInMemDict();
      if(BrilleDefinitions.DEBUG) logger.debug("done inserting");
      docMan.unBlackList(docNr);
      if(BrilleDefinitions.DEBUG) logger.debug("adddocument finished");
      currdoc = fp.getNextDocument();
    }
    if(fp.isStopped()){
      throw new NoMoreDocsException();
    }
  }
  else{
```

```
    if(BrilleDefinitions.DEBUG)
      logger.debug("should add document "+uri.toString());
    int docNr = docMan.addDoc(uri);
    if(BrilleDefinitions.DEBUG)
      logger.debug("done adding to docma, and got docnr "+docNr);
    ArrayList<ByteArr> terms = new ArrayList<ByteArr>();
    int roomInInMemIndex =0;

    String inp = in.readLine();
    HashMap<String,ExtendableDocTermEntry> tokensfromDoc =
      new HashMap<String,ExtendableDocTermEntry>();
    int currloc=1;
    while(inp!=null){
      inp = inp.toLowerCase();
      String[] ins = inp.split("\\W+");
      for(int i=0;i<ins.length;i++){
        if(ins[i].equals(""))continue;
        if(!tokensfromDoc.containsKey(ins[i])){
          ByteArr toAdd = new ByteArr(ins[i]);
          if(toAdd.getL()>BrilleDefinitions.MAX_TERM_LENGTH){
            logger.warn("skipping term "+toAdd.decodeString());
            continue;
          }
          tokensfromDoc.put(ins[i],new ExtendableDocTermEntry());
          if(toAdd.getL()<1)
            throw new Error("have a term with less than 1 in length!");
          terms.add(toAdd);
          roomInInMemIndex+=(6+toAdd.getL());
        }
        tokensfromDoc.get(ins[i]).addLoc(currloc++);
        roomInInMemIndex+=4;
      }

      inp=in.readLine();
    }
    docMan.setLengthForDoc(currloc,docNr);
    if(BrilleDefinitions.DEBUG) logger.debug("done reading in doc");
    while(!currPartialDict.addDoc(tokensfromDoc,terms,
        docNr,roomInInMemIndex))
      createNewInMemDict();
    if(BrilleDefinitions.DEBUG) logger.debug("done inserting");
    docMan.unBlackList(docNr);
    if(BrilleDefinitions.DEBUG) logger.debug("adddocument finished");
  }
}

public void fixSingleDocument(URI uri, int currloc,HashMap<ByteArr,
    ExtendableDocTermEntry> tokensfromDoc, ArrayList<ByteArr> terms,
    int roomInInMemIndex){
  int docNr = docMan.addDoc(uri);
  docMan.setLengthForDoc(currloc,docNr);
  if(BrilleDefinitions.DEBUG) logger.debug("done reading in doc");
  while(!currPartialDict.addDocument(tokensfromDoc,terms,
      docNr,roomInInMemIndex))
    createNewInMemDict();
  if(BrilleDefinitions.DEBUG) logger.debug("done inserting");
  docMan.unBlackList(docNr);
  if(BrilleDefinitions.DEBUG) logger.debug("adddocument finished");
}

public void finishedBatch() {

  if(BrilleDefinitions.DEBUG) logger.debug("finishedBatch() called");
  if(currPartialDict!=null){
    if(BrilleDefinitions.DEBUG)
      logger.debug("adding mergedicts and mergeinvs");
    mergedicts.add(currPartialDict.getDictNr());
    mergeinvs.add(currPartialDict.getInvNr());
    if(BrilleDefinitions.DEBUG)
```

```
        logger.debug("flushing currPartialDict");
      if(gath!=null){
        IOStats[] res = gath.endGatheringAndReturnIOStats();
        System.out.println("done with accumulation in memory");
        for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
        gath.startGatheringIOStats();
      }
      if(BrilleDefinitions.BASELINERUN){
        numBytesAddedToMemIndex+=currPartialDict.getUsedBytes();
        timeSpentWithAddingDocsInMem+=currPartialDict.getTimeSpentAddingDocs();
      }
      currPartialDict.flush();
      if(BrilleDefinitions.BASELINERUN){
        long flushing = currPartialDict.getTimeSpentFlushing();
        System.out.println("Used "+flushing+
            " nanoseconds to flush the new docs in inmemindex. "+
            "There are now "+docMan.getNumActiveDocs()+" docs flushed.");
      }
      if(gath!=null){
        IOStats[] res = gath.endGatheringAndReturnIOStats();
        System.out.println("done flushing partial dict");
        for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
        gath.startGatheringIOStats();
      }
      if(BrilleDefinitions.DEBUG)
        logger.debug("Done flushing currPartialDict");
    }
    if(mergedicts.size()>0){
      SortedListDictionary oldSearchDict = searchDict;
      int limit = mergedicts.size();
      if(searchDict!=null){
        mergedicts.add(searchDict.getDictNr());
        mergeinvs.add(searchDict.getInvNr());
      }
      if(BrilleDefinitions.DEBUG)logger.info("creating searchDict");
      if(mergedicts.size()>1){
        searchDict =
          new SortedListDictionary(getNextNr(),indexbp,
              mergedicts,mergeinvs,docMan,null);
        if(gath!=null){
          IOStats[] res = gath.endGatheringAndReturnIOStats();
          System.out.println("done merging all files. "+
              docMan.getNumActiveDocs()+" docs searchable");
          for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
          gath.startGatheringIOStats();
        }
        if(BrilleDefinitions.DEBUG)
          logger.info("One max on disk usage is "+diskSize());
        for(int i=0;i<limit;i++)
          new UndirtyThread(indexbp,mergedicts.get(i)).start();
        for(int i=0;i<limit;i++)
          new UndirtyThread(indexbp,mergeinvs.get(i)).start();
        if(oldSearchDict!=null)dft.addNrToDel(oldSearchDict);
      }
      else{
        searchDict =
          new SortedListDictionary(mergedicts.get(0),
            mergeinvs.get(0),indexbp,docMan,null,0);
        if(gath!=null){
          IOStats[] res = gath.endGatheringAndReturnIOStats();
          System.out.println("done merging one file. "+
              docMan.getNumActiveDocs()+" docs searchable");
          for(int i=0;i<res.length;i++) System.out.println(res[i].toString());
          gath.startGatheringIOStats();
        }
      }
      mergedicts = new IntArrayList();
      mergeinvs = new IntArrayList();
    }
```

```
  if(BrilleDefinitions.BASELINERUN){
    System.out.println("Used "+timeSpentWithAddingDocsInMem+
        " nanoseconds adding "+numBytesAddedToMemIndex+
        " bytes to memory resident index.");
  }
  if(BrilleDefinitions.DEBUG) logger.debug("nothing more to be done!");
}


public long diskSize() {
  return docMan.diskSize()+indexbp.totalDiskSize();
}


private void runGC(){
  //gc needs to be run several times...
  for(int i=0;i<6;i++) System.gc();
}


public long memUsage() {
  runGC();
  return Runtime.getRuntime().totalMemory()-
    Runtime.getRuntime().freeMemory();
}


public void deleteDoc(int docNr) {
  throw new Error("delete not implemented");
}


public void deleteDoc(URI uri) {
  deleteDoc(docMan.getDocNrForURI(uri));
}


public List<Result> search(List<ByteArr> searchList, int resfrom, int resto){
  SortedListDictionary currSearchDict = searchDict;
  while(!currSearchDict.increads()) currSearchDict=searchDict;
  ArrayList<SearchResultHandle> l = new ArrayList<SearchResultHandle>();
  for(int i=0;i<searchList.size();i++){
    l.add(i,currSearchDict.search(searchList.get(i)));
  }
  SearchResultMerger srm = new SearchResultMerger(l,docMan);
  ArrayList<Result> ret = new ArrayList<Result>();
  if(resto==-1){
    while(srm.hasMoreResults()){
      Result res = srm.getNextResult();
      if(res!=null)ret.add(res);
    }
    Collections.sort(ret);
    for(int i=0;i<ret.size();i++){
      URI ins=docMan.getUriForDocNr(ret.get(i).getDocNr());
      if(ins==null)
        ret.remove(i--);
      else
        ret.get(i).setURI(ins);
    }
    ArrayList<Result> newret = new ArrayList<Result>();
    for(int i=ret.size()-1;i>=0;i--) newret.add(ret.get(i));
    ret=newret;
  }
  else{
    ResultHeap rh = new ResultHeap(resto);
    while(srm.hasMoreResults()) rh.insertIfGudd(srm.getNextResult());
    for(int i=0;i<resfrom && rh.getCurrNr()>0;i++)rh.extractMin();
    for(int i=resfrom;i<resto && rh.getCurrNr()>0;i++)
      ret.add(rh.extractMin());
```

```
      for(int i=0;i<ret.size();i++){
        URI ins=docMan.getUriForDocNr(ret.get(i).getDocNr());
        if(ins==null)
          ret.remove(i--);
        else
          ret.get(i).setURI(ins);
      }
      ArrayList<Result> newret = new ArrayList<Result>();
      for(int i=ret.size()-1;i>=0;i--) newret.add(ret.get(i));
      ret=newret;
    }
    return ret;
  }


  public void deleteIndex() {
    dft.addNrToDel(searchDict);
    searchDict=null;
  }

  /**
   * Method used to dump all terms in the dictionary, to be able to pick
   * search terms that are interesting to test searches with.
   *
   * @param out - A printwriter to dump the dictionary to.
   */
  public void dumpDictionary(PrintWriter out){
    SortedListDictionary currSearchDict = searchDict;
    while(!currSearchDict.increads()) currSearchDict=searchDict;
    currSearchDict.flushDictionary(out);
    currSearchDict.decreads();
  }
}
```

C.1.11 brille.Result

```
package brille;

import java.net.URI;

/**
 * This class represents a result from a search in brille.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.ResultHeap
 */
public class Result implements Comparable<Result>{
  private int docNr;
  private double relevance;
  private URI doc;

  /**
   * Constructor with the document number and relevance of the result.
   * The uri is set later on when we know that this result should be
   * returned to the user.
   *
   * @param docNr - the document number of the result
   * @param relevance - the computed relevance of the result
   */
  public Result(int docNr, double relevance){
    this.docNr=docNr;
    this.relevance=relevance;
  }

  /**
   * Method for setting the uri when we know that this result should be
   * returned to the user.
```

```
   *
   * @param doc - the uri to this document
   */
  public void setURI(URI doc){
    this.doc=doc;
  }

  /**
   * Getter for the document number
   *
   * @return the doc nr
   */
  public int getDocNr(){
    return docNr;
  }

  /**
   * Getter for the relevance
   *
   * @return the relevance
   */
  public double getRelevance(){
    return relevance;
  }

  /**
   * Getter for the uri
   *
   * @return the uri
   */
  public URI getURI(){
    return doc;
  }

  public int compareTo(Result other){
    if(relevance<other.getRelevance()) return -1;
    else if(other.getRelevance()<relevance) return 1;
    return 0;
  }

  public String toString(){
    if(doc==null) return "Doc: "+docNr+" with relevance: "+relevance;
    return "Doc: "+docNr+" with URI: "+
      doc.toString()+" and relevance: "+relevance;
  }
}
```

C.1.12 brille.ResultHeap

```
package brille;

/**
 * This class is basically a min-heap implementation with a few additional
 * methods. It is very handy when searching for the top k results, or the
 * results from t to k. In both cases we build a min-heap with k elements.
 * We first insert the first k results, and then build the heap. When we
 * want to evaluate the next result, we check its relevance to the smallest
 * one in the heap (the top of the heap). If it has higher ranking, wi insert
 * it in the heap, replacing the previous smallest value.
 *
 * In this way finding the top k results can be done in O(n log(k)) with only
 * O(k) memory.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class ResultHeap {
  private int cap;
```

```
private int currNr;
private Result[] l;
private boolean sorted;

/**
 * Constructor
 *
 * @param cap - the capacity of this min-heap
 */
public ResultHeap(int cap){
  this.cap=cap;
  currNr=0;
  l=new Result[cap];
  sorted=false;
}

/**
 * Method for testing whether the next result has got a better relevance
 * than the currently smallest one, in which case it is inserted in the
 * heap.
 *
 * @param r - the new result to test
 */
public void insertIfGudd(Result r){
  if(currNr<cap){
    l[currNr++]=r;
    if(currNr==cap) buildHeap();
  }
  else if(r.compareTo(l[0])>0){
    l[0]=r;
    minHeapify(0);
  }
}

private void buildHeap(){
  for(int i=(cap/2)-1;i>=0;i--){
    minHeapify(i);
  }
  sorted=true;
}

private void minHeapify(int ind){
  int le = ind*2+1;
  int ri = ind*2+2;
  if(le>=currNr)return;
  int min= ri>=currNr?le:l[le].compareTo(l[ri])<0?le:ri;
  if(l[ind].compareTo(l[min])>0){
    Result tmp = l[ind];
    l[ind]=l[min];
    l[min]=tmp;
    minHeapify(min);
  }
}

/**
 * Method for retrieving the current number of entries in this heap.
 *
 * @return the number of entries in the heap
 */
public int getCurrNr(){
  return currNr;
}

/**
 * Method for extracting the lowest ranked result in this heap.
 *
 * @return the lowest ranked result in this heap
 */
public Result extractMin(){
```

```
    if(!sorted)buildHeap();
    Result ret = l[0];
    if(currNr>1){
      l[0]=l[--currNr];
      minHeapify(0);
    }else currNr=0;
    return ret;
  }
}
```

C.1.13 brille.Searcher

```
package brille;

import java.util.List;

import brille.utils.ByteArr;

/**
 * Interface used to define what kind of methods an object supporting
 * searches should provide.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public interface Searcher {
  /**
   * Method for searching for a single term.
   *
   * @param term - the term to search for
   * @return a list of termrefs to hits
   */
  public List<Result> search(List<ByteArr> searchList, int from, int to);


  /**
   * Called when you want to change the index. This method will delete it,
   * and you can build a new one from scratch by adding documents
   */
  public void deleteIndex();

  /**
   * @return The amount of disk used by the index.
   */
  public long diskSize();

}
```

C.1.14 brille.SearchResultHandle

```
package brille;

import brille.inv.IndexEntry;

/**
 * This method describes overall how all result iterators in Brille should
 * provide. All results ar returned through iterators to avoid using to much
 * memory.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public abstract class SearchResultHandle {

  /**
   * Retrieving the index entry the iterator is currently at.
   *
```

```
   * @return the current entry
   */
  public abstract IndexEntry getEntry();

  /**
   * Method used to move the iterator one step forwards, to the next resulting
   * index entry.
   *
   */
  public abstract void next();

  /**
   * Method for retrieving the number of results in the inverted list
   * currently beeing read by the iterator.
   *
   * @return the number of results
   */
  public abstract int getNrOfResults();

  /**
   * Method for retrieving the total number of occurrences of the term searched
   * for in the current inverted list.
   *
   * @return total number of occurrences of the term searched for
   */
  public abstract int getNrOcc();

  /**
   * Method used to release the iterator. That should be done when we know we
   * want use it anymore because locks will be unlocked and buffers unpinned
   * when calling this method.
   *
   */
  public abstract void release();
}
```

C.1.15 brille.SearchResultMerger

```
package brille;

import java.util.ArrayList;

import org.apache.log4j.Logger;

import brille.inv.IndexEntry;
import brille.docman.DocMan;

/**
 * This class is a heap of iterators. It contains one iterator for each term
 * searched for. These iterators are sorted on their document number in a
 * min-heap. When a complete result should be returned, results are extracted
 * from this heap until an entry with a different document number is
 * extracted. In this way, it is simple to calculate the complete rank for
 * the given document.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class SearchResultMerger {

  protected final Logger logger = Logger.getLogger(getClass());

  private ArrayList<SearchResultHandle> l;
  private int cap;
  private DocMan docMan;

  /**
   * Constructor taking the list of iterators, and the document manager
```

```
 * from which to retrieve information about ranking values for the
 * different documents.
 *
 * @param l - the list of iterators
 * @param docMan - the document manager containing information about
 *  the indexed documents
 */
public SearchResultMerger(ArrayList<SearchResultHandle> l, DocMan docMan){
  this.l=l;
  if(BrilleDefinitions.DEBUG && l==null) logger.info("l is null!");
  for(int i=0;i<l.size();i++)
    if(l.get(i).getEntry()==null)
      l.remove(i--);
  cap=l.size();
  buildMinHeap();
  this.docMan=docMan;
}


private void buildMinHeap(){
  for(int i=(cap-1)/2;i>=0;i--)minHeapify(i);
}


private void minHeapify(int ind){
  int le = ind*2+1;
  int ri = ind*2+2;
  if(le>=cap)return;
  int min= ri>=cap?
      le:l.get(le).getEntry().compareTo(l.get(ri).getEntry())<0?le:ri;
  if(l.get(ind).getEntry().compareTo(l.get(min).getEntry())>0){
    SearchResultHandle tmp = l.get(ind);
    l.set(ind,l.get(min));
    l.set(min,tmp);
    minHeapify(min);
  }
}

/**
 * Method for testing whether this heap has more results left.
 *
 * @return whether or not this heap has more results to return
 */
public boolean hasMoreResults(){
  return cap!=0;
}

private void inc(int i){
  l.get(i).next();
  if(l.get(i).getEntry()==null){
    l.get(i).release();
    if(cap>1)l.set(i,l.remove(--cap));
    else{
      cap=0;
      l=new ArrayList<SearchResultHandle>();
      return;
    }
  }
  minHeapify(i);
}

/**
 * Method for obtaining the next result as described above.
 *
 * @return the next result
 */
public Result getNextResult(){
  boolean gudd = false;
  double rank=0.0;
```

```
    int docNr=-1;
    while(!gudd){
      if(cap==0) return null;
      IndexEntry first = l.get(0).getEntry();
      while(first==null){
        l.get(0).release();
        l.remove(0);
        cap--;
        if(cap==0) return null;
        first=l.get(0).getEntry();
      }
      rank=(double)first.getLocations().length;
      rank*=Math.log((double)docMan.getNumActiveDocs()/
          (double)l.get(0).getNrOfResults());
      inc(0);
      docNr=first.getDocNr();
      while(nextEquals(first)){
        IndexEntry n = l.get(0).getEntry();
        rank+=(n.getLocations().length*
            Math.log((double)docMan.getNumActiveDocs()/
                (double)l.get(0).getNrOfResults()));
        inc(0);
      }
      if(!docMan.isBlackList(docNr)) gudd=true;

    }
    if(BrilleDefinitions.DEBUG)logger.info("returning one result");
    return new Result(docNr,rank/docMan.getRankingValueForDoc(docNr));
  }


  private boolean nextEquals(IndexEntry ie){
    if(l.size()==0) return false;
    else return ie.getDocNr()==l.get(0).getEntry().getDocNr();
  }
}
```

C.2 brille.btree

C.2.1 brille.btree.BTree

```
package brille.btree;

import java.util.ArrayList;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.RandomAccessFile;
import java.nio.channels.FileChannel;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.NotImplementedException;
import brille.buffering.Buffer;
import brille.buffering.BuffPool;
import brille.utils.IntArrayList;

/**
 * This class contains the implementation of the general B-tree in Brille.
 * It is capable of containing arbitrary entries. The entries should inherit
 * from brille.btree.Entry, and one must also make internal entries inheriting
 * from brille.btree.InternalEntry.
 *
 * The B-tree is chained on all levels. This is done to enable a B-tree that
 * guarantees serilizable executions regardless of the number of threads
 * accessing it simultaneously. All actual entries are stored on the leaf
 * level.
 *
 * The B-tree supports insertions, deletions, searches, and two special
 * operations needed in the FullBTreeIndex called increment-or-insert and
 * decrement-or-delete. Deletions of entries are supported, but to enable more
 * efficient locking schemes and to avoid a too complex implementation,
 * deletion of nodes is not supported. This implies that the upper limit on
 * space overhead most B-trees have in common does not hold here, but it is
 * considered unlikely that a document collection will shrink dramatically.
 *
 * When performing one of the mentioned operations (except search), the
 * operation will be committed when it is performed on the leaf level, and
 * the performing thread will return from the function. If the operations
 * requires updates in higher levels in the tree (for instance when a node is
 * splitted), this will be performed in a thread on its own, called the
 * UpdateThread.
 *
 * A search in the B-tree will return a BTreeHandle, which is an iterator over
 * hits in the B-tree. This handle must be released when it should not be used
 * any more.
 *
 * Note the two factories required as arguments in the constructor of the
 * B-tree. Factories are needed to create new instances of the entries. The
 * reason why generics in Java could not be used is the, in my opinion, stupid
 * implementation of generics in java, where the type only exists compile time,
 * making the structure far less useful.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.btree.Node
 * @see brille.btree.UpdateThread
 */
public class BTree {

  protected final Logger logger = Logger.getLogger(getClass());
```

```
private PrintWriter td;
private InternalEntryFactory ief;
private EntryFactory lef;
private int nextNr;
private int rootlevel;
private int root;
private ArrayList<Update> updates;
private File f;
private FileChannel channel;
private BuffPool bp;
private UpdateThread ut;
private int fileNr;
private ReentrantReadWriteLock rootLock;

private long numInserts;
private long numUpdates;



/**
 * The constructor where all parameters can be set for the B-tree.
 *
 * @param filename - the name of the file where the B-tree will be stored
 * @param bp - the buffer pool to pin buffers from
 * @param sleepTimeForUpdateThread - the number of milliseconds the
 *  UpdateThread should sleep before it checks for new updates
 * @param ief - the factory for internal entries
 * @param lef - the factory for leaf entries
 */
public BTree(String filename, BuffPool bp,long sleepTimeForUpdateThread,
    InternalEntryFactory ief, EntryFactory lef){
  numInserts = 0;
  numUpdates = 0;
  rootLock = new ReentrantReadWriteLock(true);
  try{
    f = new File(filename);
    if(f.exists())f.delete();
    channel = new RandomAccessFile(f, "rw").getChannel();
  }catch(FileNotFoundException e){
    e.printStackTrace();
    System.exit(1);
  }
  this.ief=ief;
  this.lef=lef;
  this.bp=bp;
  fileNr=bp.addFile(channel,f);
  root = 0;
  rootlevel=0;
  Buffer r = bp.pinBuffer(root,fileNr,false,true);
  try{
    Entry in = lef.make();
    in.setKeyToMax();
    Node rootNode = createNode(r,(short)0,root,in);
    if(BrilleDefinitions.DEBUG){
      logger.info("have created rootnode:");
      logger.info(rootNode.toString());
    }
  }finally{
    bp.unpinBuffer(r,false,true);
  }
  nextNr=1;
  updates=new ArrayList<Update>();
  ut=new UpdateThread(sleepTimeForUpdateThread,this);
  ut.start();
  updates = new ArrayList<Update>();
  nextNr=1;
  if(BrilleDefinitions.DEBUG)logger.debug("new tree created");
  try{
    td = new PrintWriter(new File(BrilleDefinitions.TREE_DRAWING));
```

```
  }catch(Exception e){
    System.out.println("Exception with treedrawer");
    e.printStackTrace();
  }
}

/**
 * Constructor using the default value of 50 for sleeptime of the
 * UpdateThread. Otherwise, the parameters are as above.
 *
 * @param filename - the name of the file where the B-tree will be stored
 * @param bp - the buffer pool to pin buffers from
 * @param ief - the factory for internal entries
 * @param lef - the factory for leaf entries
 */
public BTree(String filename, BuffPool bp, InternalEntryFactory ief,
    EntryFactory lef){
  this(filename, bp, 50,ief,lef);
}

/**
 * This method is used when considering the efficiency of the B-tree.
 * It retrieves the number of user initiated insertions that have been
 * performed in this tree, counting both normal inserts, and inserts
 * caused by incorins.
 *
 * @return the number of user initiated inserts in the B-tree
 */
public long getNumInserts(){
  return numInserts;
}

/**
 * This method is also used for gathering statistics about the efficiency
 * of the B-tree. It retrieves the number of updates caused by incorins.
 *
 * @return the number of updates caused by incorins in this B-tree
 */
public long getNumUpdates(){
  return numUpdates;
}

/**
 * Convenience method for creating a new Node with the given parameters.
 *
 * @param buf - the buffer where the node should be initialized
 * @param level - the level of the node in the B-tree
 * @param part - the part number in the file of the B-tree
 * @param maxKey - the maximum key in the B-tree
 * @param nextPointer - the pointer to the next node from this new node
 * @return the new Node.
 */
private Node createNode(Buffer buf, short level,int part, Entry maxKey,
    int nextPointer){
  Node ret = new Node(buf,level==0?lef:ief,bp,part,fileNr);
  ret.init(level,maxKey,nextPointer);
  return ret;
}

/**
 * Convenience method for creating a new Node when the next-pointer should
 * point to null.
 *
 * @param buf - the buffer where the node should be initialized
 * @param level - the level of the node in the B-tree
 * @param part - the part number in the file of the B-tree
 * @param maxKey - the maximum key in the B-tree
 * @return the new Node
 */
```

```
private Node createNode(Buffer buf, short level,int part, Entry maxKey){
  return createNode(buf,level,part,maxKey,Integer.MAX_VALUE);
}

/**
 * This method is used to get the next part of the file that should be
 * used when creating a new node.
 *
 * @return the next part of the file to use for a new node
 */
private synchronized int getNextFree(){
  return nextNr++;
}

/**
 * Method used to delete Entry e from the leaf level in the B-tree.
 *
 * @param e - the entry to delete
 */
public void delete(Entry e){
  deleteAtLevel(e,(short)0);
}

private void readLockRoot(){
  rootLock.readLock().lock();
}

private void readUnlockRoot(){
  rootLock.readLock().unlock();
}

private void writeLockRoot(){
  rootLock.writeLock().lock();
}

private void writeUnlockRoot(){
  rootLock.writeLock().unlock();
}

/**
 * Method that actually performs the deletion. It can perform
 * it at a given level in the tree, but public methods only allow deleting
 * at the leaf level.
 *
 * @param en - the entry to delete
 * @param level - the level to delete the entry from
 */
private void deleteAtLevel(Entry en, short level){
  if(BrilleDefinitions.DEBUG)logger.info("in delete at level");
  int currentPart = -1;
  int prev=-1;
  readLockRoot();
  currentPart=root;
  Buffer s = null;
  if(rootlevel==level){
    s = bp.pinBuffer(currentPart,fileNr,false,true);
    readUnlockRoot();
    prev=currentPart;
  }
  else{
    s = bp.pinBuffer(currentPart,fileNr,true,true);
    readUnlockRoot();
    prev=findCorrectAtCorrectLevel(en,level,currentPart,s);
    s = bp.pinBuffer(prev,fileNr,false,true);
  }
  deleteAtCurrentLevel(en,prev,s,level);
}

/**
```

```
 * Convenience method to call when we have found the correct level to delete
 * from.
 *
 * @param en - the entry to delete
 * @param partWithWriteLock - the part in the file we think we should delete
 *  from
 * @param buf - the buffer holding the part we think we should delete from
 * @param level - the level to delete from (which is the current level)
 */
private void deleteAtCurrentLevel(Entry en, int partWithWriteLock,
    Buffer buf, short level){
  Node n = new Node(buf,level==0?lef:ief,bp,partWithWriteLock,fileNr);
  int prev=partWithWriteLock;
  while(!n.shouldGoInThis(en)){
    if(BrilleDefinitions.DEBUG)
      logger.debug("realised that we should not place it in this one");
    int next = n.getNextPointer();
    if(BrilleDefinitions.DEBUG)logger.debug("got nextpointer which is "+1);
    bp.unpinBuffer(buf,false,true);
    if(BrilleDefinitions.DEBUG) logger.debug("have unlocked "+prev);
    prev=next;
    if(BrilleDefinitions.DEBUG){
      logger.debug(" locking "+prev+" when next was "+next);
      logger.debug("Trying to get writelock for next");
    }
    buf=bp.pinBuffer(prev,fileNr,false,true);
    n = new Node(buf,level==0?lef:ief,bp,prev,fileNr);
    if(BrilleDefinitions.DEBUG){
      logger.debug("That worked");
      logger.debug("The NODE FROM NEXTPOINTER IS:");
    }
  }
  n.deleteEntry(en);
  if(BrilleDefinitions.DEBUG){
    logger.info("after deleting "+en.toString());
    logger.info(n.toString());
  }
  bp.unpinBuffer(buf,false,true);
}


/**
 * This method is used to decrement values in an entry, or delete it if the
 * value of the entry becomes zero. This is used in the FullBTreeIndex when
 * storing the approximate number of documents containing a term, and the
 * approximate total number of occurrences of the term.
 *
 * @param e - the entry with the same key as the one to decrement, and
 *  containing the values to decrement the currently existing entry with.
 */
public void decOrDelete(Entry e){
  decOrDelete(e,(short)0);
}

/**
 * Convenience method to perform the actual decrementation at a specified
 * level. The public method above only allows this at level 0.
 *
 * @param en - entry with the same key as the one to decrement, and
 *  containing the values with which to decrement the existing entry
 * @param level - the level of the entry which should be decremented
 */
private void decOrDelete(Entry en, short level){
  if(BrilleDefinitions.DEBUG)
    logger.debug("dec or del of "+en+" at level "+level);
  int currentPart = -1;
  int prev=-1;
  readLockRoot();
  currentPart=root;
```

```
    Buffer s = null;
    if(rootlevel==level){
      s = bp.pinBuffer(currentPart,fileNr,false,true);
      readUnlockRoot();
      prev=currentPart;
    }
    else{
      if(BrilleDefinitions.DEBUG)logger.debug("need to search from root");
      s = bp.pinBuffer(currentPart,fileNr,true,true);
      if(BrilleDefinitions.DEBUG)logger.debug("Got lock on root");
      readUnlockRoot();
      if(BrilleDefinitions.DEBUG)logger.debug("done locking stuff");
      prev=findCorrectAtCorrectLevel(en,level,currentPart,s);
      if(BrilleDefinitions.DEBUG)
        logger.debug("realised that i should go to node "+prev);
      s = bp.pinBuffer(prev,fileNr,false,true);
      if(BrilleDefinitions.DEBUG)logger.debug("locking "+prev);
    }
    decOrDelAtCurrentLevel(en,s,prev,level);
  }

  /**
   * Convenience method to decrement or delete a node from the current level.
   * Used by the above method.
   *
   * @param en - the entry with the same key as the one to decrement,
   *  containing the values with which to decrement the current entry
   * @param partWithWriteLock - the part at the current level we have reached
   * @param buf - the buffer holding the current node
   * @param level - the current level
   */
  public void decOrDelAtCurrentLevel(Entry en, Buffer buf,
      int partWithWriteLock, short level){
    Node n = new Node(buf,level==0?lef:ief,bp,partWithWriteLock,fileNr);
    int prev=partWithWriteLock;
    while(!n.shouldGoInThis(en)){
      if(BrilleDefinitions.DEBUG)
        logger.debug("realised that we should not place it in this one");
      int next = n.getNextPointer();
      if(BrilleDefinitions.DEBUG)logger.debug("got nextpointer which is "+1);
      bp.unpinBuffer(buf,false,true);
      if(BrilleDefinitions.DEBUG)logger.debug("have unlocked "+prev);
      prev=next;
      if(BrilleDefinitions.DEBUG){
        logger.debug("locking "+prev+" when next was "+next);
        logger.debug("Trying to get writelock for next");
      }
      buf=bp.pinBuffer(prev,fileNr,false,true);
      n = new Node(buf,level==0?lef:ief,bp,prev,fileNr);
      if(BrilleDefinitions.DEBUG){
        logger.debug("That worked");
        logger.debug("The NODE FROM NEXTPOINTER IS:");
      }
    }
    try{
      BTreeHandle bth = n.searchForLargerOrEqual(en);
      Entry t = bth.getEntry();
      if(t.compareTo(en)!=0)
        throw new
          IllegalArgumentException("trying to decordel entry that " +
              "does not exist");
      else{
        n.deleteEntry(t);
        if(!t.decEntry(en)){
          if(n.hasRoomFor(t)) n.insert(t);
          else splitNode(n,t,prev);
        }
      }
    }finally{
```

```
      bp.unpinBuffer(buf,false,true);
    }
  }

  /**
   * Method used to increment an entry with a specified key (similar to the
   * key in the argument entry), with the values specified by values in the
   * argument entry. If no entry with such a key exists, this one is inserted.
   *
   * This method is used by the FullBTreeIndex.
   *
   * @param e - the entry with the key the entry in the B-tree to increment,
   *   and values with wich to increment
   * @return - the resulting entry after the increment
   */
  public Entry incOrInsert(Entry e){
    return incOrInsert(e,(short)0);
  }

  /**
   * Method actually performing the increment-or-insert-operation. It is
   * available here for this operation to be performed at any level, but the
   * operation is only available outside this class for level zero.
   *
   * @param e - the entry with the key the entry in the B-tree to increment,
   *   and values with wich to increment
   * @param level - the level at which the entry is
   * @return - the resulting entry after the increment
   */
  private Entry incOrInsert(Entry e,short level){
    if(BrilleDefinitions.DEBUG)
      logger.debug("inc or insert of "+e+" at level "+level);
    int currentPart = -1;
    int prev=-1;
    readLockRoot();
    currentPart=root;
    Buffer s = null;
    if(rootlevel==level){
      if(BrilleDefinitions.DEBUG)
        logger.debug("the incorins was at the root level");
      s = bp.pinBuffer(currentPart,fileNr,false,true);
      readUnlockRoot();
      prev=currentPart;
    }
    else{
      s = bp.pinBuffer(currentPart,fileNr,true,true);
      readUnlockRoot();
      if(BrilleDefinitions.DEBUG)
        logger.debug("the incorins was NOT at root level. " +
            "Finding correct level");
      prev=findCorrectAtCorrectLevel(e,level,currentPart,s);
      s = bp.pinBuffer(prev,fileNr,false,true);
    }
    if(BrilleDefinitions.DEBUG)
      logger.debug("have found a node at correct level");
    return incOrInsAtCurrentLevel(e,s,prev,level);
  }

  /**
   * When we have found a probably correct node at the correct level, this
   * method is called to do the actual increment (or insert).
   *
   * @param en  - the entry with the key the entry in the B-tree to increment,
   *   and values with wich to increment
   * @param partWithWriteLock - the part we think holds the node we should
   *   find the correct entry in.
   * @param buf - the buffer holding the current node
   * @param level - the level we are at
   * @return - the resulting entry after increment
```

```
 */
private Entry incOrInsAtCurrentLevel(Entry en,Buffer buf,
    int partWithWriteLock,short level){
  Entry ret=null;
  if(BrilleDefinitions.DEBUG)logger.debug("incorins at current level");
  Node n = new Node(buf,level==0?lef:ief,bp,partWithWriteLock,fileNr);
  if(BrilleDefinitions.DEBUG)logger.debug("is at node "+n.toString());
  int prev=partWithWriteLock;
  if(BrilleDefinitions.DEBUG)logger.debug("update level "+level);
  while(!n.shouldGoInThis(en)){
    if(BrilleDefinitions.DEBUG)logger.debug("nextpointer");
    int next = n.getNextPointer();
    if(BrilleDefinitions.DEBUG)logger.debug("got nextpointer which is "+1);
    bp.unpinBuffer(buf,false,true);
    prev=next;
    if(BrilleDefinitions.DEBUG)
      logger.debug("Trying to get writelock for next");
    buf=bp.pinBuffer(prev,fileNr,false,true);
    n = new Node(buf,level==0?lef:ief,bp,prev,fileNr);
    if(BrilleDefinitions.DEBUG)logger.debug("next node is: "+n.toString());
  }
  try{
    BTreeHandle bth = n.searchForLargerOrEqual(en);
    Entry t = bth.getEntry();
    if(BrilleDefinitions.DEBUG){
      if(t==null){
        logger.debug("the t searched for is null");
      }
      else logger.debug("the t is "+t.toString());
    }
    if(t==null || t.compareTo(en)!=0){
      if(BrilleDefinitions.SLEEP_TIME_FOR_STATISTICSCOLLECTOR>0)
        numInserts++;
      if(BrilleDefinitions.DEBUG)logger.info("the incorins is an ins");
      ret=en;
      if(n.hasRoomFor(en)){
        if(BrilleDefinitions.DEBUG)logger.info("has room");
        n.insert(en);
        if(BrilleDefinitions.DEBUG){
          logger.info("resulting node");
          logger.info(n.toString());
        }
      }
      else{
        if(BrilleDefinitions.DEBUG)logger.info("must split");
        splitNode(n,en,prev);
      }
    }
    else{
      if(BrilleDefinitions.SLEEP_TIME_FOR_STATISTICSCOLLECTOR>0)
        numUpdates++;
      if(BrilleDefinitions.DEBUG)logger.info("incorins is update");
      n.deleteEntry(t);
      t.incEntry(en);
      ret=t;
      if(n.hasRoomFor(t)){
        if(BrilleDefinitions.DEBUG)logger.info("has room");
        n.insert(t);
        if(BrilleDefinitions.DEBUG){
          logger.info("resulting node");
          logger.info(n.toString());
        }
      }
      else{
        if(BrilleDefinitions.DEBUG)logger.info("must split");
        splitNode(n,t,prev);
      }
    }
  }catch(Exception e){
```

```
      e.printStackTrace();
    }finally{
      bp.unpinBuffer(buf,false,true);
    }
    return ret;
  }


  /**
   * Method for updating an entry. The argument entry must have the same key
   * as the one we should update, but may of course have a new value.
   *
   * @param e - entry with the key we want to update the value for, and the
   *  new value we want it to have
   */
  public void updateEntry(Entry e){
    update(e,(short)0);
  }

  /**
   * The method performing the actual update. It is here available for any
   * level in the B-tree, but only publicly available for level zero, the
   * lowest level in the tree.
   *
   * @param e - the update entry
   * @param level - the level the entry to be updated is found at
   */
  private void update(Entry e, short level){
    if(BrilleDefinitions.DEBUG)
      logger.debug("updating entry "+e+" at level "+level);
    int currentPart = -1;
    int prev=-1;
    readLockRoot();
    currentPart=root;
    Buffer s = null;
    if(rootlevel==level){
      if(BrilleDefinitions.DEBUG)
        logger.debug("the update was at the root level");
      s = bp.pinBuffer(currentPart,fileNr,false,true);
      readUnlockRoot();
      prev=currentPart;
    }
    else{
      s = bp.pinBuffer(currentPart,fileNr,true,true);
      readUnlockRoot();
      if(BrilleDefinitions.DEBUG)
        logger.debug("the update was NOT at root level. " +
            "Finding correct level");
      prev=findCorrectAtCorrectLevel(e,level,currentPart,s);
      s = bp.pinBuffer(prev,fileNr,false,true);
    }
    if(BrilleDefinitions.DEBUG)
      logger.debug("have found a node at correct level");
    updateAtCurrentLevel(e,prev,s,level);
  }

  /**
   * When the a probably correct node at the level the entry should be updated
   * at is found, this method is called to perform the actual update.
   *
   * @param en - the update entry
   * @param partWithWriteLock - the part of the file where the node we believe
   *   the update entry is found
   * @param buf - the buffer holding the part with writelock
   * @param level - the current level
   */
  private void updateAtCurrentLevel(Entry en, int partWithWriteLock,
      Buffer buf, short level){
    if(BrilleDefinitions.DEBUG)logger.debug("updating at current level");
```

```
  Node n = new Node(buf,level==0?lef:ief,bp,partWithWriteLock,fileNr);
  if(BrilleDefinitions.DEBUG)logger.debug("is at node "+n.toString());
  int prev=partWithWriteLock;
  if(BrilleDefinitions.DEBUG)logger.debug("update level "+level);
  while(!n.shouldGoInThis(en)){
    if(BrilleDefinitions.DEBUG)logger.debug("nextpointer");
    int next = n.getNextPointer();
    if(BrilleDefinitions.DEBUG)logger.debug("got nextpointer which is "+1);
    bp.unpinBuffer(buf,false,true);
    prev=next;
    if(BrilleDefinitions.DEBUG)
      logger.debug("Trying to get writelock for next");
    buf=bp.pinBuffer(prev,fileNr,false,true);
    n = new Node(buf,level==0?lef:ief,bp,prev,fileNr);
    if(BrilleDefinitions.DEBUG)logger.debug("next node is: "+n.toString());
  }
  try{
    n.deleteEntry(en);
    if(BrilleDefinitions.DEBUG)logger.debug("has deleted old");
    if(n.hasRoomFor(en)){
      if(BrilleDefinitions.DEBUG)logger.debug("has room for new");
      n.insert(en);
    }
    else{
      splitNode(n,en,prev);
      if(BrilleDefinitions.DEBUG)logger.debug("splitted node in update");
    }
  }catch(Exception e){
    if(BrilleDefinitions.DEBUG)logger.debug("Exception");
    e.printStackTrace();
  }finally{
    bp.unpinBuffer(buf,false,true);
  }
}

/**
 * Method to insert an entry. The entry is inserted into the level 0 in the
 * B-tree. Note that levels are counted beginning with 0 as the lowest, and
 * increasing upwards.
 *
 * @param e - the entry to insert
 */
public void insertEntry(Entry e){
  if(BrilleDefinitions.SLEEP_TIME_FOR_STATISTICSCOLLECTOR>0)numInserts++;
  insertEntryIntoLevel(e,(short)0);
}

/**
 * This is a convenience method to insert an entry at a specified level in
 * the B-tree. Outsiders may only insert into level 0, the leaf level, but
 * this method is also used by the update thread after a node split.
 *
 * @param en - the entry to insert
 * @param level - the level in the btree where this entry should be inserted
 */
private void insertEntryIntoLevel(Entry en, int level){
  if(BrilleDefinitions.DEBUG)
    logger.debug("should insert entry "+en.toString()+" int level "+level);
  int currentPart = -1;
  int prev=-1;
  readLockRoot();
  currentPart=root;
  Buffer s = null;
  if(rootlevel==level){
    if(BrilleDefinitions.DEBUG)logger.debug("is at root level");
    s = bp.pinBuffer(currentPart, fileNr,false,true);
    readUnlockRoot();
    prev=currentPart;
  }
```

```
    else{
      if(BrilleDefinitions.DEBUG)logger.debug("need to search from root");
      s = bp.pinBuffer(currentPart,fileNr,true,true);
      readUnlockRoot();
      prev=findCorrectAtCorrectLevel(en,(short)level,currentPart,s);
      if(BrilleDefinitions.DEBUG)
        logger.debug("realised that i should go to node "+prev);
      s = bp.pinBuffer(prev,fileNr,false,true);
    }
    if(BrilleDefinitions.DEBUG)
      logger.debug("ready to insert at current level");
    insertAtCurrentLevel(en,prev,s,(short)level);
    //the buffer is unpinned in the above method.
  }

  /**
   * This method is called when inserting an entry, and when we have reached
   * the correct level to insert it into. It does the actual insertion in a
   * node and splits the node if it becomes overfull.
   *
   * @param en - the entry to insert
   * @param partWithWriteLock - the part currently in the Buffer buf, and where
   *  we think we should insert this entry
   * @param buf - the buffer holding the current node
   * @param level - the level we are at
   */
  private void insertAtCurrentLevel(Entry en, int partWithWriteLock,
      Buffer buf,short level){
    Node n = new Node(buf,level==0?lef:ief,bp,partWithWriteLock,fileNr);
    if(BrilleDefinitions.DEBUG){
      logger.debug("Inserting "+en.toString()+" at current level="+level);
      logger.debug("starting with node "+n.toString());
    }
    int prev=partWithWriteLock;
    while(!n.shouldGoInThis(en)){
      if(BrilleDefinitions.DEBUG)
        logger.debug("realised that we should not place it in this one");
      int next = n.getNextPointer();
      if(BrilleDefinitions.DEBUG)logger.debug("got nextpointer which is "+1);
      bp.unpinBuffer(buf,false,true);
      if(BrilleDefinitions.DEBUG)logger.debug("have unlocked "+prev);
      prev=next;
      if(BrilleDefinitions.DEBUG){
        logger.debug("locking "+prev+" when next was "+next);
        logger.debug("Trying to get writelock for next");
      }
      buf=bp.pinBuffer(prev,fileNr,false,true);
      n = new Node(buf,level==0?lef:ief,bp,prev,fileNr);
      if(BrilleDefinitions.DEBUG){
        logger.debug("jumping along nextpointer");
        logger.debug("That worked");
        logger.debug("The NODE FROM NEXTPOINTER IS:");
        logger.debug(n.toString());
      }
    }
    if(level==0 && n.isNotUnique(en))
      throw new
        IllegalArgumentException("Inserting an entry with nonunique key");
    if(n.hasRoomFor(en)){
      if(BrilleDefinitions.DEBUG)logger.debug("has room");
      n.insert(en);
      if(BrilleDefinitions.DEBUG){
        logger.debug("inserted entry "+en.toString()+" into node:");
        logger.debug(n.toString());
      }
    }
    else{
      if(BrilleDefinitions.DEBUG){
        logger.debug("SPLITTING NODE when inserting ");
```

```
        logger.debug(en.toString());
        logger.debug("Into node:");
        logger.debug(n.toString());
      }
      splitNode(n,en,prev);
      if(BrilleDefinitions.DEBUG)logger.debug("DONE SPLITTING NODE!");
    }
    bp.unpinBuffer(buf,false,true);
    if(BrilleDefinitions.DEBUG)logger.info("should unlock "+prev);
  }

  /**
   * This method is a convenience method used by all methods in this B-tree.
   * It starts from a given node and searches down the tree until it finds a
   * node at the correct level where, according to the pointers, the operation
   * should be performed. Its node number is then returned. Note that it might
   * not be the correct node: It might be so that the update thread has not
   * yet performed the updates needed to have a correct pointer to the node
   * where the operation should be performed.
   *
   * @param en - the entry we should find a tentative node for
   * @param level - the level at which we should find this node
   * @param readLockedRoot - the part we start from, which is the root node
   *  (that should be readlocked) in all methods in this implementation
   * @return the number of the node where we believe (according to the current
   *  pointers) that the operation should be performed
   */
  private int findCorrectAtCorrectLevel(Entry en, short level,
      int readLockedRoot, Buffer b){
    if(BrilleDefinitions.DEBUG)
      logger.debug("finding correct at correct level " +
          "for entry "+en.toString());
    int currentPart= readLockedRoot;
    Node current = new Node(b,ief,bp,currentPart,fileNr);

    while(true){
      if(BrilleDefinitions.DEBUG)logger.info("searching");
      int next = 0;
      if(current.getMaxKey().compareTo(en)<0){
        if(BrilleDefinitions.DEBUG)logger.debug("nextpointer");
        next = current.getNextPointer();
        bp.unpinBuffer(b,true,true);
      }
      else{
        if(BrilleDefinitions.DEBUG){
          logger.debug("go downwards from node:");
          logger.info(current.toString());
        }
        BTreeHandle bh = current.searchForLargerOrEqual(en);
        next = ((InternalEntry)bh.getEntry()).getPointer();
        if(BrilleDefinitions.DEBUG)logger.debug("going to pointer "+next);

        if(current.getLevel()==level+1){
          bh.release();
          return next;
        }
        bh.release();
      }
      if(BrilleDefinitions.DEBUG)logger.info("going to node "+next);
      currentPart=next;
      b=bp.pinBuffer(currentPart,fileNr,true,true);
      current=new Node(b,ief,bp,currentPart,fileNr);
      if(BrilleDefinitions.DEBUG)
        logger.info("the new node is "+current.toString());
    }
  }

  /**
   * When a node is splitted, an update needs to occur further up in the tree.
```

```
 * To let the update thread know this, this method is used to add a new
 * update.
 *
 * @see brille.btree.Update
 * @param up - the update to add
 */
private synchronized void addUpdate(Update up){
  updates.add(up);
}

/**
 * This method is only used by the update thread when a new update should be
 * processed. It removes the next entry in the queue, and returns null if
 * the queue of updates is empty.
 *
 * @return
 */
public synchronized Update removeNextUpdate(){
  if(updates.size()==0) return null;
  else return updates.remove(0);
}

/**
 * Method used to split a node when an insertion (or update) makes it
 * overfull.
 *
 * @param n - the node that should be splitted
 * @param en - the entry inserted (or updated) that makes it necesseary to
 *  split the node
 * @param currentPart - the part of the file where this node is represented
 */
private void splitNode(Node n, Entry en, int currentPart){
  if(BrilleDefinitions.DEBUG){
    logger.debug("splitting node (LOOKLOOKLOOK):");
    logger.debug(n.toString());
    logger.debug("when inserting ");
    logger.debug(en.toString());
  }
  int nr = getNextFree();
  Buffer buf = bp.pinBuffer(nr,fileNr,false,true);
  Node newNode =
    createNode(buf,n.getLevel(),nr,n.getMaxKey(),n.getNextPointer());
  if(BrilleDefinitions.DEBUG)logger.info("This nextpointer is set to "+nr);
  n.setNextPointer(nr);
  if(BrilleDefinitions.DEBUG){
    logger.info("The new node now looks like:");
    logger.info(newNode.toString());
    logger.info("");
  }
  n.findWhereToSplitAndSplit(en,newNode);
  if(BrilleDefinitions.DEBUG){
    logger.debug("Gives nodes:");
    logger.debug(n.toString());
    logger.debug("and");
    logger.debug(newNode.toString());
  }
  Entry newMax = newNode.getMaxKey();
  Entry nMax = n.getMaxKey();
  InternalEntry old1 =
    ief.make(ief.makeEntryWithSameKey(newMax),newNode.getPart());
  InternalEntry new1 = ief.make(ief.makeEntryWithSameKey(nMax),n.getPart());
  addUpdate(new Update(old1,(short)(n.getLevel()+1),new1));
  bp.unpinBuffer(buf,false,true);
}

/**
 * Method used to search for a key in the B-tree.
 * It returns a handle to the entry found in the B-tree with the smallest
 * key larger or equal to the argument one. The argument is an entry, but
```

```
 * only the value of the key in this entry matters.
 *
 * @param key - an entry containing the key to search for
 * @return a BTreeHandle starting at the entry with the smallest key equal or
 *  larger than the key searched for
 */
public BTreeHandle searchForEntry(Entry key){
  return searchForEntryAtLevel(key,0);
}

/**
 * The method actually performing the search, but is also able to do
 * it for a specific level. This method is only public for testing
 * purposes.
 *
 * @param key - an entry containing the key to search for
 * @param level - the level at which the search is performed
 * @return a BTreeHandle starting at the entry with the smallest key equal or
 *  larger than the key searched for
 */
private BTreeHandle searchForEntryAtLevel(Entry key, int level){
  int currentPart = -1;
  int prev=-1;
  readLockRoot();
  currentPart=root;
  Buffer s = null;
  if(rootlevel==level){
    s = bp.pinBuffer(currentPart,fileNr,true,true);
    readUnlockRoot();
    prev=currentPart;
  }
  else{
    s = bp.pinBuffer(currentPart,fileNr,true,true);
    readUnlockRoot();
    prev=findCorrectAtCorrectLevel(key,(short)level,currentPart,s);
    s = bp.pinBuffer(prev,fileNr,true,true);
  }
  return searchAtCorrectLevel(key,prev,s);
}

/**
 * Convenience method used to penform the actual search once the a
 * tentative node at the correct level to search at is found.
 *
 * @param key - the key to search for
 * @param readLockedPart - the part of the file where the node we think
 *  contains the entry searched for is found
 * @param buf - a buffer containing the node we think contains the key
 *  searched for
 * @return a handle starting at the entry with the smallest key larger or
 *  equal to the key searched for
 */
private BTreeHandle searchAtCorrectLevel(Entry key, int readLockedPart,
    Buffer buf){
  int curr = readLockedPart;
  Node n = new Node(buf,lef,bp,curr,fileNr);
  while(n.getMaxKey().compareTo(key)<0){
    curr = n.getNextPointer();
    bp.unpinBuffer(buf,true,true);
    buf= bp.pinBuffer(curr,fileNr,true,true);
    n = new Node(buf,lef,bp,curr,fileNr);
  }
  return n.searchForLargerOrEqual(key);
}

/**
 * Method used to handle an update by the UpdateThread.
 * It does various things depending on the level of the update,
 * it might represent a need for a new root, in which case a new one is
```

```
 * made, or it might cause an insert or an update at a given level.
 *
 * @param u - the update to handle
 */
public void handleUpdate(Update u){
  if(BrilleDefinitions.DEBUG)logger.debug("Handling update");
  readLockRoot();
  if(rootlevel<u.getLevel()){
    readUnlockRoot();
    writeLockRoot();
    if(rootlevel<u.getLevel()){
      if(BrilleDefinitions.DEBUG)
        logger.debug("need to update root, and have writelock");
      int nr = getNextFree();
      Buffer r = bp.pinBuffer(nr,fileNr,false,true);
      if(BrilleDefinitions.DEBUG)
        logger.debug("got writelock on the sweet new nr "+nr);
      root = nr;
      rootlevel++;
      writeUnlockRoot();
      Entry in = ief.make();
      in.setKeyToMax();
      Node newNode = createNode(r,u.getLevel(),nr,in);
      if(BrilleDefinitions.DEBUG)
        logger.debug("the update has created the new node");
      newNode.insert(u.getNew());
      newNode.insert(u.getOld());
      if(BrilleDefinitions.DEBUG){
        logger.debug("The root update created was:");
        logger.debug(newNode.toString());
      }
      bp.unpinBuffer(r,false,true);
      if(BrilleDefinitions.DEBUG)logger.debug("new root created by update");
      return;
    }
    writeUnlockRoot();
    throw new Error("Am I using several updatethreads??");
  }
  if(BrilleDefinitions.DEBUG)logger.debug("NO NEW ROOT");
  readUnlockRoot();
  if(BrilleDefinitions.DEBUG)logger.debug("Should insert new ");
  insertEntryIntoLevel(u.getNew(),u.getLevel());
  if(BrilleDefinitions.DEBUG)logger.debug("should update");
  update(u.getOld(),u.getLevel());
  if(BrilleDefinitions.DEBUG)logger.debug("Done handling update");
}

/**
 * This is a hack for retrieving a handle to the smallest entry in this
 * B-tree. Because of the way this tree grows, this will always be the
 * smallest entry in node 0, and this is a constant time method for obtaining
 * this node. It is used in the FullBTreeIndex when the rankings are
 * updated by the FixRankingsThread
 *
 * @return a handle starting at the smallest entry in this tree, which can be
 *  used to iterate over all entries.
 */
public BTreeHandle getSmallestEntry(){
  Buffer buf = bp.pinBuffer(0,fileNr,true,true);
  Node n = new Node(buf,lef,bp,0,fileNr);
  BTreeHandle bh = new BTreeHandle(n,n.getNr()-1);
  return bh;
}

/**
 * This is a method used to delete all entries in this tree. It is
 * not implemented here, but the same thing can be achieved by deleting
 * the file this B-tree uses for storing its nodes, release the pointer
 * to this tree (and let the garbage collector delete everything in memory),
```

```
 * and create a new empty tree.
 *
 * @throws NotImplementedException
 */
public void deleteAllEntries() throws NotImplementedException{
  throw new
  NotImplementedException("have not implemented deletion of all entries");
}

/**
 * This method is called when the application shuts down, and it stops the
 * update thread.
 *
 */
public void shutDown(){
  ut.shutDown();
}

/**
 * This method is only used for printing a representation of this
 * tree that can be used to make a drawing of the tree.
 *
 * WARNING: Do not use when tree is modified, as the representation
 * may not be correct if the update thread has not performed all its
 * updates yet.
 *
 */
public void pt(){
  IntArrayList q = new IntArrayList();
  IntArrayList level = new IntArrayList();
  readLockRoot();
  q.add(root);
  level.add(rootlevel);

  readUnlockRoot();
  while(q.size()!=0){
    int n = q.remove(0);
    int l = level.remove(0);
    Buffer buf = bp.pinBuffer(n,fileNr,true,true);
    Node no = new Node(buf,l==0?lef:ief,bp,n,fileNr);
    td.println(no.toString());
    td.println();
    if(l>0){
      Entry en = ief.make();
      for(int i=no.getNr()-1;i>=0;i--){
        no.readEntryFromArrayPos(en,i);
        q.add(((InternalEntry)en).pointer);
        level.add(l-1);
      }
    }
    bp.unpinBuffer(buf,true,true);
  }
  td.close();
}

/**
 * Method for calculating the size of this B-tree on disk.
 *
 * @return the size of the file containing this B-tree
 */
public long diskSize(){
  return f.length();
}

/**
 * Convenience method to retrieve the height of the B-tree.
 * NOTE: This method is not thread safe, and is only used for testing
 * purposes.
 *
```

```
   * @return the current height of the B-tree
   */
  public int getRootLevel(){
    return rootlevel+1;
  }
}
```

C.2.2 brille.btree.BTreeHandle

```
package brille.btree;

/**
 * This class is returned as a result when searching in a B-tree. It is an
 * iterator which starts at an entry. The entry is usually the one searched
 * for, if such an entry exists.
 *
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class BTreeHandle {
  private Node node;
  private int pos;

  /**
   * Constructor taking the node where the current entry is found, and the
   * position of the entry within the node as arguments.
   *
   * @param node - the node where the entry is found
   * @param pos - the position of the entry within the node
   */
  public BTreeHandle(Node node, int pos){
    this.node=node;
    this.pos=pos;
  }


  /**
   * Method for moving the iterator one step forward in the B-tree.
   * Calling this method will result in a new Entry to be returned on calls
   * to getEntry()
   *
   */
  public void getNext(){
    pos--;
    while(pos<0){
      Node newNode = node.getNodeFromNextPointer();
      node.release();
      node=newNode;
      if(node==null) return;
      pos=node.getNr()-1;
    }
  }

  /**
   * When this iterator should not be used anymore,this method should be called
   * to ensure that all locks and buffers this iterator has will be unlocked.
   *
   */
  public void release(){
    if(node!=null)node.release();
  }

  /**
   * Method for retrieving the entry currently held by this iterator. If there
   * are no more entries, null is returned.
   *
   * @return the current entry held by this iterator
```

```
   */
  public Entry getEntry(){
    if(node==null || pos==-1) return null;
    return node.getEntryFromArrayPos(pos);
  }

  /**
   * Method used for debugging to retrieve the node the iterator is currently
   * holding
   *
   * @return the current node this iterator holds
   */
  public String getNodeString(){
    if(node==null) return "no node!";
    return node.toString();
  }
}
```

C.2.3 brille.btree.Entry

```
package brille.btree;

import brille.buffering.Buffer;

/**
 * This is the abstract class all entries in a B-tree must extend.
 * It contains methods to write and read entries and keys, compare
 * different entries, and to do several things with keys.
 *
 * All kinds of entries must have a specified way to denote a key that is
 * larger than all other possible keys. This is used as the max-key in the
 * nodes placed to the farthest right in the B-tree. When using integers
 * as keys, one can choose to use Integer.MAX_VALUE as the maximum key for
 * instance, but this will give us a smaller set of keys. In entries with
 * terms as part of the key, terms with length -1 is used as max-keys. This
 * will not have a negative effect on the number of valid keys.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public abstract class Entry implements Comparable<Entry>{

  /**
   * Method used to decide whether the key in this entry is the maximum key
   * for this kind of entry.
   *
   * @return whether or not this entry has a maximum key
   */
  public abstract boolean keyIsMax();

  /**
   * Method used to set the key of this entry to the maximum key for this kind
   * of entries.
   *
   */
  public abstract void setKeyToMax();

  /**
   * Method for writing this entry to the specified buffer in the given
   * position, and returning the number of bytes used, that is the next
   * position where a write can occur without damaging this one is
   * pos + the return value.
   *
   * This method assumes that there is room for this entry in the given
   * buffer. This could be checked beforehand by calling the writeLength()-
   * method, which will tell you how many bytes this entry will consume.
   *
   * @param buf - the buffer to write this entry to
```

```
 * @param pos - the position in the buffer to write this entry to
 * @return the number of bytes used to write this entry
 */
public abstract int write(Buffer buf,int pos);

/**
 * Method for writing the key of this entry to the given buffer at the
 * given position. This is typically used when writing max-keys in the
 * different nodes. This one also assumes that there is room for this key
 * in the buffer, and returns the number of bytes used for representing the
 * key.
 *
 * @param buf - the buffer to write to
 * @param pos - the position in the buffer to write to
 * @return the number of bytes used to write the key
 */
public abstract int writeKey(Buffer buf, int pos);

/**
 * This method will change all object variables in this object. According
 * to standard object orientation, this should create a new object, but for
 * efficiency reasons, that is not the case here. It is expensive to create
 * a lot of new objects, and it is more efficient to change the contents on
 * a few of them.
 *
 * The method reads in all variables of this entry from the given position
 * in the given buffer, and returns the number of bytes read.
 *
 * @param buf - the buffer to read from
 * @param pos - the position within the buffer to read from
 * @return - the number of bytes read
 */
public abstract int readIn(Buffer buf, int pos);

/**
 * Method used to read in a new key into this entry from the specified
 * position within the specified buffer.
 *
 * @param buf - the buffer to read from
 * @param pos - the position within the given buffer to read from
 * @return - the number of bytes read
 */
public abstract int readInKey(Buffer buf, int pos);

/**
 * Method for retrieving the number of bytes used to represent the key of
 * this entry in a buffer. This is useful when testing whether there is room
 * for this entry's key in a particular node.
 *
 * @return the number of bytes used to represent the key of this entry in a
 *  buffer
 */
public abstract int keyLength();

/**
 * Method for retrieving the number of bytes used to represent this entry
 * in a buffer. This is useful for testing whether there is room for this
 * entry in a buffer.
 *
 * @return the number of bytes used to represent this entry in a buffer
 */
public abstract int length();

/**
 * Method used to compare this entry to another one. It follows the same
 * principles as normal compareTo-methods, but only compares the key of the
 * two entries.
 *
 * @param e - the entry containing the key to compare this ones to
```

```
     * @return an integer value describing which key is smallest, 0 if they
     * are equal
     */
    public abstract int compareTo(Entry e);


    public abstract String toString();

    /**
     * Method used to decide whether the key in this entry may be compared to
     * the key in the argument entry. This is usually decided upon by
     * checking which kind of object the calling entry is.
     *
     * @param en - the entry to test for key compatibility
     * @return whether or not this entry is key compatible with the argument
     *  entry
     */
    public abstract boolean isKeyCompatible(Entry en);

    /**
     * Method to return the key in this entry as a String.
     *
     * @return a String representation of the key of this entry
     */
    public abstract String getKeyAsString();

    /**
     * Method added to support the incorinsert-method in the B-tree.
     * If that method is called with an entry with a key similar to this
     * entry's key, this entry's values should be decremented with the values
     * in the argument entry's values.
     *
     * @param en - the entry containing the values with which to increment this
     *  entry's values
     */
    public abstract void incEntry(Entry en);

    /**
     * Method added to support the decordel-operation in the B-tree.
     * If we call decordel with an entry with a key similar to this one,
     * this method is used to decrement values in this entry in the way
     * specified by this method. In most implementation, this method just
     * throws an IllegalArgumentException, but it is used for
     * FullBTreeIndexLeafEntry.
     *
     * @param en - the entry with the values with which to decrement this
     *  entries values
     * @return whether or not the values could be considered to be zero now,
     *  in which case this entry should be deleted.
     */
    public abstract boolean decEntry(Entry en);
}
```

C.2.4 brille.btree.EntryFactory

```
package brille.btree;

/**
 * Because of the way generics are implemented in Java, the only way to
 * instantiate an object defined run-time is by making a factory.
 * This class is an abstract factory all entries must have an implementation
 * of to be valid entries in a B-tree.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public abstract class EntryFactory {
```

```
  /**
   * The only method defined in this class, which should return a new instance
   * of an entry, with unspecified member variables.
   *
   * @return a new entry of the kind this EntryFactory produces
   */
  public abstract Entry make();

}
```

C.2.5 brille.btree.InternalEntry

```
package brille.btree;

/**
 * InternalEntries differs slightly from leaf entries because leaf entries
 * have values while internal entries consists of keys and a pointer. This
 * class defines this behavior, and all leaf entries should have a "sibling"
 * object inheriting from this class.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public abstract class InternalEntry extends Entry {
  protected int pointer;

  /**
   * The only constructor in this class, which defines that all internal
   * entries must have a pointer.
   *
   * @param pointer the pointer of this internal entry
   */
  public InternalEntry(int pointer){
    this.pointer=pointer;
  }

  /**
   * Getter for the pointer.
   *
   * @return the pointer of this internal entry
   */
  public int getPointer(){
    return pointer;
  }
}
```

C.2.6 brille.btree.InternalEntryFactory

```
package brille.btree;

/**
 * This class defines methods that should be available in factories for
 * internal entries.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public abstract class InternalEntryFactory extends EntryFactory {

  /**
   * Method to make an entry with the specified key, and the given
   * pointer.
   *
   * @param maxKey - the key of the entry to make
   * @param pointer - the pointer of the new entry
   * @return the new entry
   */
```

```
  public abstract InternalEntry make(Entry maxKey,int pointer);

  /**
   * Method used to create a new entry with a key equal to the argument entry.
   *
   * @param newMax - the entry with the same key as the new one should have
   * @return the new entry
   */
  public abstract InternalEntry makeEntryWithSameKey(Entry newMax);

}
```

C.2.7 brille.btree.Node

```
package brille.btree;

import brille.BrilleDefinitions;
import brille.buffering.Buffer;
import brille.buffering.BuffPool;
import brille.utils.IntArrayList;

import java.util.ArrayList;

import org.apache.log4j.Logger;

/**
 * This class represents the node in a B-tree. Essentially all variables in
 * the node are stored in the buffer member variable, and are read from there.
 * This node supports entries with variable length keys and values, and still
 * supports binary searching the entries (which, when you think about it, is
 * not completely straight-forward).
 *
 * The buffer has the following structure, even though it may have variable
 * length:
 *
 *  _________________________________________________________________________
 * | level |    num    |  next    | bytes | max | entries     | ... | pointers to |
 * |       | entries | pointer | used   | key |             |     |   entries   |
 * |_______|_________|_________|_______|_____|____________|_____|_____________|
 *
 * The level is the level this node is found at in the B-tree, and is
 * represented in a short, and thus using 2 bytes in the buffer. Num entries
 * is obviously the number of entries in this node. This is represented in an
 * int to support very large nodes, but it should probably have been a short as
 * well. The next pointer is an int pointing to the next node at the same level
 * as this one. The bytes used is also an int. It represents the size,
 * in bytes, of the maxkey and all the entries listed, not counting the
 * pointers to the entries. Hence a node is full when:
 * buffersize - 14 - bytes used - (num entries + 1)*4 = 0
 *
 * The pointers to the entries points into the entries to denote where the
 * different entries start and stop. And entries is obviously a listing of
 * the entries in this node represented quite compact (but without compression
 * in the current implementation).
 *
 * As entries are inserted, the listings of entries and pointers into this list
 * will both grow, and we will not allow more insertions when there is not
 * enough room between these two lists.
 *
 * This class provides methods for inserting and deleting entries within a
 * node, and it also provides methods for manipulating the various "invisible"
 * member variables.
 *
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class Node {
```

```
protected final Logger logger = Logger.getLogger(getClass());

private Buffer buf;
private EntryFactory ef;
private BuffPool bp;
private int part;
private int fileNr;

/**
 * The constructor for nodes. It takes the buffer representing this node as
 * an argument, together with an entry factory for creating new entries of
 * the kind represented in this node. This factory will be for leaf entries
 * if the level==0, or internal entries otherwise.
 *
 * The tree last arguments represents within which bufferpool, and which file
 * and part number this node's buffer is gotten from. This enables getting
 * the node from the nextpointer of this node as well, because all nodes in
 * the same B-tree will be found in the same buffer pool and within the same
 * file.
 *
 * @param buf - the buffer within which this node is represented
 * @param ef - a factory for creating entries of the kind found in this node
 * @param bp - the buffer pool where this entry's buffer is pinned from
 * @param part - the part of the file where the buffer representing this
 *   node is found
 * @param fileNr - the number in the bufferpool representing the file where
 *   this node is found
 */
public Node(Buffer buf, EntryFactory ef, BuffPool bp, int part,int fileNr){
  this.buf=buf;
  this.ef=ef;
  this.bp=bp;
  this.part=part;
  this.fileNr=fileNr;
}

/**
 * When we have created a node, there are some fields that should be
 * initialized, and this is one of the two methods to do this. It sets
 * the level and maxkey to the arguments, and sets the nextpointer to
 * null (by setting its integer-pointer to Integer.MAX_VALUE even though
 * it should possibly have been represented by a negative number).
 *
 * @param level - the value the level of this node should be initiated to
 * @param maxKey - the maximum key of this new node
 */
public void init(short level, Entry maxKey){
  init(level,maxKey,Integer.MAX_VALUE);
}

/**
 * This method also initializes a new node, but differs from the one above
 * in that the nextpointer can also be specified.
 *
 * @param level - the level of this new node in the B-tree
 * @param maxKey - the maximum key in this new node
 * @param nextPointer - the pointer to the next node, that is the part of
 *   the file where the next node on the same level is found
 */
public void init(short level, Entry maxKey, int nextPointer){
  setLevel(level);
  setBytesUsed(0);
  setNextPointer(nextPointer);
  setNr(0);
  setMaxKey(maxKey);
  if(BrilleDefinitions.DEBUG){
    logger.info("new node initialized:");
    logger.info(toString());
```

```
    }
  }

  private void setMaxKey(Entry en){
    int len=en.keyLength();
    en.writeKey(buf,14);
    if(BrilleDefinitions.DEBUG)
      logger.info("setting maxkey, and incing bytesused with "+len);
    incBytesUsed(len);
  }

  /**
   * Method for retrieving the maximum key stored in this node.
   *
   * @return the maximum key
   */
  public Entry getMaxKey(){
    Entry ret = ef.make();
    ret.readInKey(buf,14);
    return ret;
  }

  /**
   * Method for setting which part the buffer representing this node currently
   * holds. It is used to make sure there is consistency between these values
   * in the B-tree and in the buffer pool, for debugging purposes.
   *
   * @param p - the number we want to set the part to
   */
  public void setPart(int p){
    buf.setPart(p);
  }

  /**
   * Method for retriving the part represented in this node.
   * This method is also used for debugging purposes to ensure
   * consistency between the buffer pool and this node.
   *
   * @return the part of the file where this node is represented
   */
  public int getPart(){
    return part;
  }

  /**
   * Method for setting the nextpointer for this node.
   * This method is typically used when splitting this node when it becomes
   * too full.
   *
   * @param p - the value we want to set the nextpointer to
   */
  public void setNextPointer(int p){
    buf.putInt(p,6);
  }

  /**
   * Method for retrieving the nextpointer in this node. This method is
   * typically used when the update-thread has not yet updated parents of
   * nodes where an operation needs to be carried out.
   *
   * @return the nextpointer
   */
  public int getNextPointer(){
    return buf.getInt(6);
  }

  /**
   * Method for retrieving the level of this node in the B-tree.
   *
```

```
 * @return the level of this node
 */
public short getLevel(){
  return buf.getShort(0);
}

/**
 * Method for setting the level of this node. This method should really
 * only be called when the node is initialized, because the level of a node
 * never changes in the current B-tree implementation.
 *
 * @param level - the value to set the node's level to
 */
public void setLevel(short level){
  buf.putShort(level,0);
}

/**
 * Method for retrieving the number of entries found in this node.
 *
 * @return the number of entries in this node
 */
public int getNr(){
  return buf.getInt(2);
}

/**
 * Method for setting the number of entries in this node.
 *
 * @param nr - the number of entries to set for this node
 */
private void setNr(int nr){
  buf.putInt(nr,2);
}

/**
 * Method for incrementing the number of entries found in this node by one.
 *
 */
private void incNr(){
  setNr(getNr()+1);
}

/**
 * Method for decrementing the number of entries found in this node by one.
 *
 */
private void decNr(){
  setNr(getNr()-1);
}


/*
 * The four following methods modyfies and retrieves the number of bytes
 * used. They are only private as this should not be done outside this
 * class.
 *
 */
private int getBytesUsed(){
  return buf.getInt(10);
}


private void setBytesUsed(int nr){
  buf.putInt(nr,10);
}


private void incBytesUsed(int inc){
```

```
  setBytesUsed(getBytesUsed()+inc);
}


private void decBytesUsed(int decBy){
  setBytesUsed(getBytesUsed()-decBy);
}

/**
 * Method for retrieving the buffer where this node is represented.
 * This method is currently only used for debugging purposes.
 *
 * @return the buffer where this node is represented
 */
public Buffer getBuffer(){
  return buf;
}


/**
 * Method for testing whether the argument entry has a key that should go
 * in this node. The evaluation is performed by checking whether the entry
 * has a key smaller or equal to the maximum key in this node. The maximum
 * key is never changed for a node, so if the argument entry key is larger,
 * we know that the entry can not be inserted here.
 *
 * @param en - the entry we want to check whether should go in this node or
 *  not
 * @return whether or not this entry should go in this node.
 */
public boolean shouldGoInThis(Entry en){
  Entry t = getMaxKey();
  return en.compareTo(t)<=0;
}

/**
 * Method for retrieving the number of free bytes in this node.
 *
 * @return the number of free bytes in this node
 */
public int getFreeBytes(){
  return BrilleDefinitions.BUFFER_SIZE-14-getBytesUsed()-4*getNr();
}

/**
 * Method for determining whether a node has room for the argument entry.
 * This is done by checking whether there are more bytes left than this
 * entry will need.
 *
 * @param en - the entry we want to know whether there is room for
 * @return whether or not there is room for the given entry
 */
public boolean hasRoomFor(Entry en){
  return getFreeBytes()>=(en.length()+4);
}

public int testpositions(){
  int nextfree = nextFreePos();
  for(int i=0;i<getNr();i++){
    int val = buf.getInt(convArrayPosToBufPos(i));
    if(val>=nextfree) return i;
  }
  return -1;
}

/**
 * This method is important for splitting nodes. It takes an entry as
 * argument, in addition to the new node. The entry is the one we tried to
 * insert when we found out that this node would be overfull. Hence, we have
```

```
 * determined to split this node, and the node we want to transfer half of
 * the bytes to is the other argument.
 *
 * This method tries to find the most fair way to split. This is not
 * necessearily to split so that each of them has the same number of entries
 * as entries may be of different size. We thus try to split so that the
 * number of bytes used in each of the nodes is as evenly distributed as
 * possible.
 *
 * This method takes care of initializing the other node so that everything
 * will go smoothly, and serializable.
 *
 * @param en - the entry causing the split
 * @param n - the other node which we should transfer some of the entries to
 */
public void findWhereToSplitAndSplit(Entry en,Node n){
  if(BrilleDefinitions.DEBUG)logger.info("in the splitmethod");
  int ideal=(en.length()+getBytesUsed()+
      ((getNr()+1)*BrilleDefinitions.INT_SIZE - getMaxKey().keyLength()))/2;
  if(BrilleDefinitions.DEBUG){
    logger.debug("ideal is "+ideal);
    logger.info("the entry uses "+en.length()+" and nr is  "+getNr()+
        " and have used "+(getBytesUsed()-getMaxKey().keyLength()));
    logger.info("This makes the ideal "+ideal);
    logger.info("The entry I should insert is :");
    logger.info(en.toString());
  }
  int prev=0;
  int next=0;
  int posForEn=-1;
  int includedUpTo=0;
  Entry test = ef.make();
  Entry tmp = ef.make();
  for(int i=getNr()-1;i>=0;i--){
    Entry prevEntry=test;
    test=tmp;
    tmp=prevEntry;
    readEntryFromArrayPos(test,i);
    if(posForEn==-1 && en.compareTo(test)<0){
      next = prev+en.length()+BrilleDefinitions.INT_SIZE;
      i++;
      if(next>=ideal){
        includedUpTo=i;
        if(next-ideal<ideal-prev){
          if(BrilleDefinitions.DEBUG)
            logger.debug("The closest i came to ideal is "+next);
          posForEn=i-1;
        }
        else if(BrilleDefinitions.DEBUG)
          logger.debug("The closest i came to ideal is "+prev);
        break;
      }
      posForEn=i-1;
    }
    else{
      next=prev+test.length()+BrilleDefinitions.INT_SIZE;
      if(next>=ideal){
        if(next-ideal<ideal-prev){
          if(BrilleDefinitions.DEBUG)
            logger.debug("The closest i came to ideal is "+next);
          includedUpTo=i;
        }
        else{
          if(BrilleDefinitions.DEBUG)
            logger.debug("The closest i came to ideal is "+prev);
          includedUpTo=i+1;
        }
        break;
      }
```

```
    }
    prev=next;
  }
  if(posForEn==-1){
    if(BrilleDefinitions.DEBUG){
      logger.info("Inserting the new entry in the new node");
      if(!n.hasRoomFor(en)){
        logger.error("MAJOR ERROR! It is not room for entry. The entry is");
        logger.error(en.toString());
      }
    }
    n.insert(en);
  }
  if(BrilleDefinitions.DEBUG)
    logger.info("And inserting the following entries in the new node:");
  Entry move = ef.make();
  if(BrilleDefinitions.DEBUG){
    logger.info("I have made the new node:");
    logger.info(n.toString());
  }
  for(int i=includedUpTo-1;i>=0;i--){
    readEntryFromArrayPos(move,i);
    if(BrilleDefinitions.DEBUG)logger.info("moving: "+move.toString());
    n.insert(move);
  }
  if(BrilleDefinitions.DEBUG){
    logger.info("new node is:");
    logger.info(n.toString());
  }
  fixInternalEntries(includedUpTo,posForEn,en);
  if(BrilleDefinitions.DEBUG){
    logger.info("old node is:");
    logger.info(toString());
  }
  if(BrilleDefinitions.DEBUG && testpositions()!=-1){
    int val = testpositions();
    logger.error("Error in testpositions nr "+val);
    logger.error("in node:");
    logger.error(toString());
    Entry err = ef.make();
    readEntryFromArrayPos(err,val);
    logger.error("the current entry there is:");
    logger.error(err.toString());
    throw new Error("fucked positions after split");
  }

  if(BrilleDefinitions.DEBUG && n.testpositions()!=-1){
    int val = n.testpositions();
    logger.error("Error in testpositions nr "+val);
    logger.error("in node:");
    logger.error(n.toString());
    Entry err = ef.make();
    n.readEntryFromArrayPos(err,val);
    logger.error("the current entry there is:");
    logger.error(err.toString());
    throw new Error("fucked positions in new node after split");
  }

  if(BrilleDefinitions.DEBUG && !isSorted()){
    logger.error("Not sorted after split");
    logger.error(toString());
    throw new Error("doh after split!");
  }
  else if(BrilleDefinitions.DEBUG && !n.isSorted()){
    logger.error("the new node is not sorted after split!");
    logger.error(n.toString());
    throw new Error("doh for new node after split!");
  }
}
```

```
private void fixInternalEntries(int uptoAndIncluding,
    int posForEn, Entry en){
  ArrayList<Entry> l = new ArrayList<Entry>();
  boolean added=false;
  for(int i=getNr()-1;i>=uptoAndIncluding;i--){
    if(i==posForEn){
      added=true;
      l.add(en);
    }
    Entry add = ef.make();
    readEntryFromArrayPos(add,i);
    l.add(add);
  }
  if(!added && posForEn!=-1)l.add(en);
  setNr(l.size());
  setBytesUsed(0);
  setMaxKey(l.get(l.size()-1));
  for(int i=0;i<l.size();i++){
    int pos = nextFreePos();
    buf.putInt(pos,convArrayPosToBufPos(l.size()-1-i));
    incBytesUsed(l.get(i).write(buf,pos));
  }
}


public String toString(){
  String ret = "Node:"+part+" Level:"+getLevel()+" #entries "+
    getNr()+" NextPointer: "+getNextPointer()+" maxkey: "+
    getMaxKey().getKeyAsString()+" Bytes used: "+getBytesUsed()+"\n";
  if(getNr()>0){
    ret+="Entries: \n";
    Entry en = ef.make();
    if(BrilleDefinitions.DEBUG)logger.debug("number is "+getNr());
    for(int i=getNr()-1;i>=0;i--){
      if(BrilleDefinitions.DEBUG)
        logger.debug("i is "+i+" and the pointer there points to "+
            buf.getInt(convArrayPosToBufPos(i)));
      readEntryFromArrayPos(en,i);
      ret+= (en.toString()+"\n");
    }
  }
  else ret+="No Entries\n";
  return ret;
}

/**
 * This method is only used when a drawing of the tree is wanted. It prints
 * stuff for dot, to enable a nice drawing of this node.
 *
 * @return a String for dot.
 */
public String fixDot(){
  int lev = getLevel();
  String ret="struct"+part+" [label=\"";
  Entry e=ef.make();
  int cnt=0;
  for(int i=getNr()-1;i>=0;i--){
    readEntryFromArrayPos(e,i);
    cnt++;
    if(lev==0) ret+="<f"+cnt+"> ("+e.toString()+")|";
    else ret+="<f"+cnt+"> "+e.getKeyAsString()+"|";
  }
  ret+="<f"+cnt+"> next\"]";
  return ret;
}

/**
 * This method is similar to the one below, except that this one will also
```

```
 * create the entry returned for you, and thus causing more object ceration.
 *
 * @param arrayPos - the arraypos to read from
 * @return the entry pointed to by the given array position
 */
public Entry getEntryFromArrayPos(int arrayPos){
  Entry ret = ef.make();
  readEntryFromArrayPos(ret,arrayPos);
  return ret;
}

/**
 * This method is capable of reading an entry from a specified position
 * in the list of pointers. It will follow the pointer and read the
 * entry from there. The entry will be read into the argument entry,
 * so that unecesseary object creation is avoided.
 *
 * @param en - the entry where we want the result
 * @param arrayPos - the number in the list of pointers we want the entry
 *   from
 */
public void readEntryFromArrayPos(Entry en, int arrayPos){
  if(arrayPos>=getNr() || arrayPos<0)
    throw new
      IllegalArgumentException("reading an arraypos that is too high: "+
          arrayPos);
  en.readIn(buf,buf.getInt(convArrayPosToBufPos(arrayPos)));
}

private int convArrayPosToBufPos(int arrayPos){
  return (BrilleDefinitions.BUFFER_SIZE-4*(arrayPos+1));
}

private int findSmallestLargerThanOrEqual(Entry en){
  if(BrilleDefinitions.DEBUG){
    logger.debug("in findSmallestLarger...");
    logger.debug("In node:");
    logger.debug(toString());
    logger.debug("Should find:");
    logger.debug(en.toString());
  }
  int r=-1;
  int l=getNr()-1;
  int m=0;
  Entry test = ef.make();
  while(l>r){
    m=(l+r)/2;
    try{
      readEntryFromArrayPos(test,m);
    }
    catch(IndexOutOfBoundsException i){
      System.out.println("TheException");
      System.out.println("m is "+m+" and num is "+getNr());
      int pos = buf.getInt(convArrayPosToBufPos(m));
      System.out.println("Getting from position "+pos);
      System.out.println("The length of uri is "+buf.getShort(pos+4));
      throw new IllegalArgumentException("Sweet error");
    }
    if(BrilleDefinitions.DEBUG){
      logger.debug("reading in");
      logger.debug(test.toString());
    }
    if(en.compareTo(test)<0){
      if(BrilleDefinitions.DEBUG) logger.debug("if");
      r=m+1;
    }
    else if(en.compareTo(test)>0){
      if(BrilleDefinitions.DEBUG) logger.debug("else if");
      l=m-1;
```

```
      }
      else{
        if(BrilleDefinitions.DEBUG) logger.debug("else");
        l=m;
        r=m;
      }
    }
    int ret=((l+r)/2);
    if(BrilleDefinitions.DEBUG) logger.debug("think I should return "+ret);
    if(ret>=0){
      readEntryFromArrayPos(test,ret);
      if(BrilleDefinitions.DEBUG)
        logger.debug("the one i think is "+test.toString());
      if(en.compareTo(test)>0){
        if(BrilleDefinitions.DEBUG) logger.debug("decing");
        ret--;
      }
    }
    return ret;
  }

  /**
   * Method for inserting a new entry into this node. This method should only
   * be used when we know that there is room for the argument entry in this
   * node.
   *
   * @param en - the entry to be inserted
   */
  public void insert(Entry en){
    if(BrilleDefinitions.DEBUG){
      logger.debug("inserting entry:");
      logger.debug(en.toString());
    }

    if(BrilleDefinitions.DEBUG && !isSorted()){
      logger.error("Not before insert");
      logger.error(" to insert:");
      logger.error(en.toString());
      logger.error("In node:");
      logger.error(toString());
      throw new Error("doh before insert!");
    }

    if(BrilleDefinitions.DEBUG && testpositions()!=-1){
      int val = testpositions();
      logger.error("Error in testpositions nr "+val+" before insert");
      logger.error("in node:");
      logger.error(toString());
      Entry err = ef.make();
      readEntryFromArrayPos(err,val);
      logger.error("the current entry there is:");
      logger.error(err.toString());
      throw new Error("fucked positions before insert");
    }

    if(BrilleDefinitions.DEBUG){
      logger.info("in insert whith nr "+getNr());
      logger.info("The node nr is "+part);
      logger.info("should insert into:");
      logger.info(toString());
      logger.info("and the entry I should insert is "+en.toString());
      logger.info("free space is "+getFreeBytes()+
          " and this one uses "+en.length());
    }

    int arrayPos = findSmallestLargerThanOrEqual(en);
    if(BrilleDefinitions.DEBUG)
      logger.info("found smallest larger than or equal at arraypos "+
          arrayPos);
```

```
Entry test = ef.make();


if(arrayPos>=0){
  if(BrilleDefinitions.DEBUG)
    logger.info("should check for uniqueness in insert");
  readEntryFromArrayPos(test,arrayPos);
  if(en.compareTo(test)==0)
    throw new
      IllegalArgumentException("Trying to insert record with " +
          "nonunique key "+en.getKeyAsString());
}
if(BrilleDefinitions.DEBUG)
  logger.debug("uniqueness test passed and nr is "+getNr());
movePosesLargerThan(arrayPos);
if(BrilleDefinitions.DEBUG)
  logger.debug("have moved poses larger than "+
      arrayPos+" and nr is "+getNr());
int putInPos = nextFreePos();
if(BrilleDefinitions.DEBUG){
  for(int i=0;i<getNr();i++){
    int val = buf.getInt(convArrayPosToBufPos(i));
    if(val==putInPos){
      logger.error("Pos already present in arraypos "+i+" in node:");
      logger.error(toString());
      Entry errorentry = ef.make();
      readEntryFromArrayPos(errorentry,i);
      logger.error("it currently contains the entry:");
      logger.error(errorentry.toString());
      throw new Error("Pos already present in arraypos "+i);
    }
  }
}
if(BrilleDefinitions.DEBUG)logger.debug("putInPos is "+
    putInPos+" and nr is "+getNr());

if(BrilleDefinitions.DEBUG)
  logger.debug("have written entry and nr is "+getNr());
incBytesUsed(en.write(buf,putInPos));
if(BrilleDefinitions.DEBUG)
  logger.debug("have incedBytes and nr is "+getNr());
incNr();
if(BrilleDefinitions.DEBUG)logger.debug("after incnr nr is "+getNr());

buf.putInt(putInPos,convArrayPosToBufPos(arrayPos+1));
if(BrilleDefinitions.DEBUG){
  logger.debug("Done Inserting. Have node:");
  logger.debug(toString());
}
if(BrilleDefinitions.DEBUG && testpositions()!=-1){
  int val = testpositions();
  logger.error("Error in testpositions nr "+val+" after insert");
  logger.error("in node:");
  logger.error(toString());
  Entry err = ef.make();
  readEntryFromArrayPos(err,val);
  logger.error("the current entry there is:");
  logger.error(err.toString());
  throw new Error("fucked positions after insert");
}


if(BrilleDefinitions.DEBUG && !isSorted()){
  logger.error("Not sorted after insert");
  logger.error("inserted:");
  logger.error(en.toString());
  logger.error("Resulting node:");
  logger.error(toString());
  logger.error("arrayPos is "+arrayPos);
```

```
      logger.error("testing for duplicates");
      for(int i=0;i<getNr();i++){
        int val = buf.getInt(convArrayPosToBufPos(i));
        if(val==putInPos) logger.error("is in arraypos "+i);
      }
      throw new Error("doh after insert!");
    }
  }


  private boolean isSorted(){
    if(getNr()>1){
      Entry last = ef.make();
      Entry curr = ef.make();
      Entry tmp = null;
      readEntryFromArrayPos(curr,0);
      for(int i=1;i<getNr();i++){
        tmp = last;
        last = curr;
        curr = tmp;
        readEntryFromArrayPos(curr,i);
        if(last.compareTo(curr)<0) return false;
      }
    }
    return true;
  }

  /**
   * Method for testing whether the argument entry has a key different from
   * all entries in this node.
   *
   * @param en - the entry we want to check uniqueness for
   * @return - whether or not the key in the argument entry is unique
   */
  public boolean isNotUnique(Entry en){
    int arrayPos = findSmallestLargerThanOrEqual(en);
    if(arrayPos>=0 && arrayPos<getNr()){
      Entry test = ef.make();
      readEntryFromArrayPos(test,arrayPos);
      return test.compareTo(en)==0;
    }
    return false;
  }

  /**
   * Method to search within this node for an entry with a key
   * similar to the argument key. A BTreeHandle to the smallest entry with a
   * key larger or equal to the argument entry is returned.
   *
   * @param en - entry with the key to search for
   * @return - a handle to the closest larger or equal hit
   */
  public BTreeHandle searchForLargerOrEqual(Entry en){
    return new BTreeHandle(this,findSmallestLargerThanOrEqual(en));
  }

  private int nextFreePos(){
    return 14+getBytesUsed();
  }

  /**
   * This method is only private, but quite important when inserting entries,
   * and thus commented. It just moves all pointers to entries at all larger
   * positions one position to the left, so that there is room for a new entry
   * in between
   *
   * @param arrayPos the position in the pointer array we want to be free.
   */
  private void movePosesLargerThan(int arrayPos){
```

```
  for(int i=getNr()-1;i>arrayPos; i--)
    buf.putInt(buf.getInt(convArrayPosToBufPos(i)),
        convArrayPosToBufPos(i+1));
}

/**
 * Method used to delete the entry with the similar key to the argument
 * entry from this node. The method assumes that an entry with the argument
 * entry's key is present in this node, otherwise, an
 * IllegalArgumentException is thrown.
 *
 * @param en - an entry with the same key as the node we will delete  from
 *  this node.
 */
public void deleteEntry(Entry en){

  int arrayPos = findSmallestLargerThanOrEqual(en);
  if(BrilleDefinitions.DEBUG){
    if(arrayPos == getNr() ||
        en.compareTo(getEntryFromArrayPos(arrayPos))!=0){
      logger.error("illargex");
      logger.error("Node is ");
      logger.error(toString());
      logger.error("and should delete:");
      logger.error(en.toString());
      logger.error("");
      if(getNextPointer()!=Integer.MAX_VALUE){
        logger.error("Also printing next node");
        logger.error(getNodeFromNextPointer().toString());
      }
    }
  }
  if(arrayPos == getNr() || en.compareTo(getEntryFromArrayPos(arrayPos))!=0)
    throw new
      IllegalArgumentException("Trying to delete a key that does not exist");
  int comp = buf.getInt(convArrayPosToBufPos(arrayPos));
  IntArrayList largerThanArrayPos = new IntArrayList();
  for(int i=getNr()-1;i>=0;i--)
    if(i!=arrayPos && buf.getInt(convArrayPosToBufPos(i))>comp)
      largerThanArrayPos.add(i);
  sortBasedOnArrayVals(largerThanArrayPos,0,largerThanArrayPos.size()-1);
  int nextpos = buf.getInt(convArrayPosToBufPos(arrayPos));
  Entry move = ef.make();
  readEntryFromArrayPos(move,arrayPos);
  decBytesUsed(move.length());
  for(int i=0;i<largerThanArrayPos.size();i++){
    readEntryFromArrayPos(move,largerThanArrayPos.get(i));
    move.write(buf,nextpos);
    buf.putInt(nextpos,convArrayPosToBufPos(largerThanArrayPos.get(i)));
    nextpos+=(move.length());
  }
  for(int i=arrayPos+1;i<getNr();i++){
    buf.putInt(buf.getInt(convArrayPosToBufPos(i)),
        convArrayPosToBufPos(i-1));
  }
  decNr();
  if(BrilleDefinitions.DEBUG && !isSorted()){
    logger.error("Not sorted after delete");
    logger.error(toString());
    throw new Error("doh after delete!");
  }
}

private void sortBasedOnArrayVals(IntArrayList l, int f, int t){
  if(f>=t) return;
  int q = partition(l,f,t);
  sortBasedOnArrayVals(l,f,q-1);
  sortBasedOnArrayVals(l,q+1,t);
}
```

```
  private int partition(IntArrayList l, int f, int t){
    int m = (f+t)/2;
    l.switchPoses(m,t);
    int i=f;
    int comp = buf.getInt(convArrayPosToBufPos(l.get(t)));
    for(int j=f;j<t;j++){
      if(buf.getInt(convArrayPosToBufPos(l.get(j)))<=comp){
        l.switchPoses(j,i++);
      }
    }
    l.switchPoses(i,t);
    return i;
  }

  /**
   * Method used to release this node when it will not be used anymore.
   * It makes sure that the buffer where this node is represented is unpinned
   * from the buffer pool.
   *
   */
  public void release(){
    bp.unpinBuffer(buf,true,true);
  }

  /**
   * Method for getting the node this node's next pointer points to.
   *
   * @return the node to the right of this one in the same level in the tree
   */
  public Node getNodeFromNextPointer(){
    int wantPart = getNextPointer();
    if(wantPart==Integer.MAX_VALUE) return null;
    Buffer newBuf = bp.pinBuffer(wantPart,fileNr,true,true);
    Node ret = new Node(newBuf,ef,bp,wantPart,fileNr);
    return ret;
  }
}
```

C.2.8 brille.btree.Update

```
package brille.btree;

/**
 * This class is used to represents update that should occur in the B-tree.
 * Updates in the B-tree should be carried out when an insert has caused a
 * split in the tree, and we need to insert new entries in the above level.
 *
 * An update is represented by the level at which it should occur. One of the
 * nodes to be pointed to existed before the split. The new node will have
 * the same key as this one, and if the update does not create a new root,
 * the old entry to the old node, could only be updated to point to the
 * new one instead. This must happen AFTER the new entry with the pointer
 * to the old node is inserted though, to ensure serializability. The new
 * and old entries mentioned here are represented as old1 and new1 in this
 * class.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.btree.UpdateThread
 */
public class Update {
  private Entry old1;
  private short level;
  private Entry new1;

  /**
   * Constructor for updates to the B-tree, taking all member variables as
```

```
   * arguments.
   *
   * @param old1 - the old entry mentioned above
   * @param level - the level for this update
   * @param new1 - the new entry
   */
  public Update(Entry old1, short level, Entry new1){
    this.level=level;
    this.old1=old1;
    this.new1=new1;
  }

  /**
   * Getter for the level at which this update should take place.
   *
   * @return the level of the update
   */
  public short getLevel() {
    return level;
  }

  /**
   * Getter for the new entry to be inserted
   *
   * @return the new entry
   */
  public Entry getNew() {
    return new1;
  }

  /**
   * Getter for the entry that should possible only be updated.
   *
   * @return the (possibly) old entry
   */
  public Entry getOld() {
    return old1;
  }


  public String toString(){
    String ret = "u|"+level;
    if(old1!=null)ret+="|o1"+old1.toString();
    if(new1!=null)ret+="|n1"+new1.toString();
    return ret;
  }
}
```

C.2.9 brille.btree.UpdateThread

```
package brille.btree;

import org.apache.log4j.Logger;

/**
 * This class is responsible for updating the B-tree when new updates
 * comes in. In its current form, it checks for new updates at regular
 * intervals. If there are any new updates, they are processed until the
 * line of updates is empty.
 *
 * In a future implementation this thread will only be run when a thread
 * inserting in the B-tree notifies it that it has added an update.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class UpdateThread  extends Thread {
```

```
  protected final Logger logger = Logger.getLogger(getClass());

  private long timeToWait;
  private BTree tree;
  private boolean keepRunning;

  /**
   * The constructor. The arguments gives the waiting time for this thread
   * between each check for new updates, and the tree to update for.
   *
   * @param timeToWait - the waiting time between each check for updates
   * @param tree - the tree to be the update thread for
   */
  public UpdateThread(long timeToWait, BTree tree){
    this.timeToWait=timeToWait;
    this.tree=tree;
    keepRunning=true;
  }

  /**
   * Alternative constructor where the time to wait between checks for updates
   * is set to the default value, 50 ms.
   *
   * @param tree - the tree to be the update thread for
   */
  public UpdateThread(BTree tree){
    this(50,tree);
  }

  /**
   * Method to be called when the application shuts down, to make sure that
   * this thread stops running.
   *
   */
  public void shutDown(){
    keepRunning=false;
  }

  public void run(){
    while(keepRunning){
      try{
        Thread.sleep(timeToWait);
      }catch(InterruptedException e){
        e.printStackTrace();
        System.out.println("Exception in updatethread");
        System.exit(1);
      }
      try{
        Update toDo = tree.removeNextUpdate();
        while(toDo!=null){
          tree.handleUpdate(toDo);
          toDo=tree.removeNextUpdate();
        }
      }catch(Exception e){
        System.out.println("Exception in updatethread");
        logger.error("Exception in updatethread",e);
        e.printStackTrace();
        System.exit(1);
      }
      catch(Error e){
        System.out.println("Error in updatethread for B-tree");
        e.printStackTrace();
      }
    }
  }
}
```

C.3 brille.buffering

C.3.1 brille.buffering.Buffer

```
package brille.buffering;

import java.nio.channels.FileChannel;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.util.concurrent.locks.ReentrantReadWriteLock;
import java.io.IOException;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;

/**
 * This class represents buffers in Brille. It is basically a wrapper around
 * java.nio.DirectByteBuffer. These buffers exists through the whole lifecycle
 * of brille. Therefore direct buffers where chosen over HeapByteBuffers.
 * HeapByteBuffers are faster to initiate, but slower in use.
 *
 * This class also provides methods for manipulating the ByteBuffer, flushing
 * it, and obtaining locks on it.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class Buffer extends LinkedListElement{

  protected final Logger logger = Logger.getLogger(getClass());

  protected ByteBuffer buf;
  protected long part;
  protected int nrReadingOrWriting;
  protected boolean dirty;
  protected long bufferSize;
  protected FileChannel channel;
  protected int filenr;
  protected int bufferNr;

  private ReentrantReadWriteLock lock;

  /**
   * Constructor for buffers, taking the size of the buffer, and its number
   * as arguments. The number is only used for debugging purposes, but is
   * considered to have an impact small enough that it is not important to
   * remove it.
   *
   * @param bufferSize - the size of the buffer to create
   * @param bufferNr - the number of this buffer (should be unique for each
   *   buffer pool)
   */
  public Buffer(int bufferSize,int bufferNr){
    buf = ByteBuffer.allocateDirect(bufferSize);
    buf.order(ByteOrder.nativeOrder());
    this.bufferSize=(long)bufferSize;
    part=-1;
    nrReadingOrWriting=0;
    dirty=false;
    this.bufferNr=bufferNr;
    filenr=-1;
    lock = new ReentrantReadWriteLock(true);
  }

  /**
   * Method for obtaining a readlock on the file and part currently held by
```

```
 * this buffer.
 *
 */
public void readLock(){
  lock.readLock().lock();
}

/**
 * Method for obtaining a writelock on the file and part currently held by
 * this buffer.
 *
 */
public void writeLock(){
  lock.writeLock().lock();
}

/**
 * Method for releasing the readlock on the file and part currently held by
 * this buffer.
 *
 */
public void readUnlock(){
  lock.readLock().unlock();
}

/**
 * Method for releasing the writelock on the file and part currently held by
 * this buffer.
 *
 */
public void writeUnlock(){
  lock.writeLock().unlock();
}

/**
 * This method makes sure this buffer is set as not dirty even though it
 * might be. This is useful when the file it caches from should be deleted.
 *
 */
public void setUnDirty(){
  dirty=false;
}

/**
 * This method is only used for testing.
 *
 */
public void setDirty(){
  dirty=true;
}

/**
 * Retrieving the number of this buffer.
 *
 * @return the buffer number
 */
public int getBufferNr(){
  return bufferNr;
}

/**
 * Getter for size of buffer
 *
 * @return buffersize
 */
public long getBufferSize(){
  return bufferSize;
}
```

```
/**
 * Method for testing whether the buffer is dirty (contains values not yet
 * flushed to disk).
 *
 * @return whether or not the buffer is dirty
 */
public synchronized boolean isDirty(){
  return dirty;
}

/**
 * Retrieving the channel this buffer contains a part from.
 *
 * @return the current channel
 */
public FileChannel getChannel(){
  return channel;
}

/**
 * Retrieving the number of the file this buffer caches from.
 *
 * @return file number
 */
public int getFileNr(){
  return filenr;
}

/**
 * Method for switching the channel this buffer should cache from.
 * Used when pinning a buffer to a new part.
 *
 * @param channel - the new channel
 * @param filenr - the file number in the buffer pool of this channel
 */
public void setChannel(FileChannel channel,int filenr){
  this.channel=channel;
  this.filenr=filenr;
  part=-1;
}


/**
 * Method for flushing the contents of this buffer to the position in the
 * file it caches from.
 *
 */
public synchronized void flush(){
  if(dirty && channel!=null){
    try{
      buf.limit((int)bufferSize);
      dirty=false;
      channel.write(buf,part*bufferSize);

      buf.clear();
    }catch(IOException e){
      e.printStackTrace();
      System.exit(1);
    }
  }
  else if(channel==null) dirty=false;
}


/**
 * Method for reading in a new part in this buffer. It reads in the argument
 * part from the channel currently set.
 *
 * @param part - the part of the current file to buffer
```

```
 */
public synchronized void readIn(long part){
  if(this.part!=part && channel!=null){
    buf.clear();
    try {
      channel.read(buf,part*bufferSize);
    } catch (IOException e) {
      e.printStackTrace();
      System.out.println("FUCKED IN BUFFER!!!");
      //System.exit(1);
    }
    this.part=part;
    buf.flip();
    buf.limit((int)bufferSize);
  }else if(channel==null) this.part=part;
}

/**
 * Method for retrieving the part currently buffered.
 *
 * @return the currently buffered part, or -1 if no part is buffered
 */
public synchronized long getPart(){
  return part;
}

/**
 * This method is only used for testing purposes and when the buffer is
 * instantiated. It sets the part this buffer currently contains, but
 * this should be done through readIn(part) when running the application.
 *
 * @param part - the part to set this buffer's part to
 */
public void setPart(long part){
  this.part=part;
}

/**
 * Method for registering that one more thread is currently reading
 * (or writing to) this buffer. This is used for knowing when a buffer
 * is free.
 *
 */
public synchronized void incNrReading(){
  nrReadingOrWriting++;
}

/**
 * Method for obtaining the number of threads reading from this buffer.
 *
 * @return the number of threads currently reading this buffer.
 */
public synchronized int getNrReading(){
  return nrReadingOrWriting;
}

/**
 * Method for registering that one less thread is currently reading
 * (or writing to) this buffer. This is used for knowing when a buffer
 * is free.
 *
 * @return whether or not there are any more threads reading this buffer
 */
public synchronized boolean decAndIsZeroReading(){
  return (--nrReadingOrWriting==0);
}

/**
 * Method for putting an int to the given position in the buffer.
```

```
 *
 * @param val - the int to write
 * @param pos - the position in the buffer to write it to
 */
public void putInt(int val, int pos){
  if(BrilleDefinitions.DEBUG && pos==0)
    logger.info("writing int to pos 0 for part "+part+" and filenr "+
        filenr+" value: "+val);
  buf.putInt(pos,val);
  dirty=true;
}

/**
 * Retrieving an int from the given position in the buffer.
 *
 * @param pos  - the position in the buffer to retrieve the int from
 * @return the int represented at the given position in the buffer
 */
public int getInt(int pos){
  return buf.getInt(pos);
}

/**
 * Method for putting a short to the given position in the buffer.
 *
 * @param val - the short to write
 * @param pos - the position in the buffer to write it to
 */
public void putShort(short val, int pos){
  buf.putShort(pos,val);
  dirty=true;
}

/**
 * Retrieving a short from the given position in the buffer.
 *
 * @param pos - the position in the buffer to retrieve the short from
 * @return the short represented at the given position in the buffer
 */
public short getShort(int pos){
  return buf.getShort(pos);
}

/**
 * Method for putting a byte to the given position in the buffer.
 *
 * @param val - the byte to write
 * @param pos - the position in the buffer to write it to
 */
public void putByte(byte val, int pos){
  buf.put(pos,val);
  dirty=true;
}

/**
 * Retrieving a byte from the given position in the buffer.
 *
 * @param pos - the position in the buffer to retrieve the byte from
 * @return the byte represented at the given position in the buffer
 */
public byte getByte(int pos){
  return buf.get(pos);
}

/**
 * Method for putting a long to the given position in the buffer.
 *
 * @param val - the long to write
 * @param pos - the position in the buffer to write it to
```

```
   */
  public void putLong(long val, int pos){
    dirty=true;
    buf.putLong(pos,val);
  }

  /**
   * Retrieving a long from the given position in the buffer.
   *
   * @param pos - the position in the buffer to retrieve the long from
   * @return the long represented at the given position in the buffer
   */
  public long getLong(int pos){
    return buf.getLong(pos);
  }


  public boolean removeFromList(boolean dirtyValue) {
    if(isDirty()!=dirtyValue)return false;
    previous.setNext(next);
    next.setPrevious(previous);
    next=null;
    previous=null;
    return true;
  }

  public String nrToString(){
    return "bufnr: "+bufferNr;
  }

  public String toString(){
    return "bufnr: "+bufferNr+" part: "+
      part+" dirty: "+(dirty?"yes":"no")+" next: "+
      (next==null?"null":next.nrToString())+" prev: "
      +(previous==null?"null":previous.nrToString());
  }

  public boolean inQueue(boolean dirtyVal) {
    if(dirtyVal==dirty && inOneQueue()) return true;
    return false;
  }
}
```

C.3.2 brille.buffering.BufferFIFOQueue

```
package brille.buffering;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;

/**
 * This class is basically a linked list used for queues of buffers.
 * The reason why this is implemented instead of using java.util.LinkedList
 * is that the LinkedListImplementation contains pointers to the objects,
 * while in this implementation, the objects themselves have pointers to their
 * neighbors. In this way, we can faciliate removing an object from the queue
 * by only calling methods on the object itself.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class BufferFIFOQueue extends LinkedListElement{

  protected final Logger logger = Logger.getLogger(getClass());

  private boolean dirtyValue;
```

```
/**
 * Constructor making a queue for either dirty or clean buffers.
 *
 * @param dirtyValue - whether or not this should be a queue for dirty
 *   buffers
 */
public BufferFIFOQueue(boolean dirtyValue){
  super();
  this.dirtyValue=dirtyValue;
  next=this;
  previous=this;
}

/**
 * Method to remove the given element from this queue.
 *
 * @param el - the element to remove
 * @return whether or not the element was successfully removed
 */
public synchronized boolean removeElement(LinkedListElement el){
  if(!el.inQueue(dirtyValue)) throw new IllegalArgumentException("");
  return el.removeFromList(dirtyValue);
}

/**
 * Method for adding an element at the end of this linked list.
 *
 * @param el - the element to add
 */
public synchronized void addElementLast(LinkedListElement el){
  previous.setNext(el);
  el.setPrevious(previous);
  el.setNext(this);
  previous=el;
  notifyAll();
}

/**
 * Method to obtain the first element in this list
 *
 * @return the first element of this list
 */
public synchronized LinkedListElement getFirst(){
  LinkedListElement ret = next;
  if(!(ret instanceof Buffer)) return null;
  return ret;
}

/**
 * Method used to remove and return the first element of this list.
 *
 * @return the first element in this list (which is also removed)
 */
public synchronized LinkedListElement removeFirst(){
  if(BrilleDefinitions.UNPIN_BUG_DEBUG)
    logger.debug("should remove first...");
  LinkedListElement ret = next;
  if(BrilleDefinitions.UNPIN_BUG_DEBUG) logger.debug("got it");
  if(!(ret instanceof Buffer)) return null;
  if(BrilleDefinitions.UNPIN_BUG_DEBUG) logger.debug("did not return null");
  if(!ret.removeFromList(dirtyValue))
    throw new
      IllegalArgumentException("FUCKED "+(dirtyValue?"dirty":"not dirty"));
  if(BrilleDefinitions.UNPIN_BUG_DEBUG) logger.debug("no exception");
  return ret;
}

/**
 * Method used to wait for this queue. This is typically used when we wait
```

```
 * for the queue to get some elements.
 *
 * @throws InterruptedException
 */
public synchronized void waitFor() throws InterruptedException{
  if(next instanceof BufferFIFOQueue)wait();
}

public boolean removeFromList(boolean dirtyValue){
  throw new IllegalArgumentException("can't remove the queue itself");
}



public boolean inQueue(boolean dirtyVal) {
  return false;
}

/**
 * Method used to remove the given element from this queue if it is part of
 * this queue.
 *
 * @param toRemove - the element to remove
 * @return whether or not the element was removed
 */
public synchronized boolean ifInRemoveElement(Buffer toRemove) {
  if(toRemove.inQueue(dirtyValue)){
    if(toRemove.removeFromList(dirtyValue)) return true;
    else throw new IllegalArgumentException("Should be in list but isn't");
  }
  else return false;
}

public String nrToString(){
  return "QUEUE";
}

public String toString(){
  return "QUEUE next: "+
    (next==null?"null":next.nrToString())+" prev: "+
      (previous==null?"null":previous.nrToString());
}

/**
 * Method used to print the contents of this linked list. It is typically
 * used for debugging purposes, and prints to the log.
 *
 * @param prefix - the prefix of all printed lines.
 */
public void printQueue(String prefix){
  LinkedListElement el = next;
  logger.debug(prefix+"Printing queue:");
  while(el instanceof Buffer){
    logger.debug(prefix+el.toString());
    el=el.getNext();
  }
}
}
```

C.3.3 brille.buffering.BufferPool

```
package brille.buffering;

import java.io.File;
import java.nio.channels.FileChannel;
import java.util.ArrayList;
```

```java
import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.locking.LockManager;
import brille.utils.IntArrayList;

/**
 * This class contains the old implementation of a buffer pool. It is
 * very slow, but still used sometimes for correctness testing. This
 * is the most thoroughly tested buffer pool, and if bugs are suspected
 * to come from the buffer pool, it can be changed to this one to make that
 * less likely.
 *
 * This bufferpool only consists of two ArrayLists, one with pinned buffers
 * and one with free buffers. When pinning a buffer, it switches queue.
 *
 * All methods here are defined in brille.buffering.BuffPool, and are thus
 * commented there.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class BufferPool implements BuffPool{

  protected final Logger logger = Logger.getLogger(getClass());

  private ArrayList<Buffer> freeBuffers;
  private ArrayList<Pin> pinnedBuffers;
  private ArrayList<FileChannel> files;
  private ArrayList<File> realfiles;
  private IntArrayList freeList;
  private LockManager lm;


  /**
   * Constructor for BufferPools. The only argument is the number of buffers
   * this pool should contain. The size of the buffers is found in
   * brille.BrilleDefinitions.BUFFER_SIZE.
   *
   * @param nrOfBuffers - the number of buffers this buffer pool should contain
   */
  public BufferPool(int nrOfBuffers){
    freeBuffers = new ArrayList<Buffer>();
    pinnedBuffers = new ArrayList<Pin>();
    files = new ArrayList<FileChannel>();
    realfiles = new ArrayList<File>();
    freeList = new IntArrayList();
    lm = new LockManager();
    for(int i=0;i<nrOfBuffers;i++)
      freeBuffers.add(new Buffer(BrilleDefinitions.BUFFER_SIZE,i));
  }


  public synchronized int addFile(FileChannel ch,File f){
    if(freeList.size()>0){
      int ret = freeList.remove(0);
      files.set(ret,ch);
      realfiles.set(ret,f);
      return ret;
    }
    files.add(ch);
    realfiles.add(f);
    return files.size()-1;
  }

  public synchronized void removeFileWithNr(int nr){
    freeList.add(nr);
  }
```

```java
public Buffer pinBuffer(long part,int fileNr, boolean read, boolean lock){
  Buffer ret =null;
  if(lock){
    if(read) lm.readLock((int)part);
    else lm.writeLock((int)part);
  }
  synchronized(this){
    outer: while(ret==null){
      for(Buffer b: freeBuffers){
        if(b.getPart()==part && b.getFileNr()==fileNr){
          if(BrilleDefinitions.DEBUG)
            logger.info("found freebuffer with correct part");
          pinnedBuffers.add(new Pin(b, part,fileNr));
          freeBuffers.remove(b);
          b.incNrReading();
          ret=b;
          break outer;
        }
      }
      if(read){
        for(Pin p:pinnedBuffers){
          if(p.getPart()==part && p.getFileNr()==fileNr){
            if(BrilleDefinitions.DEBUG)
              logger.info("found pinned buff with correct part");
            ret=p.getBuffer();
            ret.incNrReading();
            break outer;
          }
        }
      }
      if(freeBuffers.size()>0){
        if(BrilleDefinitions.DEBUG){
          logger.info("need new buffer!");
          logger.info("freebuffers.size()="+freeBuffers.size());
        }
        ret=freeBuffers.remove(0);
        ret.flush();
        ret.incNrReading();
        ret.setChannel(files.get(fileNr),fileNr);
        ret.readIn((int)part);
        break;
      }
      try{
        wait();
      }catch(InterruptedException e){
        e.printStackTrace();
      }
    }
  }
  return ret;
}


public void unpinBuffer(Buffer b, boolean read, boolean lock){
  int part = (int)b.getPart();
  synchronized(this){
    if(b.decAndIsZeroReading()){
      pinnedBuffers.remove(b);
      removeFromPinned(b);
      freeBuffers.add(b);
      notify();
    }
  }
  if(lock){
    if(read) lm.readUnlock(part);
    else lm.writeUnlock(part);
  }

}
```

```
  private void removeFromPinned(Buffer b){
    for(Pin p : pinnedBuffers){
      if(p.getBuffer().equals(b)){
        pinnedBuffers.remove(p);
        break;
      }
    }
  }


  public void unDirtyAllWithAndDelete(int i) {
    for(Buffer b: freeBuffers){
      if(b.getFileNr()==i && b.isDirty()){
        b.setUnDirty();
      }
    }
    for(Pin p:pinnedBuffers){
      if(p.getFileNr()==i){
        logger.error("deleting a file with a pin! Filenr "+i);
      }
    }
    if(realfiles.get(i)!=null)realfiles.get(i).delete();

  }


  public long totalDiskSize(){
    long ret = 0;
    for(int i=0;i<realfiles.size();i++)
      if(realfiles.get(i)!=null)
        ret+= realfiles.get(i).length();
    return ret;
  }


  public void shutDown(){
    //NO-OP
  }
}
```

C.3.4 brille.buffering.BuffPool

```
package brille.buffering;

import java.io.File;
import java.nio.channels.FileChannel;

/**
 * This interface defines what an implementation of a BufferPool should
 * provide. It has proven to be useful to have such an interface because
 * the bufferpool code is extensively used in Brille, and making slight
 * improvements regarding its efficiency has been rewarding. Hence, it is
 * likely that new versions will come in the future.
 *
 * The buffer pools primary task is to give an abstraction removing the access
 * to and buffering from files, and the possibility to pin part of files
 * is memory.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public interface BuffPool {

  /**
   * Method for adding a new file from which buffers can be pinned
   *
```

```
     * @param ch - the channel to the file
     * @param f - the physical file the channel represents
     * @return - the file number of this file. This value is used as a handle to
     *  the file when we want to pin buffers from it later on.
     */
    public int addFile(FileChannel ch,File f);

    /**
     * Method for removing a file from the buffer pool before we delete it.
     *
     * @param nr - the number of the file we want to remove
     */
    public void removeFileWithNr(int nr);

    /**
     * Probably one of the most important methods in a buffer pool. It is used
     * when we want to ensure that a buffer contains a given part of a given
     * file until the method unpinBuffer is called with this buffer as an
     * argument. This method also enables locking. If the lock variable is
     * set to true, the buffer part the buffer contains will be read or write
     * locked (depending on the read variable) before the buffer is returned.
     *
     * @param part - the part we want the buffer to contain.
     * @param fileNr - the file number for the file the buffer should cache
     *  from
     * @param read - whether or not this is a read (only used when lock is true)
     * @param lock - whether or not a lock should be obtained before the buffer
     *   is returned
     * @return a buffer containing the correct part from the given file
     */
    public Buffer pinBuffer(long part,int fileNr, boolean read, boolean lock);

    /**
     * Method to unpin a given buffer. It may also release a lock (read or write)
     * depending on the argument variables.
     *
     * @param b - the buffer to unpin
     * @param read - whether or not there was a read or a write that occurred
     *  (this argument is only used if lock is true)
     * @param lock - whether or not a lock should be released
     */
    public void unpinBuffer(Buffer b, boolean read, boolean lock);

    /**
     * When a file should be deleted, it is important to make sure no buffers
     * will flush to it after it is deleted. This method first marks all buffers
     * from the argument file as not dirty, and then deletes the file.
     *
     * @param i - the file number of the file we want to delete
     */
    public void unDirtyAllWithAndDelete(int i);

    /**
     * Method used to calculate the disk space occupied by the files this
     * buffer pool buffers from.
     *
     * @return
     */
    public long totalDiskSize();

    /**
     * Method to be called when the application shuts down, to give the buffer
     * pool a chance to shut down all its threads.
     *
     */
    public void shutDown();
}
```

C.3.5 brille.buffering.FlushingThread

```
package brille.buffering;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.utils.IntArrayList;

/**
 * This thread is responsible for flushing buffers in the dirty queue, and
 * moving them to the free queue. By organising the buffer pool this way, a
 * normal thread does not have to perform disk writes, but may have to wait
 * a while for a free buffer when the flushing thread has got a lot of
 * work to do.
 *
 * The threads pinning buffers are themselves responsible for reading in the
 * correct part if it is not currently in the buffer pinned.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FlushingThread extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private IntArrayList skip;
  private boolean shouldRun;
  private NewBufferPool bp;

  /**
   * Constructor for flushing thread. It takes the buffer pool to flush
   * buffers from as an argument.
   *
   * @param bp - the buffer pool to flush buffers from
   */
  public FlushingThread(NewBufferPool bp){
    this.bp=bp;
    skip = new IntArrayList();
    shouldRun=true;
  }

  /**
   * When we want to make sure that buffers will not be flushed to deleted
   * files, this method is called with the file number of the file we want to
   * delete. If that is done, no buffers with content from the given file
   * will be flushed, but rather set to be not dirty.
   *
   * @param toSkip - the file number to skip flushing to
   */
  public synchronized void addSkip(int toSkip){
    skip.add(toSkip);
  }

  /**
   * This is the method called at regular intervals in this thread, and it is
   * the method doing all the work. It removes the first buffer from the dirty
   * queue as long as there are buffers there, flushes it, and adds it to the
   * end of the free queue.
   *
   */
  private synchronized void doARound(){
    if(BrilleDefinitions.UNPIN_BUG_DEBUG){
      logger.error(Thread.currentThread()+" DOING A RUN!!!");
      logger.debug("doing run");
    }
    Buffer curr = bp.removeFirstFromDirty();
    while(curr!=null){
      if(BrilleDefinitions.UNPIN_BUG_DEBUG)
        logger.debug("Flushing removed "+curr.toString());
```

```
        boolean shouldFlush=true;
        for(int i=0;i<skip.size();i++)
          if(skip.get(i)==curr.getFileNr()) shouldFlush=false;
        if(shouldFlush){
          if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("should flush");
          curr.flush();
        }
        else{
          if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("should not flush");
          curr.setUnDirty();
        }

        if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("unpinning");
        bp.unpinBuffer(curr, true, true);
        if(BrilleDefinitions.UNPIN_BUG_DEBUG){
          logger.debug(Thread.currentThread()+" Buffer with nr "+
              curr.getBufferNr()+" part "+curr.getPart()+" moved to free");
          logger.debug(Thread.currentThread()+" FREE");
          bp.printFreeQueue(Thread.currentThread()+" ");
          logger.debug(Thread.currentThread()+" DIRTY");
          bp.printDirtyQueue(Thread.currentThread()+" ");
          logger.debug(Thread.currentThread()+" ");
          logger.debug("pinning new");
        }
        curr = bp.removeFirstFromDirty();
      }
      skip = new IntArrayList();
      if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("done with run");
    }

    /**
     * Method used to make sure this thread will be stopped. This method is
     * typically called when the application shuts down.
     *
     */
    public void stopRunning(){
      shouldRun=false;
    }

    public void run(){
      while(shouldRun){
        try{
          doARound();
        }catch(Exception e){
          System.out.println("FlushingThread exception: ");
          e.printStackTrace();
          //System.exit(1);
        }catch(Error e){
          System.out.println("FlushingThread error:");
          e.printStackTrace();
        }
        try{
          Thread.sleep(
              BrilleDefinitions.SLEEP_TIME_FOR_BUFFERPOOL_FLUSHINGTHREAD);
        }catch(InterruptedException ie){
          logger.error("FlushingThread couldn't sleep",ie);
          shouldRun=false;
        }
      }
    }
  }
```

C.3.6 brille.buffering.LinkedListElement

```
package brille.buffering;

/**
```

```java
 * This class defines an element in a doubly linked list. It is used because
 * it is impossible to say to an element in a java.util.LinkedList that it
 * should remove itself, because that implementation only contains pointers
 * to the element from the linked list itself.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public abstract class LinkedListElement {
  protected LinkedListElement next;
  protected LinkedListElement previous;

  /**
   * Default constructor, initiates both pointers to null.
   *
   */
  public LinkedListElement(){
    next=null;
    previous=null;
  }

  /**
   * Getter for the next pointer.
   *
   * @return the next pointer
   */
  public LinkedListElement getNext() {
    return next;
  }

  /**
   * Setter for the next pointer.
   *
   * @param next - the pointer to the next element
   */
  public void setNext(LinkedListElement next) {
    this.next = next;
  }

  /**
   * Getter for the pointer to the previous element in the list.
   *
   * @return the previous element
   */
  public LinkedListElement getPrevious() {
    return previous;
  }

  /**
   * Setter for the previous element in the list.
   *
   * @param previous - the new previous element
   */
  public void setPrevious(LinkedListElement previous) {
    this.previous = previous;
  }

  /**
   * This method is especially adapted to the way linked lists are used
   * in NewBufferPool. It will try to remove this element from the queue
   * if this queue contains dirty buffers, if the argument is true, and
   * clean buffers otherwise. It returns whether or not it succeeded removing
   * itself from the given list.
   *
   * @param dirtyValue - whether or not it is the dirty list we try to remove
   *   this entry from
   * @return whether or not we succeeded removing the entry from the given
   *   list
   */
  public abstract boolean removeFromList(boolean dirtyValue);
```

```
  /**
   * Method for testing whether this element is in a linked list.
   *
   * @return whether or not the element is in a list
   */
  public boolean inOneQueue(){
    return next!=null;
  }

  /**
   * Method for testing whether this element is part of the given queue
   * (dirty or free).
   *
   * @param dirtyVal - whether or not it is the dirty queue we should test
   *  whether this element is part of
   * @return whether or not this element is part of the given queue
   */
  public abstract boolean inQueue(boolean dirtyVal);

  /**
   * Method for getting the number of this element as a string.
   *
   * @return a string representation of this element's number
   */
  public abstract String nrToString();


  public abstract String toString();
}
```

C.3.7 brille.buffering.NewBufferPool

```
package brille.buffering;

import java.io.File;
import java.io.IOException;
import java.nio.channels.FileChannel;
import java.util.ArrayList;
import java.util.HashMap;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.utils.IntArrayList;

/**
 * This class contains the implementation of the second version of a buffer
 * pool, with severly improved performance. Its primary task is to be a
 * container for all buffers, and provide ways to pin them (that means to
 * make sure their contents are kept until the same thread unpins it).
 *
 * This bufferpool uses three main structures, two linked lists of buffers,
 * and a java.util.HashMap. If a buffer is pinned by at least one thread, it
 * is only available through the HashMap. It can be found through the HashMap
 * by getting the entry with the file number and part this buffer is pinned
 * to contain. If a buffer is not pinned, it may also be available through the
 * HashMap, and that happens when it contains a part of a file (which it always
 * does as long as it has been used since it was initialized).
 *
 * When a buffer is not pinned, it will also be present in one of the two
 * linked lists. They are called dirty and free. If the buffer is dirty, which
 * means that it contains changes that are not flushed to disk yet, it will be
 * found in the dirty list, otherwise it is in the free list. A thread of type
 * brille.btree.FlushingThread moves entries from the dirty list to the free
 * list after it has flushed the buffer.
 *
```

```
 * Most of the methods here are declared in brille.buffering.BuffPool, and are
 * thus not commented here. The current implementation does actually not
 * support reuse of file numbers, because it could cause synchronization
 * issues, but this will be supported to save memory in future implementations.
 *
 * Warning: This class should not be modyfied if you are not perfectly aware
 * of what you are doing. Moving something into or out of the synchronized
 * regions may cause severe errors.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class NewBufferPool implements BuffPool{

  protected final Logger logger = Logger.getLogger(getClass());

  private HashMap<Pair, Buffer> map;
  private BufferFIFOQueue dirty;
  private BufferFIFOQueue free;
  private ArrayList<FileChannel> files;
  private ArrayList<File> realfiles;
  private IntArrayList freeList;
  private FlushingThread ft;

  /**
   * Constructor to create a new buffer pool. The only argument is the number
   * of buffers to be contained in this bufferpool. The size of each of the
   * buffers is found from BrilleDefinitions.BUFFER_SIZE.
   *
   * @param nrOfBuffers - the number of buffers in this bufferpool
   */
  public NewBufferPool(int nrOfBuffers){
    files = new ArrayList<FileChannel>();
    realfiles = new ArrayList<File>();
    freeList = new IntArrayList();
    dirty = new BufferFIFOQueue(true);
    free = new BufferFIFOQueue(false);
    map = new HashMap<Pair,Buffer>();
    for(int i=0;i<nrOfBuffers;i++)
      free.addElementLast(new Buffer(BrilleDefinitions.BUFFER_SIZE,i));
    ft = new FlushingThread(this);
    ft.start();
  }


  public synchronized int addFile(FileChannel ch,File f){
    if(freeList.size()>0){
      int ret = freeList.remove(0);
      files.set(ret,ch);
      realfiles.set(ret,f);
      return ret;
    }
    files.add(ch);
    realfiles.add(f);
    return files.size()-1;
  }

  public synchronized void removeFileWithNr(int nr){
    //Not doing this removes the problem that caused the memory problems..
    //freeList.add(nr);
    //this should be supported in a future implementation
  }

  public Buffer pinBuffer(long part,int fileNr, boolean read,boolean lock){
    Buffer ret = null;
    Pair p = new Pair(fileNr,part);
    while(true){
      synchronized(this){
        if(map.containsKey(p)){
          ret = map.get(p);
```

```
          ret.incNrReading();
          //if we get something with the wrong channel... but the same filenr.
          //therefore, the reuse of filenumbers is disabled at the moment
          while(ret.inOneQueue()){
            if(dirty.ifInRemoveElement(ret)) break;
            if(free.ifInRemoveElement(ret)) break;
          }
          break;
        }
        else{
          ret = (Buffer)free.removeFirst();
          if(ret!=null){
            map.remove(new Pair(ret.getFileNr(),ret.getPart()));
            ret.setChannel(files.get(fileNr),fileNr);
            map.put(p,ret);
            ret.incNrReading();
            break;
          }
        }
      }
      try{
        free.waitFor();
      }catch(InterruptedException ie){
        logger.error("could not wait",ie);
        ie.printStackTrace();
        System.exit(1);
      }
    }
    ret.readIn(part);
    if(lock){
      if(read) ret.readLock();
      else ret.writeLock();
    }
    return ret;
  }


  public void unpinBuffer(Buffer b, boolean read, boolean lock){
    synchronized(this){
      if(BrilleDefinitions.UNPIN_BUG_DEBUG){
        logger.debug("in unpinbuffer");
        logger.debug("with buffer with "+b.getNrReading()+" reading");
      }
      if(b.decAndIsZeroReading()){
        if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("zero reading");
        if(b.isDirty()){
          dirty.addElementLast(b);
          if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("dirtylist");
        }
        else{
          if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("freelist");
          free.addElementLast(b);
        }
      }
    }
    if(lock){
      if(read) b.readUnlock();
      else b.writeUnlock();
    }
  }


  public void unDirtyAllWithAndDelete(int i) {
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)
      logger.debug("in undirtyallwithanddelete");
    ft.addSkip(i);
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)
      logger.debug("have added skip");
    removeFileWithNr(i);
```

```
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)
      logger.debug("have removed file");
    if(files.get(i)!=null) try{
      files.get(i).close();
    }catch(IOException ioe){
      ioe.printStackTrace();
      logger.error(ioe);
    }
    if(realfiles.get(i)!=null)realfiles.get(i).delete();
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)
      logger.debug("returning from undirty");
  }


  public long totalDiskSize(){
    long ret = 0;
    for(int i=0;i<realfiles.size();i++)
      if(realfiles.get(i)!=null)
        ret+= realfiles.get(i).length();
    return ret;
  }


  public void shutDown(){
    ft.stopRunning();
  }

  /**
   * Method used for debugging, which prints the contents of the list of
   * free buffers.
   *
   * @param prefix - a prefix that will be the first characters on each line
   */
  public void printFreeQueue(String prefix){
    free.printQueue(prefix);
  }

  /**
   * Method used for debugging, which prints the contents of the list of
   * dirty buffers.
   *
   * @param prefix - a prefix that will be the first characters on each line
   */
  public void printDirtyQueue(String prefix){
    dirty.printQueue(prefix);
  }

  /**
   * This method is used by the flushingthread to to pin the first buffer in
   * the dirty queue, to flush it.
   *
   * @return the first buffer in the dirty queue, or null if there are no
   *  buffers in that queue.
   */
  public synchronized Buffer removeFirstFromDirty(){
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)
      logger.debug("should remove first from...");
    Buffer ret = (Buffer)dirty.removeFirst();
    if(ret!=null){
      ret.readLock();
      ret.incNrReading();
    }
    return ret;
  }
}
```

C.3.8 brille.buffering.Pair

```
package brille.buffering;

/**
 * This class is basically the standard Pair, but this one is a pair
 * consisting of one int, the number of a file, and one long, the part of
 * the file. This object is used for hashing in the NewBufferPool.
 *
 * Its hash-function is a XOR of filenumber and an int-version of the part.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class Pair{
  private int fileNum;
  private long part;

  /**
   * Default constructor taking both member variables as arguments.
   *
   * @param fileNum - the number of the file in the bufferpool from which the
   *   buffered part pointed to by the HashMap for this object is found
   * @param part - the part of the file where the buffer pointed to is cached
   *   from
   */
  public Pair(int fileNum,long part){
    this.fileNum=fileNum;
    this.part=part;
  }

  /**
   * Getter for the file number.
   *
   * @return the file number
   */
  public int getFileNum() {
    return fileNum;
  }

  /**
   * Getter for the part.
   *
   * @return the part
   */
  public long getPart() {
    return part;
  }

  public int hashCode(){
    return fileNum^((int)part);
  }

  public boolean equals(Object o){
    if(!(o instanceof Pair)) return false;
    Pair p = (Pair)o;
    return p.getFileNum()==fileNum && p.getPart()==part;
  }
}
```

C.3.9 brille.buffering.Pin

```
package brille.buffering;


/**
 * This class is used in brille.buffering.BufferPool, the old version of
 * the buffer pool to represent a pinned buffer. This class only contains
```

```
 * a reference to the pinned buffer, and the file number of the file the
 * buffer is pinned to, and the part it contains.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class Pin {
  private Buffer buf;
  private long part;
  private int filenr;

  /**
   * Constructor taking all member variables as arguments.
   *
   * @param buf - the pinned buffer
   * @param part - the part of the file this buffer is pinned to
   * @param filenr - the file number of the file the pinned buffer has a
   *  part from
   */
  public Pin(Buffer buf, long part, int filenr){
    this.buf=buf;
    this.part=part;
    this.filenr=filenr;
  }

  /**
   * Method for retrieving the buffer.
   *
   * @return the pinned buffer
   */
  public Buffer getBuffer() {
    return buf;
  }

  /**
   * Getter for the part the buffer is pinned to.
   *
   * @return the part of the file this buffer has cached
   */
  public long getPart() {
    return part;
  }

  /**
   * Getter for the file number in the buffer pool the buffer caches a part
   * from.
   *
   * @return the file number
   */
  public int getFileNr(){
    return filenr;
  }
}
```

C.3.10 brille.buffering.UndirtyThread

```
package brille.buffering;

/**
 * This class is a Thread capable of making sure that all buffers containing
 * parts of a specified file will be set as undirty. This is useful when
 * a file is deleted, because we want to make sure that no buffers will
 * be flushed to a deleted file. The reason why this happens in a thread on
 * its own is that the method called from here is synchronized, and other
 * threads might hold that monitor for a long time, and we do not want to
 * slow down indexing speed waiting for things like that, so the overhead
 * incurred by starting a thread is considered smaller.
 *
```

```
 * @author Truls A. Ã¸Bjrklund
 */
public class UndirtyThread extends Thread{
  private BuffPool bp;
  private int undirty;

  /**
   * Constructor taking all member variables as arguments.
   *
   * @param bp - the bufferpool the file has been added to
   * @param undirty - the file number of the file we want to undirty all
   *   buffers for
   */
  public UndirtyThread(BuffPool bp, int undirty){
    this.bp=bp;
    this.undirty=undirty;
  }

  public void run(){
    bp.unDirtyAllWithAndDelete(undirty);
  }
}
```

C.4 brille.dict

C.4.1 brille.dict.BTreeSearchResultHandle

```
package brille.dict;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.SearchResultHandle;
import brille.btree.BTreeHandle;
import brille.inv.IndexEntry;
import brille.utils.ByteArr;

/**
 * This is the SearchResultHandle for B-trees. It provides an iterator
 * through the list of occurrences of a term searched for.
 *
 * It also contains information about an approximate number of documents
 * containing the term, and an approximate number of total occurrences.
 * The reasons why these numbers are not exact is the way the B-tree index
 * works. When a new occurrence is added, the entry containing information
 * about the number of occurrences is first updated, before the entry is
 * actually added. See the btreeindex-class for more information.
 *
 * All important methods are declared in brille.SearchResultHandle, and are
 * thus commented there as well.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.dict.FullBTreeIndex
 * @see brille.SearchResultHandle
 */
public class BTreeSearchResultHandle extends SearchResultHandle{

  protected final Logger logger = Logger.getLogger(getClass());

  private ByteArr term;
  private BTreeHandle bh;
  private int approxnum;
  private int approxtotocc;

  /**
   * Constructor. This takes in a BTreeHandle, which is an iterator through the
   * entries in the B-tree. The first entry should be one giving info about the
   * number of occurrences. This is stored and the next entries this iterator
   * returns are returned directly from the BTreeHandle.
   *
   * @param bh - the BTreeHandle that represents the hit for this term
   * @param term - the term searched for
   */
  public BTreeSearchResultHandle(BTreeHandle bh,ByteArr term){
    this.bh=bh;
    this.term=term;
    FullBTreeIndexLeafEntry e = (FullBTreeIndexLeafEntry)bh.getEntry();
    if(e==null || e.getTerm().compareTo(term)!=0){
      approxnum=0;
    }
    else if(e.getDocNr()!=-1){
      if(BrilleDefinitions.DEBUG)
        logger.error("could not find approxnum for term "+term.decodeString());
    }
    else{
      int[] inf = e.getLocations();
      approxnum=inf[0];
      approxtotocc=inf[1];
      bh.getNext();
```

```
    }
  }

  public int getNrOfResults(){
    return approxnum;
  }

  public int getNrOcc(){
    return approxtotocc;
  }

  public void next() {
    bh.getNext();
  }

  public IndexEntry getEntry(){
    FullBTreeIndexLeafEntry ent = (FullBTreeIndexLeafEntry)bh.getEntry();
    if(ent ==null || !term.equals(ent.getTerm())) return null;
    return ent.getIndexEntry();
  }

  public void release() {
    bh.release();
  }
}
```

C.4.2 brille.dict.Dictionary

```
package brille.dict;

import brille.BrilleDefinitions;
import brille.SearchResultHandle;
import brille.utils.ByteArr;

/**
 * This abstract class defines the minimum of what a dictionary should
 * provide with respect to methods. All dictionaries are stored in a file
 * and this class constructs the filename. It also defines a search-method
 * returning a SearchResultHandle.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.SearchResultHandle
 */
public abstract class Dictionary {
  protected int nr;
  protected String filename;

  /**
   * Constructor taking the number this dictionaries has as an argument,
   * and which constructs a file name based on this argument.
   *
   * @param nr - the number of the dictionary
   */
  public Dictionary(int nr){
    this.nr=nr;
    filename=BrilleDefinitions.INDEX_PATH+"/dict"+nr+".dat";
  }

  /**
   * Getter for this dictionary's number
   *
   * @return the number of this dict.
```

```
   */
  public int getNr(){
    return nr;
  }

  /**
   * Getter for the name of the file this dictionary is stored to.
   *
   * @return the filename for this dictionary
   */
  public String getFilename(){
    return filename;
  }

  /**
   * This is a declaration of the search-method that should be available in
   * all dictionaries. It should return a SearchResultHandle. A
   * SearchResultHandle is an iterator capable of iterating through all
   * occurrences in the term's occurrenc list.
   *
   * @param term - the term searched for
   * @return - an iterator (SearchResultHandle) over all documents that
   *   contains the term
   */
  public abstract SearchResultHandle search(ByteArr term);
}
```

C.4.3 brille.dict.DictIteratorHeap

```
package brille.dict;

import java.util.ArrayList;

/**
 * This class is a heap of a set of SortedListDictIterators. This implies
 * that when the smallest entry is extracted from the heap, it might be so
 * that there are more entries from the dictionary, and the dictiterator
 * should not be removed from the heap, but rather moved to the next entry.
 * That is the reason why the java.util.PriorityQueue-class could not be used.
 *
 * This classed is used when merging several dictionaries into a new one. This
 * is done both in the construction of the index in the RemergeIndexMaster (and
 * in updates), and likewise for the HierarchicIndexMaster.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class DictIteratorHeap {
  private ArrayList<SortedListDictIterator> l;

  /**
   * Constructor taking a list of the SortedListDictIterators for the
   * dictionaries to merge.
   *
   * @param l - the list of dictiterators
   */
  public DictIteratorHeap(ArrayList<SortedListDictIterator> l){
    this.l=l;
    buildHeap();
  }

  /**
   * Method used for constructing the heap. As usual, this consists of several
   * to minHeapify().
   *
   */
  private void buildHeap(){
    for(int i=(l.size()/2)-1;i>=0;i--) minHeapify(i);
```

```
  }

  /**
   * The implementation of minHeapify for this heap. It starts at the given
   * index, and minheapifies it with its two children found at indexes
   * 2*ind+1 and 2*ind+2.
   *
   * @param ind - the index to minheapify
   */
  private void minHeapify(int ind){
    int le = ind*2+1;
    int ri = ind*2+2;
    if(le>=l.size())return;
    //sweet one-liner..
    int min= ri>=l.size()?le:
      l.get(le).getEntry().getTerm().
      compareTo(l.get(ri).getEntry().getTerm())<0?le:ri;
    if(l.get(ind).getEntry().getTerm().
        compareTo(l.get(min).getEntry().getTerm())>0){
      SortedListDictIterator tmp = l.get(ind);
      l.set(ind,l.get(min));
      l.set(min,tmp);
      minHeapify(min);
    }
  }

  /**
   * Method for retrieving the next entry from this iterator. Note that once
   * this entry is retrieved, it will no longer be available from this
   * iterator.
   *
   * @return the next MergeSortedListDictEntry from this iterator
   */
  public MergeSortedListDictEntry getEntry(){
    if(l.size()==0) return null;
    MergeSortedListDictEntry ret = l.get(0).getEntry();

    l.get(0).next();
    if(l.get(0).getEntry()==null){
      l.get(0).release();
      if(l.size()>1)l.set(0,l.get(l.size()-1));
      l.remove(l.size()-1);
    }
    if(l.size()>0)minHeapify(0);
    return ret;
  }

  /**
   * Methof for testing whether this heap is empty, that means that all
   * dictionaries have returned all of their terms.
   *
   * @return whether or not the heap is empty
   */
  public boolean isEmpty(){
    return l.size()==0;
  }
}
```

C.4.4 brille.dict.ExtendableDocTermEntry

```
package brille.dict;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.utils.ByteArrayList;
import brille.utils.IntArrayList;
```

```
/**
 * This class is used to represent the occurrences of a term in a document
 * beeing parsed.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class ExtendableDocTermEntry {

  protected final Logger logger = Logger.getLogger(getClass());

  private IntArrayList locs;
  private ByteArrayList descr;

  /**
   * Constructor creating the list of locations of the term in the document
   * and, if the nr of bits used from the description is larger than zero,
   * the descriptions-list is also initialised
   *
   */
  public ExtendableDocTermEntry(){
    locs = new IntArrayList();
    descr = BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0?new ByteArrayList():null;
  }

  /**
   * getter
   *
   * @return descriptions of occurrences
   */
  public ByteArrayList getDescr() {
    return descr;
  }

  /**
   * getter
   *
   * @return the occerrences of the given term
   */
  public IntArrayList getLocs() {
    return locs;
  }

  /**
   * Add a location of term in the document.
   *
   * @param termnr - the token number of the term in the document
   */
  public void addLoc(int termnr){
    locs.add(termnr);
  }

  /**
   * add description for the most recently added location.
   *
   * @param description
   */
  public void addDescr(byte description){
    descr.add(description);
  }
}
```

C.4.5 brille.dict.FullBTreeIndex

```
package brille.dict;

import org.apache.log4j.Logger;
```

```
import brille.BrilleDefinitions;
import brille.SearchResultHandle;
import brille.btree.BTree;
import brille.btree.BTreeHandle;
import brille.buffering.BuffPool;
import brille.inv.IndexEntry;
import brille.utils.ByteArr;

/**
 * This class represents the B-tree index. It is part of the
 * brille.dict-package allthough it is more than a dictionary. This method
 * actually is a combined dictionary and inverted file. All of it is stored
 * in a single B-tree.
 *
 * When inserting an entry into this index, one is inserting the occurrences
 * of a term in a specific document. When this happens, one first increment
 * the number of occurrences of the term by accessing the B-tree once. Then,
 * the actual entry in the document is added.
 *
 * Because the nodes in the B-tree has limited size. This implies that it
 * cannot hold entries of arbitrary size. In brille, we set two restrictions
 * to cope with this. First of all, a term that is inserted can not be longer
 * than a maximum number of bytes, defined in brille.BrilleDefinitions. To
 * avoid splitting nodes in the B-tree into more than two, there is also a
 * restriction on the maximum amount of locations for a term with a specified
 * length. This is calculated in this class.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FullBTreeIndex extends Dictionary{

  protected final Logger logger = Logger.getLogger(getClass());

  private BTree btree;

  /**
   * Constructor for a FullBTreeIndex, giving the number that will be the
   * suffix in the name of the file for this index, and the buffer pool
   * from which buffers are pinned.
   *
   * @param nr - nr of the dictionary/index
   * @param bp - the buffer pool
   */
  public FullBTreeIndex(int nr, BuffPool bp){
    super(nr);
    if(BrilleDefinitions.DEBUG)logger.info("creating fullbtreeindex");
    btree = new BTree(filename,bp,
        BrilleDefinitions.SLEEP_TIME_FOR_INDEX_UPDATE_THREAD,
        new FullBTreeIndexInternalEntryFactory(),
        new FullBTreeIndexLeafEntryFactory());
    if(BrilleDefinitions.DEBUG)logger.info("btree created");
  }

  /**
   * Getter which is only here for gathering of statistics.
   *
   * @return the BTree with the index
   */
  public BTree getBTree(){
    return btree;
  }

  /**
   * Method for inserting a new IndexEntry for the specified term.
   *
   * @param term - the term to insert an entry for
   * @param ie - the entry to insert
   * @return - the approximate number of occurrences of the inserted term,
```

```
 *  including the one just inserted.
 */
public int insert(ByteArr term, IndexEntry ie) {
  if(term.getL()>getMaxBytesInTerm()){
    return 0;
  }
  if(BrilleDefinitions.DEBUG)
    logger.info("inserting term "+term.decodeString());
  int[] locs = new int[2];
  locs[0]=1;
  locs[1]=ie.getLocations().length;
  if(BrilleDefinitions.DEBUG){
    logger.info("Printing incorinsvalues:");
    logger.info(locs[0]+" "+locs[1]);
  }
  FullBTreeIndexLeafEntry incEnt =
    new FullBTreeIndexLeafEntry(term,-1,locs,
        BrilleDefinitions.NR_OF_BITS_FROM_DESCR==0?null:new byte[2]);
  if(BrilleDefinitions.DEBUG)logger.info("adding/incing inc-entry:");
  if(BrilleDefinitions.DEBUG)logger.info(incEnt.toString());
  FullBTreeIndexLeafEntry rankvals =
    (FullBTreeIndexLeafEntry)btree.incOrInsert(incEnt);
  if(BrilleDefinitions.DEBUG)
    logger.info("should add new entry "+term.decodeString()+
        " ie "+ie.toString());
  btree.insertEntry(new FullBTreeIndexLeafEntry(term,ie));
  if(BrilleDefinitions.DEBUG)
    logger.info("done inserting term "+term.decodeString());
  return rankvals.getLocations()[0];
}


public SearchResultHandle search(ByteArr term) {
  FullBTreeIndexLeafEntry k = new FullBTreeIndexLeafEntry(term,-1,null,null);
  BTreeHandle bh = btree.searchForEntry(k);
  return new BTreeSearchResultHandle(bh,term);
}

/**
 * Method for deleting the specified indexentry for the given term.
 *
 * @param term - the term to delete an entry for
 * @param ie - the entry to delete
 */
public void delete(ByteArr term, IndexEntry ie){
  int[] locs = new int[2];
  locs[0]=1;
  locs[1]=ie.getLocations().length;
  btree.delete(new FullBTreeIndexLeafEntry(term,ie));
  btree.decOrDelete(new FullBTreeIndexLeafEntry(term,-1,locs,new byte[2]));
}

/**
 * Method for retrieving a BTreeHandle capable of iterating through the
 * whole index. This iterator is used when the FixRankingThread iterates
 * through to update all ranking values. This method actually
 * iterates through the lowest level in the B-tree.
 *
 * @return a BTreeHandle capable of iterating through the B-Tree
 */
public BTreeHandle getIndexIterator(){
  return btree.getSmallestEntry();
}

/**
 * Method for finding the disk space used by this index
 *
 * @return the disk space used in bytes
 */
```

```
  public long diskSize(){
    return btree.diskSize();
  }

  /**
   * Method called when the application shuts down, so that this index
   * can shut down the thread updating the B-tree.
   *
   */
  public void shutDown(){
    btree.shutDown();
  }

  /**
   * Retrieving the maximum length of a term in the B-tree in bytes.
   *
   * @return maxlength of a term to insert
   */
  public int getMaxBytesInTerm(){
    return BrilleDefinitions.MAX_KEY_LENGTH-6;
  }

  /**
   * Retrieving the maximum number of locations that will be stored for a term
   * with the given length in bytes.
   *
   * @param l - the length of the term in bytes
   * @return - the maximum number of locations that will be stored for the
   * given term
   */
  public int getMaxNumEntriesForTermWithL(int l){
    return (BrilleDefinitions.MAX_ENTRY_LENGTH-(8 + l))
      /BrilleDefinitions.INT_SIZE;
  }
}
```

C.4.6 brille.dict.FullBTreeIndexInternalEntry

```
package brille.dict;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;

import org.apache.log4j.Logger;

import brille.btree.Entry;
import brille.btree.InternalEntry;
import brille.buffering.Buffer;
import brille.utils.ByteArr;

/**
 * This class represents the internal entries in a FullBTreeIndex.
 * the primary key in such a tree is term and document number, and these
 * entries thus contain those two values, and a pointer to a child node in the
 * B-tree.
 *
 * Most methods here are declared in brille.btree.InternalEntry, and are thus
 * commented there.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.btree.InternalEntry
 */
public class FullBTreeIndexInternalEntry extends InternalEntry{

  protected final Logger logger = Logger.getLogger(getClass());

  private ByteArr term;
```

```
private int docNr;

/**
 * The constructor used to make new kinds of this Entry. It takes all
 * values stored here as arguments.
 *
 * @param pointer
 * @param term
 * @param docNr
 */
public FullBTreeIndexInternalEntry(int pointer, ByteArr term, int docNr){
  super(pointer);
  this.term=term;
  this.docNr=docNr;
}

/**
 * Constructor used when we want a new FullBTreeIndexInternalEntry which
 * is capable of reading in entries from the B-tree. This is used in the
 * B-tree to avoid to many object creations, and thus some garbage
 * collection as well.
 *
 */
public FullBTreeIndexInternalEntry(){
  this(-1,new ByteArr(-1),-1);
}


public boolean keyIsMax() {
  return term.getL()==-1;
}

/**
 * Getting the term in this entry.
 *
 * @return the term
 */
public ByteArr getTerm(){
  return term;
}

/**
 * Getting the document number, the other part of the primary key here.
 *
 * @return the document number of this entry.
 */
public int getDocNr(){
  return docNr;
}


public void setKeyToMax() {
  term=new ByteArr(-1);
}


public int write(Buffer buf, int pos) {
  int currPos = pos+writeKey(buf,pos)+4;
  buf.putInt(pointer,currPos-4);
  return currPos-pos;
}


public int writeKey(Buffer buf, int pos) {
  buf.putShort((short)term.getL(),pos);
  if(term.getL()==-1) return 2;
  int currPos=pos+2;
  for(int i=0;i<term.getL();i++,currPos++)
    buf.putByte(term.getBuffer().get(i),currPos);
```

```
  buf.putInt(docNr,currPos);
  return currPos+4-pos;
}


public int readIn(Buffer buf, int pos) {
  int currPos = pos+readInKey(buf,pos);
  pointer=buf.getInt(currPos);
  return currPos+4-pos;
}


public int readInKey(Buffer buf, int pos) {
  int currPos=pos;
  int l = buf.getShort(currPos);
  if(l==-1){
    term = new ByteArr(-1);
    return 2;
  }
  currPos+=2;
  ByteBuffer bf = ByteBuffer.allocate(l);
  bf.order(ByteOrder.nativeOrder());
  for(int i=0;i<l;i++,currPos++) bf.put(i,buf.getByte(currPos));
  term = new ByteArr(bf);
  docNr=buf.getInt(currPos);
  return currPos+4-pos;
}


public int keyLength() {
  if(keyIsMax()) return 2;
  return 6+term.getL();
}


public int length() {
  return keyLength()+4;
}


public int compareTo(Entry e) {
  if(e instanceof FullBTreeIndexLeafEntry){
    FullBTreeIndexLeafEntry b = (FullBTreeIndexLeafEntry)e;
    if(b.getTerm().equals(term)){
      if(b.keyIsMax()) return 0;
      return getDocNr()-b.getDocNr();
    }
    return term.compareTo(b.getTerm());
  }
  else if(e instanceof FullBTreeIndexInternalEntry){
    FullBTreeIndexInternalEntry b = (FullBTreeIndexInternalEntry)e;
    if(b.getTerm().equals(term)){
      if(b.keyIsMax()) return 0;
      return getDocNr()-b.getDocNr();
    }
    return term.compareTo(b.getTerm());
  }
  else
    throw new
      IllegalArgumentException("not a valid comparison: "+e.toString());
}


public String toString() {
  if(keyIsMax()) return "max";
  return term.decodeString()+", "+getDocNr()+"\n pointer: "+pointer;
}
```

```
  public boolean isKeyCompatible(Entry en) {
    return
      (en instanceof FullBTreeIndexInternalEntry)||
      (en instanceof FullBTreeIndexLeafEntry);
  }


  public String getKeyAsString() {
    if(keyIsMax()) return "max";
    return term.decodeString()+", "+docNr;
  }


  public void incEntry(Entry en) {
    throw new IllegalArgumentException("not allowed to inc");
  }


  public boolean decEntry(Entry en) {
    throw new IllegalArgumentException("not allowed to dec");
  }

  /**
   * Method for setting the key similar to the key of another entry.
   *
   * @param entry - the entry with the key we want to set this one to
   */
  public void setKey(FullBTreeIndexLeafEntry entry) {
    if(entry.getTerm().getL()==-1) term = new ByteArr(-1);
    else{
      term=entry.getTerm();
      docNr=entry.getDocNr();
    }
  }
}
```

C.4.7 brille.dict.FullBTreeIndexInternalEntryFactory

```
package brille.dict;

import brille.btree.Entry;
import brille.btree.InternalEntry;
import brille.btree.InternalEntryFactory;

/**
 * This is the factory for FullBTreeIndexInternalEntries, and needed
 * to insert these into the general B-tree. All methods are declared in
 * brille.btree.InternalEntryFactory, and are thus commented there.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FullBTreeIndexInternalEntryFactory extends InternalEntryFactory{

  public InternalEntry make(Entry maxKey, int pointer) {
    FullBTreeIndexInternalEntry mk =(FullBTreeIndexInternalEntry)maxKey;
    return new FullBTreeIndexInternalEntry(pointer,mk.getTerm(),mk.getDocNr());
  }

  public Entry make() {
    return new FullBTreeIndexInternalEntry();
  }

  public InternalEntry makeEntryWithSameKey(Entry newMax) {
    if(newMax instanceof FullBTreeIndexInternalEntry) return
      (FullBTreeIndexInternalEntry)newMax;
    else if(newMax instanceof FullBTreeIndexLeafEntry){
      FullBTreeIndexInternalEntry ret = new FullBTreeIndexInternalEntry();
```

```
      ret.setKey((FullBTreeIndexLeafEntry)newMax);
      return ret;
    }
    return null;
  }


}
```

C.4.8 brille.dict.FullBTreeIndexLeafEntry

```
package brille.dict;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.btree.Entry;
import brille.buffering.Buffer;
import brille.inv.IndexEntry;
import brille.utils.ByteArr;

/**
 * This class represents the leaf entries in a FullBTreeIndex.
 * It contains the primary key in that B-tree, the term and document number,
 * and a listing of occurrences possibly with descriptions, for all occurrences
 * of the term within the given document.
 *
 * Most methods here are declared in brille.btree.Entry, and are thus commented
 * there.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FullBTreeIndexLeafEntry extends Entry{

  protected final Logger logger = Logger.getLogger(getClass());

  private ByteArr term;
  private IndexEntry en;
  private int maxEntriesToWrite;

  /**
   * Default constructor. Used by the make method in the factory.
   * It is used to instantiate an object capable of reading in values from a
   * node, to avoid some unnessececary object creation, and thus garbage
   * collection.
   *
   */
  public FullBTreeIndexLeafEntry(){
    this(null,-2,null,null);
  }

  /**
   * Constructor taking all values stored in this document as arguments.
   *
   * @param term - the term represented
   * @param docNr - the document number
   * @param locations - the occurrences of the term in the given document
   * @param descr - descriptions of the occurrences within the document
   */
  public FullBTreeIndexLeafEntry(ByteArr term, int docNr,
      int[] locations, byte[] descr){
    this.term=term;
    en = new IndexEntry(docNr, locations, descr);
    if(term!=null) calculateMaxEntriesToWrite();
```

```
}

/**
 * Constructor taking the term and the indexentry describing all other
 * values.
 *
 * @param term - the term
 * @param en - the index entry for this term
 */
public FullBTreeIndexLeafEntry(ByteArr term, IndexEntry en){
  this.term =term;
  this.en=en;
  if(term!=null) calculateMaxEntriesToWrite();
}

/**
 * This method is important in this class. It is used each time the term
 * in this object is changed, to calculate the maximum number of occurences
 * within the indexentry than can be written to disk.
 *
 */
private void calculateMaxEntriesToWrite(){
  if(term.getL()==-1)
    maxEntriesToWrite = (BrilleDefinitions.MAX_ENTRY_LENGTH-8)
      /BrilleDefinitions.INT_SIZE;
  else
    maxEntriesToWrite = (BrilleDefinitions.MAX_ENTRY_LENGTH-(8+term.getL()))
      /BrilleDefinitions.INT_SIZE;
}

/**
 * Getting the term represented in this entry. This is the first part of the
 * primary key in this tree.
 *
 * @return the term
 */
public ByteArr getTerm(){
  return term;
}

/**
 * Getting the document number that is a part of the primary key here.
 *
 * @return the document number
 */
public int getDocNr(){
  return en.getDocNr();
}

/**
 * Getting the occurrences of the given term within the given document
 *
 * @return the locations
 */
public int[] getLocations(){
  return en.getLocations();
}

/**
 * Getting the list of descriptions of the occurrences.
 *
 * @return descriptions
 */
public byte[] getDescr(){
  return en.getDescriptions();
}

/**
 * Getting the index-entry inherently represented by this entry.
```

```
 *
 * @return the index entry
 */
public IndexEntry getIndexEntry(){
  return en;
}


public boolean keyIsMax() {
  return term.getL()==-1;
}


public void setKeyToMax() {
  term=new ByteArr(-1);
}


public int write(Buffer buf, int pos) {
  int currPos = pos+writeKey(buf,pos);
  int l=0;
  if(en.getLocations()!=null)l=en.getLocations().length;
  if(BrilleDefinitions.DEBUG) logger.debug("have written key, and l is "+l);
  buf.putShort((short)l,currPos);
  if(en.getLocations()!=null)
    l=en.getLocations().length>maxEntriesToWrite?maxEntriesToWrite:
      en.getLocations().length;
  currPos+=BrilleDefinitions.SHORT_SIZE;
  for(int i=0;i<l;i++,currPos+=BrilleDefinitions.INT_SIZE){
    buf.putInt(en.getLocations()[i],currPos);
    if(BrilleDefinitions.DEBUG)
      logger.debug("writing out location "+en.getLocations()[i]);
  }
  if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
    for(int i=0;i<l;i++,currPos++)
      buf.putByte(en.getDescriptions()[i],currPos);
  }
  return currPos-pos;
}


public int writeKey(Buffer buf, int pos) {
  int currPos =pos;
  int l = term.getL();
  buf.putShort((short)l,currPos);
  currPos+=BrilleDefinitions.SHORT_SIZE;
  if(l!=-1) for(int i=0;i<l;i++,currPos++)
    buf.putByte(term.getBuffer().get(i),currPos);
  else return BrilleDefinitions.SHORT_SIZE;
  buf.putInt(en.getDocNr(),currPos);
  currPos+=BrilleDefinitions.INT_SIZE;
  return currPos-pos;
}


public int readIn(Buffer buf, int pos) {
  int currPos = pos+readInKey(buf,pos);
  calculateMaxEntriesToWrite();
  if(keyIsMax()) return currPos-pos;
  int l=buf.getShort(currPos);
  if(BrilleDefinitions.DEBUG) logger.debug("reading in with l="+l);
  currPos+=BrilleDefinitions.SHORT_SIZE;
  int[] locations;
  byte[] descr=null;
  if(l==0) locations=null;
  else{
    locations=new int[l];
    if(l>maxEntriesToWrite) l=maxEntriesToWrite;
    for(int i=0;i<l;i++,currPos+=BrilleDefinitions.INT_SIZE)
```

```
        locations[i]=buf.getInt(currPos);
      if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
        descr=new byte[l];
        for(int i=0;i<l;i++,currPos++) descr[i]=buf.getByte(currPos);
      }
      else{
        descr=null;
      }
    }
    en.setLocations(locations);
    en.setDescr(descr);
    return currPos-pos;
  }


  public int readInKey(Buffer buf, int pos) {
    int currPos=pos;
    int l = buf.getShort(currPos);
    currPos+=BrilleDefinitions.SHORT_SIZE;
    if(l==-1){
      term = new ByteArr(-1);
      return 2;
    }
    else{
      ByteBuffer bf = ByteBuffer.allocate(l);
      bf.order(ByteOrder.nativeOrder());
      for(int i=0;i<l;i++,currPos++) bf.put(i,buf.getByte(currPos));
      term = new ByteArr(bf);
    }
    en = new IndexEntry(buf.getInt(currPos),null,null);
    currPos+=BrilleDefinitions.INT_SIZE;
    return currPos-pos;
  }


  public int keyLength() {
    if(keyIsMax()) return 2;
    return 6+term.getL();
  }


  public int length() {
    int ret = keyLength()+2;
    if(en.getLocations()!=null){
      int length =en.getLocations().length;
      ret+=(Math.min(length,maxEntriesToWrite)*BrilleDefinitions.INT_SIZE);
    }
    if(en.getDescriptions()!=null){
      int length =en.getDescriptions().length;
      ret+=(Math.min(length,maxEntriesToWrite));
    }
    return ret;
  }


  public int compareTo(Entry e) {
    if(e instanceof FullBTreeIndexLeafEntry){
      FullBTreeIndexLeafEntry b = (FullBTreeIndexLeafEntry)e;
      if(b.getTerm().equals(term)){
        if(b.keyIsMax()) return 0;
        return en.getDocNr()-b.getDocNr();
      }
      if(BrilleDefinitions.DEBUG) logger.debug("Terms are not equal");
      return term.compareTo(b.getTerm());
    }
    else if(e instanceof FullBTreeIndexInternalEntry){
      FullBTreeIndexInternalEntry b = (FullBTreeIndexInternalEntry)e;
      if(b.getTerm().equals(term)){
        if(b.keyIsMax()) return 0;
```

```
        return en.getDocNr()-b.getDocNr();
      }
      return term.compareTo(b.getTerm());
    }
    else throw
      new IllegalArgumentException("not a valid comparison: "+e.toString());
  }


  public String toString() {
    if(keyIsMax()) return "max";
    String ret = term.decodeString()+", "+en.getDocNr()+"\n";
    if(en.getLocations()!=null){
      for(int i=0;i<en.getLocations().length;i++){
        ret+=(en.getLocations()[i]+" ");
      }
      ret+=("\n");
      if(en.getDescriptions()!=null){
        for(int i=0;i<en.getDescriptions().length;i++){
          ret+=(en.getDescriptions()[i]+" ");
        }
      }
      ret+=("\n");
    }
    return ret;
  }

  public boolean isKeyCompatible(Entry en) {
    return (en instanceof FullBTreeIndexLeafEntry)
      ||(en instanceof FullBTreeIndexInternalEntry);
  }

  public String getKeyAsString() {
    if(keyIsMax()) return "max";
    return term.decodeString()+", "+en.getDocNr();
  }

  public void incEntry(Entry ent) {
    if(!(ent instanceof FullBTreeIndexLeafEntry))
      throw new
        IllegalArgumentException("not legal to incentry with another class");
    FullBTreeIndexLeafEntry b = (FullBTreeIndexLeafEntry)ent;
    if((!b.getTerm().equals(term)) || b.getDocNr()!=-1 || en.getDocNr()!=-1)
      throw new IllegalArgumentException("different entries");
    for(int i=0;i<b.getLocations().length;i++)
      en.getLocations()[i]+=b.getLocations()[i];
  }


  public boolean decEntry(Entry ent) {
    if(!(ent instanceof FullBTreeIndexLeafEntry))
      throw new IllegalArgumentException("not legal to deccentry " +
          "with another class");
    FullBTreeIndexLeafEntry b = (FullBTreeIndexLeafEntry)ent;
    if((!b.getTerm().equals(term)) || b.getDocNr()!=-1 || en.getDocNr()!=-1)
      throw new IllegalArgumentException("different entries");
    for(int i=0;i<b.getLocations().length;i++)
      en.getLocations()[i]-=b.getLocations()[i];
    return en.getLocations()[0]==0;
  }
}
```

C.4.9 brille.dict.FullBTreeIndexLeafEntryFactory

```
package brille.dict;

import brille.btree.Entry;
```

```java
import brille.btree.EntryFactory;

/**
 * An entry factory for FullBTreeIndexLeafEntries. This is
 * made to enable insertions of these objects into a general B-tree. The
 * method is declared in brille.btree.EntryFactory, and is thus commented
 * there.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FullBTreeIndexLeafEntryFactory extends EntryFactory{

  public Entry make() {
    return new FullBTreeIndexLeafEntry();
  }
}
```

C.4.10 brille.dict.InMemPartialIndex

```java
package brille.dict;

import brille.BrilleDefinitions;
import brille.buffering.Buffer;
import brille.buffering.BuffPool;
import brille.buffering.UndirtyThread;
import brille.utils.ByteArr;
import brille.utils.ByteArrayList;
import brille.utils.IntArrayList;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.channels.FileChannel;

import org.apache.log4j.Logger;

/**
 * This class is the index accumulated in memory while index
 * construction. It is used in both the Remerge index and the
 * hierarchic one. It is based on the nio-buffers used
 * throughout Brille.
 *
 * This class is given a maximum number of buffers that may be
 * pinned simultaneously while building the index. It reserves two
 * of these for the dictionary and inverted file when it should flush,
 * and uses pins the rest as documents are added.
 *
 * The index is built using the following overall method: When a document is
 * added, we mark where the representation of this document starts in the
 * buffers. Then, all terms for this document are written lexicographically.
 * with information about all occurrences of this term in the document.
 *
 * When a non-empty document is added, an entry into a heap is also inserted.
 * In order to avoid using unbounded memory, this heap is build from the
 * end of the buffers, so that the heap and the representation of documents
 * will grow towards each other. When there is not enough room for the next
 * document to be added, this index will be  flushed to disk.
 *
 * In order to have a regular inverted file on disk, the occurrences in all
 * documents are sorted. This means that we use the heap to extract the next
 * term lexicographically, and the smallest doc number that has this term.
 * When the next term differs from the previous one, an entry is added
```

```
 * in the dictionary, and the complete occurrence list is added to the inverted
 * file.
 *
 * Because initial testing revealed that parsing the next term for each
 * document several times was inefficient, the class now uses some more memory
 * during flushing. All next terms for each document is now cached, causing
 * a significant speed-up by comprosing some memory usage.
 *
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class InMemPartialIndex {

  protected final Logger logger = Logger.getLogger(getClass());

  private static final int HEAP_ENTRY_SIZE=12;
  private final int LARGEST_DATATYPE_USED_IN_INDEX=BrilleDefinitions.INT_SIZE;

  private BuffPool bp;
  private Buffer[] buffers;
  private int numDocs;
  private int invnr;
  private int dictnr;
  private int nullnr;
  private int currBufferInd;
  private int currBufferPart;
  private int dictpos;
  private int dictpart;
  private int numterms;
  private Buffer currDictBuffer;
  private ArrayList<ByteArr> cachedNextTerms;
  //variables for baseline run
  private long startTime;
  private long timeSpentAddingDocs;
  private long timeSpentFlushing;

  /**
   * The constructor for this class, telling which buffer pool to use buffers
   * from, the maximum allowed number of buffers to use, the path to the
   * inverted file it should flush to, and the path to the dictionary-file it
   * should flush to.
   *
   * @param bp - the buffer pool to pin buffers from
   * @param maxNrOfBuffers - the maximum number of buffers the class may pin
   *  simultaneously
   * @param invpath - path to the inverted file it should flush to
   * @param dictpath - path to the dictionary file it should flush to
   */
  public InMemPartialIndex(BuffPool bp, int maxNrOfBuffers,
      String invpath, String dictpath){
    this.bp=bp;
    numDocs=0;
    if(BrilleDefinitions.BASELINERUN)
      timeSpentAddingDocs=0;
    File chf =null;
    File dictf = null;
    FileChannel ch = null;
    FileChannel dictch = null;
    try{
      if(BrilleDefinitions.DEBUG)
        logger.debug("trying to create file with path "+invpath.toString());
      chf = new File(invpath);
      ch= new RandomAccessFile(chf,"rw").getChannel();
      if(BrilleDefinitions.DEBUG)
        logger.debug("trying to create file with path "+dictpath.toString());
      dictf = new File(dictpath);
      dictch = new RandomAccessFile(dictf,"rw").getChannel();
    }catch(FileNotFoundException fnfe){
      logger.error("problems with file for InMemPartialIndex",fnfe);
```

```
  }
  invnr=bp.addFile(ch,chf);
  dictnr = bp.addFile(dictch,dictf);
  nullnr=bp.addFile(null,null);
  if(BrilleDefinitions.DEBUG)logger.info("dictnr is "+dictnr);
  if(BrilleDefinitions.DEBUG)logger.info("invtnr is "+invnr);
  buffers = new Buffer[maxNrOfBuffers-2];
  currBufferPart=0;
  currBufferInd=0;
  buffers[currBufferPart]=bp.pinBuffer(currBufferPart,nullnr,false,false);
}

/**
 * Getter
 *
 * @return the number the dictionary-file has in the buffer pool
 */
public int getDictNr(){
  return dictnr;
}

/**
 * Getter
 *
 * @return the number the inverted file has in the buffer pool
 */
public int getInvNr(){
  return invnr;
}


/**
 * Private method used to add a new term to the dictionary. It will
 * simultaneously add the number of occurrences of the last entry in the
 * dictionary, as this is only clear when all occurrences for this term
 * have been added to the inverted file.
 *
 * @param currTerm - the last term added
 * @param newTerm - the new term to add
 * @param currcnt - the number of documents containing currTerm
 * @param currnumoccs - the total number of occurrences of currTerm
 * @param invpart - the current part of the inverted file
 * @param invpos - the current position within the current part of the
 *  inverted file
 */
private void writeToDict(ByteArr currTerm, ByteArr newTerm, int currcnt,
    int currnumoccs,int invpart, int invpos){
  int le = newTerm.getL();

  if(currTerm.getL()!=-1){
    currDictBuffer.putInt(currcnt,dictpos);
    dictpos+=BrilleDefinitions.INT_SIZE;
    currDictBuffer.putInt(currnumoccs,dictpos);
    dictpos+=BrilleDefinitions.INT_SIZE;
  }

  if(newTerm.getL()==-1){
    if(numterms!=0)currDictBuffer.putInt(dictpos,
        BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE*numterms);
    currDictBuffer.putInt(dictpos-1,BrilleDefinitions.BUFFER_SIZE-
        BrilleDefinitions.INT_SIZE*(numterms+1));
    currDictBuffer.putInt(numterms+1,0);
    return;
  }

  int needed = 24+(le>0?le:0);
  int rest = BrilleDefinitions.BUFFER_SIZE
  -(numterms*BrilleDefinitions.INT_SIZE + dictpos);
  if(needed>rest){
```

```
    if(BrilleDefinitions.DEBUG) logger.info("need new dictBuffer");
    currDictBuffer.putInt(numterms,0);
    if(BrilleDefinitions.DEBUG) logger.info("Setting numterms to "
        +numterms+" for dictpart "+dictpart);
    currDictBuffer.putInt(dictpos,BrilleDefinitions.BUFFER_SIZE
        -BrilleDefinitions.INT_SIZE*numterms);
    bp.unpinBuffer(currDictBuffer,false,false);
    dictpos=BrilleDefinitions.INT_SIZE;
    dictpart++;
    currDictBuffer = bp.pinBuffer(dictpart,dictnr,false,false);
    numterms=0;
  }

  if(numterms!=0) currDictBuffer.putInt(dictpos,BrilleDefinitions.BUFFER_SIZE
      -BrilleDefinitions.INT_SIZE*(numterms));
  numterms++;
  if(le>0) for(int i=0;i<le;i++,dictpos++)
    currDictBuffer.putByte(newTerm.getBuffer().get(i),dictpos);

  long pos = (long)((long)invpart)*((long)BrilleDefinitions.BUFFER_SIZE)
  +(long)invpos;
  currDictBuffer.putLong(pos,dictpos);
  dictpos+=BrilleDefinitions.LONG_SIZE;
}

/**
 * This is the method that caches the next term for a given document in the
 * heap in the list of cached terms. It incs the next read-position in the
 * heap so that it is tracked whether we have reached the end of this
 * document.
 *
 * @param i - the index in the heap that should have its next term cached
 * @return - the ByteArr representing the next term for the given document
 */
private ByteArr readByteArrForDocAndInc(int i){
  int firstind = i*HEAP_ENTRY_SIZE+BrilleDefinitions.INT_SIZE;
  int posbuf = buffers.length-1-firstind/BrilleDefinitions.BUFFER_SIZE;
  int posind=BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE
  -firstind%BrilleDefinitions.BUFFER_SIZE;

  int pos = buffers[posbuf].getInt(posind);

  int readbuf = pos/BrilleDefinitions.BUFFER_SIZE;
  int readind = pos - readbuf*BrilleDefinitions.BUFFER_SIZE;

  if(BrilleDefinitions.BUFFER_SIZE-readind<BrilleDefinitions.SHORT_SIZE){
    pos+=(BrilleDefinitions.BUFFER_SIZE-readind); readind=0;readbuf++;
  }

  int l = buffers[readbuf].getShort(readind);
  pos+=BrilleDefinitions.SHORT_SIZE;
  readind+=BrilleDefinitions.SHORT_SIZE;

  if(l==-1){
    buffers[posbuf].putInt(pos,posind);
    return new ByteArr(-1);
  }
  ByteBuffer b = ByteBuffer.allocate(l);
  b.order(ByteOrder.nativeOrder());
  for(int j=0;j<l;j++,pos++,readind++){
    if(BrilleDefinitions.BUFFER_SIZE==readind){readind=0;readbuf++;}
    b.put(j,buffers[readbuf].getByte(readind));
  }
  buffers[posbuf].putInt(pos,posind);
  ByteArr ret = new ByteArr(b);
  return ret;
}

/**
```

```
 * This method is called when this partial index should be flushed to disk.
 * As explained above, it merges all documents in a multi-way merge performed
 * by keeping all documents in a heap at the end of the array of buffers.
 * For each step, the heap needs to be minheapified and so on.
 *
 * @return the number of blocks used in the inverted file
 */
public int flush(){
  if(BrilleDefinitions.BASELINERUN)
    startTime=System.nanoTime();
  if(BrilleDefinitions.DEBUG)
    logger.debug("In flush. Should output all documents");
  if(BrilleDefinitions.DEBUG)
    for(int j=0;j<numDocs;j++) logAllTermsForDoc(j);
  if(BrilleDefinitions.DEBUG) logger.debug("Done outputting all");

  cachedNextTerms = new ArrayList<ByteArr>();
  for(int i=0;i<numDocs;i++) cachedNextTerms.add(readByteArrForDocAndInc(i));

  if(BrilleDefinitions.DEBUG) logger.debug("outputting all cached terms");
  if(BrilleDefinitions.DEBUG) for(int i=0;i<numDocs;i++)
    logger.debug(cachedNextTerms.get(i).decodeString());

  if(BrilleDefinitions.DEBUG) logger.debug("in flush, should build heap");
  buildHeap();
  if(BrilleDefinitions.DEBUG) logger.debug("Done building heap");
  if(BrilleDefinitions.DEBUG) logHeap();

  int invpos=0;
  int invpart=0;
  //make room for numterms at beginning
  dictpos=BrilleDefinitions.INT_SIZE;
  dictpart=0;
  Buffer currInvBuffer = bp.pinBuffer(invpart,invnr,false,false);
  currDictBuffer = bp.pinBuffer(dictpart,dictnr,false,false);
  int currcnt=0;
  int currnumoccs=0;
  ByteArr currTerm = new ByteArr(-1);
  if(BrilleDefinitions.DEBUG) logger.debug("starting emptying the heap");
  while(!heapEmpty()){
    if(BrilleDefinitions.DEBUG) logger.debug("the heap is still not empty");
    if(BrilleDefinitions.DEBUG) logger.debug("outputting all cached terms");
    if(BrilleDefinitions.DEBUG) for(int i=0;i<numDocs;i++)
      logger.debug(cachedNextTerms.get(i).decodeString());
    int startpos = getNextPos();
    if(BrilleDefinitions.DEBUG) logger.debug("startpos is "+startpos);
    int currBuffer = startpos/BrilleDefinitions.BUFFER_SIZE;
    int currpos=startpos-BrilleDefinitions.BUFFER_SIZE*currBuffer;
    if(BrilleDefinitions.DEBUG)
      logger.debug("should read bytearr from there");
    int inc=0;
    ByteArr newTerm = cachedNextTerms.get(0);
    if(BrilleDefinitions.DEBUG)
      logger.debug("read term "+newTerm.decodeString());
    if(newTerm.compareTo(currTerm)!=0){
      if(BrilleDefinitions.DEBUG)
        logger.debug("got new term "+newTerm.decodeString());
      writeToDict(currTerm, newTerm,currcnt, currnumoccs,invpart,invpos);
      currTerm=newTerm;
      currcnt=0;
      currnumoccs=0;
    }
    currcnt++;
    if(BrilleDefinitions.BUFFER_SIZE-invpos<BrilleDefinitions.INT_SIZE){
      invpos=0;invpart++; bp.unpinBuffer(currInvBuffer,false,false);
      currInvBuffer=bp.pinBuffer(invpart,invnr,false,false);
    }
    currInvBuffer.putInt(getNextDocNr(),invpos);
    invpos+=BrilleDefinitions.INT_SIZE;
```

```
      if(BrilleDefinitions.DEBUG)
        logger.debug("have put docnr to invbuffer,"+getNextDocNr());
      if(BrilleDefinitions.BUFFER_SIZE-currpos<BrilleDefinitions.INT_SIZE){
        inc+=(BrilleDefinitions.BUFFER_SIZE-currpos); currBuffer++; currpos=0;
      }
      int length = buffers[currBuffer].getInt(currpos);
      if(BrilleDefinitions.DEBUG) logger.debug("length is "+length);
      if(BrilleDefinitions.BUFFER_SIZE-invpos<BrilleDefinitions.INT_SIZE){
        invpos=0;invpart++; bp.unpinBuffer(currInvBuffer,false,false);
        currInvBuffer=bp.pinBuffer(invpart,invnr,false,false);
      }
      currInvBuffer.putInt(length,invpos);
      if(BrilleDefinitions.DEBUG) logger.debug("have put length to invbuffer");
      invpos+=BrilleDefinitions.INT_SIZE;
      currpos+=BrilleDefinitions.INT_SIZE;
      inc+=BrilleDefinitions.INT_SIZE;
      currnumoccs+=length;
      if(BrilleDefinitions.DEBUG)
        logger.debug("should output occurrence list");
      for(int i=0;i<length;i++){
        if(BrilleDefinitions.BUFFER_SIZE-currpos<BrilleDefinitions.INT_SIZE){
          inc+=(BrilleDefinitions.BUFFER_SIZE-currpos);
          currBuffer++; currpos=0;
        }
        if(BrilleDefinitions.BUFFER_SIZE-invpos<BrilleDefinitions.INT_SIZE){
          invpos=0;invpart++; bp.unpinBuffer(currInvBuffer,false,false);
          currInvBuffer=bp.pinBuffer(invpart,invnr,false,false);
          if(BrilleDefinitions.DEBUG)
            logger.debug("getting next buffer for inv");
        }
        int transpos = buffers[currBuffer].getInt(currpos);
        currInvBuffer.putInt(transpos,invpos);
        currpos+=BrilleDefinitions.INT_SIZE;
        invpos+=BrilleDefinitions.INT_SIZE;
        inc+=BrilleDefinitions.INT_SIZE;
      }
      if(BrilleDefinitions.DEBUG)
        logger.debug("have outputted occurrence list");
      if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
        for(int i=0;i<length;i++,currpos++){
          if(currpos==BrilleDefinitions.BUFFER_SIZE){ currBuffer++; currpos=0;}
          if(BrilleDefinitions.BUFFER_SIZE==invpos){
            invpos=0;invpart++; bp.unpinBuffer(currInvBuffer,false,false);
            currInvBuffer=bp.pinBuffer(invpart,invnr,false,false);
          }
          currInvBuffer.putByte(buffers[currBuffer].getByte(currpos),invpos);
          inc++;
        }
      }
      if(BrilleDefinitions.DEBUG) logger.debug("should incnextpos with "+inc);
      boolean newOne =incnextpos(inc);
      if(newOne){
        if(numDocs==1){
          numDocs=0;
          break;
        }
        move(--numDocs,0);
        ByteArr m = cachedNextTerms.remove(cachedNextTerms.size()-1);
        cachedNextTerms.set(0,m);
      }
      else cachedNextTerms.set(0,readByteArrForDocAndInc(0));
      minHeapify(0);
    }
    if(currTerm.getL()!=-1){
      writeToDict(currTerm,new ByteArr(-1),currcnt,currnumoccs,invpart,invpos);
    }
    else{
      currDictBuffer.putInt(dictpos-1,BrilleDefinitions.BUFFER_SIZE
          -BrilleDefinitions.INT_SIZE*(numterms+1));
```

```
      currDictBuffer.putInt(numterms+1,0);
    }
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)
      logger.debug("done with the actual flushing in flush()." +
          " unpinning invbuff");
    bp.unpinBuffer(currInvBuffer,false,false);
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)
      logger.debug("unpinning dictbuff");
    bp.unpinBuffer(currDictBuffer,false,false);
    for(int i=0;i<buffers.length;i++) if(buffers[i]!=null){
      if(BrilleDefinitions.UNPIN_BUG_DEBUG)
        logger.debug("unpinning nullbuf nr "+i);
      bp.unpinBuffer(buffers[i],false,false);
    }
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("undirty nullfile");
    if(BrilleDefinitions.DEBUG) logger.info("undirtyNullFile");
    new UndirtyThread(bp,nullnr).start();
    if(BrilleDefinitions.DEBUG) logger.info("done with undirtyNullFile");
    if(BrilleDefinitions.UNPIN_BUG_DEBUG)logger.debug("all done");
    if(BrilleDefinitions.BASELINERUN)
      timeSpentFlushing = (System.nanoTime()-startTime);
    return invpart+1;
  }

  /**
   * Retrieving the document-number at the top of the min-heap.
   *
   * @return the document-number with the lexicographically smallest term
   */
  private int getNextDocNr(){
    return buffers[buffers.length-1].getInt(BrilleDefinitions.BUFFER_SIZE-4);
  }

  /**
   * Getting the position to read from in the document with the
   * lexicographically smallest term.
   *
   * @return position in document with lexicographically smallest term
   */
  private int getNextPos(){
    return buffers[buffers.length-1].getInt(BrilleDefinitions.BUFFER_SIZE-8);
  }

  /**
   * Method used to inc read-from-position in the document at the top of the
   * heap. If this method returns true, we know that we have reached the end of
   * this document.
   *
   * @param inc - the amount to increment the read-from-position in the
   *   document at the top of the heap.
   * @return whether or not we have reached the end of the document at the
   *   top of the heap.
   */
  private boolean incnextpos(int inc){
    buffers[buffers.length-1].putInt(
        buffers[buffers.length-1].getInt(BrilleDefinitions.BUFFER_SIZE-8)+inc,
        BrilleDefinitions.BUFFER_SIZE-8);
    return buffers[buffers.length-1].getInt(BrilleDefinitions.BUFFER_SIZE-8)==
      buffers[buffers.length-1].getInt(BrilleDefinitions.BUFFER_SIZE-12);
  }

  /**
   * Testing whether the heap is empty, in which case we have no more documents
   * to flush.
   *
   * @return - heap empty
   */
  private boolean heapEmpty(){
    return numDocs==0;
```

```
}

/**
 * Method for adding a new document to this partil index if there is room for
 * it. The document is represented in the index as explained above, if a
 * worst-case estimate of its space usage is smaller than the remaining
 * number of bytes in the partial index.
 *
 * @param name - a hashmap-representation of this document, mapping from
 *  terms to ExtendableDocTermEntry, which contains both an occurrence-list,
 *  and a list of descriptions of these occurrences if the number of bits
 *  used from the description is above 0.
 * @param terms - An ArrayList of ByteArrs. A term is represented in the
 *  index as a ByteArr, and terms are sorted lexicographically on the
 *  ByteArr-representation.
 * @param docNr - the document number of this document, that will be stored
 *  in the heap.
 * @param docLength the amount of bytes used to store this document
 *  compactly.
 * @return whether or not this document is added to the partial index, that
 *  is whether or not there is room for it in this one.
 */
public boolean addDoc(HashMap<String, ExtendableDocTermEntry> doc,
    ArrayList<ByteArr> terms,int docNr,int docLength){
  int realWorstCaseDocLength = docLength +
    ((docLength/BrilleDefinitions.BUFFER_SIZE)+1)*
    LARGEST_DATATYPE_USED_IN_INDEX;
  if(BrilleDefinitions.DEBUG) logger.debug("should add doc with nr "+docNr);
  if(!roomFor(realWorstCaseDocLength)) return false;
  if(BrilleDefinitions.BASELINERUN)
    startTime = System.nanoTime();
  if(BrilleDefinitions.DEBUG) logger.debug("have room for it");
  int[] docinfo = new int[3];
  docinfo[0]=docNr;
  docinfo[1]=(int)currBufferPos();
  if(BrilleDefinitions.DEBUG)
    logger.debug("starting at position "+docinfo[1]);
  Collections.sort(terms);
  if(BrilleDefinitions.DEBUG) logger.debug("terms sorted");
  for(ByteArr b:terms){
    if(b.getL()>BrilleDefinitions.MAX_TERM_LENGTH)
      throw new Error("the term is to long!");
    if(BrilleDefinitions.BUFFER_SIZE-currBufferInd<
        BrilleDefinitions.SHORT_SIZE){
      currBufferInd=0;currBufferPart++;
      if(buffers[currBufferPart]==null)
        buffers[currBufferPart] =
          bp.pinBuffer(currBufferPart,nullnr,false,false);
    }

    buffers[currBufferPart].putShort((short)b.getL(),currBufferInd);

    if(BrilleDefinitions.DEBUG) logger.debug("l written");

    currBufferInd+=BrilleDefinitions.SHORT_SIZE;

    for(int i=0;i<b.getL();i++,currBufferInd++){
      if(BrilleDefinitions.BUFFER_SIZE==currBufferInd){
        currBufferInd=0;currBufferPart++;
        if(buffers[currBufferPart]==null)
          buffers[currBufferPart] =
            bp.pinBuffer(currBufferPart,nullnr,false,false);
      }
      buffers[currBufferPart].putByte(b.getBuffer().get(i),currBufferInd);
    }

    if(BrilleDefinitions.DEBUG) logger.debug("complete term written, "+
        b.decodeString());
    IntArrayList toAdd = doc.get(b.decodeString()).getLocs();
```

```
    if(BrilleDefinitions.DEBUG) logger.debug("adding occurrences");


    if(BrilleDefinitions.BUFFER_SIZE-currBufferInd<
        BrilleDefinitions.INT_SIZE){
      currBufferInd=0;currBufferPart++;
      if(buffers[currBufferPart]==null)
        buffers[currBufferPart] =
          bp.pinBuffer(currBufferPart,nullnr,false,false);
    }

    buffers[currBufferPart].putInt(toAdd.size(),currBufferInd);
    currBufferInd+=BrilleDefinitions.INT_SIZE;
    for(int i=0;i<toAdd.size();i++){
      if(BrilleDefinitions.BUFFER_SIZE-currBufferInd<
          BrilleDefinitions.INT_SIZE){
        currBufferInd=0;currBufferPart++;
        if(buffers[currBufferPart]==null)
          buffers[currBufferPart] =
            bp.pinBuffer(currBufferPart,nullnr,false,false);
      }
      buffers[currBufferPart].putInt(toAdd.get(i),currBufferInd);
      currBufferInd+=BrilleDefinitions.INT_SIZE;
    }

    if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
      ByteArrayList addDescr = doc.get(b.decodeString()).getDescr();
      for(int i=0;i<addDescr.size();i++,currBufferInd++){
        if(BrilleDefinitions.BUFFER_SIZE==currBufferInd){
          currBufferInd=0;currBufferPart++;
          if(buffers[currBufferPart]==null)
            buffers[currBufferPart] =
              bp.pinBuffer(currBufferPart,nullnr,false,false);
        }
        buffers[currBufferPart].putByte(addDescr.get(i),currBufferInd);
      }
    }
  }
  docinfo[2]=(int)currBufferPos();
  if(docinfo[1]!=docinfo[2]){
    if(BrilleDefinitions.DEBUG)
      logger.debug("ending at position "+docinfo[2]);
    setInd(numDocs++,docinfo);
  }
  if(BrilleDefinitions.BASELINERUN)
    timeSpentAddingDocs += (System.nanoTime()-startTime);
  return true;
}

public boolean addDocument(HashMap<ByteArr,ExtendableDocTermEntry> doc,
    ArrayList<ByteArr> terms,int docNr,int docLength){
  int realWorstCaseDocLength = docLength +
    ((docLength/BrilleDefinitions.BUFFER_SIZE)+1)*
    LARGEST_DATATYPE_USED_IN_INDEX;
  if(BrilleDefinitions.DEBUG) logger.debug("should add doc with nr "+docNr);
  if(!roomFor(realWorstCaseDocLength)) return false;
  if(BrilleDefinitions.BASELINERUN)
    startTime = System.nanoTime();
  if(BrilleDefinitions.DEBUG) logger.debug("have room for it");
  int[] docinfo = new int[3];
  docinfo[0]=docNr;
  docinfo[1]=(int)currBufferPos();
  if(BrilleDefinitions.DEBUG)
    logger.debug("starting at position "+docinfo[1]);
  Collections.sort(terms);
  if(BrilleDefinitions.DEBUG) logger.debug("terms sorted");
  for(ByteArr b:terms){
    if(b.getL()>BrilleDefinitions.MAX_TERM_LENGTH)
      throw new Error("the term is to long!");
```

```
  if(BrilleDefinitions.BUFFER_SIZE-currBufferInd<
      BrilleDefinitions.SHORT_SIZE){
    currBufferInd=0;currBufferPart++;
    if(buffers[currBufferPart]==null)
      buffers[currBufferPart] =
        bp.pinBuffer(currBufferPart,nullnr,false,false);
  }

  buffers[currBufferPart].putShort((short)b.getL(),currBufferInd);

  if(BrilleDefinitions.DEBUG) logger.debug("l written");

  currBufferInd+=BrilleDefinitions.SHORT_SIZE;

  for(int i=0;i<b.getL();i++,currBufferInd++){
    if(BrilleDefinitions.BUFFER_SIZE==currBufferInd){
      currBufferInd=0;currBufferPart++;
      if(buffers[currBufferPart]==null)
        buffers[currBufferPart] =
          bp.pinBuffer(currBufferPart,nullnr,false,false);
    }
    buffers[currBufferPart].putByte(b.getBuffer().get(i),currBufferInd);
  }

  if(BrilleDefinitions.DEBUG) logger.debug("complete term written, "+
      b.decodeString());
  IntArrayList toAdd = doc.get(b).getLocs();
  if(BrilleDefinitions.DEBUG) logger.debug("adding occurrences");


  if(BrilleDefinitions.BUFFER_SIZE-currBufferInd<
      BrilleDefinitions.INT_SIZE){
    currBufferInd=0;currBufferPart++;
    if(buffers[currBufferPart]==null)
      buffers[currBufferPart] =
        bp.pinBuffer(currBufferPart,nullnr,false,false);
  }

  buffers[currBufferPart].putInt(toAdd.size(),currBufferInd);
  currBufferInd+=BrilleDefinitions.INT_SIZE;
  for(int i=0;i<toAdd.size();i++){
    if(BrilleDefinitions.BUFFER_SIZE-currBufferInd<
        BrilleDefinitions.INT_SIZE){
      currBufferInd=0;currBufferPart++;
      if(buffers[currBufferPart]==null)
        buffers[currBufferPart] =
          bp.pinBuffer(currBufferPart,nullnr,false,false);
    }
    buffers[currBufferPart].putInt(toAdd.get(i),currBufferInd);
    currBufferInd+=BrilleDefinitions.INT_SIZE;
  }

  if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
    ByteArrayList addDescr = doc.get(b.decodeString()).getDescr();
    for(int i=0;i<addDescr.size();i++,currBufferInd++){
      if(BrilleDefinitions.BUFFER_SIZE==currBufferInd){
        currBufferInd=0;currBufferPart++;
        if(buffers[currBufferPart]==null)
          buffers[currBufferPart] =
            bp.pinBuffer(currBufferPart,nullnr,false,false);
      }
      buffers[currBufferPart].putByte(addDescr.get(i),currBufferInd);
    }
  }
}
docinfo[2]=(int)currBufferPos();
if(docinfo[1]!=docinfo[2]){
  if(BrilleDefinitions.DEBUG)
    logger.debug("ending at position "+docinfo[2]);
```

```
      setInd(numDocs++,docinfo);
    }
    if(BrilleDefinitions.BASELINERUN)
      timeSpentAddingDocs += (System.nanoTime()-startTime);
    return true;
  }

  /**
   * Getting the current overall position in the index.
   * @return - current pos in index
   */
  private long currBufferPos(){
    return ((long)BrilleDefinitions.BUFFER_SIZE)*((long)currBufferPart)+
      ((long)currBufferInd);
  }

  /**
   * Method to test whether there is room for a document with the given
   * worst-case length in bytes. The reason why the worst-case length may
   * differ from the compact length, is that there might not be room for
   * datatypes larger than 1 byte at the end of a buffer.
   *
   * @param docLength - the worst-case length of the document
   * @return whether or not there is room for such a document.
   */
  private boolean roomFor(int docLength){
    if(BrilleDefinitions.DEBUG)
      logger.debug("in roomFor with doclength "+docLength);
    if(BrilleDefinitions.DEBUG)
      logger.debug("buffers.size() is "+buffers.length+
          " and buffersize is "+BrilleDefinitions.BUFFER_SIZE);
    long restpos = ((long)buffers.length*(long)BrilleDefinitions.BUFFER_SIZE)-
      getUsedBytes();
    if(BrilleDefinitions.DEBUG) logger.debug("restpos is "+restpos);
    return ((long)(HEAP_ENTRY_SIZE+docLength))<=restpos;
  }

  public long getUsedBytes(){
    return currBufferPos() + ((long)(HEAP_ENTRY_SIZE*numDocs));
  }

  /**
   * This method is called when this index should be flushed. It builds
   * a heap of the documents represented at the end of the buffers. The heap
   * remains at the end of the buffers, except that the next term for each
   * document is cached in memory.
   *
   */
  private void buildHeap(){
    if(BrilleDefinitions.DEBUG)
      logger.debug("buildheap with numdocs = "+numDocs);
    for(int i=(numDocs/2)-1;i>=0;i--) minHeapify(i);
  }

  /**
   * Usual heap-routine
   *
   * @param i - the index to min-heapify
   */
  private void minHeapify(int i){
    if(BrilleDefinitions.DEBUG) logger.debug("minHeapify with i="+i);
    if(BrilleDefinitions.DEBUG)
      logger.debug("this is "+cachedNextTerms.get(i).decodeString()+
          " "+getDocNrForDocPos(i));
    int le = i*2+1;
    if(BrilleDefinitions.DEBUG && le<numDocs)
      logger.debug("le is "+cachedNextTerms.get(le).decodeString()+
          " "+getDocNrForDocPos(le));
    int ri = i*2+2;
```

```
    if(BrilleDefinitions.DEBUG && ri<numDocs)
      logger.debug("ri is "+cachedNextTerms.get(ri).decodeString()+
          " "+getDocNrForDocPos(ri));
    if(le>=numDocs){
      if(BrilleDefinitions.DEBUG)
        logger.debug("returning because le is to high");
      return;
    }
    if(BrilleDefinitions.DEBUG && ri<numDocs)
      logger.debug("compare(le,ri) gives "+compare(le,ri));
    int min= ri>=numDocs?le:compare(le,ri)<0?le:ri;
    if(BrilleDefinitions.DEBUG && min==ri) logger.debug("ri is min");
    if(BrilleDefinitions.DEBUG && min==le) logger.debug("le is min");
    if(BrilleDefinitions.DEBUG)
      logger.debug(cachedNextTerms.get(min).decodeString()+
          " "+getDocNrForDocPos(min));
    if(compare(i,min)>0){
      if(BrilleDefinitions.DEBUG) logger.debug("should move");
      int[] tmp = getInd(i);
      move(min,i);
      setInd(min,tmp);
      ByteArr tmpb = cachedNextTerms.get(i);
      cachedNextTerms.set(i,cachedNextTerms.get(min));
      cachedNextTerms.set(min,tmpb);
      if(BrilleDefinitions.DEBUG) logger.debug("after move");
      if(BrilleDefinitions.DEBUG)
        logger.debug(i+": "+cachedNextTerms.get(i).decodeString()+
            " "+getDocNrForDocPos(i));
      if(BrilleDefinitions.DEBUG)
        logger.debug(min+": "+cachedNextTerms.get(i).decodeString()+
            " "+getDocNrForDocPos(i));
      minHeapify(min);
    }
  }

  /**
   * Moving a position to another one in the heap.
   *
   * @param from - the position to move from
   * @param to - the position to move to
   */
  private void move(int from, int to){
    int firstind1=from*HEAP_ENTRY_SIZE;
    int firstind2=to*HEAP_ENTRY_SIZE;
    int currbuf1 = buffers.length-1-(firstind1/BrilleDefinitions.BUFFER_SIZE);
    int currind1 = BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE
      -(firstind1%BrilleDefinitions.BUFFER_SIZE);
    int currbuf2 = buffers.length-1-(firstind2/BrilleDefinitions.BUFFER_SIZE);
    int currind2 = BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE
      -(firstind2%BrilleDefinitions.BUFFER_SIZE);
    buffers[currbuf2].putInt(buffers[currbuf1].getInt(currind1),currind2);
    for(int j=1;j<3;j++){
      currind1-=BrilleDefinitions.INT_SIZE;
      currind2-=BrilleDefinitions.INT_SIZE;
      if(currind1<0){
        currbuf1--;
        currind1=BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE;
      }
      if(currind2<0){
        currbuf2--;
        currind2=BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE;
      }
      buffers[currbuf2].putInt(buffers[currbuf1].getInt(currind1),currind2);
    }
  }

  /**
   * Setting the values at a specified index in the heap
   *
```

```
 * @param i - the index at which the values should be set
 * @param v - the values to set in the specified position in the heap
 */
private void setInd(int i, int[] v){
  int firstind=i*HEAP_ENTRY_SIZE;
  int currbuf = buffers.length-1-(firstind/BrilleDefinitions.BUFFER_SIZE);
  int currind = BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE-
    (firstind%BrilleDefinitions.BUFFER_SIZE);
  if(buffers[currbuf]==null)
    buffers[currbuf] = bp.pinBuffer(currbuf,nullnr,false,false);
  buffers[currbuf].putInt(v[0],currind);
  for(int j=1;j<3;j++){
    currind-=BrilleDefinitions.INT_SIZE;
    if(currind<0){
      currbuf--;
      currind=BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE;
      if(buffers[currbuf]==null)
        buffers[currbuf] = bp.pinBuffer(currbuf,nullnr,false,false);
    }
    buffers[currbuf].putInt(v[j],currind);
  }
}

/**
 * Getting the values at the specified position in the heap.
 *
 * @param i - the index in the heap to retrieve the values  from
 * @return - the values at index i in the heap
 */
private int[] getInd(int i){
  int[] ret = new int[3];
  int firstind = i*HEAP_ENTRY_SIZE;
  int currbuf = buffers.length-1-firstind/BrilleDefinitions.BUFFER_SIZE;
  int currind=BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE-
    firstind%BrilleDefinitions.BUFFER_SIZE;
  ret[0]=buffers[currbuf].getInt(currind);
  for(int j=1;j<3;j++){
    currind-=BrilleDefinitions.INT_SIZE;
    if(currind<0){
      currbuf--;
      currind=BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE;
      if(buffers[currbuf]==null)
        buffers[currbuf] = bp.pinBuffer(currbuf,nullnr,false,false);
    }
    ret[j]=buffers[currbuf].getInt(currind);
  }
  return ret;
}

/**
 * Retriving the document number represented at position pos in the heap.
 *
 * @param pos - the position in the heap we want the document number for
 * @return - the document number represented at the given position in the
 *   heap
 */
private int getDocNrForDocPos(int pos){
  int firstind = pos*HEAP_ENTRY_SIZE;
  int currbuf = buffers.length-1-firstind/BrilleDefinitions.BUFFER_SIZE;
  int currind=BrilleDefinitions.BUFFER_SIZE-BrilleDefinitions.INT_SIZE-
    firstind%BrilleDefinitions.BUFFER_SIZE;
  return buffers[currbuf].getInt(currind);
}

/**
 * Comparing two position in the heap.
 *
 * @param i - one of the positions
 * @param j - the other position
```

```
 * @return - negative if position i is smaller than position j and vice versa
 */
private int compare(int i, int j){
  if(BrilleDefinitions.DEBUG) logger.debug("in compare of "+i+" and "+j);
  int comp = cachedNextTerms.get(i).compareTo(cachedNextTerms.get(j));
  if(comp!=0) return comp;
  return getDocNrForDocPos(i)-getDocNrForDocPos(j);
}

/**
 * Debugging method for logging the contents of the heap.
 *
 */
public void logHeap(){
  for(int i=0;i<numDocs;i++){
    int[] vals = getInd(i);
    int currBuffer = vals[1]/BrilleDefinitions.BUFFER_SIZE;
    int currpos = vals[1]-BrilleDefinitions.BUFFER_SIZE*currBuffer;
    if(BrilleDefinitions.BUFFER_SIZE-currpos<BrilleDefinitions.SHORT_SIZE){
      currBuffer++; currpos=0;
    }
    int currl = (int)buffers[currBuffer].getShort(currpos);
    currpos+=BrilleDefinitions.SHORT_SIZE;
    ByteBuffer t = ByteBuffer.allocate(currl);
    t.order(ByteOrder.nativeOrder());
    for(int j=0;j<currl;j++){
      if(currpos==BrilleDefinitions.BUFFER_SIZE){
        currpos=0;currBuffer++;
      }
      t.put(j,buffers[currBuffer].getByte(currpos++));
    }
    logger.info(new ByteArr(t).decodeString()+" "+vals[0]);
  }
}

public void logAllTermsForDoc(int i){
  int[] vals = getInd(i);
  logger.info("");
  logger.info("Outputting all terms for docnr "+vals[0]);
  int start = vals[1];
  int currBuffer = start/BrilleDefinitions.BUFFER_SIZE;
  int currpos = start-BrilleDefinitions.BUFFER_SIZE*currBuffer;
  int stop = vals[2];
  int endBuffer = stop/BrilleDefinitions.BUFFER_SIZE;
  int endpos = stop-BrilleDefinitions.BUFFER_SIZE*endBuffer;
  while(currBuffer<endBuffer || (currBuffer==endBuffer && currpos<endpos)){
    if(BrilleDefinitions.BUFFER_SIZE-currpos<BrilleDefinitions.SHORT_SIZE){
      currBuffer++; currpos=0;
    }
    int currl = (int)buffers[currBuffer].getShort(currpos);
    currpos+=BrilleDefinitions.SHORT_SIZE;
    ByteBuffer t = ByteBuffer.allocate(currl);
    t.order(ByteOrder.nativeOrder());
    for(int j=0;j<currl;j++){
      if(currpos==BrilleDefinitions.BUFFER_SIZE){currpos=0;currBuffer++;}
      t.put(j,buffers[currBuffer].getByte(currpos++));
    }
    if(BrilleDefinitions.BUFFER_SIZE-currpos<BrilleDefinitions.INT_SIZE){
      currBuffer++; currpos=0;
    }
    int length = buffers[currBuffer].getInt(currpos);
    currpos+=BrilleDefinitions.INT_SIZE;
    String print = new ByteArr(t).decodeString()+" Locations: ";
    for(int j=0;j<length;j++){
      if(BrilleDefinitions.BUFFER_SIZE-currpos<BrilleDefinitions.INT_SIZE){
        currBuffer++; currpos=0;
      }
      print+=(buffers[currBuffer].getInt(currpos)+" ");
      currpos+=BrilleDefinitions.INT_SIZE;
```

```
      }
      logger.info(print);
    }
  }

  public long getTimeSpentAddingDocs() {
    return timeSpentAddingDocs;
  }

  public long getTimeSpentFlushing(){
    return timeSpentFlushing;
  }
}
```

C.4.11 brille.dict.MergerThread

```
package brille.dict;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.DelFromBPThread;
import brille.HierarchicIndex;
import brille.HierarchicIndexMaster;
import brille.buffering.NewBufferPool;
import brille.docman.StaticDocManager;
import brille.utils.IntArrayList;

/**
 * This thread is responsible for merging indexes into the hierarchy in the
 * hierarchic index. It continues by sleeping for a while, and when new
 * indexes of the smallest size has been flushed, it gathers all of them
 * and merges them into the hierarchy, possibly merging some of the indexes
 * already in the hierarchy as well. It will continue as long as there
 * are new small indexes.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class MergerThread extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private StaticDocManager docMan;
  private HierarchicIndexMaster m;
  private NewBufferPool bp;
  private boolean running;
  private DelFromBPThread bpdel;

  /**
   * Constructor taking references to the document manager where the ranking
   * values should be updated, the index master, and the thread used to delete
   * files from the buffer pool.
   *
   * @param docMan - the document manager managing the documents represented
   *   in the indexes to be merged
   * @param m - the index master
   * @param bpdel - the thread deleting files from the buffer pool
   */
  public MergerThread(StaticDocManager docMan, HierarchicIndexMaster m,
      DelFromBPThread bpdel){
    this.docMan=docMan;
    this.m=m;
    bp = m.getIndexBP();
    this.bpdel = bpdel;
  }

  /**
```

```
     * Method to call when you want this thread to stop running.
     *
     */
    public void stopRunning(){
      running = false;
    }

    public void run(){
      try{
      running = true;
      int k =0;
      while(running){
        if(BrilleDefinitions.HIERARCHIC_DEBUG && (k%1000)==0)
          logger.debug("I am running");
        k++;
        HierarchicIndex myH = m.getIndex();
        while(myH.getSmall().size()>0){
          long time = System.nanoTime();
          System.out.println("Starting merge at time "+time);
          int sizeToMerge = myH.getSmall().size();
          if(BrilleDefinitions.HIERARCHIC_DEBUG)
            logger.debug("Should merge, size is "+sizeToMerge);
          int numBlocksTogether = 0;
          IntArrayList toRemoveSmall = new IntArrayList();
          IntArrayList dictsToMerge = new IntArrayList();
          IntArrayList invsToMerge = new IntArrayList();
          for(int i=0;i<sizeToMerge;i++){
            numBlocksTogether += myH.getSmallBlocks().get(i);
            toRemoveSmall.add(myH.getSmall().get(i).getDictNr());
            dictsToMerge.add(myH.getSmall().get(i).getDictNr());
            invsToMerge.add(myH.getSmall().get(i).getInvNr());
          }
          int index=0;
          IntArrayList removed = new IntArrayList();
          int maxsize = m.getMaxNrBuffersForBuilding();
          if(myH.getNumBlocks().size()>0){
            numBlocksTogether+=myH.getNumBlocks().get(0);
            if(myH.getNumBlocks().get(0)>0){
              bpdel.addNrToDel(myH.getSearch().get(0));
              dictsToMerge.add(myH.getSearch().get(0).getDictNr());
              invsToMerge.add(myH.getSearch().get(0).getInvNr());
              removed.add(myH.getSearch().get(0).getDictNr());
            }
          }
          while(numBlocksTogether>maxsize){
            index++;
            if(myH.getNumBlocks().size()>index &&
                myH.getNumBlocks().get(index)>0){
              bpdel.addNrToDel(myH.getSearch().get(index));
              numBlocksTogether += myH.getNumBlocks().get(index);
              dictsToMerge.add(myH.getSearch().get(index).getDictNr());
              invsToMerge.add(myH.getSearch().get(index).getInvNr());
              removed.add(myH.getSearch().get(index).getDictNr());
            }
            maxsize*=BrilleDefinitions.K;
          }
          SortedListDictionary newOne = null;
          if(BrilleDefinitions.HIERARCHIC_DEBUG)
            logger.debug("the index it should go into is "+index+
                " and the number of dicts is "+dictsToMerge.size());
          if(dictsToMerge.size()==1) newOne = myH.getSmall().get(0);
          else{
            for(int i=0;i<sizeToMerge;i++)
              bpdel.addNrToDel(myH.getSmall().get(i));
            newOne = new SortedListDictionary(m.getNextNr(),bp,dictsToMerge,
                invsToMerge,docMan,index>=myH.getSearch().size()-1?null:m);
          }
          if(BrilleDefinitions.HIERARCHIC_DEBUG)
            logger.debug("the merging is done");
```

```
        int sweetBlocks = newOne.getBlocks();
        m.switchIndexes(removed,toRemoveSmall,newOne,sweetBlocks,index);
        m.releaseIndex(myH);
        long elapsedTime = System.nanoTime()-time;
        System.out.println("Done with the merge starting at time "+
            time+" after "+elapsedTime+" nanoseconds ");
        myH = m.getIndex();
      }
      m.releaseIndex(myH);
      try{
        Thread.sleep(BrilleDefinitions.SLEEP_TIME_FOR_HIERARCHY_THREAD);
      } catch(InterruptedException ie){
        logger.error("could not sleep",ie);
        ie.printStackTrace();
        System.exit(1);
      }
    }
    } catch(Exception e){
      e.printStackTrace();
      System.out.println("ERROR!!!");
    }
    bp.shutDown();
  }
}
```

C.4.12 brille.dict.MergeSortedListDictEntry

```
package brille.dict;

import brille.utils.ByteArr;

/**
 * This class is just a small extension to SortedListDictEntry
 * where we also have the number of the inverted file where
 * this SortedListDictEntry points to as a parameter. By having it so, we
 * can access the correct inverted file by only having a reference to an
 * object of this kind, when we are merging entries from different inverted
 * files.
 *
 * @author Truls A. Ã¸Bjrklund
 */

public class MergeSortedListDictEntry extends SortedListDictEntry{
  private int invnum;

  /**
   * Standard constructor for this kind of object. Note the invnum which
   * is the difference between this kind of object and SortedListDictEntry.
   *
   * @param pointer - pointer into the inverted file
   * @param term - the term this entry is an entry for
   * @param cnt - the number of documents containing this term
   * @param numoccs - the total number of occurrences of the term in this
   *  document collection indexed in this index
   * @param invnum - the number of the inverted file in the bufferpool
   */
  public MergeSortedListDictEntry(long pointer, ByteArr term, int cnt,
      int numoccs, int invnum) {
    super(pointer, term, cnt, numoccs);
    this.invnum=invnum;
  }

  /**
   * Getter for the number of the inverted file in the bufferpool.
   *
   * @return the number of the inverted file in the bufferpool
   */
```

```
  public int getInvNum(){
    return invnum;
  }

}
```

C.4.13 brille.dict.SortedListDictEntry

```
package brille.dict;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;

import brille.BrilleDefinitions;
import brille.buffering.Buffer;
import brille.utils.ByteArr;

import org.apache.log4j.Logger;

/**
 * This class represents an entry in a SortedListDictionary.
 * It holds the term it represents, together with a pointer to where in the
 * inverted file the inverted list for this term is found. The entry also
 * stores the total number of documents in this index (dict+invfile) that
 * contains this term, and the total number of occurrences.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class SortedListDictEntry {

  protected final Logger logger = Logger.getLogger(getClass());

  private long pointer;
  private ByteArr term;
  private int cnt;
  private int numoccs;

  /**
   * Default constructor. Doesn't do anything:)
   *
   */
  public SortedListDictEntry(){}

  /**
   * Constructor used to make a new SortedListDictEntry.
   * It takes all the member variable of such an entry as arguments.
   *
   * @param pointer - the pointer to where in the inverted file the term
   *  represented is found
   * @param term - the term represented by this entry
   * @param cnt - the number of documents in this index containing this term
   * @param numoccs - the total number of occurrences of this term in this
   *  index
   */
  public SortedListDictEntry(long pointer, ByteArr term, int cnt, int numoccs){
    this.pointer=pointer;
    this.term=term;
    this.cnt=cnt;
    this.numoccs=numoccs;
  }

  /**
   * Get the pointer into the inverted file
   *
   * @return pointer in invfile
   */
  public long getPointer(){
```

```
  return pointer;
}

/**
 * Get the term represented by this entry.
 *
 * @return - the term
 */
public ByteArr getTerm(){
  return term;
}

/**
 * Get the number of documents containg this term in this index.
 *
 * @return - number of docs with this term
 */
public int getCnt(){
  return cnt;
}

/**
 * Get total number of occurrences of this term in this index.
 *
 * @return totoccs of this term in this index
 */
public int getNumoccs(){
  return numoccs;
}

/**
 * Write this SortedListDictEntry to the given buffer at the given position,
 * and return the number of bytes used.
 *
 * @param buf - the buffer to write to
 * @param pos - the position to begin writing at
 * @return - the number of bytes used
 */
public int write(Buffer buf, int pos){
  int l = term.getL();
  int ret=l;
  if(l!=-1) for(int i=0;i<l;i++) buf.putByte(term.getBuffer().get(i),pos+i);
  else throw new Error("this should not happen");
  buf.putLong(pointer,pos+ret);
  buf.putInt(cnt,pos+ret+8);
  buf.putInt(numoccs,pos+ret+12);
  return ret+16;
}

/**
 * Reads in all values in this sortedlistdictentry from the given buffer.
 * It starts at the given position, and based on the to-value, the length
 * of the term to read is given. Return the number of bytes read.
 *
 * @param buf - the buffer to read from
 * @param pos - the position to start reading at
 * @param to - the position to read to
 * @return - the number of bytes read
 */
public int readIn(Buffer buf, int pos, int to){
  int l = to-pos-16;
  int ret = l;
  ByteBuffer b = ByteBuffer.allocate(l);
  b.order(ByteOrder.nativeOrder());
  if(l!=-1) for(int i=0;i<l;i++) b.put(i,buf.getByte(pos+i));
  else throw new Error("this should not happen");
  term = new ByteArr(b);
  pointer=buf.getLong(pos+ret);
  cnt=buf.getInt(pos+ret+8);
```

```
    numoccs=buf.getInt(pos+ret+12);
    return ret+16;
  }

  /**
   * This method caltulates the number of bytes needed to write this entry to
   * a buffer.
   *
   * @return - number of bytes used to represent this entry in a buffer
   */
  public int writeLength(){
    if(BrilleDefinitions.DEBUG){
      logger.info("in writelength and term.getL() is "+term.getL());
      logger.info("this means I should return "+(18+term.getL())+" or 16");
    }
    return 16+(term.getL()==-1?0:term.getL());
  }


  public String toString(){
    return "term: "+term.decodeString()+" pointer: "+pointer+" cnt: "+
      cnt+" numoccs: "+numoccs;
  }

  public String toDumpString(){
    return "term: "+term.decodeString()+" cnt: "+ cnt+" numoccs: "+numoccs;
  }
}
```

C.4.14 brille.dict.SortedListDictionary

```
package brille.dict;

import brille.BrilleDefinitions;
import brille.HierarchicIndex;
import brille.HierarchicIndexMaster;
import brille.NotImplementedException;
import brille.SearchResultHandle;
import brille.buffering.Buffer;
import brille.buffering.BuffPool;
import brille.docman.StaticDocManager;
import brille.inv.IndexEntry;
import brille.inv.InvListIterator;
import brille.inv.InvListMergerHeap;
import brille.inv.MergeInvListIterator;
import brille.utils.ByteArr;
import brille.utils.IntArrayList;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.io.RandomAccessFile;
import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.channels.FileChannel;
import java.util.ArrayList;
import java.util.HashMap;

import org.apache.log4j.Logger;

/**
 * This class is the most basic dictionary in brille.
 * It is essentially a sorted list with some added functionality to store
 * some parts of the dictionary in main memory, while the rest of it is stored
 * on disk, but buffered using the bufferpool.
 *
 * The bufferpool contains several buffers of the same size. The
```

```
 * sortedlistdictionary stores as many dictionary-entries as there are room
 * for in each buffer, and it also keeps an arraylist of bytearrs where
 * the smallest entry in each buffer (except the first) is stored.
 * A search proceedes by binary searching the list in memory first, and
 * using the hit there to define which bufferpart of the dictionary that
 * should be looked up.
 *
 * The given buffer-part is pinned from the bufferpool (read in if it is not
 * currently cached), and a binary search is performed in the buffer as well.
 * The terms in such a buffer obviously have different lenghths. To enable a
 * binary search in such a buffer, a list of where the different terms
 * start, except the first one, and where the last one stops is also
 * stored in each buffer, in a list starting at the end of the buffer, and
 * growing towards the end of the listing of terms.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class SortedListDictionary extends Dictionary{

  protected final Logger logger = Logger.getLogger(getClass());

  private ArrayList<ByteArr> inMemList;
  private BuffPool bp;
  private int dictNum;
  private int invNum;
  private boolean readyForSearch;
  private StaticDocManager docMan;
  private int dictpos;
  private int dictpart;
  private Buffer currDictBuffer;
  private int invpart;
  private int invpos;
  private Buffer currInvBuffer;
  private int numterms;
  private HierarchicIndexMaster master;
  private int numberOfAccessingThreads;
  private boolean deleted;
  private int numBlocks;
  private HashMap<Integer,Double> rankings;
  private HierarchicIndex myH;

  //statistics for baseline run
  private long numentries;
  private long elapsedTime;
  private long startTime;
  private long startSearch;
  private long searchTime;

  /**
   * This constructor is the one to use when you want to merge several
   * sortedlistdictionaries into one new one. It makes a heap over the
   * SortedListDictIterators for each of them, and merges them in a multi-way
   * merge.
   *
   * @param nr - the number to add at the end of the filename of both the
   *  dictionary-file and the inverted file
   * @param bp - the bufferpool from which to pin buffers
   * @param dictnrs - the numbers in the buffer pool of the dictionaries to
   *  merge
   * @param indexnrs - the numbers in the buffer pool of the inverted files
   *  to be merged
   * @param docMan - the document manager where the ranking values of the
   *  merged documents should be stored
   * @param master - this is null if this dictionary is used in the
   *  remerge-method, and refers to the HierarchicIndexMaster if that should
   *  be used. The reason why we need a reference to the hierarchic
   *  index-master is that we need to know the number of occurrences of
   *  different terms in the largest dictionary for ranking purposes.
   *
```

```
 */
public SortedListDictionary(int nr, BuffPool bp, IntArrayList dictnrs,
    IntArrayList indexnrs, StaticDocManager docMan,
    HierarchicIndexMaster master){
  super(nr);
  inMemList = new ArrayList<ByteArr>();
  readyForSearch=false;
  this.master=master;
  if(BrilleDefinitions.HIERARCHIC_DEBUG)
    logger.debug("in the constructor for dict");
  if(master!=null) myH = master.getIndex();
  if(BrilleDefinitions.HIERARCHIC_DEBUG)
    logger.debug("gotten master index");
  this.bp=bp;
  this.docMan=docMan;
  deleted=false;
  dictpos=4;
  dictpart=0;
  invpos=0;
  numterms=0;
  try{
    File f = new File(filename);
    FileChannel channel = new RandomAccessFile(f, "rw").getChannel();
    dictNum=bp.addFile(channel,f);
    f=new File(BrilleDefinitions.INDEX_PATH+"/inv"+nr+".dat");
    channel = new RandomAccessFile(f, "rw").getChannel();
    invNum=bp.addFile(channel,f);
    if(BrilleDefinitions.DEBUG) logger.info("pinning for dict");
    currDictBuffer = bp.pinBuffer(dictpart,dictNum,false,false);
    if(BrilleDefinitions.DEBUG) logger.info("pinning for inv");
    currInvBuffer = bp.pinBuffer(invpart,invNum,false,false);
    if(BrilleDefinitions.HIERARCHIC_DEBUG)
      logger.debug("starting building index");
    buildIndex(dictnrs, indexnrs);
    if(BrilleDefinitions.HIERARCHIC_DEBUG)
      logger.debug("done building index");
    if(BrilleDefinitions.DEBUG) logger.info("printing inmemlist:");
    if(BrilleDefinitions.DEBUG) for(int i=0;i<inMemList.size();i++)
      logger.info(inMemList.get(i).decodeString());
  }catch(FileNotFoundException e){
    e.printStackTrace();
    System.exit(1);
  }finally{
    if(master!=null) master.releaseIndex(myH);
    if(currDictBuffer!=null) bp.unpinBuffer(currDictBuffer,false,false);
    if(currInvBuffer!=null) bp.unpinBuffer(currInvBuffer,false,false);
  }
}

/**
 * This constructor should be used when we want to make a dictionary with
 * cached terms from a single flushed file. It only reads through the index
 * to fix the rankingvalues and cache the entries that should be in the
 * arraylist in memory.
 *
 * @param dictNum - number of the dictionary to create this one from in
 *   the bufferpool
 * @param invNum - number of the associated inverted file in the buffer pool
 * @param bp - the buffer pool
 * @param docMan - the document manager where ranking values should be stored
 * @param master - the hierarchic master if that should be used. It is
 *   included for the same reasons as in the method above.
 */
public SortedListDictionary(int dictNum, int invNum, BuffPool bp,
    StaticDocManager docMan, HierarchicIndexMaster master,int blocks){
  super(dictNum);
  if(BrilleDefinitions.DEBUG)
    logger.info("only one file to merge, so do it the special way!");
  this.bp=bp;
```

```
    this.docMan=docMan;
    this.dictNum=dictNum;
    this.invNum=invNum;
    this.master=master;
    if(master!=null) myH = master.getIndex();
    this.numBlocks=blocks;
    rankings = new HashMap<Integer,Double>();
    inMemList = new ArrayList<ByteArr>();
    readyForSearch=false;
    deleted=false;
    dictpos=4;
    dictpart=0;
    invpos=0;
    currDictBuffer = bp.pinBuffer(dictpart,dictNum,true,false);
    if(BrilleDefinitions.BASELINERUN){
      elapsedTime=0;
      numentries=0;
      startTime = System.nanoTime();
    }
    addRankValuesForBuffer();
    if(BrilleDefinitions.BASELINERUN)
      elapsedTime += (System.nanoTime()-startTime);
    while(!lastInBuf(currDictBuffer)){
      bp.unpinBuffer(currDictBuffer,true,false);
      dictpart++;
      currDictBuffer = bp.pinBuffer(dictpart, dictNum, true,false);
      if(BrilleDefinitions.BASELINERUN)
        startTime = System.nanoTime();
      ByteArr add = addRankValuesForBuffer();
      if(BrilleDefinitions.BASELINERUN)
        elapsedTime += (System.nanoTime()-startTime);
      inMemList.add(add);
    }
    if(BrilleDefinitions.BASELINERUN)
      startTime = System.nanoTime();
    docMan.setRankingValues(rankings);
    if(BrilleDefinitions.BASELINERUN){
      elapsedTime += (System.nanoTime()-startTime);
      System.out.println("Used "+elapsedTime+
          " nanoseconds adding rankings for "+numentries+" entries.");
    }
    rankings=null;
    readyForSearch = true;
    //NOTE:
    if(master!=null) master.releaseIndex(myH);
    bp.unpinBuffer(currDictBuffer,true,false);
  }

  /**
   * This constructor is only used for the JUnit-tests in brille, and is
   * not important.
   *
   * @param inMemList - the ByteArr-list to test binsearch in
   */
  public SortedListDictionary(ArrayList<ByteArr> inMemList){
    super(0);
    this.inMemList=inMemList;
  }

  public int getBlocks(){
    return numBlocks;
  }

  /**
   * This method is the one used when we only want to add ranking values
   * and cache terms from an existing dictionary  file. It adds all
   * ranking values for a given dictionary buffer to the document
   * manager, and returns the lexicographically smallest term in the
   * buffer.
```

```
 *
 * @return the lexicographically smallest term represented in the buffer
 */
public ByteArr addRankValuesForBuffer(){
  int numterms = currDictBuffer.getInt(0);
  double numDocsInDocMan = (double)docMan.getNumActiveDocs();
  ByteArr ret = null;
  SortedListDictEntry en = new SortedListDictEntry();
  for(int i=0;i<numterms;i++){
    int from = getDictPos(currDictBuffer,i);
    int to = getDictPos(currDictBuffer, i+1);
    if(to<from) return ret==null?new ByteArr(-1):ret;
    en.readIn(currDictBuffer, from, to);
    if(BrilleDefinitions.BASELINERUN)
      numentries+=(en.getCnt());
    if(i==0) ret = en.getTerm();
    int cnt = en.getCnt()+
    (master==null?0:master.getCntInLargest(en.getTerm(),myH));
    InvListIterator invit = new InvListIterator(en,bp,invNum);
    IndexEntry inen = invit.getEntry();
    while(inen!=null){
      int docNr = inen.getDocNr();
      if(!rankings.containsKey(docNr)) rankings.put(docNr,0.0);
      int numoccshere = inen.getLocations().length;
      double add = Math.pow(((double)numoccshere)*Math.log(
          ((double)numDocsInDocMan)/(double)cnt),2.0);
      rankings.put(docNr,rankings.get(docNr)+add);
      invit.next();
      inen = invit.getEntry();
    }
    invit.release();
  }
  return ret;
}

/**
 * Method for testing whether this buffer is the last one in the current
 * dictionary file. This is checked by checking whether the length of that
 * term is below zero from the array of pointers at the end of the buffer.
 * This array of pointers is explained above. It is a convention in brille
 * that this is how the last term in a dictionary is represented.
 *
 * @param currBuffer - the buffer we want to check whether holds the last
 *  term in the dictionary
 * @return whether or not the given buffer contains the last term in this
 *  dictionary
 */
private boolean lastInBuf(Buffer currBuffer){
  int numTermsInBuf = currBuffer.getInt(0);
  int from = getDictPos(currBuffer,numTermsInBuf-1);
  int to = getDictPos(currBuffer, numTermsInBuf);
  return to<from;
}

/**
 * Increment the number of searches reading this dictionary.
 * This is done to ensure secure deletions of old dictionary files.
 * This method will return false if this file is already scheduled for
 * deletion. This means that the new searchdictionaries are added in
 * the master, but that the search has stalled in between. This implies
 * that the search should start over with new dictionaries with more updated
 * information.
 *
 * @return whether or not the search calling this method is allowed to read
 *  from this dictionary
 */
public synchronized boolean increads(){
  if(deleted) return false;
  numberOfAccessingThreads++;
```

```
    return true;
  }

  /**
   * Opposite of the above. When a search is done reading from a dicionary, it
   * should call this method to make sure that the dictionary will eventually
   * be scheduled for deletion when it is replaced by a newer and more updated
   * one.
   *
   */
  public synchronized void decreads(){
    numberOfAccessingThreads--;
  }

  /**
   * Method called by the thread deleting dictionaries. If this method returns
   * true, the thread is allowed to delete the dictionary. That happens when no
   * searches reads from it, and a variable is set to make sure that no more
   * searches is allowed to read from it later on.
   *
   * @return whether or not this dictionary should be deleted now.
   */
  public synchronized boolean isZeroAccessingAndDelete(){
    if(numberOfAccessingThreads==0){
      deleted=true;
      return true;
    }
    return false;
  }

  /**
   * During a merge of several dictionaries, this method is used when a new
   * term with associated information should be written to the new dictionary.
   *
   * @param s - the sortedlistdicentry to add to the new dictionary
   */
  private void writeToDict(SortedListDictEntry s){
    int needed = s.writeLength()+8;
    int rest = BrilleDefinitions.BUFFER_SIZE-dictpos-(numterms*4);
    if(rest<needed){
      if(BrilleDefinitions.DEBUG)
        logger.info("need new dictbuffer in sortedlistdict");
      currDictBuffer.putInt(numterms,0);
      currDictBuffer.putInt(dictpos,BrilleDefinitions.BUFFER_SIZE-4*numterms);
      bp.unpinBuffer(currDictBuffer,false,false);
      dictpart++;
      currDictBuffer = bp.pinBuffer(dictpart,dictNum,false,false);
      dictpos=4;
      inMemList.add(s.getTerm());
      numterms=0;
    }
    if(numterms!=0)
      currDictBuffer.putInt(dictpos,BrilleDefinitions.BUFFER_SIZE-4*numterms);
    numterms++;
    s.write(currDictBuffer,dictpos);
    dictpos+=s.writeLength();
    if(BrilleDefinitions.DEBUG)
      logger.info("dictpos is now "+dictpos+" after adding "+s.writeLength());
  }

  /**
   * Convenience method to write an int safely to the inverted file, and
   * incrementing the current position there accordingly.
   *
   * @param i - the integer to write
   */
  private void writeIntToInvList(int i){
    if(BrilleDefinitions.BUFFER_SIZE-invpos<BrilleDefinitions.INT_SIZE){
      bp.unpinBuffer(currInvBuffer,false,false);
```

```
      invpart++;
      invpos=0;
      currInvBuffer = bp.pinBuffer(invpart,invNum,false,false);
    }
    currInvBuffer.putInt(i,invpos);
    invpos+=BrilleDefinitions.INT_SIZE;
  }

  /**
   * Convenience method to write a byte safely to the inverted file, and
   * incrementing the current position there accordingly.
   *
   * @param b - the byte to write
   */
  private void writeByteToInvList(byte b){
    if(BrilleDefinitions.BUFFER_SIZE==invpos){
      bp.unpinBuffer(currInvBuffer,false,false);
      invpart++;
      invpos=0;
      currInvBuffer = bp.pinBuffer(invpart,invNum,false,false);
    }
    currInvBuffer.putByte(b,invpos++);
  }

  /**
   * Method used to merge all occurrences of a given term into the new
   * inverted file. It uses an InvListMergerHeap to do so.
   *
   * @param l - a list of entries into different dictionaries that
   * contains the given term
   * @see brille.inv.InvListMergerHeap
   */
  private void merge(ArrayList<MergeSortedListDictEntry> l){
    int cnt = 0;
    int numoccs = 0;
    for(int i=0;i<l.size();i++){
      cnt+=(l.get(i).getCnt());
      numoccs+=(l.get(i).getNumoccs());
    }
    ByteArr term = l.get(0).getTerm();
    if(BrilleDefinitions.DEBUG)
      logger.info("merging for term "+term.decodeString());
    if(BrilleDefinitions.DEBUG)
      logger.info("invpart is "+invpart+" and invpos is "+invpos);
    writeToDict(new SortedListDictEntry(
        (((long)invpart)*((long)BrilleDefinitions.BUFFER_SIZE))+
        ((long)invpos),term,cnt,numoccs));

    int cntInLargest = master==null?0:master.getCntInLargest(term,myH);
    ArrayList<MergeInvListIterator> iil =
      new ArrayList<MergeInvListIterator>();
    if(BrilleDefinitions.DEBUG)
      logger.info("adding to list of mergeinvlistiterators");
    if(BrilleDefinitions.DEBUG)
      logger.info("should add "+l.size()+" elements.");
    for(int i=0;i<l.size();i++) iil.add(new MergeInvListIterator(l.get(i),bp));
    if(BrilleDefinitions.DEBUG) logger.info("should make invlistmergerheap");
    InvListMergerHeap h = new InvListMergerHeap(iil);
    if(BrilleDefinitions.DEBUG) logger.info("have made invlistmergerheap");
    while(!h.isEmpty()){
      if(BrilleDefinitions.DEBUG) logger.info("heap is not empty");
      IndexEntry n = h.getEntry();
      if(BrilleDefinitions.DEBUG) logger.info("entry is "+n.toString());
      int docnr = n.getDocNr();
      int cnthere = n.getLocations().length;
      double add;

      if(master!=null)
        add=Math.pow(((double)cnthere)*Math.log(
```

```
            ((double)docMan.getNumActiveDocs())/
            (double)(cnt+cntInLargest)),2.0);
      else add=Math.pow(((double)(cnthere))*
          Math.log(((double)docMan.getNumActiveDocs())/(double)cnt),2.0);
      docMan.addToNewRankingValue(docnr,add);
      writeIntToInvList(docnr);
      writeIntToInvList(n.getLocations().length);
      for(int i=0;i<n.getLocations().length;i++)
        writeIntToInvList(n.getLocations()[i]);
      if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0)
        for(int i=0;i<n.getLocations().length;i++)
          writeByteToInvList(n.getDescriptions()[i]);
    }

  }

  /**
   * The overall method used to merge several dictionaries and inverted files
   * into new ones. It uses a SortedListDictIterator for each file, which it
   * builds a DictIteratorHeap for, and flushes to the new files through a
   * multi-way merge.
   *
   * @param dictnrs - the numbers in the buffer pool of the dictionaries
   *   to merge
   * @param indexnrs - the numbers in the buffer pool of the inverted files
   *   to merge
   * @see brille.dict.DictIteratorHeap
   */
  private void buildIndex(IntArrayList dictnrs, IntArrayList indexnrs){
    docMan.resetNewRankingValues();
    ArrayList<SortedListDictIterator> l =
      new ArrayList<SortedListDictIterator>();
    for(int i=0;i<dictnrs.size();i++){
      l.add(new SortedListDictIterator(bp,dictnrs.get(i),indexnrs.get(i)));
    }
    if(BrilleDefinitions.DEBUG) logger.info("have added the dictiterators");
    DictIteratorHeap h = new DictIteratorHeap(l);
    if(BrilleDefinitions.DEBUG) logger.info("have made the dictiteratorheap");
    MergeSortedListDictEntry last = h.getEntry();

    ArrayList<MergeSortedListDictEntry> currEntries =
      new ArrayList<MergeSortedListDictEntry>();
    currEntries.add(last);
    if(BrilleDefinitions.DEBUG) logger.info("added the first entry");
    while(!h.isEmpty()){
      MergeSortedListDictEntry next = h.getEntry();
      if(BrilleDefinitions.DEBUG)
        logger.info("the next entry is "+next.toString());
      if(next.getTerm().compareTo(last.getTerm())==0){
        if(BrilleDefinitions.DEBUG)
          logger.info("it is similar to the last one");
        currEntries.add(next);
      }
      else{
        if(BrilleDefinitions.DEBUG)
          logger.info("it is not similar to the last one. Merging..");
        merge(currEntries);
        if(BrilleDefinitions.DEBUG) logger.info("done merging");
        currEntries = new ArrayList<MergeSortedListDictEntry>();
        currEntries.add(next);
        last = next;
      }
    }
    if(BrilleDefinitions.DEBUG)
      logger.info("merging for the last time with term "+
          last.getTerm().decodeString());
    merge(currEntries);
    if(BrilleDefinitions.BUFFER_SIZE-dictpos-(numterms*4)<8){
      if(BrilleDefinitions.DEBUG)
```

```
        logger.info("Need new buffer for last dictpos");
      currDictBuffer.putInt(numterms,0);
      currDictBuffer.putInt(dictpos, BrilleDefinitions.BUFFER_SIZE-4*numterms);
      bp.unpinBuffer(currDictBuffer,false,false);
      dictpart++;
      currDictBuffer = bp.pinBuffer(dictpart,dictNum,false,false);
      dictpos=4;
      inMemList.add(new ByteArr(-1));
      numterms=0;
    }

    currDictBuffer.putInt(numterms+1,0);
    if(numterms!=0)
      currDictBuffer.putInt(dictpos,BrilleDefinitions.BUFFER_SIZE-4*numterms);
    currDictBuffer.putInt(dictpos-1,
        BrilleDefinitions.BUFFER_SIZE-4*(numterms+1));
    bp.unpinBuffer(currDictBuffer,false,false);
    bp.unpinBuffer(currInvBuffer,false,false);
    currDictBuffer=null;
    currInvBuffer=null;
    docMan.finishComputationOfAllNewRankingValuesAndSwitch();
    readyForSearch=true;
    numBlocks = invpart+1;
  }

  /**
   * Retrieving the number this dictionary has in the buffer pool.
   *
   * @return - file number for this dictionary in the buffer pool
   */
  public int getDictNr(){
    return dictNum;
  }

  /**
   * Retrieving the number this inverted file associated with this dictionary
   * has in the buffer pool.
   *
   * @return - file number for inverted file in the buffer pool
   */
  public int getInvNr(){
    return invNum;
  }

  /**
   * Method used to test whether this dictionary is ready for searches, that
   * means whether its construction is done.
   *
   * @return whether this dictionary is ready for searches
   */
  public boolean isReadyForSearch(){
    return readyForSearch;
  }

  /**
   * This is the search method for SortedListDictionaries. It proceeds
   * by performimg a binary search in the terms in the memory list first,
   * before it goes to the buffer part it is pointed to and finds
   * the correct SortedListDictEntry if any. This entry is used to
   * make an InvListIterator over the results, which is returned.
   *
   * @param term - the term to search for
   * @return an InvListIterator for the results
   * @see brille.inv.InvListIterator
   */
  public SearchResultHandle search(ByteArr term){
    if(!readyForSearch)
      throw new
      IllegalArgumentException("the dictionary is not ready for search");
```

```
    if(BrilleDefinitions.DEBUG)
      logger.info("searching for "+term.decodeString());
    if(BrilleDefinitions.BASELINERUN)
      startSearch = System.nanoTime();
    int part = binSearchInMemList(term);
    if(BrilleDefinitions.DEBUG) logger.info("found it to be in part "+part);
    Buffer corrBuf = bp.pinBuffer(part,dictNum,true,false);
    SortedListDictEntry e = binSearch(corrBuf,term);
    if(e==null && BrilleDefinitions.DEBUG) logger.info("ret is null");
    else if(BrilleDefinitions.DEBUG)
      logger.info("the entry i got was "+e.toString());
    bp.unpinBuffer(corrBuf,true,false);
    if(BrilleDefinitions.BASELINERUN)
      searchTime+=(System.nanoTime()-startSearch);
    return new InvListIterator(e,bp,invNum);
  }


  /**
   * This is the other search method for SortedListDictionaries. It proceeds
   * by performimg a binary search in the terms in the memory list first,
   * before it goes to the buffer part it is pointed to and finds
   * the correct SortedListDictEntry if any. This entry is used to
   * make an InvListIterator over the results, which is returned.
   * This method is used in hierarchic indexes where we for the rankings
   * sake wish to add the number of hits in the largest index to the number
   * of results to achieve a more accurate ranking.
   *
   * @param term - the term to search for
   * @return an InvListIterator for the results
   * @see brille.inv.InvListIterator
   */
  public SearchResultHandle search(ByteArr term, int addNrRes){
    if(!readyForSearch)
      throw new
      IllegalArgumentException("the dictionary is not ready for search");
    if(BrilleDefinitions.DEBUG)
      logger.info("searching for "+term.decodeString());
    int part = binSearchInMemList(term);
    if(BrilleDefinitions.DEBUG) logger.info("found it to be in part "+part);
    Buffer corrBuf = bp.pinBuffer(part,dictNum,true,false);
    SortedListDictEntry e = binSearch(corrBuf,term);
    if(e==null && BrilleDefinitions.DEBUG) logger.info("ret is null");
    else if(BrilleDefinitions.DEBUG)
      logger.info("the entry i got was "+e.toString());
    bp.unpinBuffer(corrBuf,true,false);
    return new InvListIterator(e,bp,invNum,addNrRes);
  }


  /**
   * This method is used when changing rankings in the HierarchicIndexMaster.
   * It gives you the number of documents containing a given term, and this
   * is needed when merging some dictionaries, because we want to have the
   * most accurate number of occurrences of a given term for ranking values.
   * This is found in the HierarchicIndexMaster by accessing this method
   * for the largest dictionary.
   *
   * @param term - the term we want to know how many documents in this index
   *   contains.
   * @return the number of documents in this index containing the given term
   */
  public int getNumDocsContainingTerm(ByteArr term){
    if(!readyForSearch)
      throw new
      IllegalArgumentException("the dictionary is not ready for search");
    if(BrilleDefinitions.DEBUG)
      logger.info("searching for "+term.decodeString());
    int part = binSearchInMemList(term);
```

```
    if(BrilleDefinitions.DEBUG) logger.info("found it to be in part "+part);
    Buffer corrBuf = bp.pinBuffer(part,dictNum,true,false);
    SortedListDictEntry e = binSearch(corrBuf,term);
    if(e==null) return 0;
    else return e.getCnt();
  }

  /**
   * Method to delete the index. That is currently not done here, but through
   * the buffer pool. It is uncertain whether this will change in future
   * versions, and the method is still kept here for this reason.
   *
   * @throws NotImplementedException
   */
  public void delete() throws NotImplementedException{
    throw new NotImplementedException("deletion of sortedlistdict is " +
    "not implemented here");
  }

  /**
   * Retrieve the position of term nr index in the given buffer.
   *
   * @param buf - the buffer to read the term from
   * @param index - the term nr we want the position for
   * @return - the position of the given term nr in the buffer
   */
  private int getDictPos(Buffer buf, int index){
    return index==0?4:buf.getInt(BrilleDefinitions.BUFFER_SIZE-4*index);
  }

  /**
   * Method for searching in the given buffer. This method is used after
   * the search has found the correct buffer to pin from a binary search
   * in the memory-list of terms, performed in the method below.
   * This method returns the entry with the given term if it exists,
   * and null otherwise.
   *
   * @param buf - the buffer to perform the binary search in
   * @param term - the term to search for
   * @return - the SortedListDictEntry for the given term, or null if the
   *  term does not exist in this dictionary
   */
  private SortedListDictEntry binSearch(Buffer buf, ByteArr term){
    int num = buf.getInt(0);
    if(BrilleDefinitions.DEBUG) logContentOfBuffer(buf);
    int l = 0;
    int r = num-1;
    int m = (l+r)/2;
    while(l<=r){
      if(BrilleDefinitions.DEBUG) logger.info("m is "+m);
      int currpos = m==0?4:getDictPos(buf,m);
      int le = getDictPos(buf,m+1)-16-currpos;
      if(BrilleDefinitions.DEBUG) logger.info("le is "+le);
      if(le<0){
        r--;
        m=(l+r)/2;
        continue;
      }
      ByteBuffer b = ByteBuffer.allocate(le);
      b.order(ByteOrder.nativeOrder());
      for(int i=0;i<le;i++) b.put(i,buf.getByte(currpos++));
      ByteArr testTerm = new ByteArr(b);
      if(BrilleDefinitions.DEBUG) logger.info(testTerm.decodeString());
      int comp = term.compareTo(testTerm);
      if(comp==0){
        if(BrilleDefinitions.DEBUG) logger.info("HIT");
        long pointer = buf.getLong(currpos);
        currpos+=8;
        int cnt = buf.getInt(currpos);
```

```
      currpos+=4;
      int numoccs = buf.getInt(currpos);
      return new SortedListDictEntry(pointer,testTerm,cnt,numoccs);
    }
    else if(comp<0) r=m-1;
    else l=m+1;
    m=(l+r)/2;
  }
  return null;
}

/**
 * This method is used to perform a binary search in the list of terms kept
 * in main memory. It will give a pointer to the correct buffer-part of the
 * dictionary to pin, where the rest of the search can be carried out.
 *
 * @param term - the term to search for
 * @return - the number of the buffer part to pin to find the
 *   sortedlistdictentry for the given term
 */
public int binSearchInMemList(ByteArr term){
  if(inMemList.size()==0) return 0;
  int l =0;
  int r = inMemList.size();
  int m =0;
  if(term.compareTo(inMemList.get(inMemList.size()-1))>=0)
    return inMemList.size();
  while(l<r-1){
    m = (l+r)/2;
    if(term.compareTo((inMemList.get(m)))<0) r=m;
    else l=m;
  }
  m = (l+r)/2;
  if(l!=r && r!=inMemList.size() && term.compareTo(inMemList.get(r))>=0)
    return r+1;
  if(m==0 && term.compareTo(inMemList.get(0))<0) return 0;
  return m+1;
}

/**
 * Method used for testing, which logs all terms in the given buffer.
 *
 * @param buf - the buffer to log all terms in
 */
public void logContentOfBuffer(Buffer buf){
  int numterms = buf.getInt(0);
  if(lastInBuf(buf)) numterms--;
  SortedListDictEntry en = new SortedListDictEntry();
  logger.info("the buffer has "+numterms+" terms:");
  for(int i=0;i<numterms;i++){
    en.readIn(buf, getDictPos(buf,i),getDictPos(buf,i+1));
    logger.info((i+1)+" "+en.toString());
  }
}


/**
 * Method used for dumping the contents of this dictionary to be able to
 * pick terms to search for.
 *
 * @param out - the printwriter to dump dict to
 */
public void flushDictionary(PrintWriter out){
  int dictp = 0;
  Buffer dictBuffer = bp.pinBuffer(dictp,dictNum,true,false);
  flushContentsOfBuffer(dictBuffer,out);
  while(!lastInBuf(dictBuffer)){
    bp.unpinBuffer(dictBuffer,true,false);
    dictp++;
```

```
      dictBuffer = bp.pinBuffer(dictp, dictNum, true,false);
      flushContentsOfBuffer(dictBuffer,out);
    }
  }

  private void flushContentsOfBuffer(Buffer buf, PrintWriter out){
    int numterms = buf.getInt(0);
    SortedListDictEntry en = new SortedListDictEntry();
    for(int i=0;i<numterms;i++){

      int from = getDictPos(buf, i);
      int to = getDictPos(buf, i+1);
      if(to<from) return;
      en.readIn(buf, from, to);
      out.println(en.toDumpString());
    }
  }

  public long getSearchTime(){
    return searchTime;
  }
}
```

C.4.15 brille.dict.SortedListDictIterator

```
package brille.dict;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.buffering.Buffer;
import brille.buffering.BuffPool;
import brille.utils.ByteArr;

/**
 * This class is an iterator capable of iterating through a complete
 * SortedListDictionary. It will do so by going through the terms
 * lexicographically, returning MergeSortedListDictEntries for each term.
 * Remember to release the iterator after using it, to ensure that no
 * buffers remains pinned forever.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class SortedListDictIterator {

  protected final Logger logger = Logger.getLogger(getClass());

  private int numterms;
  private int readterms;
  private MergeSortedListDictEntry currEntry;
  private int currpos;
  private int filenr;
  private BuffPool bp;
  private int currpart;
  private Buffer currBuf;
  private int invfilenr;
  private boolean done;

  /**
   * This is the only constructor for this class. It takes the buffer pool the
   * dictionary to be indexed is represented in, the file number of the
   * dictionary in the buffer pool, and the filenumber of the inverted file it
   * points to, as arguments.
   *
```

```
 * @param bp - the buffer pool where the dictionary is represented
 * @param filenr - the file number of the dictionary in the buffer pool.
 * @param invfilenr - the file number of the assosiated inverted file in the
 *  buffer pool.
 */
public SortedListDictIterator(BuffPool bp, int filenr, int invfilenr){
  this.bp=bp;
  this.filenr=filenr;
  this.invfilenr=invfilenr;
  currpart=0;
  currpos=4;
  if(BrilleDefinitions.DEBUG) logger.info("pinning for iterator... dict");
  currBuf = bp.pinBuffer(currpart,filenr,true,false);
  done=false;
  numterms = currBuf.getInt(0);
  if(BrilleDefinitions.DEBUG) logger.info("in <init> of " +
      "SortedListDictIterator, and numterms is "+numterms+" for dictnr "+
      filenr+" and part "+currpart);
  readterms=0;
  readNextTerm();
}

/**
 * Getting the file number of the assosiated inverted file in the
 * buffer pool.
 *
 * @return file number of the assosiated inverted file in the buffer pool
 */
public int getInvFileNr(){
  return invfilenr;
}

/**
 * Getting the entry the iterator is currently at.
 *
 * @return current entry in iterator
 */
public MergeSortedListDictEntry getEntry(){
  return currEntry;
}

private int readIntUnsafe(){
  currpos+=BrilleDefinitions.INT_SIZE;
  return currBuf.getInt(currpos-BrilleDefinitions.INT_SIZE);
}

private long readLongUnsafe(){
  currpos+=BrilleDefinitions.LONG_SIZE;
  return currBuf.getLong(currpos-BrilleDefinitions.LONG_SIZE);
}

private byte readByteUnsafe(){
  currpos++;
  return currBuf.getByte(currpos-1);
}

private void readNextTerm(){
  if(done) return;
  if(readterms==numterms){
    bp.unpinBuffer(currBuf,true,false);
    currpart++;
    currpos=BrilleDefinitions.INT_SIZE;
    currBuf = bp.pinBuffer(currpart,filenr,true,false);
    numterms = currBuf.getInt(0);
    if(BrilleDefinitions.DEBUG) logger.info("NEEDED TO READ NEW IN " +
        "ITERATOR: numterms is "+numterms);
    readterms=0;
  }
  int from = readterms==0?BrilleDefinitions.INT_SIZE:currBuf.getInt(
```

```
        BrilleDefinitions.BUFFER_SIZE-4*(readterms));
    int to = currBuf.getInt(BrilleDefinitions.BUFFER_SIZE
        -BrilleDefinitions.INT_SIZE*(readterms+1));
    if(BrilleDefinitions.DEBUG) logger.info("from is "+from+" and to is "+to);
    if(to<from){
      if(BrilleDefinitions.DEBUG)
        logger.info("to is less than from after reading "+readterms+" terms.");
      if(BrilleDefinitions.DEBUG)
        logger.info("to is "+to+" and from is "+from);
      currEntry=null;
      done=true;
      return;
    }
    int l = to-from -16;
    ByteBuffer buf = ByteBuffer.allocate(l);
    buf.order(ByteOrder.nativeOrder());
    if(BrilleDefinitions.DEBUG) logger.info("this gives an l of "+l);
    for(int i=0;i<l;i++) buf.put(i,readByteUnsafe());
    ByteArr t = new ByteArr(buf);
    if(BrilleDefinitions.DEBUG) logger.info("the term is "+t.decodeString());
    long pointer = readLongUnsafe();
    int cnt = readIntUnsafe();
    int numoccs = readIntUnsafe();
    currEntry = new MergeSortedListDictEntry(pointer,t,cnt,numoccs,invfilenr);
    readterms++;
  }

  /**
   * Important method! This should be called when the iterator is no longer
   * used to ensure that no buffers remains pinned endlessly.
   *
   */
  public void release(){
    bp.unpinBuffer(currBuf,true,false);
  }

  /**
   * This method moves the iterator to the next term in the dictionary, moving
   * to the next buffer if needed. The implementatino is really in the private
   * method readNextTerm().
   *
   */
  public void next(){
    readNextTerm();
  }
}
```

C.5 brille.docman

C.5.1 brille.docman.BlackList

```
package brille.docman;

import org.apache.log4j.Logger;

import brille.utils.IntArrayList;

/**
 * This class is the main tool for providing atomic document deletions.
 * It is also used to ensure atomic document insertions in dynamic
 * docmument managers, like DocManager. This class holds a bit for
 * every document in the collection. If this bit is set to 0, the document
 * is active and can thus be returned as a search result. If it is set
 * to 1, the document should not be returned.
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public class BlackList {
  private IntArrayList bl;
  protected final Logger logger = Logger.getLogger(getClass());

  /**
   * Constructor
   *
   */
  public BlackList(){
    bl=new IntArrayList();
  }

  /**
   * Method for testing wether a document with the given document number is
   * blacklisted.
   *
   * @param docNr - the document we want to check
   * @return - true or false depending on wether the document in question
   *  is blacklisted.
   */
  public synchronized boolean isBlackList(int docNr){
    int ind = docNr/32;
    if(ind>bl.size()) return true;
    int pos = docNr-32*docNr;
    return ((bl.get(ind)>>>pos) & 1)>0;
  }

  /**
   * Adding a bit for the given document number. If the list is to short, it
   * is extended. The method also makes sure that this document number is
   * initially blaclisted (its bit is set to 1).
   *
   * @param docNr - the document number we want to add as a blacklisted one.
   */
  public synchronized void addBlackListedDocNr(int docNr){
    int ind = docNr/32;
    while(bl.size()<=ind) bl.add(-1);
    int pos = docNr-32*ind;
    if(((bl.get(ind)>>>pos) & 1)==0) bl.replace(bl.get(ind)^((1<<pos)), ind);
  }

  /**
   * Set the blacklist-bit to one for the given document number.
   * This method is typically called when the document is deleted, and
   * we want to ensure that it will not be returned as a result any more.
```

```
   *
   * @param docNr - the document number of the document we want to blacklist.
   * @return   whether or not the given document number was blacklisted before.
   */
  public synchronized boolean setBlackList(int docNr){
    int ind = docNr/32;
    int pos = docNr-32*ind;
    if(((bl.get(ind)>>>pos) & 1)==0){
      bl.replace(bl.get(ind)^((1<<pos)), ind);
      return true;
    }
    return false;
  }

  /**
   * Set the blacklist-bit to zero for the given document number.
   * This method is typically called when the document is added, and
   * we know that all terms for the document are added to the index
   * so that this document can be returned as a result. It might be so
   * that an ongoing search has retrived some search results before all
   * terms for this document was added, and still return it as a result.
   * This will lead to a situation where the document is returned with
   * a worse rank than it should have, but that is not considered an
   * error in Brille.
   *
   * @param docNr - the document number of the document we want to blacklist.
   * @return   whether or not the given document number was blacklisted before.
   */
  public synchronized boolean setUnBlackList(int docNr){
    int ind = docNr/32;
    int pos = docNr-32*ind;
    if(((bl.get(ind)>>>pos) & 1)>0){
      bl.replace(bl.get(ind)^((1<<pos)), ind);
      return true;
    }
    return false;
  }

  /**
   * Method for blacklisting all documents.
   *
   */
  public synchronized void blacklistAll(){
    for(int i=0;i<bl.size();i++) bl.replace(-1,i);
  }
}
```

C.5.2 brille.docman.DocEntry

```
package brille.docman;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.btree.Entry;
import brille.buffering.Buffer;
import brille.utils.ByteArr;

/**
 * Class representing leaf entries in the B-tree in the document manager.
 * These entries have a single int as the primary key, which is the document
 * number, and the rest of the entry contains the URI to the document.
 * It could easily be extended to store more values for the document,
 * for example pagerank, only by changing this class. Most of the
```

```
 * methods here overrides methods in brille.btree.Entry, and are thus not
 * commented here.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.btree.Entry
 */
public class DocEntry extends Entry{

  protected final Logger logger = Logger.getLogger(getClass());

  private int docNr;
  private ByteArr uri;

  /**
   * Constructor taking the document number and the uri to the document.
   * @param docNr - the document number
   * @param uri - the uri to the document.
   */
  public DocEntry(int docNr,ByteArr uri){
    this.docNr=docNr;
    this.uri=uri;
  }

  /**
   * Constructor used by the factory. Such an entry will be used for reading
   * in entries, to ensure that not to many objects needs to be created
   * and garbage collected in the B-tree implementation.
   *
   */
  public DocEntry(){
    this(-1,null);
  }

  /**
   * Getting the document number.
   * @return - the document number for this entry (the primary key).
   */
  public int getDocNr(){
    return docNr;
  }

  /**
   * Getting the URI to the document represented.
   * @return - the URI to the document represented.
   */
  public ByteArr getURI(){
    return uri;
  }

  public int write(Buffer buf,int pos) {
    if(uri==null) throw new
    IllegalArgumentException("Trying to write an entry without a URI");
    int l = writeKey(buf,pos);
    buf.putShort((short)uri.getL(),pos+l);
    l+=BrilleDefinitions.SHORT_SIZE;
    for(int i=0;i<uri.getL();i++,l++)
      buf.putByte(uri.getBuffer().get(i),pos+l);
    return l;
  }

  public int writeKey(Buffer buf, int pos){
    if(docNr==-1) throw new IllegalArgumentException("Trying to write a key" +
    " wich is not instantiated");
    buf.putInt(docNr,pos);
    return BrilleDefinitions.INT_SIZE;
  }

  public boolean isKeyCompatible(Entry en){
    return (en instanceof InternalDocEntry || en instanceof DocEntry);
```

```
  }

  public int readIn(Buffer buf, int pos) {
    int ret = readInKey(buf,pos);
    int l = (int)buf.getShort(pos+ret);
    ret+=BrilleDefinitions.SHORT_SIZE;
    ByteBuffer c=ByteBuffer.allocate(l);
    c.order(ByteOrder.nativeOrder());
    for(int i=0;i<l;i++,ret++) c.put(i,buf.getByte(pos+ret));
    uri = new ByteArr(c);
    return ret;
  }

  public int readInKey(Buffer buf, int pos){
    docNr=buf.getInt(pos);
    return BrilleDefinitions.INT_SIZE;
  }

  public int keyLength(){
    return BrilleDefinitions.INT_SIZE;
  }

  public int length() {
    if(uri.getL()==-1) return
      BrilleDefinitions.INT_SIZE+BrilleDefinitions.SHORT_SIZE;
    else return
      BrilleDefinitions.INT_SIZE+BrilleDefinitions.SHORT_SIZE+uri.getL();
  }

  public int compareTo(Entry e) {
    if(!isKeyCompatible(e)) throw new
    IllegalArgumentException("can not compare different entries");
    if(e instanceof DocEntry) return docNr -((DocEntry)e).getDocNr();
    else return docNr-((InternalDocEntry)e).getKey();
  }

  public boolean keyIsMax() {
    return docNr==Integer.MAX_VALUE;
  }

  public void setKeyToMax() {
    docNr=Integer.MAX_VALUE;
  }

  public String toString(){
    return docNr+","+(uri==null?"":uri.decodeString());
  }

  public String getKeyAsString(){
    return ""+docNr;
  }

  public void incEntry(Entry en) {
    throw new IllegalArgumentException("not allowed to inc");
  }

  public boolean decEntry(Entry en) {
    throw new IllegalArgumentException("not allowed to dec");
  }
}
```

C.5.3 brille.docman.DocEntryFactory

```
package brille.docman;

import brille.btree.EntryFactory;
```

```
/**
 * A factory for DocEntry.
 * Factories are needed because of the way generics are implemented
 * in java. They are only available compile time, and it is thus
 * impossible to istantiate an object of type T within a MyObject<T>-class.
 * To make a truly dynamic structure like the B-tree in brille, factories
 * are thus needed. The method here is inherited from
 * brille.btreeEntryFactory.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see brille.btree.EntryFactory
 */
public class DocEntryFactory extends EntryFactory{

  public DocEntry make() {
    return new DocEntry();
  }
}
```

C.5.4 brille.docman.DocMan

```
package brille.docman;

/**
 * This is an interface defining what a document manager should provide
 * regardless of whether it supports incremental or inherently static
 * indexes.
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public interface DocMan {
  /**
   * This method retrieves the document length for the given document.
   * The document length is the length as calculated in a normal
   * tf-idf-score, as follows: sqrt(sum(f(i,j)*log(n(i)/N))),
   * where f(i,j) is the frequency of term i in document j, n(i) is
   * the number of documents in the collection that contains term i,
   * and N is the total number of documents in the collection.
   *
   * @param doc - the docnumber we want the tf-idf-length for.
   * @return - the tf-idf-length of the document with doc nr doc.
   */
  public double getRankingValueForDoc(int doc);

  /**
   * Retrieving the number of active documents in the collection.
   * @return - the number of active documents(N).
   */
  public int getNumActiveDocs();

  /**
   * Method to test whether the document with the given document number
   * is blaclisted or not.
   *
   * @param docNr - the document number of the document we want to test.
   * @return - true/false depending on whether the document is blacblisted.
   */
  public boolean isBlackList(int docNr);
}
```

C.5.5 brille.docman.DocManager

```
package brille.docman;
```

```java
import java.net.URI;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.NotImplementedException;
import brille.btree.BTree;
import brille.btree.BTreeHandle;
import brille.buffering.BuffPool;
import brille.dict.FullBTreeIndex;
import brille.utils.ByteArr;
import brille.utils.DoubleArrayList;
import brille.utils.IntArrayList;

/**
 * This class is the document manager for incremental indexes.
 * It serves two main purposes:
 *  1. Keeping track of the mapping between documents and their
 *  numbers.
 *  2. Storing the tf-idf-lengths of all documents, and maintaining
 *  these.
 *
 * The first part is done through a small database-system, implemented
 * as a B-tree, where searches by primary key is the only thing
 * that is really supported. One may thus search for the uri of the document
 * with a given document number. It is also possible to search by uri,
 * but that requires a search through the complete index, and is thus
 * strongly prohibited. The reason why this part is supported through a
 * disk-based structure is that uris are only needed when a document is
 * returned as a result. That is usually true for 10 documents for each
 * search. Because of buffering, it is likely that it takes less than
 * 10 diskaccesses to retrieve these.
 *
 * tf-idf-lengths are needed every time a document is a candidate hit in
 * a search. All documents containing the terms the search searches for
 * must be evaluated. It should be obivous that this could be a lot of
 * documents, and it is essential to avoid performing one disk access per
 * document. Hence, these values are kept in main memory, in a DoubleArrayList.
 *
 * This class also uses a FixRankingsThread. When documents are added
 * incrementally, the ranking values of the first added documents are
 * deteriorating. To avoid doing a complete update of all ranking values
 * each time a document is inserted, we rather have a thread that goes
 * through the whole index, and updates all ranking values. To enable this,
 * this class keeps a shadow copy of the rankingvalues, and switches
 * when the thread has traversed the whole index. Care is taken to avoid
 * updating lengths of documents that are currently beeing added and so on.
 *
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public class DocManager implements DocMan{

  protected final Logger logger = Logger.getLogger(getClass());

  private DoubleArrayList docVectorLengths;
  private DoubleArrayList newDocVectorLengths;
  private IntArrayList freeList;
  private IntArrayList soonFreeList;
  private IntArrayList doNotUpdateDocs;
  private int numActiveDocs;
  private boolean fixingRank;


  private IntArrayList numTokensInDocs;
  private BlackList bl;
  private BTree bt;
  private BuffPool bp;
```

```
private FreeThread ft;
private FixRankingsThread frt;

private int newDocNr;

/**
 * Default constructor
 *
 */
public DocManager() {
  newDocNr=1;
  numActiveDocs=0;
  docVectorLengths=new DoubleArrayList();
  newDocVectorLengths = new DoubleArrayList();
  numTokensInDocs=new IntArrayList();
  freeList = new IntArrayList();
  soonFreeList = new IntArrayList();
  bl = new BlackList();
  fixingRank=false;
  doNotUpdateDocs=new IntArrayList();
  ft = new FreeThread(freeList,soonFreeList);
  ft.start();

  if(BrilleDefinitions.DEBUG)logger.info("docman created");
}

/**
 * Method for finding the disksize used by this document manager.
 *
 * @return - the number of bytes used by the document manager, in a long
 */
public long diskSize(){
  return bt.diskSize();
}

/**
 * Setting the bufferpool this class should use.
 *
 * @param bp
 */
public void setBufferPool(BuffPool bp){
  this.bp=bp;
  makeBTree();
}

/**
 * Setting the index which the fixrankingsthread should traverse
 * to update the rankingvalues.
 * @param index
 */
public void setIndexForUpdateThread(FullBTreeIndex index){
  frt = new FixRankingsThread(this,index);
  frt.start();
}

private void makeBTree(){
  if(BrilleDefinitions.DEBUG)logger.info("all variables set, and btree " +
  "will be created");
  bt=new
  BTree(BrilleDefinitions.INDEX_PATH+"/"+BrilleDefinitions.DOCMAN_FILE_NAME
      ,bp,BrilleDefinitions.SLEEP_TIME_FOR_DOCMAN_UPDATE_THREAD,
      new InternalDocEntryFactory(), new DocEntryFactory());
}

private synchronized int getDocNr(){
  if(freeList.size()>0) return freeList.remove(freeList.size()-1);
  return newDocNr++;
}
```

```
/**
 * Method for adding a new document to this document manager.
 * The document is initally blacklisted, and has a length of zero.
 * The mapping between docnr and uri is inserted into the B-tree,
 * and the document number this document got is returned.
 *
 * @param uri - the uri of the document to add
 * @return - the document number assigned to this document
 */
public int addDoc(URI uri){
  if(BrilleDefinitions.DEBUG)logger.info("adding doc "+uri.toString());
  int ret = getDocNr();
  if(BrilleDefinitions.DEBUG)logger.info("got docnr "+ret);
  addBlackListedDocNr(ret);
  numTokensInDocs.ensureSize(ret);
  docVectorLengths.ensureSize(ret);
  newDocVectorLengths.ensureSize(ret);
  docVectorLengths.replace(0.0,ret);
  newDocVectorLengths.replace(0.0,ret);
  DocEntry ins = new DocEntry(ret,new ByteArr(uri));
  if(BrilleDefinitions.DEBUG)logger.info("should insert in btree");
  if(BrilleDefinitions.DEBUG)logger.info(ins.toString());
  bt.insertEntry(ins);
  if(BrilleDefinitions.DEBUG)logger.info("done inserting in btree");
  return ret;
}

/**
 * Method for setting the number of tokens in a document. This is
 * usually needed in some compression techniques, and is currently not used.
 *
 * @param lengthTokens - the number of tokens in the document
 * @param docNr - the document number
 */
public void setLengthForDoc(int lengthTokens, int docNr){
  numTokensInDocs.replace(lengthTokens,docNr);
}

/**
 * Getting the uri for the document with the given document number.
 * This is a search in the B-tree based on primary key, and should
 * be fairly efficient.
 *
 * @param docNr
 * @return
 */
public URI getUriForDocNr(int docNr){
  if(BrilleDefinitions.DEBUG)logger.info("should find uri for docnr "+docNr);
  DocEntry searchFor = new DocEntry(docNr,null);
  BTreeHandle bh = bt.searchForEntry(searchFor);
  DocEntry ret = (DocEntry)bh.getEntry();
  bh.release();
  if(ret!=null){
    if(BrilleDefinitions.DEBUG)logger.info("returning "
        +ret.getURI().decodeString());
    return ret.getURI().decodeToURI();
  }
  if(BrilleDefinitions.DEBUG)logger.info("return null");
  return null;
}

/**
 * Method to delete all documents in the document manager.
 *
 * @throws NotImplementedException
 */
public void deleteAllDocs() throws NotImplementedException{
  if(BrilleDefinitions.DEBUG)logger.info("deleting all docs");
  bl.blacklistAll();
```

```
    ft.shutDown();
    bt.deleteAllEntries();
    freeList=new IntArrayList();
    soonFreeList=new IntArrayList();
    ft = new FreeThread(freeList,soonFreeList);
    ft.start();
    newDocNr=0;
    if(BrilleDefinitions.DEBUG)logger.info("done deleting all docs");
  }

  /**
   * Method for deleting a document from the document manager.
   *
   * @param docNr - the number of the document to delete.
   */
  public void deleteDoc(int docNr){
    blackList(docNr);
    //Note: should be handled in a more clever way, but since deletions
    // are not fully supported, this is deferred until later.
    DocEntry de = new DocEntry(docNr,null);
    bt.delete(de);
    soonFreeList.add(docNr);
    if(BrilleDefinitions.DEBUG)logger.info("deleted doc with nr "+docNr);
  }

  public int getNumActiveDocs(){
    return numActiveDocs;
  }


  /**
   * This method should be used as little as possible.
   * It is capable of obtaining the document number assigned to
   * a given uri. It performs a search through the whole B-tree
   * until it finds the correct uri, and returns the document number.
   *
   * @param uri
   * @return
   */
  public int getDocNrForURI(URI uri){
    int ret =-1;
    BTreeHandle bh = bt.getSmallestEntry();
    try{
      DocEntry de = (DocEntry)bh.getEntry();
      while(de!=null){
        if(de.getURI().equals(uri)){
          ret = de.getDocNr();
          break;
        }
        bh.getNext();
        de=(DocEntry)bh.getEntry();
      }
    }finally{
      bh.release();
    }
    return ret;
  }

  /**
   * Methods for setting all rankingvalues in the shadow copy to zero.
   *
   */
  public void resetNewRankingValues() {
    if(BrilleDefinitions.DEBUG)logger.info("resetting new ranking values");
    for(int i=0;i<newDocVectorLengths.size();i++)
      newDocVectorLengths.replace(0.0,i);
  }

  /**
```

```
 * Method called by FixRankingsThread when a cycle of rankingupdates is
 * ended. This is done to ensure that documents will get their correct
 * value when they are added within such a cycle, and to take the square
 * root of all values and switching the shadow copy with the active one.
 *
 */
public synchronized void endFixingRank(){
  if(BrilleDefinitions.DEBUG)logger.info("endFixingRank");
  for(int i=0;i<newDocVectorLengths.size();i++)
    newDocVectorLengths.replace(Math.sqrt(newDocVectorLengths.get(i)),i);
  fixingRank=false;
  for(int i=0;i<doNotUpdateDocs.size();i++)
    newDocVectorLengths.replace(docVectorLengths.get(
        doNotUpdateDocs.get(i)),doNotUpdateDocs.get(i));
  DoubleArrayList tmp = docVectorLengths;
  docVectorLengths=newDocVectorLengths;
  newDocVectorLengths=tmp;
  doNotUpdateDocs=new IntArrayList();
  if(BrilleDefinitions.DEBUG)logger.info("done endFixingRank");
}

/**
 * Method called by FixRankingsThread when a cycle of rankingupdates begins.
 * This is done to ensure that documents will get their correct value when
 * they are added within such a cycle.
 *
 */
public synchronized void startFixingRank(){
  fixingRank=true;
}

/**
 * Method for setting the rankingvalue for the specified document.
 *
 * @param docNr - the document number of the document we want to set the
 *  rankingvalue for
 * @param value - the value we want to set it to
 */
public synchronized void setRankingValue(int docNr, double value){
  if(fixingRank) doNotUpdateDocs.add(docNr);
  docVectorLengths.replace(value,docNr);
}

/**
 * Add the given value to the ranking value for the given document in the
 * shadow copy.
 *
 * @param docNr - the document number where the value should be added
 * @param addval - the value to add
 */
public void addToNewRankingValue(int docNr, double addval){
  newDocVectorLengths.replace(newDocVectorLengths.get(docNr)+addval,docNr);
}

/**
 * Method setting the rankingvalue for the given document to the square root
 * of its current value. This method is called after all parts of the sum
 * in tf-idf is added.
 *
 * @param docNr
 */
public void finishComputationOfRankingValueForDoc(int docNr){
  docVectorLengths.replace(Math.sqrt(docVectorLengths.get(docNr)),docNr);
}

/**
 * Method meant to recover DocManager from failiure in a future
 * implementation.
 *
```

```
   * @return the DocManager recovered
   * @throws NotImplementedException
   */
  public static DocManager recover() throws NotImplementedException{
    throw new NotImplementedException("Recovery of docman is not implemented");
  }

  /**
   * Method to blacklist the given document.
   *
   * @param docNr - the document number of the document to blacklist.
   */
  public void blackList(int docNr){
    if(bl.setBlackList(docNr)) numActiveDocs--;
  }

  /**
   * Method for unblacklisting the given document.
   *
   * @param docNr - the document number to unblacklist
   */
  public void unBlackList(int docNr){
    if(bl.setUnBlackList(docNr))numActiveDocs++;
  }


  public boolean isBlackList(int docNr){
    return bl.isBlackList(docNr);
  }

  private void addBlackListedDocNr(int docNr){
    bl.addBlackListedDocNr(docNr);
  }

  /**
   * Shutting down this document manager, that means stopping
   * all threads it runs.
   *
   */
  public void shutDown(){
    frt.stopRunning();
    ft.shutDown();
    bt.shutDown();
  }


  public double getRankingValueForDoc(int doc){
    return docVectorLengths.get(doc);
  }
}
```

C.5.6 brille.docman.FixRankingsThread

```
package brille.docman;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.btree.BTreeHandle;
import brille.dict.FullBTreeIndex;
import brille.dict.FullBTreeIndexLeafEntry;
import brille.utils.ByteArr;

/**
 * This class is used in DocManager for incremental indexes.
 * It is responsible for keeping the ranking values fairly
 * up-to-date. It sleeps for a while, before it traverses
```

```
 * the complete index, and updates the lengths of all documents
 * accordingly. This update process does of course not affect
 * ongoing searches.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FixRankingsThread extends Thread{
  protected final Logger logger = Logger.getLogger(getClass());

  private DocManager docMan;
  private FullBTreeIndex index;
  private boolean running;

  /**
   * The constructor taking the docman that keeps the rankingvalues
   * and the index to traverse as arguments.
   *
   * @param docMan - the document manager that keeps the rankingvalues.
   * @param index - the index to traverse.
   */
  public FixRankingsThread(DocManager docMan, FullBTreeIndex index){
    this.docMan=docMan;
    this.index=index;
  }

  /**
   * Stopping this thread when the application shuts down.
   *
   */
  public void stopRunning(){
    running=false;
  }


  public void run(){
    running = true;
    while(running){
      if(BrilleDefinitions.DEBUG) logger.info("THREAD RUNNING");

      docMan.startFixingRank();
      if(BrilleDefinitions.DEBUG) logger.info("starting fixing rank");
      BTreeHandle handle = index.getIndexIterator();
      FullBTreeIndexLeafEntry entry =
        (FullBTreeIndexLeafEntry)handle.getEntry();
      while(entry!=null && running){
        int docNr = entry.getDocNr();
        if(docNr!=-1){
          logger.error("the doc-nr for first entry of kind is not -1");
          throw new Error("error in btree found by ranker");
        }
        double numdocs = (double)entry.getLocations()[0];
        ByteArr currterm = entry.getTerm();
        handle.getNext();
        entry = (FullBTreeIndexLeafEntry)handle.getEntry();
        while(entry!=null && entry.getTerm().equals(currterm)){
          if(BrilleDefinitions.DEBUG)
            logger.info("processing entry "+entry.toString());
          docMan.addToNewRankingValue(entry.getDocNr(),
              Math.pow(((double)entry.getLocations().length)*
                    Math.log(((double)docMan.getNumActiveDocs())/numdocs),2.0));
          handle.getNext();
          entry = (FullBTreeIndexLeafEntry)handle.getEntry();
        }
      }
      handle.release();
      if(BrilleDefinitions.DEBUG) logger.info("ending fixing rank");
      docMan.endFixingRank();
      docMan.resetNewRankingValues();
      try{
```

```
        if(running)
          Thread.sleep(
              BrilleDefinitions.SLEEP_TIME_FOR_RANKINGS_UPDATE_THREAD);
      }catch(InterruptedException ie){
        logger.error("could not sleep in fixrankingsthread",ie);
      }
    }
  }
}
```

C.5.7 brille.docman.FreeThread

```
package brille.docman;

import org.apache.log4j.Logger;

import brille.utils.IntArrayList;

/**
 * This class will be removed in future improvements, but its current
 * role is to make sure that document numbers of deleted documents
 * will not be reused at once, because this may lead to problems regarding
 * old documents that are still reflected in the index a search reads.
 * Its current implementation assumes that no search takes more than 5 seconds,
 * which is a fairly crappy assumption.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FreeThread extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private IntArrayList freelist;
  private IntArrayList soonFreelist;
  private boolean run;

  /**
   * Constructor taking the lists to move document numbers between as
   * arguments.
   *
   * @param freelist - the list of document numbers that can be reused.
   * @param soonFreelist - the list of document numberst that can be reused
   * soon.
   */
  public FreeThread(IntArrayList freelist, IntArrayList soonFreelist){
    this.freelist=freelist;
    this.soonFreelist=soonFreelist;
    run=true;
  }

  /**
   * Stopping this thread when the application shuts down.
   *
   */
  public void shutDown(){
    run=false;
  }


  public void run(){
    try{
      while(run){
        Thread.sleep(5000);
        if(soonFreelist.size()!=0){
          int move = soonFreelist.size();
          Thread.sleep(1000);
          for(int i=0;i<move;i++){
```

```
                freelist.add(soonFreelist.remove(0));
              }
            }
          }
        }catch(Exception e){
          logger.error("exception in freethread",e);
          logger.warn("The thread is shutting down");
        }
      }
    }
```

C.5.8 brille.docman.InternalDocEntry

```
package brille.docman;

import brille.btree.InternalEntry;
import brille.btree.Entry;
import brille.buffering.Buffer;

/**
 * This class represents a DocEntry as an internal entry in the B-tree.
 * Most methods here are inherited from brille.btree.InternalEntry and
 * brille.btree.Entry, and are thus not commented any further.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class InternalDocEntry extends InternalEntry{
  private int key;

  /**
   * Constructor taking the key, which is a document number, and
   * a pointer to the correct node in the subtree.
   *
   * @param key - document number
   * @param pointer - pointer to correct subtree node.
   */
  public InternalDocEntry(int key, int pointer){
    super(pointer);
    this.key=key;
  }

  /**
   * Constructor used by make() in InternalDocEntryFactory.
   *
   */
  public InternalDocEntry(){
    this(-1,-1);
  }

  /**
   * Getting the document number that is the key.
   *
   * @return - the document number in this key
   */
  public int getKey(){
    return key;
  }

  public boolean keyIsMax() {
    return key==Integer.MAX_VALUE;
  }


  public void setKeyToMax() {
    key=Integer.MAX_VALUE;
  }
```

```
  public int writeKey(Buffer buf, int pos){
    buf.putInt(key,pos);
    return 4;
  }

  public int write(Buffer buf,int pos) {
    writeKey(buf,pos);
    buf.putInt(pointer,pos+4);
    return 8;
  }

  public int readInKey(Buffer buf, int pos){
    key = buf.getInt(pos);
    return 4;
  }

  public int readIn(Buffer buf,int pos) {
    readInKey(buf, pos);
    pointer=buf.getInt(pos+4);
    return 8;
  }

  public int keyLength(){
    return 4;
  }

  public int length() {
    return 8;
  }

  public int compareTo(Entry e) {
    if(!isKeyCompatible(e))
      throw new
        IllegalArgumentException("not valid to compare different entries");
    if(e instanceof InternalDocEntry)
      return key-((InternalDocEntry)e).getKey();
    else return key -((DocEntry)e).getDocNr();
  }

  public String toString(){
    return key+","+pointer;
  }

  public String getKeyAsString(){
    return ""+key;
  }

  public boolean isKeyCompatible(Entry en){
    return en instanceof InternalDocEntry || en instanceof DocEntry;
  }

  public void incEntry(Entry en) {
    throw new IllegalArgumentException("not allowed to inc");
  }

  public boolean decEntry(Entry en) {
    throw new IllegalArgumentException("not allowed to dec");
  }

  /**
   * Method for setting the key to the key the argument has.
   * This is used to change maxkey.
   *
   * @param newMax - a DocEntry with the same key we want this one to have
   */
  public void setKey(DocEntry newMax) {
    key = newMax.getDocNr();
  }
}
```

C.5.9 brille.docman.InternalDocEntryFactory

```
package brille.docman;

import brille.btree.Entry;
import brille.btree.InternalEntry;
import brille.btree.InternalEntryFactory;


/**
 * This class is the factory for InternalDocEntry.
 * It implements the methods inherited from brille.btree.InternalEntryFactory
 * and brille.btree.EntryFactory, and these are thus not commented on here.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class InternalDocEntryFactory extends InternalEntryFactory{
  public InternalDocEntry make(){
    return new InternalDocEntry();
  }

  public InternalEntry make(Entry maxKey, int pointer) {
    return new InternalDocEntry(
        Integer.parseInt(maxKey.getKeyAsString()),pointer);
  }

  public InternalEntry makeEntryWithSameKey(Entry newMax) {
    if(newMax instanceof InternalDocEntry) return (InternalDocEntry)newMax;
    else if(newMax instanceof DocEntry){
      InternalDocEntry ret = new InternalDocEntry();
      ret.setKey((DocEntry)newMax);
      return ret;
    }
    else throw new IllegalArgumentException("not valid key");
  }

}
```

C.5.10 brille.docman.StaticDocManager

```
package brille.docman;

import java.net.URI;
import java.util.HashMap;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.NotImplementedException;
import brille.btree.BTree;
import brille.btree.BTreeHandle;
import brille.buffering.BuffPool;
import brille.utils.ByteArr;
import brille.utils.DoubleArrayList;
import brille.utils.IntArrayList;

/**
 * This class is the document manager for indexes that are
 * inherently static, like the hierarchic and remerge-methods
 * in brille. These differs from incremental ones in that it
 * is more natural for the indexing thread to update the ranking
 * values as new documents are added. Thus, we do not need a
 * fixrankingthread here, and methods for switching to new rankingvalues
 * differ. Otherwise, the overall thought behind this class is similar
 * to DocManager. The mapping between document numbers and uris are
 * stored in a b-tree with buffering, and the ranking values for
 * all documents are stored in memory.
```

```
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class StaticDocManager implements DocMan{

  protected final Logger logger = Logger.getLogger(getClass());

  private DoubleArrayList docVectorLengths;
  private DoubleArrayList newDocVectorLengths;
  private IntArrayList freeList;
  private IntArrayList soonFreeList;
  private int numActiveDocs;

  private IntArrayList numTokensInDocs;
  private BlackList bl;
  private BTree bt;
  private BuffPool bp;
  private FreeThread ft;


  private int newDocNr;

  /**
   * Default constructor
   *
   */
  public StaticDocManager() {
    newDocNr=1;
    numActiveDocs=0;
    docVectorLengths=new DoubleArrayList();
    newDocVectorLengths=new DoubleArrayList();
    numTokensInDocs=new IntArrayList();
    freeList = new IntArrayList();
    soonFreeList = new IntArrayList();
    bl = new BlackList();
    ft = new FreeThread(freeList,soonFreeList);
    ft.start();
    if(BrilleDefinitions.DEBUG) logger.debug("staticdocman created");
  }

  /**
   * Method for finding the disksize used by this document manager.
   *
   * @return - the number of bytes used by the document manager, in a long
   */
  public long diskSize(){
    return bt.diskSize();
  }

  /**
   * Setting the bufferpool this class should use.
   *
   * @param bp
   */
  public void setBufferPool(BuffPool bp){
    this.bp=bp;
    makeBTree();
  }

  public BTree getBTree(){
    return bt;
  }


  private void makeBTree(){
    if(BrilleDefinitions.DEBUG) logger.debug("making btree");
    bt=new BTree(BrilleDefinitions.INDEX_PATH+"/"+
        BrilleDefinitions.DOCMAN_FILE_NAME,bp,
        BrilleDefinitions.SLEEP_TIME_FOR_DOCMAN_UPDATE_THREAD,
```

```
      new InternalDocEntryFactory(), new DocEntryFactory());
  if(BrilleDefinitions.DEBUG) logger.debug("btree created");
}


private synchronized int getDocNr(){
  if(freeList.size()>0) return freeList.remove(freeList.size()-1);
  return newDocNr++;
}

/**
 * Method for adding a new document to this document manager.
 * The documenthas an initial length of zero. The mapping between
 * docnr and uri is inserted into the B-tree, and the document number
 * this document got is returned.
 *
 * @param uri - the uri of the document to add
 * @return - the document number assigned to this document
 */
public int addDoc(URI uri){
  if(BrilleDefinitions.DEBUG)
    logger.debug("should add doc "+uri.toString());
  int ret = getDocNr();
  if(BrilleDefinitions.DEBUG) logger.debug("got docnr "+ret);
  addBlackListedDocNr(ret);
  if(BrilleDefinitions.DEBUG) logger.debug("have added blacklisted docnr");
  numTokensInDocs.ensureSize(ret);
  newDocVectorLengths.ensureSize(ret);
  newDocVectorLengths.replace(0.0,ret);
  if(BrilleDefinitions.DEBUG) logger.debug("have ensured sizes");
  DocEntry ins = new DocEntry(ret,new ByteArr(uri));
  bt.insertEntry(ins);
  if(BrilleDefinitions.DEBUG) logger.debug("have inserted entry in btree");
  return ret;
}

/**
 * Method for setting the number of tokens in a document. This is
 * usually needed in some compression techniques, and is currently not used.
 *
 * @param lengthTokens - the number of tokens in the document
 * @param docNr - the document number
 */
public void setLengthForDoc(int lengthTokens, int docNr){
  numTokensInDocs.replace(lengthTokens,docNr);
}

/**
 * Getting the uri for the document with the given document number.
 * This is a search in the B-tree based on primary key, and should
 * be fairly efficient.
 *
 * @param docNr
 * @return
 */
public URI getUriForDocNr(int docNr){
  DocEntry searchFor = new DocEntry(docNr,null);
  BTreeHandle bh = bt.searchForEntry(searchFor);
  DocEntry ret = (DocEntry)bh.getEntry();
  bh.release();
  if(ret!=null) return ret.getURI().decodeToURI();
  return null;
}

/**
 * Method to delete all documents in the document manager.
 *
 * @throws NotImplementedException
 */
```

```
public void deleteAllDocs() throws NotImplementedException{
  bl.blacklistAll();
  ft.shutDown();
  bt.deleteAllEntries();
  freeList=new IntArrayList();
  soonFreeList=new IntArrayList();
  ft = new FreeThread(freeList,soonFreeList);
  ft.start();
  newDocNr=0;
}

/**
 * Method for deleting a document from the document manager.
 *
 * @param docNr - the number of the document to delete.
 */
public void deleteDoc(int docNr){
  blackList(docNr);
  // Note: should be handled in a more clever way, but since deletions
  // are not fully supported, this is deferred until later.
  DocEntry de = new DocEntry(docNr,null);
  bt.delete(de);
  soonFreeList.add(docNr);
}


public int getNumActiveDocs(){
  return numActiveDocs;
}

/**
 * This method should be used as little as possible.
 * It is capable of obtaining the document number assigned to
 * a given uri. It performs a search through the whole B-tree
 * until it finds the correct uri, and returns the document number.
 *
 * @param uri
 * @return
 */
public int getDocNrForURI(URI uri){
  int ret =-1;
  BTreeHandle bh = bt.getSmallestEntry();
  try{
    DocEntry de = (DocEntry)bh.getEntry();
    while(de!=null){
      if(de.getURI().equals(uri)){
        ret = de.getDocNr();
        break;
      }
      bh.getNext();
      de=(DocEntry)bh.getEntry();
    }
  }finally{
    bh.release();
  }
  return ret;
}

/**
 * Methods for setting all rankingvalues in the shadow copy to zero.
 *
 */
public void resetNewRankingValues() {
  for(int i=0;i<newDocVectorLengths.size();i++)
    newDocVectorLengths.replace(0.0,i);
}

/**
 * Add the given value to the ranking value for the given document in the
```

```
 * shadow copy.
 *
 * @param docNr - the document number where the value should be added
 * @param addval - the value to add
 */
public void addToNewRankingValue(int docNr, double addval){
  newDocVectorLengths.ensureSize(docNr);
  newDocVectorLengths.replace(newDocVectorLengths.get(docNr)+addval,docNr);
}

/**
 * This finishes the computation of all ranking values in the shadow copy,
 * by replacing them with their square root. Then it switches all old values
 * with these new ones. This reflects the way this class differs from
 * DocManager. Here, it is the indexing thread that performs the ranking
 * computations, because this is, in my opinion the only sound way to do it.
 * Hence, we know that all the values should switch, and the problem with
 * keeping track of which documents that doesn't need to be updated is
 * eliminated.
 *
 */
public synchronized void finishComputationOfAllNewRankingValuesAndSwitch(){
  for(int i=0;i<newDocVectorLengths.size();i++){
    if(newDocVectorLengths.get(i)>0.0)
      newDocVectorLengths.replace(Math.sqrt(newDocVectorLengths.get(i)),i);
    else if(docVectorLengths.size()>i)
      newDocVectorLengths.replace(docVectorLengths.get(i),i);
  }
  DoubleArrayList tmp = docVectorLengths;
  docVectorLengths=newDocVectorLengths;
  newDocVectorLengths=tmp;
  newDocVectorLengths.ensureSize(docVectorLengths.size()-1);
}


/**
 * Method used when only one inmempartialindex is flushed and ranked. This
 * method is capable of updating only the given document lengths. This is
 * typically needed in the hierarchic method because two merges can go on
 * at the same time. If that was to be supported in other ways, several
 * methods would have to be synchronized causing a slower execution.
 *
 * @param theNewRankingValues
 */
public synchronized void setRankingValues(
    HashMap<Integer,Double> theNewRankingValues){
  for(int i :theNewRankingValues.keySet()){
    if(docVectorLengths.size()<=i)docVectorLengths.ensureSize(i);
    docVectorLengths.replace(Math.sqrt(theNewRankingValues.get(i)),i);
  }
}

/**
 * Method meant to recover DocManager from failiure in a future
 * implementation.
 *
 * @return the DocManager recovered
 * @throws NotImplementedException
 */
public static DocManager recover() throws NotImplementedException{
  throw new NotImplementedException("Recovery of docman is not implemented");
}

/**
 * Method to blacklist the given document.
 *
 * @param docNr - the document number of the document to blacklist.
 */
public void blackList(int docNr){
```

```
    if(bl.setBlackList(docNr)) numActiveDocs--;
  }

  /**
   * Method for unblacklisting the given document.
   *
   * @param docNr - the document number to unblacklist
   */
  public void unBlackList(int docNr){
    if(bl.setUnBlackList(docNr))numActiveDocs++;
  }


  public boolean isBlackList(int docNr){
    return bl.isBlackList(docNr);
  }

  private void addBlackListedDocNr(int docNr){
    bl.addBlackListedDocNr(docNr);
  }

  /**
   * Shutting down this document manager, that means stopping
   * all threads it runs.
   *
   */
  public void shutDown(){
    ft.shutDown();
    bt.shutDown();
  }


  public double getRankingValueForDoc(int doc){
    return docVectorLengths.get(doc);
  }
}
```

C.6 brille.inv

C.6.1 brille.inv.IndexEntry

```java
package brille.inv;

import brille.BrilleDefinitions;

/**
 * This class represents an entry into an inverted list, and is
 * the object-representation of the structure that usually resides on disk.
 * It consists of a document nr which contains the term, a list of
 * locations, and to enable more fancy ranking in the future, it also contains
 * a byte-array, where each location can be further described.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class IndexEntry implements Comparable<IndexEntry>{


  private int docNr;
  private int[] locations;
  private byte[] descr;

  public IndexEntry(int docNr, int[] locations, byte[] descr){
    this.docNr=docNr;
    this.locations=locations;
    this.descr=descr;
  }


  /**
   * @return the document nr
   */
  public int getDocNr(){
    return docNr;
  }

  /**
   * @return the list of locations
   */
  public int[] getLocations(){
    return locations;
  }

  /**
   * @return the array of descriptions
   */
  public byte[] getDescriptions(){
    return descr;
  }

  /**
   * setter for locations within doc
   * @param locations
   */
  public void setLocations(int[] locations){
    this.locations=locations;
  }

  /**
   * setter for descriptions
   * @param descr
   */
  public void setDescr(byte[] descr){
    this.descr=descr;
```

```
}

/**
 * Compare this IndexEntry to another, used when
 * sorting entries
 * @param other - the IndexEntry to compare with
 * @return an int. Negative is this one is smaller, and
 *     positive if this one is larger.
 */
public int compareTo(IndexEntry other) {
  if (this.docNr < other.docNr) {
    return -1;
  }
  else if (this.docNr > other.docNr) {
    return 1;
  }
  if(locations==null || locations.length==0){
    if(other.locations!=null && other.locations.length!=0){
      return -1;
    }
    else{
      int ret =checkDescr(other,0);
      return ret;
    }
  }
  else if(other.locations==null || other.locations.length==0){
    return 1;
  }
  if(locations.length<other.locations.length){
    return -1;
  }
  else if(locations.length>other.locations.length){
    return 1;
  }
  for(int i=0;i<locations.length;i++){
    if(locations[i]<other.locations[i]){
      return 1;
    }
    else if(locations[i]>other.locations[i]){
      return -1;
    }
  }
  int ret =checkDescr(other,0);
  return ret;
}

private int checkDescr(IndexEntry other, int length) {
  if(length==0)return 0;
  if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
    if(descr.length<other.descr.length) return -1;
    else if(descr.length>other.descr.length) return 1;
    for(int i=0;i<descr.length;i++){
      if(descr[i]<other.descr[i]) return -1;
      else if(descr[i]>other.descr[i]) return 1;
    }
  }
  return 0;
}

public boolean equals(Object other){
  if(other instanceof IndexEntry){
    IndexEntry t = (IndexEntry)other;
    return 0==compareTo(t);
  }
  else return false;
}

/**
 * @return a string representation of this IndexEntry
```

```
   */
  public String toString(){
    String ret = "docnr: "+docNr+"\nLocations: ";
    for(int i=0;i<locations.length;i++) ret+=(locations[i]+" ");
    return ret;
  }
}
```

C.6.2 brille.inv.InvListIterator

```
package brille.inv;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.SearchResultHandle;
import brille.buffering.Buffer;
import brille.buffering.BuffPool;
import brille.dict.SortedListDictEntry;

/**
 * This is the SearchResultHandle for normal inverted lists.
 * Such an object recieves information about the number of occurrences
 * and where to read them from from the dictionary, and provides an
 * iterator going through all those entries, and returning null when
 * there no more entries. This provides a way for searching without
 * using too much memory.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class InvListIterator extends SearchResultHandle{

  protected final Logger logger = Logger.getLogger(getClass());

  private BuffPool bp;
  private int currpart;
  private int currpos;
  private Buffer currBuffer;
  private int numEntriesLeft;
  private int numres;
  private int totocc;
  private int filenum;
  private IndexEntry currEntry;
  private int returnRes;
  private boolean returnOther;

  /**
   * The only constructor
   * @param e - The entry from the dictionary giving the number of
   *     occurrences and pointer into the file
   * @param bp - The bufferpool where buffers containing the inverted list
   *     can be pinned
   * @param invNum - the filenumber of the inverted file in the bufferpool.
   */
  public InvListIterator(SortedListDictEntry e, BuffPool bp, int invNum) {
    returnOther=false;
    if(e==null){
      numEntriesLeft=0;
      currEntry=null;
    }
    else{
      this.bp=bp;
      currpart = (int)(e.getPointer()/
          ((long)BrilleDefinitions.BUFFER_SIZE));
      currpos = (int)(e.getPointer()-
          (((long)currpart)*((long)BrilleDefinitions.BUFFER_SIZE)));
      if(BrilleDefinitions.DEBUG) logger.info("pinning buffer in invit");
```

```
    currBuffer = bp.pinBuffer(currpart,invNum,true,false);
    numEntriesLeft = e.getCnt();
    numres = numEntriesLeft;
    totocc = e.getNumoccs();
    filenum = invNum;
    if(BrilleDefinitions.DEBUG)
      logger.info("Have made invit with numres = "+numres);
    if(BrilleDefinitions.DEBUG)
      logger.info("currpart is "+currpart+
          "and currpos is "+currpos);
    next();
  }
}

public InvListIterator(SortedListDictEntry e, BuffPool bp,
    int invNum, int toReturn) {
  this(e,bp,invNum);
  this.returnRes = toReturn;
  returnOther=true;
}

/**
 * Method for retrieving the current entry in the iterator.
 * This entry will stay the same until next() is called.
 * @return The current entry in the iterator
 */
public IndexEntry getEntry() {
  return currEntry;
}


private int readInt(){
  if(BrilleDefinitions.BUFFER_SIZE-currpos<4){
    bp.unpinBuffer(currBuffer,true,false);
    currpart++;currpos=0;
    currBuffer= bp.pinBuffer(currpart,filenum,true,false);
  }
  currpos+=4;
  return currBuffer.getInt(currpos-4);
}

private byte readByte(){
  if(BrilleDefinitions.BUFFER_SIZE==currpos){
    bp.unpinBuffer(currBuffer,true,false);
    currpart++;
    currpos=0;
    currBuffer= bp.pinBuffer(currpart,filenum,true,false);
  }
  return currBuffer.getByte(currpos++);
}

/**
 * Method for moving the iterator along the list, so that the next entry
 * will become current.
 */
public void next() {
  if(numEntriesLeft==0){
    currEntry=null;
    return;
  }
  int docNr = readInt();
  int num = readInt();
  int[] locs = new int[num];
  for(int i=0;i<num;i++) locs[i]=readInt();
  byte[] descr = null;
  if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
    descr = new byte[num];
    for(int i=0;i<num;i++) descr[i]=readByte();
  }
```

```
    currEntry = new IndexEntry(docNr,locs,descr);
    if(BrilleDefinitions.DEBUG) logger.info("have read new entry "+
        currEntry.toString());
    numEntriesLeft--;
  }

  /**
   * Getting the nr of results in the current inverted list.
   * @return -nr of entries in the iterator in total
   */
  public int getNrOfResults() {
    return numres+(returnOther?returnRes:0);
  }

  /**
   * Getting the total number of occurrences of the term in this
   * inverted file. It is the sum of the number of occurrences in all
   * docs.
   * @return -total number of occurrences of term
   */
  public int getNrOcc() {
    return totocc;
  }

  /**
   * This is an important method. When one has finished using this iterator,
   * this method should be called, to make sure the buffer used in this
   * iterator is unpinned.
   */
  public void release() {
    bp.unpinBuffer(currBuffer,true,false);
  }

}
```

C.6.3 brille.inv.InvListMergerHeap

```
package brille.inv;

import java.util.ArrayList;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;

/**
 * This class is a sorted iterator of several several inverted
 * lists for the same term. It takes a list of MergeInvListIterators,
 * builds a heap of them which is maintained throughout the lifetime
 * of this iterator. The heap assumes that no inverted list for this term
 * contains the same document number.
 *
 * This class is used when different inverted files are merged, both in
 * RemergeIndexMaster and HierarchicIndexMaster. In the hiererarchic one,
 * it is used for both searching and index merging.
 *
 * This class could not have been replaced with java.util.PriorityQueue<T>
 * because it is a heap of iterators, not of simple objects.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class InvListMergerHeap {

  protected final Logger logger = Logger.getLogger(getClass());

  private ArrayList<MergeInvListIterator> l;
```

```
  /**
   * Constructor taking a list of MergeInvListIterators.
   * @param l
   */
  public InvListMergerHeap(ArrayList<MergeInvListIterator> l){
    this.l=l;
    buildHeap();
  }


  private void buildHeap(){
    for(int i=(l.size()/2)-1;i>=0;i--) minHeapify(i);
  }

  private void minHeapify(int ind){
    if(BrilleDefinitions.DEBUG) logger.info("minheapify with i="+ind);
    int le = ind*2+1;
    int ri = ind*2+2;
    if(le>=l.size())return;
    int min= ri>=l.size()?le:
      l.get(le).getEntry().getDocNr()<l.get(ri).getEntry().getDocNr()?
          le:ri;
    if(l.get(ind).getEntry().getDocNr()>l.get(min).getEntry().getDocNr()){
      MergeInvListIterator tmp = l.get(ind);
      l.set(ind,l.get(min));
      l.set(min,tmp);
      minHeapify(min);
    }
  }

  /**
   * Getting the next entry. Once this method has returned an entry, the
   * same entry will never be returned from this heap again.
   *
   * @return - the current entry
   */
  public IndexEntry getEntry(){
    if(BrilleDefinitions.DEBUG) logger.info("in getentry");
    IndexEntry ret = l.get(0).getEntry();
    if(BrilleDefinitions.DEBUG)
      logger.info("the entry i should return is "+ret.toString());
    l.get(0).next();
    if(l.get(0).getEntry()==null){
      l.get(0).release();
      if(l.size()>0)l.set(0,l.get(l.size()-1));
      l.remove(l.size()-1);
    }
    if(l.size()>0)minHeapify(0);
    return ret;
  }


  /**
   * Testing whether the heap is empty.
   * @return
   */
  public boolean isEmpty(){
    return l.size()==0;
  }
}
```

C.6.4 brille.inv.MergeInvListIterator

```
package brille.inv;

import brille.BrilleDefinitions;
import brille.buffering.Buffer;
```

```java
import brille.buffering.BuffPool;
import brille.dict.MergeSortedListDictEntry;

/**
 * This class serves much the same purpose as InvListIterator, but
 * differs in the way that it can serve different inverted files,
 * needed when merging inverted files. The two classes should
 * probably have been merged, but that is currently postponed:)
 * No further comments will be provided here. Look at InvListIterator.
 *
 * @author Truls A. Ã¸Bjrklund
 * @see InvListIterator
 */
public class MergeInvListIterator{

  private int currpos;
  private int currpart;
  private Buffer currBuffer;
  private IndexEntry currEntry;
  private int numEntriesLeft;
  private BuffPool bp;
  private int filenum;

  public MergeInvListIterator(MergeSortedListDictEntry e, BuffPool bp){
    this.bp=bp;
    currpart = (int)(e.getPointer()/((long)BrilleDefinitions.BUFFER_SIZE));
    currpos = (int)(e.getPointer()-
        (((long)currpart)*((long)BrilleDefinitions.BUFFER_SIZE)));
    currBuffer = bp.pinBuffer(currpart,e.getInvNum(),true,false);
    numEntriesLeft = e.getCnt();
    filenum = e.getInvNum();
    next();
  }


  public IndexEntry getEntry() {
    return currEntry;
  }

  private int readInt(){
    if(BrilleDefinitions.BUFFER_SIZE-currpos<4){
      bp.unpinBuffer(currBuffer,true,false);
      currpart++;currpos=0;
      currBuffer= bp.pinBuffer(currpart,filenum,true,false);
    }
    currpos+=4;
    return currBuffer.getInt(currpos-4);
  }

  private byte readByte(){
    if(BrilleDefinitions.BUFFER_SIZE==currpos){
      bp.unpinBuffer(currBuffer,true,false);
      currpart++;currpos=0;
      currBuffer= bp.pinBuffer(currpart,filenum,true,false);
    }
    return currBuffer.getByte(currpos++);
  }

  public void next() {
    if(numEntriesLeft==0){
      currEntry=null;
      return;
    }
    int docNr = readInt();
    int num = readInt();
    int[] locs = new int[num];
    for(int i=0;i<num;i++) locs[i]=readInt();
    byte[] descr = null;
    if(BrilleDefinitions.NR_OF_BITS_FROM_DESCR>0){
```

```
      descr = new byte[num];
      for(int i=0;i<num;i++) descr[i]=readByte();
    }
    currEntry = new IndexEntry(docNr,locs,descr);
    numEntriesLeft--;
  }

  public boolean isEmpty(){
    return numEntriesLeft==0;
  }

  public void release() {
    bp.unpinBuffer(currBuffer,true,false);
  }
}
```

C.7 brille.locking

C.7.1 brille.locking.LockManager

```
package brille.locking;

import java.util.ArrayList;
import java.util.concurrent.locks.ReentrantReadWriteLock;

/**
 * This class maintain control of all locks for a B-tree,
 * but only when brille.buffering.BufferPool is used, not the
 * new version implemented in brille.buffering.NewBufferPool.
 * It uses the ReentrantReadWriteLock from the jdk for each
 * of the locks.
 *
 * @author Truls A. Ã¸Bjrklund
 */

public class LockManager {
  private ArrayList<ReentrantReadWriteLock> locks;

  /**
   * Default constructor
   *
   */
  public LockManager(){
    locks = new ArrayList<ReentrantReadWriteLock>();
  }

  private synchronized void ensureEnoughLocks(int part){
    for(int i=locks.size();i<=part;i++){
      //true says fair.
      locks.add(new ReentrantReadWriteLock(true));
    }
  }

  /**
   * Method for obtaining a read lock.
   * @param part - the part one wants to read lock.
   */
  public void readLock(int part){
    if(part+1>=locks.size()) ensureEnoughLocks(part+1);
    locks.get(part+1).readLock().lock();
  }

  /**
   * Method for obtaining a write lock.
   * @param part - the part one wants to write lock.
   */
  public void writeLock(int part){
    if(part+1>=locks.size()) ensureEnoughLocks(part+1);
    locks.get(part+1).writeLock().lock();
  }

  /**
   * Method for releasing a read lock.
   * @param part - the part one wants to read unlock.
   */

  public void writeUnlock(int part){
    locks.get(part+1).writeLock().unlock();
  }

  /**
   * Method for releasing a write lock.
```

```
   * @param part - the part one wants to write unlock.
   */
  public synchronized void readUnlock(int part){
    locks.get(part+1).readLock().unlock();
  }
}
```

C.8 brille.utils

C.8.1 brille.utils/AlternativeGOV2FileParser

```
package brille.utils;

import java.io.BufferedReader;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.nio.CharBuffer;
import java.util.ArrayList;
import java.util.HashMap;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.dict.ExtendableDocTermEntry;

/**
 * This class does the same thing as GOV2FileParser, except that it
 * supports that several threads can retrieve parsed documents. It is therefore
 * used by the B-tree, which supports several feeding threads.
 *
 * Most of this implementation works just like GOV2FileParser, and it is thus
 * not commented here.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class AlternativeGOV2FileParser extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private BufferedReader in;
  private boolean doneReading;
  private Document[] cachedDocs;
  private int currentnr;
  private int nextPosToRead;
  private int nextPosToWrite;
  private boolean running;
  private Object nextFileWaiter;
  private Object waitForDoneReadingObj;
  private int numDocsRead;

  //variables for timing a baseline run
  private long timeSpent;
  private long startTime;
  private long termsIndexed;


  /**
   * Default constructor
   *
   */
  public AlternativeGOV2FileParser(){
    in=null;
    doneReading = true;
    cachedDocs = new Document[5];
    currentnr = 0;
    nextPosToRead = 0;
    nextPosToWrite = 0;
    nextFileWaiter = new Object();
    waitForDoneReadingObj = new Object();
    running = true;
    numDocsRead=0;
    termsIndexed=0;
```

```
    timeSpent=0;
  }



  /**
   * Method for testing whether this reader has shut down.
   *
   * @return whether or not this reader has shut down
   */
  public boolean isStopped(){
    return !running;
  }

  private void nextFile(BufferedReader t){
    synchronized(nextFileWaiter){
      in = t;
      doneReading = false;
      if(BrilleDefinitions.BTREE_READING_DEBUG)
        logger.info("notifying for next file");
      nextFileWaiter.notify();

    }
  }

  private void waitForNextFile(){
    synchronized(nextFileWaiter){
      try{
        if(BrilleDefinitions.BTREE_READING_DEBUG)
          logger.info("waiting for next file");
        if(in==null){
          if(BrilleDefinitions.BTREE_READING_DEBUG)
            logger.info("actually waiting for next file");
          nextFileWaiter.wait();
        }
        else if(BrilleDefinitions.BTREE_READING_DEBUG){
          logger.info("Does not have to wait because in!=null");
        }
        if(BrilleDefinitions.BTREE_READING_DEBUG)
          logger.info("done waiting for next file");

      }
      catch(InterruptedException ie){
        logger.error(ie);
        System.exit(1);
      }
    }
  }


  public void setNewFile(BufferedReader t){
    if(t==null) throw new IllegalArgumentException("cannot add a null-file");
    nextFile(t);
  }

  public void waitForDoneReading(){
    synchronized(waitForDoneReadingObj){
      try{
        if(!doneReading) waitForDoneReadingObj.wait();
      }catch(InterruptedException ie){
        ie.printStackTrace();
        logger.error(ie);
      }
    }
  }

  private void setDoneReading(){
    synchronized(waitForDoneReadingObj){
      doneReading=true;
```

```
      in = null;
      waitForDoneReadingObj.notify();
    }
  }


  /**
   * Getter for the next document from this reader. If the reader has not yet
   * parsed the next document, the calling thread will have to wait.
   *
   * @return the next document from this reader, or null if no more docs are
   *  available
   */
  public synchronized Document getNextDocument(){
    if(currentnr==0 && doneReading) return null;
    while(currentnr<1 && !doneReading){
      try{
        wait();
      }
      catch(InterruptedException ie){
        logger.error(ie);
        System.exit(1);
      }
    }
    if(currentnr==0 && doneReading) return null;
    Document ret = cachedDocs[nextPosToRead++];
    if(nextPosToRead>=cachedDocs.length) nextPosToRead-=cachedDocs.length;
    currentnr--;
    if(currentnr==4){
      if(BrilleDefinitions.BTREE_READING_DEBUG)
        logger.debug("notifying reader.");
      notifyAll();
    }
    return ret;
  }

  public void run(){
    try{
    while(running){
      if(BrilleDefinitions.BASELINERUN)startTime = System.nanoTime();
      CharBuffer buff =
        CharBuffer.allocate(BrilleDefinitions.MAX_TERM_LENGTH+4);
      CharBuffer intagBuff = CharBuffer.allocate(400);
      CharBuffer docNr = CharBuffer.allocate(50);
      boolean skiptag=false;
      boolean addtodocnr = false;
      boolean skip = false;
      boolean intag = false;
      boolean inword = false;
      boolean skipword = false;
      boolean maybeskipnext = false;

      char[] curr = new char[65536];

      int room = 0;
      int i =0;
      int read = 0;
      outer: while(!doneReading){
        HashMap<ByteArr, ExtendableDocTermEntry> docterms =
          new HashMap<ByteArr,ExtendableDocTermEntry>();
        ArrayList<ByteArr> terms = new ArrayList<ByteArr>();
        int termnr=1;
        while(true){
          if(i==read){
            i = 0;
            if(BrilleDefinitions.BASELINERUN)
              timeSpent += (System.nanoTime()-startTime);
            try{
              read = in.read(curr);
```

```
}
catch(IOException ioe){
  logger.error(ioe);
  System.exit(1);
}
if(BrilleDefinitions.BASELINERUN) startTime = System.nanoTime();
if(read < 1){
  try{
    cachedDocs[nextPosToWrite++] =
      new Document(new URI(docNr.toString()),docterms,
          room,termnr-1,terms);
    if(nextPosToWrite>=cachedDocs.length)
      nextPosToWrite-=cachedDocs.length;
  }
  catch(URISyntaxException use){
    logger.error(use);
    System.exit(1);
  }

  numDocsRead++;
  if(BrilleDefinitions.OUTPUT_DISK_STATISTICS &&
      (numDocsRead%BrilleDefinitions.OUTPUT_INTERVAL)==0){
    System.out.println("Have read "+numDocsRead+" after "+
        System.nanoTime()+" nanoseconds");
  }
  if(BrilleDefinitions.BASELINERUN){
    timeSpent+=(System.nanoTime()-startTime);
    termsIndexed+=(termnr-1);
  }

  synchronized(this){
    currentnr++;
    if(currentnr==1){
      if(BrilleDefinitions.BTREE_READING_DEBUG)
        logger.debug("notifying all because currnr is 1");
      notifyAll();
    }

    if(numDocsRead==BrilleDefinitions.STOP_AFTER){
      if(BrilleDefinitions.BASELINERUN)
        System.out.println("Used "+timeSpent+
            " nanoseconds to parse "+termsIndexed+" terms.");
      if(BrilleDefinitions.BTREE_READING_DEBUG)
        logger.debug("setting done reading because finished");
      running = false;
      setDoneReading();
      notifyAll();
      return;
    }
  }

  if(BrilleDefinitions.BTREE_READING_DEBUG)
    logger.debug("setting done reading because end of file");
  setDoneReading();



  terms = new ArrayList<ByteArr>();
  docterms = new HashMap<ByteArr,ExtendableDocTermEntry>();
  room = 0;
  termnr = 1;
  synchronized(this){
    if(currentnr == 5){
      try{
        if(BrilleDefinitions.BTREE_READING_DEBUG)
          logger.debug("must wait");
        wait();
        if(BrilleDefinitions.BTREE_READING_DEBUG)
          logger.debug("done waiting");
```

```
          }
          catch(InterruptedException ie){
            logger.error(ie);
            System.exit(1);
          }
        }
      }
      break outer;
    }
  }

  while(i<read){
    char currchar = curr[i];
    if(!intag && currchar=='<'){
      if(inword){
        buff.flip();
        ByteArr currArr = new ByteArr(buff);
        if(currArr.getL()<=BrilleDefinitions.MAX_TERM_LENGTH){
          if(!docterms.containsKey(currArr)){
            docterms.put(currArr,new ExtendableDocTermEntry());
            terms.add(currArr);
            room+=(6+currArr.getL());
          }
          docterms.get(currArr).addLoc(termnr);
          termnr++;
          room+=4;
        }
        inword = false;
        buff.clear();
        maybeskipnext = true;
      }
      intag=true;
    }
    else if(intag && currchar=='>'){
      intag=false;
      if(!skiptag){
        intagBuff.flip();
        String val = intagBuff.toString().toLowerCase();
        if(val.equals("doc")){
          try{
            cachedDocs[nextPosToWrite++] =
              new Document(new URI(docNr.toString()),docterms,
                  room,termnr-1,terms);
            if(nextPosToWrite>=cachedDocs.length)
              nextPosToWrite-=cachedDocs.length;
          }
          catch(URISyntaxException use){
            logger.error(use);
            System.exit(1);
          }

          numDocsRead++;
          if(BrilleDefinitions.OUTPUT_DISK_STATISTICS &&
              (numDocsRead%BrilleDefinitions.OUTPUT_INTERVAL)==0){
            System.out.println("Have read "+numDocsRead+" after "+
                System.nanoTime()+" nanoseconds");
          }
          if(BrilleDefinitions.BASELINERUN){
            timeSpent+=(System.nanoTime()-startTime);
            termsIndexed+=(termnr-1);
          }


          synchronized(this){
            currentnr++;
            if(currentnr==1) notifyAll();
            if(numDocsRead==BrilleDefinitions.STOP_AFTER){
              if(BrilleDefinitions.BASELINERUN)
                System.out.println("Used "+timeSpent+
```

```
                " nanoseconds to parse "+termsIndexed+" terms.");
          running = false;
          if(BrilleDefinitions.BTREE_READING_DEBUG)
            logger.debug("setting done reading because finished");
          setDoneReading();
          if(BrilleDefinitions.BTREE_READING_DEBUG)
            logger.debug("notifying all");
          notifyAll();
          return;
        }
      }

      terms = new ArrayList<ByteArr>();
      docterms = new HashMap<ByteArr,ExtendableDocTermEntry>();
      room = 0;
      termnr = 1;
      docNr.clear();

      synchronized(this){
        if(currentnr == 5){
          try{
            if(BrilleDefinitions.BTREE_READING_DEBUG)
              logger.debug("must wait");
            wait();
            if(BrilleDefinitions.BTREE_READING_DEBUG)
              logger.debug("done waiting");
          }
          catch(InterruptedException ie){
            logger.error(ie);
            System.out.println("COULD NOT WAIT!!!");
          }
        }
      }
      if(BrilleDefinitions.BASELINERUN)
        startTime = System.nanoTime();
      skip=true;
    }
    else if(val.equals("/dochdr")){
      skip=false;
    }
    else if(val.equals("docno")){
      addtodocnr = true;
    }
    else if(val.equals("/docno")){
      addtodocnr = false;
      docNr.flip();
    }
    else if(val.regionMatches(0,"script",0,6) ||
        val.regionMatches(0,"style",0,5)){
      skip = true;
    }
    else if(val.equals("/script") || val.equals("/style")){
      skip = false;
    }
  }
  intagBuff.clear();
  skiptag=false;
}
else if(intag && !skiptag){
  intagBuff.put(currchar);
  if(intagBuff.position()>=399) skiptag=true;
}
else if(addtodocnr){
  docNr.put(currchar);
}
else if(!skip && !intag && inword){
  if((currchar>='a' && currchar<='z') ||
      (currchar>='0' && currchar<='9')){
    if(!skipword) buff.put(currchar);
```

```
            }
            else if(currchar>='A' && currchar<='Z'){
              currchar+=32;
              if(!skipword)buff.put(currchar);
            }
            else{
              buff.flip();
              if(!skipword && !(maybeskipnext && currchar==';')){
                ByteArr currArr = new ByteArr(buff);
                if(currArr.getL()<=BrilleDefinitions.MAX_TERM_LENGTH){
                  if(!docterms.containsKey(currArr)){
                    docterms.put(currArr,new ExtendableDocTermEntry());
                    terms.add(currArr);
                    room+=(6+currArr.getL());
                  }
                  docterms.get(currArr).addLoc(termnr);
                  termnr++;
                  room+=4;
                }
              }
              inword=false;
              skipword = false;
              if(currchar!='&')maybeskipnext = false;
              else maybeskipnext = true;
              buff.clear();
            }
            if(buff.position()>=BrilleDefinitions.MAX_TERM_LENGTH)
              skipword = true;
          }
          else if(!skip && !intag && !inword){
            if((currchar>='a' && currchar<='z')
                || (currchar>='0' && currchar<='9')){
              inword = true;
              buff.put(currchar);
            }
            else if(currchar>='A' && currchar<='Z'){
              inword = true;
              currchar+=32;
              buff.put(currchar);
            }
            else if(currchar=='&') maybeskipnext = true;
            else maybeskipnext = false;
          }
          i++;
        }
      }
    }
    if(running)waitForNextFile();
  }
  }catch(Exception e){
    System.out.println("Exception");
    e.printStackTrace();
    logger.error(e);
  }catch(Error e){
    System.out.println("Error in altgov2...");
    e.printStackTrace();
  }
 }
}
```

C.8.2 brille.utils.ByteArr

```
package brille.utils;

import java.net.URI;
import java.net.URISyntaxException;
import java.nio.ByteBuffer;
```

```java
import java.nio.CharBuffer;
import java.nio.charset.Charset;

import org.apache.log4j.Logger;

/**
 * This is an important class in the Search Engine. It represents
 * a searchable term. The reason why this class is used is two-fold
 *   - When storing terms in a b-tree (and other buffer-based dictionaries)
 *   it is essential to know the length of the term. When storing a String,
 *   some characters might have representation lengths longer than one byte
 *   in UTF-8. I need to know that before it is stored, and hence use this
 *   class.
 *   - In a B-tree, one needs to be able to create a key that is larger than
 *   any key one will ever insert. Making a String with a lot of funky chars
 *   is not a general solution, so this class has a constructor where you can
 *   set the length to be -1, and this will evaluate to larger than any other
 *   ByteArr in compareTo(ByteArr other).
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class ByteArr implements Comparable<ByteArr>{

  protected final Logger logger = Logger.getLogger(getClass());

  public static final Charset cs = Charset.forName("UTF-8");

  private ByteBuffer b;
  int l;
  int hashCode;

  /**
   * Constructor where you can make a ByteArr from a String.
   * @param s
   */
  public ByteArr(String s){
    b=cs.encode(s);
    l=b.limit();
  }

  public ByteArr(CharBuffer buf){
    b=cs.encode(buf);
    l=b.limit();
    calcHashCode();
  }

  private void calcHashCode(){
    if(l<0) hashCode = 1;
    else{
      hashCode = 0;
      for(int i=0;i<l;i++){
        hashCode*=31;
        hashCode+=b.get(i);
      }
    }
  }

  /**
   * Constructor to use when you have already made the ByteBuffer yourself.
   * @param b
   */
  public ByteArr(ByteBuffer b){
    this.b=b;
    l=b.limit();
  }

  /**
   * Constructor to create a ByteArr from an URI. Used in the
   * DocumentManager.
```

```
 * @param u
 */
public ByteArr(URI u){
  this(u.toString());
}

/**
 * Constructor to make a ByteArr with the specified length. This is handy
 * when you want different special values for maxkeys and such.
 * @param l
 */
public ByteArr(int l){
  this.l=l;
}

/**
 * Returning the nio.ByteBuffer that holds the representation of the term.
 * @return
 */
public ByteBuffer getBuffer(){
  return b;
}

/**
 * Method for decoding the string representing this term.
 * @return the String representation of this term.
 */
public String decodeString(){
  String ret = cs.decode(b).toString();
  b.rewind();
  return ret;
}

/**
 * Getting the length of this term.
 * @return length of term.
 */
public int getL(){
  return l;
}

/**
 * Method for setting the length. Be sure not to use this one
 * if you do not know what you're doing:)
 * @param l
 */
public void setL(int l){
  this.l=l;
}

/**
 * Method to return the URI represented by this ByteArr.
 * This method is used in the docman. Be sure not to use
 * it if this bytearr does not store an URI:)
 * @return
 */
public URI decodeToURI(){
  try{
    return new URI(decodeString());
  }catch(URISyntaxException use){
    logger.error("Problems decoding URI",use);
    return null;
  }

}

public int compareTo(ByteArr other){
  if(l==-1){
    if(other.getL()==-1) return 0;
```

```
      return 1;
    }
    else if(other.getL()==-1) return -1;
    int m = l<other.getL()?l:other.getL();
    for(int i=0;i<m;i++){
      if(b.get(i)<other.getBuffer().get(i)) return -1;
      else if (b.get(i)>other.getBuffer().get(i)) return 1;
    }
    if(l>m)return 1;
    else if(other.getL()>m) return -1;
    return 0;
  }

  public boolean equals(Object oth){
    if(!(oth instanceof ByteArr)) return false;
    ByteArr other = (ByteArr) oth;
    if(l!=other.getL()) return false;
    for(int i=0;i<l;i++) if(b.get(i)!=other.getBuffer().get(i)) return false;
    return true;
  }

  public int hashCode(){
    return hashCode;
  }
}
```

C.8.3 brille.utils.ByteArrayList

```
package brille.utils;

/**
 * This class serves the same right as IntArrayList, to create a more
 * space-efficient and fast version of ArrayList<Byte>
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public class ByteArrayList {
  private static final int stdInitialCapacity=10;
  private static final double stdFactor=1.1;
  private double factor;
  private byte[] currArray;
  private int currLength;

  /**
   * Constructor where all parameters can be set
   * @param initialCapacity - the initial size of the underlying array
   * @param factor - the factor at which the size should grow (or shrink).
   */
  public ByteArrayList(int initialCapacity,double factor){
    this.factor=factor;
    currArray=new byte[initialCapacity];
    currLength=0;
  }

  /**
   * Constructor that uses the standard growing factor of 1.1
   * @param initialCapacity  - the initial size of the underlying array
   */
  public ByteArrayList(int initialCapacity){
    this(initialCapacity,stdFactor);
  }

  /**
   * Constructtor that uses the standard initial capacity, 10.
   * @param factor - the factor at which the size should grow (or shrink).
   */
```

```
public ByteArrayList(double factor){
  this(stdInitialCapacity,factor);
}

/**
 * Constructor with standard values for initial capacity(10) and
 * factor(1.1).
 */
public ByteArrayList(){
  this(stdInitialCapacity,stdFactor);
}

/**
 * Append d to the end of the list. Extend the list if necesseary.
 * Note that n calls to this method will give a running time of O(n).
 * The amortized complexity is thus O(1).
 *
 * @param d - the byte that should be appended.
 */
public void add(byte d){
  if(currLength==currArray.length) extend();
  currArray[currLength++]=d;
}

private void extend(){
  int newSize=(int)(currArray.length*factor);
  byte[] newCurr = new byte[newSize];
  for(int i=0;i<currLength;i++) newCurr[i]=currArray[i];
  currArray=newCurr;
}

private void extend(int index){
  int newSize=(int)(currArray.length*factor);
  byte[] newCurr = new byte[newSize];
  for(int i=0;i<index;i++) newCurr[i]=currArray[i];
  for(int i=index;i<currLength;i++)newCurr[i+1]=currArray[i];
  currArray=newCurr;
}

/**
 * Add a byte at a specified position (the list size will grow with 1).
 * @param d - the byte to add.
 * @param index - the index at which the byte should be added.
 */
public void add(byte d, int index){
  if(currLength==currArray.length){
    extend(index);
    currArray[index]=d;
  }
  else{
    byte tmp=currArray[index];
    byte tmp2=0;
    for(int i=index+1;i<currLength;i++){
      tmp2=currArray[i];
      currArray[i]=tmp;
      tmp=tmp2;
    }
  }

}

/**
 * Just as set in ArrayList, but I think replace is a better name.
 *
 * @param d - the byte that should replace the current
 * @param index - at the given position.
 */
public void replace(byte d,int index){
  currArray[index]=d;
```

```
  }

  private void shrink(int index){
    int newSize=(int)(currArray.length/factor);
    byte[] newCurr = new byte[newSize];
    for(int i=0;i<index;i++)newCurr[i]=currArray[i];
    for(int i=index+1;i<currLength;i++)newCurr[i-1]=currArray[i];
    currArray=newCurr;
    currLength--;
  }

  /**
   * Remove a byte at the given index. Warning: This is not a very efficient
   * method
   * @param index
   * @return
   */
  public byte remove(int index){
    byte ret = currArray[index];
    int minsize = (int)(((double)currArray.length)/(factor*factor));
    if(currLength<minsize && currArray.length>10) shrink(index);
    else{
      for(int i=index;i<currLength-1;i++) currArray[i]=currArray[i+1];
      currArray[--currLength]=0;
    }
    return ret;
  }

  /**
   * The size of the list
   * @return number of bytes in list
   */
  public int size(){
    return currLength;
  }

  /**
   * Method to make sure that the given index can be accessed
   * @param index - the index we will ensure we can access.
   */
  public synchronized void ensureSize(int index){
    while(currLength<index+1) add((byte)0);
  }

  /**
   * Getting the byte at the given position
   * @param index - the index at which we want to get the byte.
   * @return - the byte at index index.
   */
  public byte get(int index){
    return currArray[index];
  }

}
```

C.8.4 brille.utils.Document

```
package brille.utils;

import java.net.URI;
import java.util.ArrayList;
import java.util.HashMap;

import brille.dict.ExtendableDocTermEntry;

/**
 * This class represents a document read from the GOV2-collection.
```

```
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class Document {
  private HashMap<ByteArr,ExtendableDocTermEntry> tokensfromDoc;
  private URI uri;
  private int roomInInMemIndex;
  private int numterms;
  private ArrayList<ByteArr> list;

  /**
   * Constructor taking all member variables as parameters.
   *
   * @param uri - the uri to the document
   * @param tokensfromDoc - a hashmap with tokens and their positions
   * @param roomInInMemIndex - the amount of room it will consume in the
   *  in-memory index.
   * @param numterms - the number of terms
   * @param list - a list of the terms
   */
  public Document(URI uri, HashMap<ByteArr,
      ExtendableDocTermEntry> tokensfromDoc, int roomInInMemIndex,
      int numterms,ArrayList<ByteArr> list){
    this.uri = uri;
    this.tokensfromDoc = tokensfromDoc;
    this.roomInInMemIndex = roomInInMemIndex;
    this.numterms = numterms;
    this.list = list;
  }

  /**
   * Getter for the uri.
   *
   * @return the uri of this document
   */
  public URI getUri(){
    return uri;
  }

  /**
   * Getter for the mappings from terms to occurrences in this document.
   *
   * @return the hashmap between terms and occurrences
   */
  public HashMap<ByteArr,ExtendableDocTermEntry> getTerms(){
    return tokensfromDoc;
  }

  /**
   * Getter for number of terms.
   *
   * @return the number of terms in the document
   */
  public int getNumterms() {
    return numterms;
  }

  /**
   * Getter for the amount of space this document will consume in the
   * in-memory index.
   *
   * @return the amonut of room needed
   */
  public int getRoomInInMemIndex() {
    return roomInInMemIndex;
  }

  /**
   * Getter for the list of terms.
```

```
   *
   * @return the terms in this document
   */
  public ArrayList<ByteArr> getTermList(){
    return list;
  }
}
```

C.8.5 brille.utils.DoubleArrayList

```
package brille.utils;

/**
 * This class serves the same right as IntArrayList, to create a more
 * space-efficient and fast version of ArrayList<Long>
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public class DoubleArrayList {
  private static final int stdInitialCapacity=10;
  private static final double stdFactor=1.1;
  private double factor;
  private double[] currArray;
  private int currLength;

  /**
   * Constructor where all parameters can be set
   * @param initialCapacity - the initial size of the underlying array
   * @param factor - the factor at which the size should grow (or shrink).
   */
  public DoubleArrayList(int initialCapacity,double factor){
    this.factor=factor;
    currArray=new double[initialCapacity];
    currLength=0;
  }

  /**
   * Constructor that uses the standard growing factor of 1.1
   * @param initialCapacity  - the initial size of the underlying array
   */
  public DoubleArrayList(int initialCapacity){
    this(initialCapacity,stdFactor);
  }

  /**
   * Constructtor that uses the standard initial capacity, 10.
   * @param factor - the factor at which the size should grow (or shrink).
   */
  public DoubleArrayList(double factor){
    this(stdInitialCapacity,factor);
  }

  /**
   * Constructor with standard values for initial capacity(10) and
   * factor(1.1).
   */
  public DoubleArrayList(){
    this(stdInitialCapacity,stdFactor);
  }

  /**
   * Append d to the end of the list. Extend the list if necesseary.
   * Note that n calls to this method will give a running time of O(n).
   * The amortized complexity is thus O(1).
   * @param d - the double that should be appended.
   */
```

```
public void add(double d){
  if(currLength==currArray.length) extend();
  currArray[currLength++]=d;
}

private void extend(){
  int newSize=(int)(currArray.length*factor);
  double[] newCurr = new double[newSize];
  for(int i=0;i<currLength;i++) newCurr[i]=currArray[i];
  currArray=newCurr;
}

private void extend(int index){
  int newSize=(int)(currArray.length*factor);
  double[] newCurr = new double[newSize];
  for(int i=0;i<index;i++) newCurr[i]=currArray[i];
  for(int i=index;i<currLength;i++)newCurr[i+1]=currArray[i];
  currArray=newCurr;
}

/**
 * Add a double at a specified position (the list size will grow with 1).
 * @param d - the double to add.
 * @param index - the index at which the double should be added.
 */
public void add(double d, int index){
  if(currLength==currArray.length){
    extend(index);
    currArray[index]=d;
  }
  else{
    double tmp=currArray[index];
    double tmp2=0;
    for(int i=index+1;i<currLength;i++){
      tmp2=currArray[i]; currArray[i]=tmp; tmp=tmp2;
    }
  }

}

/**
 * Just as set in ArrayList, but I think replace is a better name.
 * @param d - the double that should replace the current
 * @param index - at the given position.
 */
public void replace(double d,int index){
  currArray[index]=d;
}

private void shrink(int index){
  int newSize=(int)((currArray.length)/factor);
  double[] newCurr = new double[newSize];
  for(int i=0;i<index;i++)newCurr[i]=currArray[i];
  for(int i=index+1;i<currLength;i++)newCurr[i-1]=currArray[i];
  currArray=newCurr;
  currLength--;
}

/**
 * Remove a double at the given index. Warning: This is not a very
 * efficient method
 * @param index
 * @return
 */
public double remove(int index){
  double ret = currArray[index];
  int minsize = (int)(((double)currArray.length)/(factor*factor));
  if(currLength<minsize && currArray.length>10) shrink(index);
  else{
```

```
      for(int i=index;i<currLength-1;i++) currArray[i]=currArray[i+1];
      currArray[--currLength]=0.0;
    }
    return ret;
  }

  /**
   * The size of the list
   * @return number of doubles in list
   */
  public int size(){
    return currLength;
  }

  /**
   * Method to make sure that the given index can be accessed
   * @param index - the index we will ensure we can access.
   */
  public synchronized void ensureSize(int index){
    while(currLength<index+1) add(0.0);
  }

  /**
   * Getting the double at the given position
   * @param index - the index at which we want to get the double.
   * @return - the double at index index.
   */
  public double get(int index){
    return currArray[index];
  }

  /**
   * Method used for debugging, to print all interresting values for this
   * object.
   *
   */
  public void print(){
    System.out.println("Real length is "+currArray.length);
    for(int i=0;i<currLength;i++){
      System.out.print(currArray[i]+" ");
    }
    System.out.println();
  }
}
```

C.8.6 brille.utils.FreeBSDIOStats

```
package brille.utils;

import java.math.BigInteger;

/**
 * This is a class capable of storing statistics for the I/O-subsystem on
 * FreeBSD.
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public class FreeBSDIOStats implements IOStats{
  private String device;
  private BigInteger numBytesRead;
  private BigInteger numBytesWritten;
  private BigInteger numBytesFree;
  private BigInteger totTransfersRead;
  private BigInteger totTransfersWritten;
  private BigInteger totTransfersOther;
  private BigInteger totTransfersFree;
```

```
private BigInteger blockSize;
private BigInteger time;

/**
 * Constructor. The parameters should be self-explaining. These values are
 * available in the FreeBSD-kernel. Look at iostat.c and devstat.c in the
 * FreeBSD source tree for more info.
 *
 * @param device
 * @param numBytesRead
 * @param numBytesWritten
 * @param numBytesFree
 * @param totTransfersRead
 * @param totTransfersWritten
 * @param totTransfersOther
 * @param totTransfersFree
 * @param blockSize
 * @param time
 */
public FreeBSDIOStats(String device, BigInteger numBytesRead,
    BigInteger numBytesWritten, BigInteger numBytesFree,
    BigInteger totTransfersRead, BigInteger totTransfersWritten,
    BigInteger totTransfersOther, BigInteger totTransfersFree,
    BigInteger blockSize, BigInteger time){
  this.device = device;
  this.numBytesWritten = numBytesWritten;
  this.numBytesRead = numBytesRead;
  this.numBytesFree = numBytesFree;
  this.totTransfersRead = totTransfersRead;
  this.totTransfersWritten = totTransfersWritten;
  this.totTransfersOther = totTransfersOther;
  this.totTransfersFree = totTransfersFree;
  this.blockSize = blockSize;
  this.time=time;
}

public String getDeviceName() {
  return device;
}

public BigInteger getBlockSize() {
  return blockSize;
}

public BigInteger getNumBytesFree() {
  return numBytesFree;
}

public BigInteger getNumBytesRead() {
  return numBytesRead;
}

public BigInteger getNumBytesWritten() {
  return numBytesWritten;
}

public BigInteger getTotTransfersOther() {
  return totTransfersOther;
}

public BigInteger getTotTransfersFree() {
  return totTransfersOther;
}

public BigInteger getTotTransfersRead() {
  return totTransfersRead;
}

public BigInteger getTotTransfersWritten() {
```

```
  return totTransfersWritten;
}

/**
 * numbytesread/blocksize
 * @return
 */
public BigInteger getTotBlocksRead(){
  if(blockSize.equals(new BigInteger("0")))
    throw new Error("blockSize is 0");
  return numBytesRead.divide(blockSize);
}

/**
 * numbyteswritten/blocksize
 * @return
 */
public BigInteger getTotBlocksWritten(){
  if(blockSize.equals(new BigInteger("0")))
    throw new Error("blockSize is 0");
  return numBytesWritten.divide(blockSize);
}

/**
 * numbytesfree/blocksize
 * @return
 */
public BigInteger getTotBlocksFree(){
  if(blockSize.equals(new BigInteger("0")))
    throw new Error("blockSize is 0");
  return numBytesFree.divide(blockSize);
}

/**
 * totalbytes/blocksize
 * @return
 */
public BigInteger getTotBlocks(){
  if(blockSize.equals(new BigInteger("0")))
    throw new Error("blockSize is 0");
  return getTotalBytes().divide(blockSize);
}

/**
 * totalbytes = numbytesread+numbyteswritten+numbytesfree.
 * @return
 */
public BigInteger getTotalBytes(){
  return numBytesRead.add(numBytesWritten.add(numBytesFree));
}

/**
 * tottransfers = tottransfersfree+tottransfersother+
 *    tottransfersread+tottransferswritten
 * @return
 */
public BigInteger getTotTransfers(){
  return totTransfersFree.add(totTransfersOther.add(
      totTransfersRead.add(totTransfersWritten)));
}

public BigInteger getTime(){
  return time;
}

/**
 * totalbytes/tottransfers
 * @return
 */
```

```
public BigInteger getTotalBytesPerTransfer(){
  if(getTotTransfers().equals(new BigInteger("0")))
    return new BigInteger("0");
  return getTotalBytes().divide(getTotTransfers());
}

/**
 * numbytesread/tottransfersread
 * @return
 */
public BigInteger getBytesReadPerTransfer(){
  if(totTransfersRead.equals(new BigInteger("0")))
    return new BigInteger("0");
  return numBytesRead.divide(totTransfersRead);
}

/**
 * numbyteswritten/tottransferswritten
 * @return
 */
public BigInteger getBytesWrittenPerTransfer(){
  if(totTransfersWritten.equals(new BigInteger("0")))
    return new BigInteger("0");
  return numBytesWritten.divide(totTransfersWritten);
}

/**
 * numbytesfree/tottransfersfree
 * @return
 */
public BigInteger getBytesFreePerTransfer(){
  if(totTransfersFree.equals(new BigInteger("0")))
    return new BigInteger("0");
  return numBytesFree.divide(totTransfersFree);
}

/**
 * tottransfers/time
 * @return
 */
public BigInteger getTotTransfersPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return getTotTransfers().divide(time);
}

/**
 * tottransfersread/time
 * @return
 */
public BigInteger getReadTransfersPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return totTransfersRead.divide(time);
}

/**
 * tottransferswritten/time
 * @return
 */
public BigInteger getWrittenTransfersPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return totTransfersWritten.divide(time);
}

/**
 * tottransfersfree/time
 * @return
 */
public BigInteger getFreeTransfersPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
```

```
  return totTransfersFree.divide(time);
}

/**
 * tottransfersother/time
 * @return
 */
public BigInteger getOtherTransfersPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return totTransfersOther.divide(time);
}

/**
 * totalbytes/time
 * @return
 */
public BigInteger getBytesTransferredPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return getTotalBytes().divide(time);
}

/**
 * numbytesread/time
 * @return
 */
public BigInteger getBytesReadPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return numBytesRead.divide(time);
}

/**
 * numbyteswritten/time
 * @return
 */
public BigInteger getBytesWrittenPerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return numBytesWritten.divide(time);
}

/**
 * numbytesfree/time
 * @return
 */
public BigInteger getBytesFreePerNanoSecond(){
  if(time.equals(new BigInteger("0"))) throw new Error("time is 0");
  return numBytesFree.divide(time);
}

public void diff(IOStats[] sto,long starttime){
  if(!(sto instanceof FreeBSDIOStats[]))
    throw new IllegalArgumentException();
  FreeBSDIOStats[] st = (FreeBSDIOStats[])sto;
  for(int i=0;i<st.length;i++) if(st[i]!=null &&
      st[i].getDeviceName().equals(device)){
    numBytesFree = numBytesFree.subtract(st[i].getNumBytesFree());
    numBytesRead = numBytesRead.subtract(st[i].getNumBytesRead());
    numBytesWritten = numBytesWritten.subtract(
        st[i].getNumBytesWritten());
    totTransfersOther = totTransfersOther.subtract(
        st[i].getTotTransfersOther());
    totTransfersFree = totTransfersFree.subtract(
        st[i].getTotTransfersFree());
    totTransfersRead = totTransfersRead.subtract(
        st[i].getTotTransfersRead());
    totTransfersWritten = totTransfersWritten.subtract(
        st[i].getTotTransfersWritten());
    time = time.subtract(st[i].getTime());
  }
}
```

```
  public String toString(){
    return "Device: "+device+"\nnumBytesRead: "+numBytesRead.toString()+
    "\nnumBytesWritten: "+numBytesWritten.toString()+"\nnumBytesFree: "
    +numBytesFree.toString()+"\ntotTransfersRead: "
    +totTransfersRead.toString()+"\ntotTransfersWritten: "
    +totTransfersWritten.toString()+"\ntotTransfersOther: "
    +totTransfersOther.toString()+"\ntotTransfersFree: "
    +totTransfersFree.toString()+"\nblocksize: "
    +blockSize.toString()+"\nTime: "+time.toString();
  }

}
```

C.8.7 brille.utils.FreeBSDIOStatsGatherer

```
package brille.utils;

import java.math.BigInteger;

import brille.BrilleDefinitions;

/**
 * This class is capable of gathering I/O-statistics on FreeBSD.
 * It uses FreeBSDLibAccesser. The rest of the documentation can be
 * found in IOStatsGatherer.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class FreeBSDIOStatsGatherer extends IOStatsGatherer{

  private FreeBSDIOStats[] curr;
  FreeBSDLibAccesser fbla;

  /**
   * Constructor
   */
  public FreeBSDIOStatsGatherer(){
    curr = null;
    fbla = new FreeBSDLibAccesser();
  }

  public void startGatheringIOStats() {
    int numdevs = fbla.get_numdevs();
    if(BrilleDefinitions.DEBUG) logger.info("numdevs is "+numdevs);
    String[] stats = new String[numdevs*9];
    stats = fbla.get_stats(stats);
    curr = new FreeBSDIOStats[numdevs];
    long time = System.nanoTime();
    for(int i=0;9*i<stats.length;i++){
      if(stats[9*i]==null) continue;
      curr[i] = new FreeBSDIOStats(stats[9*i],
          new BigInteger(stats[9*i+1]), new BigInteger(stats[9*i+2]),
          new BigInteger(stats[9*i+3]), new BigInteger(stats[9*i+4]),
          new BigInteger(stats[9*i+5]), new BigInteger(stats[9*i+6]),
          new BigInteger(stats[9*i+7]), new BigInteger(stats[9*i+8]),
          new BigInteger(""+time));
    }
  }

  public IOStats[] endGatheringAndReturnIOStats() {
    long time = System.nanoTime();
    if(curr==null) throw new IllegalArgumentException("the gathering of "+
    "iostats has not been initiated");
    int numdevs = fbla.get_numdevs();
    String[] stats = new String[numdevs*9];
    stats = fbla.get_stats(stats);
```

```
    FreeBSDIOStats[] diffs = new FreeBSDIOStats[numdevs];
    for(int i=0;9*i<stats.length;i++){
      if(stats[9*i]==null) continue;
      diffs[i] = new FreeBSDIOStats(stats[9*i],
          new BigInteger(stats[9*i+1]), new BigInteger(stats[9*i+2]),
          new BigInteger(stats[9*i+3]), new BigInteger(stats[9*i+4]),
          new BigInteger(stats[9*i+5]), new BigInteger(stats[9*i+6]),
          new BigInteger(stats[9*i+7]), new BigInteger(stats[9*i+8]),
          new BigInteger(""+time));
    }
    for(int i=0;i<diffs.length;i++) if(diffs[i]!=null) diffs[i].diff(curr,0);
    curr=null;
    return diffs;
  }
}
```

C.8.8 brille.utils.FreeBSDLibAccesser

```
package brille.utils;

/**
 * Class using jni (Java Native Interface) to access a C-library
 * written by the author on FreeBSD.
 *
 * This class only declares the methods, and javah helps us
 * create the header-file for C.
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public class FreeBSDLibAccesser {
  /**
   * Method to get the I/O-stats at a given time.
   * It assumes that enough place is reserved in the parameters
   * to store all return values. The returnstrings-array must have
   * 9*numdevs entries.
   * @param returnStrings - the array where you want the return-values
   * @return - the returnStrings-array, or an empty arary if something
   *     was wrong.
   */
  public native String[] get_stats(String[] returnStrings);

  /**
   * Method for finding the number of disk-devices on the given computer.
   * This method must be used to ensure you create the array of the correct
   * size for the method above.
   * @return - number of disk devices on the computer.
   */
  public native int get_numdevs();

  static{
    System.loadLibrary("realiostat");
  }

}
```

C.8.9 brille.utils.GOV2FileParser

```
package brille.utils;

import java.io.BufferedReader;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.nio.CharBuffer;
```

```java
import java.util.ArrayList;
import java.util.HashMap;

import org.apache.log4j.Logger;

import brille.BrilleDefinitions;
import brille.dict.ExtendableDocTermEntry;

/**
 * This class is sort of a buffered reader from a GOV2-file, because it is
 * capable of buffering documents from it. The files in the GOV2 collection
 * actually consists of several files, which are separated by some xml-like
 * tags (it is not a valid xml-document because it hasn't got a root tag).
 *
 * The documents are html-documents. This parser will only work for correct
 * html. It will find all terms in a document that is not inside a tag, and
 * is not between <style> and </style>, or between <script> and </script>.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class GOV2FileParser extends Thread{

  protected final Logger logger = Logger.getLogger(getClass());

  private BufferedReader in;
  private boolean doneReading;
  private Document[] cachedDocs;
  private int currentnr;
  private int nextPosToRead;
  private int nextPosToWrite;
  private Object readWaitingObject;
  private Object writeWaitingObject;
  private boolean running;
  private Object nextFileWaiter;
  private int numDocsRead;

  //variables for timing a baseline run
  private long timeSpent;
  private long startTime;
  private long termsIndexed;


  /**
   * Default constructor
   *
   */
  public GOV2FileParser(){
    in=null;
    doneReading = true;
    cachedDocs = new Document[5];
    currentnr = 0;
    nextPosToRead = 0;
    nextPosToWrite = 0;
    readWaitingObject = new Object();
    writeWaitingObject = new Object();
    nextFileWaiter = new Object();
    running = true;
    numDocsRead=0;
    termsIndexed=0;
    timeSpent=0;
    start();
  }

  /**
   * Method used to shut this reader down when no more documents should be
   * read.
   *
   */
  public void stopRunning(){
```

```
  running = false;
  nextFile();
}

/**
 * Method for testing whether this reader has shut down.
 *
 * @return whether or not this reader has shut down
 */
public boolean isStopped(){
  return !running;
}

private void nextFile(){
  synchronized(nextFileWaiter){
    nextFileWaiter.notify();
  }
}

private void waitForNextFile(){
  synchronized(nextFileWaiter){
    try{
      if(in==null)nextFileWaiter.wait();
    }
    catch(InterruptedException ie){
      logger.error(ie);
      System.exit(1);
    }
  }
}


public void setNewFile(BufferedReader in){
  this.in = in;
  doneReading = false;
  nextFile();
}

/**
 * Getter for the next document from this reader. If the reader has not yet
 * parsed the next document, the calling thread will have to wait.
 *
 * @return the next document from this reader, or null if no more docs are
 *   available
 */
public Document getNextDocument(){
  if(currentnr==0 && doneReading) return null;
  while(currentnr<1 && !doneReading) readWait();
  Document ret = cachedDocs[nextPosToRead++];
  if(nextPosToRead>=cachedDocs.length) nextPosToRead-=cachedDocs.length;
  synchronized(readWaitingObject){
    currentnr--;
  }
  writeNotify();
  return ret;
}

private void readWait(){
  synchronized(readWaitingObject){
    if(currentnr == 0 && !doneReading){
      try {
        readWaitingObject.wait();
      } catch (InterruptedException e) {
        logger.error(e);
        System.exit(1);
      }
    }
  }
}
```

```
private void readNotify(){
  synchronized(readWaitingObject){
    currentnr++;
    readWaitingObject.notify();
  }
}

private void writeWait(){
  synchronized(writeWaitingObject){
    if(currentnr == cachedDocs.length){
      try {
        writeWaitingObject.wait();
      } catch (InterruptedException e) {
        logger.error(e);
        System.exit(1);
      }
    }
  }
}

private void writeNotify(){
  synchronized(writeWaitingObject){
    if(currentnr<cachedDocs.length){
      writeWaitingObject.notify();
    }
  }
}


public void run(){
  try{
    if(BrilleDefinitions.BASELINERUN)startTime = System.nanoTime();
    CharBuffer buff =
      CharBuffer.allocate(BrilleDefinitions.MAX_TERM_LENGTH+4);
    CharBuffer intagBuff = CharBuffer.allocate(400);
    CharBuffer docNr = CharBuffer.allocate(50);
    while(running){
      buff.clear();
      intagBuff.clear();
      docNr.clear();
      boolean skiptag=false;
      boolean addtodocnr = false;
      boolean skip = false;
      boolean intag = false;
      boolean inword = false;
      boolean skipword = false;
      boolean maybeskipnext = false;

      char[] curr = new char[65536];

      int room = 0;
      int i =0;
      int read = 0;
      outer: while(!doneReading){
        HashMap<ByteArr, ExtendableDocTermEntry> docterms =
          new HashMap<ByteArr,ExtendableDocTermEntry>();
        ArrayList<ByteArr> terms = new ArrayList<ByteArr>();
        int termnr=1;
        while(true){
          if(i==read){
            i = 0;
            if(BrilleDefinitions.BASELINERUN)
              timeSpent += (System.nanoTime()-startTime);
            try{
              read = in.read(curr);
            }
            catch(IOException ioe){
              logger.error(ioe);
```

```
      System.exit(1);
    }
    if(BrilleDefinitions.BASELINERUN) startTime = System.nanoTime();
    if(read < 1){
      doneReading = true;
      try{
        cachedDocs[nextPosToWrite++] =
          new Document(new URI(docNr.toString()),docterms,
              room,termnr-1,terms);
        if(nextPosToWrite>=cachedDocs.length)
          nextPosToWrite-=cachedDocs.length;
      }
      catch(URISyntaxException use){
        logger.error(use);
        System.exit(1);
      }
      if(BrilleDefinitions.BASELINERUN) termsIndexed+=(termnr-1);
      terms = new ArrayList<ByteArr>();
      docterms = new HashMap<ByteArr,ExtendableDocTermEntry>();
      in = null;
      room = 0;
      termnr = 1;
      numDocsRead++;
      if(BrilleDefinitions.BASELINERUN)
        timeSpent+=(System.nanoTime()-startTime);
      if(numDocsRead==BrilleDefinitions.STOP_AFTER){
        if(BrilleDefinitions.BASELINERUN)
          System.out.println("Used "+timeSpent+
              " nanoseconds to parse "+termsIndexed+" terms.");
        running = false;
        doneReading = true;
      }
      readNotify();
      if(numDocsRead==BrilleDefinitions.STOP_AFTER){
        return;
      }
      break outer;
    }
  }

  while(i<read){
    char currchar = curr[i];
    if(!intag && currchar=='<'){
      if(inword){
        buff.flip();
        ByteArr currArr = new ByteArr(buff);
        if(currArr.getL()<=BrilleDefinitions.MAX_TERM_LENGTH){
          if(!docterms.containsKey(currArr)){
            docterms.put(currArr,new ExtendableDocTermEntry());
            terms.add(currArr);
            room+=(6+currArr.getL());
          }
          docterms.get(currArr).addLoc(termnr);
          termnr++;
          room+=4;
        }
        inword = false;
        buff.clear();
        maybeskipnext = true;
      }
      intag=true;
    }
    else if(intag && currchar=='>'){
      intag=false;
      if(!skiptag){
        intagBuff.flip();
        String val = intagBuff.toString().toLowerCase();
        if(val.equals("doc")){
          try{
```

```
          cachedDocs[nextPosToWrite++] =
            new Document(new URI(docNr.toString()),docterms,
                room,termnr-1,terms);
          if(nextPosToWrite>=cachedDocs.length)
            nextPosToWrite-=cachedDocs.length;
        }
        catch(URISyntaxException use){
          System.out.println("URISyntaxException!!!");
          logger.error(use);
          System.exit(1);
        }
        if(BrilleDefinitions.BASELINERUN) termsIndexed+=(termnr-1);
        terms = new ArrayList<ByteArr>();
        docterms = new HashMap<ByteArr,ExtendableDocTermEntry>();
        room = 0;
        termnr = 1;
        docNr.clear();

        numDocsRead++;
        if(BrilleDefinitions.BASELINERUN)
          timeSpent+=(System.nanoTime()-startTime);
        if(numDocsRead==BrilleDefinitions.STOP_AFTER){
          if(BrilleDefinitions.BASELINERUN)
            System.out.println("Used "+timeSpent+
                " nanoseconds to parse "+termsIndexed+" terms.");
          doneReading = true;
          running = false;
        }
        readNotify();
        if(numDocsRead==BrilleDefinitions.STOP_AFTER) return;
        if(currentnr==cachedDocs.length){
          writeWait();
        }
        if(BrilleDefinitions.BASELINERUN)
          startTime = System.nanoTime();
        skip=true;
      }
      else if(val.equals("/dochdr")){
        skip=false;
      }
      else if(val.equals("docno")){
        addtodocnr = true;
      }
      else if(val.equals("/docno")){
        addtodocnr = false;
        docNr.flip();
      }
      else if(val.regionMatches(0,"script",0,6) ||
          val.regionMatches(0,"style",0,5)){
        skip = true;
      }
      else if(val.equals("/script") || val.equals("/style")){
        skip = false;
      }
    }
    intagBuff.clear();
    skiptag=false;
  }
  else if(intag && !skiptag){
    intagBuff.put(currchar);
    if(intagBuff.position()>=399) skiptag=true;
  }
  else if(addtodocnr){
    docNr.put(currchar);
  }
  else if(!skip && !intag && inword){
    if((currchar>='a' && currchar<='z') ||
        (currchar>='0' && currchar<='9')){
      if(!skipword) buff.put(currchar);
```

```
                }
                else if(currchar>='A' && currchar<='Z'){
                  currchar+=32;
                  if(!skipword)buff.put(currchar);
                }
                else{
                  buff.flip();
                  if(!skipword && !(maybeskipnext && currchar==';')){
                    ByteArr currArr = new ByteArr(buff);
                    if(currArr.getL()<=BrilleDefinitions.MAX_TERM_LENGTH){
                      if(!docterms.containsKey(currArr)){
                        docterms.put(currArr,new ExtendableDocTermEntry());
                        terms.add(currArr);
                        room+=(6+currArr.getL());
                      }
                      docterms.get(currArr).addLoc(termnr);
                      termnr++;
                      room+=4;
                    }
                  }
                  inword=false;
                  skipword = false;
                  if(currchar!='&')maybeskipnext = false;
                  else maybeskipnext = true;
                  buff.clear();
                }
                if(buff.position()>=BrilleDefinitions.MAX_TERM_LENGTH)
                  skipword = true;
              }
              else if(!skip && !intag && !inword){
                if((currchar>='a' && currchar<='z')
                    || (currchar>='0' && currchar<='9')){
                  inword = true;
                  buff.put(currchar);
                }
                else if(currchar>='A' && currchar<='Z'){
                  inword = true;
                  currchar+=32;
                  buff.put(currchar);
                }
                else if(currchar=='&') maybeskipnext = true;
                else maybeskipnext = false;
              }
              i++;
            }
          }
        }
        if(running)waitForNextFile();
      }
    }catch(Exception e){
      System.out.println("GOV2FileParser exception:");
      e.printStackTrace();
    }catch(Error e){
      System.out.println("GOV2FileParser error:");
      e.printStackTrace();
    }
  }
}
```

C.8.10 brille.utils.IntArrayList

```
package brille.utils;

/**
 * This class has approximately the same functionality as ArrayList<Integer>.
 * The reason why it is exists is:
 *  - An ArrayList<Integer> stores pointer to Integer-objects in an Array.
```

```
 *  - That causes the memory usage to be 32 bits per int on a 64-bit machine
 *  - The caching effects are also horrible, because we havo no guarantee
 *  that the different objects will be stored sequentially in memory.
 *
 * This class on the other hand, uses approx 4 byte per int, and everything
 * is stored as sequentially as an array of ints, and the caching effects
 * should be good.
 *
 * @author Truls A. Ã¸Bjrklund
 */

public class IntArrayList {

  private static final int stdInitialCapacity=10;
  private static final double stdFactor=1.1;
  private double factor;
  private int[] currArray;
  private int currLength;

  /**
   * Constructor with all parameters
   * @param initialCapacity - the length of the initial array that is stored
   * behind the scenes.
   * @param factor - the factor with which the size should grow when the
   * current capacity is exhausted.
   */
  public IntArrayList(int initialCapacity,double factor){
    this.factor=factor;
    currArray=new int[initialCapacity];
    currLength=0;
  }

  /**
   * Constructor where the factor is set to 1.1
   * @param initialCapacity - the length of the initial array that is stored
   * behind the scenes.
   */
  public IntArrayList(int initialCapacity){
    this(initialCapacity,stdFactor);
  }

  /**
   * Constructor where the initialCapacity is set to 10.
   * @param factor - the factor with which the size should grow when the
   * current capacity is exhausted.
   */
  public IntArrayList(double factor){
    this(stdInitialCapacity,factor);
  }

  /**
   * Constructor where the initialCapacity is set to 10, and the factor
   * to 1.1
   */
  public IntArrayList(){
    this(stdInitialCapacity,stdFactor);
  }

  /**
   * Method for appending an int to the list. If the array is full, all
   * entries are moved to a larger array.
   * @param d - the int that should be appended to the list
   */
  public void add(int d){
    if(currLength==currArray.length) extend();
    currArray[currLength++]=d;
  }

  private void extend(){
```

```
    int newSize=(int)(currArray.length*factor);
    int[] newCurr = new int[newSize];
    for(int i=0;i<currLength;i++) newCurr[i]=currArray[i];
    currArray=newCurr;
  }

  private void extend(int index){
    int newSize=(int)(currArray.length*factor);
    int[] newCurr = new int[newSize];
    for(int i=0;i<index;i++) newCurr[i]=currArray[i];
    for(int i=index;i<currLength;i++)newCurr[i+1]=currArray[i];
    currArray=newCurr;
  }

  /**
   * Method for inserting an int in the list at the specified index.
   * @param d - the int that should be inserted
   * @param index - the index at which de should be inserted
   */
  public void add(int d, int index){
    if(currLength==currArray.length){
      extend(index);
      currArray[index]=d;
    }
    else{
      int tmp=currArray[index];
      int tmp2=0;
      for(int i=index+1;i<currLength;i++)
      {tmp2=currArray[i]; currArray[i]=tmp; tmp=tmp2;}
    }
  }

  /**
   * Method for replacing the integer at position index with d. It is
   * analogue to the set-method in ArrayList, but I find replace to be a
   * better name:)
   *
   * @param d
   * @param index
   */
  public void replace(int d,int index){
    currArray[index]=d;
  }

  private void shrink(int index){
    int newSize=(int)(currArray.length/factor);
    int[] newCurr = new int[newSize];
    for(int i=0;i<index;i++)newCurr[i]=currArray[i];
    for(int i=index+1;i<currLength;i++)newCurr[i-1]=currArray[i];
    currArray=newCurr;
    currLength--;
  }

  /**
   * Method for removing the int at index index.
   * @param index - the index to remove the int from
   * @return the int that was removed
   */
  public int remove(int index){
    int ret = currArray[index];
    int minsize = (int)(((double)currArray.length)/(factor*factor));
    if(currLength<minsize && currArray.length>10) shrink(index);
    else{
      for(int i=index;i<currLength-1;i++) currArray[i]=currArray[i+1];
      currArray[--currLength]=0;
    }
    return ret;
  }
```

```
  /**
   * The size of the list
   * @return - the size of the list
   */
  public int size(){
    return currLength;
  }

  /**
   * Method  for ensuring that an index can be accessed.
   * @param index - the index we want to make sure the list holds.
   */
  public synchronized void ensureSize(int index){
    while(currLength<index+1) add(0);
  }

  /**
   * Getting an int from index index.
   * @param index - the index at which we want the value
   * @return - the value at index index.
   */
  public int get(int index){
    return currArray[index];
  }

  /**
   * Switch the values at positions ind1 and ind2
   * @param ind1
   * @param ind2
   */
  public void switchPoses(int ind1,int ind2){
    //no exception
    int tmp = currArray[ind1];
    currArray[ind1]=currArray[ind2];
    currArray[ind2]=tmp;
  }

  /**
   * Get an int[] that represents the current list. Be aware that the
   * running time is O(n) because a new int[] needs to be made.
   * @return - an int[]-representation of the list
   */
  public int[] getIntArray(){
    int[] ret = new int[currLength];
    for(int i=0;i<currLength;i++) ret[i]=currArray[i];
    return ret;
  }

  /**
   * Same as getIntArray except that you can specify that you do not
   * want a int[] longer than maxlength.
   * @param maxlength - the longest list you want
   * @return - an int[]-representation of the list
   */
  public int[] getIntArrayWithMaxLength(int maxlength){
    int l = maxlength<currLength?maxlength:currLength;
    int[] ret = new int[l];
    for(int i=0;i<l;i++) ret[i]=currArray[i];
    return ret;
  }
}
```

C.8.11 brille.utils.IOStats

```
package brille.utils;
/**
 * This is a simple interface to define some methods that all different kinds
```

```
 * of IOStats on different systems should have in common. It has proven
 * difficult to find the same kind of info on different Operating Systems,
 * so this turned out to be the only solution.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public interface IOStats {

  /**
   * Get a String representing the statistics for this device
   * @return
   */
  public String toString();

  /**
   * Getter for the device name
   *
   * @return the name of the device
   */
  public String getDeviceName();

  /**
   * This method is important for the way I/O-statistics is used here. The
   * goal of gathering I/O-statistics here, is to be able to measure the
   * amount of I/O-action on a given device from a start-time to an
   * end-time. To be able to do so, we gather the current statistics when we
   * start, and the same when we stop. This method calculates the difference
   * between the two, and hence we know what has happened between the start
   * and the end.
   * @param st - the old I/O-statistics we want to remove from the new ones.
   */
  public void diff(IOStats[] st,long starttime);
}
```

C.8.12 brille.utils.IOStatsGatherer

```
package brille.utils;

import org.apache.log4j.Logger;

/**
 * This class is an abstract class defining the overall workings of an
 * I/O-statistics gatherer. Unfortunately, gathering of I/O-statistics
 * is system dependent, and all operating systems where this is run needs
 * its own IOStatsGatherer. It should implement to methods,
 * startGatheringIOStats() and endGatheringIOStats(), which faciliate
 * taking I/O-statistics for a time interval, between the two calls.
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public abstract class IOStatsGatherer {

  protected final Logger logger = Logger.getLogger(getClass());

  /**
   * Method for start gathering I/O-statistics. When you want to
   * start, this method should be called. When you want to know
   * what has happened since you called this method, call the other
   * one.
   *
   */
  public abstract void startGatheringIOStats();

  /**
   * Call this method when you want the results of the I/O-statistics.
   *
```

```
   * @return - an array with IOStats-objects, one for each disk on
   * your system.
   */
  public abstract IOStats[] endGatheringAndReturnIOStats();
}
```

C.8.13 brille.utils.LinuxIOStats

```
package brille.utils;

import java.math.BigInteger;

/**
 * This class represents the I/O-statistics gathered on Linux.
 * They are all available from /proc/diskstats.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class LinuxIOStats implements IOStats{

  private String deviceName;
  private BigInteger readsCompleted;
  private BigInteger readsMerged;
  private BigInteger sectorsRead;
  private BigInteger nrOfMillisecondsSpentReading;
  private BigInteger writesCompleted;
  private BigInteger writesMerged;
  private BigInteger sectorsWritten;
  private BigInteger nrOfMillisecondsSpentWriting;
  private BigInteger nrOfMillisecondsSpentDoingIO;
  private BigInteger time;
  private long starttime;

  private static BigInteger zero = new BigInteger("0");
  private static BigInteger add = new BigInteger("4294967295");

  /**
   * Constructor
   * @param deviceName - name of the device we gather statistics for.
   * @param readsCompleted - number of reads completed
   * @param readsMerged - number of the completed reads that have been
   *     merged with other reads.
   * @param sectorsRead - number of sectors read alltogether
   * @param nrOfMillisecondsSpentReading - self-explaining
   * @param writesCompleted - same as above for writes.
   * @param writesMerged - same as above for writes.
   * @param sectorsWritten - same as above for writes
   * @param nrOfMillisecondsSpentWriting - self-explaining
   * @param nrOfMillisecondsSpentDoingIO - total nr of ms spent doing I/O.
   * @param time
   */
  public LinuxIOStats(String deviceName, BigInteger readsCompleted,
      BigInteger readsMerged, BigInteger sectorsRead,
      BigInteger nrOfMillisecondsSpentReading,
      BigInteger writesCompleted, BigInteger writesMerged,
      BigInteger sectorsWritten, BigInteger nrOfMillisecondsSpentWriting,
      BigInteger nrOfMillisecondsSpentDoingIO, BigInteger time){
    this.deviceName=deviceName;
    this.readsCompleted = readsCompleted;
    this.readsMerged = readsMerged;
    this.sectorsRead = sectorsRead;
    this.nrOfMillisecondsSpentReading = nrOfMillisecondsSpentReading;
    this.writesCompleted = writesCompleted;
    this.writesMerged = writesMerged;
    this.sectorsWritten = sectorsWritten;
    this.nrOfMillisecondsSpentWriting = nrOfMillisecondsSpentWriting;
    this.nrOfMillisecondsSpentDoingIO = nrOfMillisecondsSpentDoingIO;
```

```
    this.time = time;
  }


  public String toString(){
    return "Device "+deviceName+"\nReads completed: "
    +readsCompleted.toString()+"\nReads merged: "
    +readsMerged.toString()+"\nSectors read: "
    +sectorsRead.toString()+"\nNr of milliseconds spent reading: "
    +nrOfMillisecondsSpentReading.toString()+"\nWrites completed: "
    +writesCompleted.toString()+"\nWrites merged: "
    +writesMerged.toString()+"\nSectors written: "
    +sectorsWritten.toString()+"\nNr of milliseconds spent writing: "
    +nrOfMillisecondsSpentWriting.toString()+
    "\nNr of milliseconds spent doing I/O: "
    +nrOfMillisecondsSpentDoingIO.toString()+"\nStart time: "
    +starttime+"\nTime spent: "+time.toString();
  }


  private BigInteger checkOverflow(BigInteger check){
    if(check.compareTo(zero)<0) return check.add(add);
    return check;
  }


  public void diff(IOStats[] st,long starttime) {
    for(int i=0;i<st.length;i++)
      if(deviceName.equals(st[i].getDeviceName())){
        this.starttime = starttime;
        LinuxIOStats lsti = (LinuxIOStats)st[i];
        readsCompleted = readsCompleted.subtract(lsti.getReadsCompleted());
        readsCompleted = checkOverflow(readsCompleted);
        readsMerged = readsMerged.subtract(lsti.getReadsMerged());
        readsMerged = checkOverflow(readsMerged);
        sectorsRead = sectorsRead.subtract(lsti.getSectorsRead());
        sectorsRead = checkOverflow(sectorsRead);
        nrOfMillisecondsSpentReading =
          nrOfMillisecondsSpentReading.subtract(
              lsti.getNrOfMillisecondsSpentReading());
        nrOfMillisecondsSpentReading =
          checkOverflow(nrOfMillisecondsSpentReading);
        writesCompleted = writesCompleted.subtract(lsti.getWritesCompleted());
        writesCompleted = checkOverflow(writesCompleted);
        writesMerged = writesMerged.subtract(lsti.getWritesMerged());
        writesMerged = checkOverflow(writesMerged);
        sectorsWritten = sectorsWritten.subtract(lsti.getSectorsWritten());
        sectorsWritten = checkOverflow(sectorsWritten);
        nrOfMillisecondsSpentWriting =
          nrOfMillisecondsSpentWriting.subtract(
              lsti.getNrOfMillisecondsSpentWriting());
        nrOfMillisecondsSpentWriting =
          checkOverflow(nrOfMillisecondsSpentWriting);
        nrOfMillisecondsSpentDoingIO =
          nrOfMillisecondsSpentDoingIO.subtract(
              lsti.getNrOfMillisecondsSpentDoingIO());
        nrOfMillisecondsSpentDoingIO =
          checkOverflow(nrOfMillisecondsSpentDoingIO);
        time = time.subtract(lsti.getTime());
        break;
      }
  }

  /**
   * Sweet getters...
   */
  public String getDeviceName() {
    return deviceName;
  }
```

```
  public BigInteger getNrOfMillisecondsSpentDoingIO() {
    return nrOfMillisecondsSpentDoingIO;
  }

  public BigInteger getNrOfMillisecondsSpentReading() {
    return nrOfMillisecondsSpentReading;
  }

  public BigInteger getNrOfMillisecondsSpentWriting() {
    return nrOfMillisecondsSpentWriting;
  }

  public BigInteger getReadsCompleted() {
    return readsCompleted;
  }

  public BigInteger getReadsMerged() {
    return readsMerged;
  }

  public BigInteger getSectorsRead() {
    return sectorsRead;
  }

  public BigInteger getSectorsWritten() {
    return sectorsWritten;
  }

  public BigInteger getTime() {
    return time;
  }

  public BigInteger getWritesCompleted() {
    return writesCompleted;
  }

  public BigInteger getWritesMerged() {
    return writesMerged;
  }
}
```

C.8.14 brille.utils.LinuxIOStatsGatherer

```
package brille.utils;

import brille.BrilleDefinitions;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.math.BigInteger;
import java.util.ArrayList;

/**
 * The Linux version of an IOStatsGatherer
 * This version reads from /proc/diskstats and parses the output. This
 * implies that it will not work on 2.4-kernels. The same thing can
 * be accomplished there by mounting sysfs, but that is not supported
 * here as brille is not a commercial product, and I don't need to support
 * people unable to get a newer kernel:)
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class LinuxIOStatsGatherer extends IOStatsGatherer{
```

```
private IOStats[] current;
private long starttime;

public LinuxIOStatsGatherer(){
  current = null;
}

private IOStats[] getCurrentIOStats(){
  BufferedReader in = null;
  try{
    in = new BufferedReader(new FileReader("/proc/diskstats"));
  } catch(FileNotFoundException fnfe){
    logger.error("did not find /proc/diskstats. If this is really "+
        "Linux, you should update your kernel. You need a" +
        " 2.6-kernel to gather statistics. If you want to use 2.4,"+
        " please recode this stuff to use sysfs instead.",fnfe);
    return null;
  }
  ArrayList<LinuxIOStats> list = new ArrayList<LinuxIOStats>();
  String[] goodprefixes =
    BrilleDefinitions.PROPER_DISK_DEVICE_PREFIXES.split("\\|");
  try{
    String line = in.readLine();
    while(line!=null){
      String[] inp = line.split("[\\s]+");
      boolean good = false;
      for(int i=0;i<goodprefixes.length && !good;i++){
        int j=0;
        for(;j<Math.min(goodprefixes[i].length(),inp[3].length())
            && goodprefixes[i].charAt(j)==inp[3].charAt(j);j++);
          good = j==goodprefixes[i].length() && inp[3].length()==j+1;
      }
      if(good)list.add(new LinuxIOStats(inp[3],
          new BigInteger(inp[4]), new BigInteger(inp[5]),
          new BigInteger(inp[6]), new BigInteger(inp[7]),
          new BigInteger(inp[8]), new BigInteger(inp[9]),
          new BigInteger(inp[10]), new BigInteger(inp[11]),
          new BigInteger(inp[13]),
          new BigInteger(""+System.nanoTime())));
      line = in.readLine();
    }
  } catch(IOException ioe){
    logger.error("Problems reading from /proc/diskstats.",ioe);
  }
  try{
    in = new BufferedReader(new FileReader("/proc/stat"));
  } catch(FileNotFoundException fnfe){
    logger.error("did not find /proc/diskstats. If this is really "+
        "Linux, you should update your kernel. You need a" +
        " 2.6-kernel to gather statistics. If you want to use 2.4,"+
        " please recode this stuff to use sysfs instead.",fnfe);
    return null;
  }
  ArrayList<LinuxCPUStats> cpulist = new ArrayList<LinuxCPUStats>();
  try{
    String line = in.readLine();
    while(line!=null){
      String[] inp = line.split("[\\s]+");
      if(inp[0].regionMatches(0,"cpu",0,3) && inp[0].length()>3)
        cpulist.add(new LinuxCPUStats(inp[0],
          new BigInteger(inp[1]), new BigInteger(inp[2]),
          new BigInteger(inp[3]), new BigInteger(inp[4]),
          new BigInteger(inp[5])));
      line = in.readLine();
    }
  } catch(IOException ioe){
    logger.error("Problems reading from /proc/diskstats.",ioe);
  }
```

```
    IOStats[] ret = new IOStats[list.size()+cpulist.size()];
    for(int i=0;i<list.size();i++)ret[i]=list.get(i);
    for(int i=0;i<cpulist.size();i++) ret[i+list.size()]=cpulist.get(i);
    return ret;
  }

  public void startGatheringIOStats() {
    current = getCurrentIOStats();
    starttime = System.nanoTime();
  }

  public IOStats[] endGatheringAndReturnIOStats() {
    if(current==null)
      throw new IllegalArgumentException("Gathering of IOStats not started.");
    IOStats[] ret = getCurrentIOStats();
    for(int i=0;i<ret.length;i++) ret[i].diff(current,starttime);
    return ret;
  }
}
```

C.8.15 brille.utils.LongArrayList

```
package brille.utils;

/**
 * This class serves the same right as IntArrayList, to create a more
 * space-efficient and fast version of ArrayList<Long>
 *
 * @author Truls A. Ã¸Bjrklund
 *
 */
public class LongArrayList {
  private static final int stdInitialCapacity=10;
  private static final double stdFactor=1.1;
  private double factor;
  private long[] currArray;
  private int currLength;

  /**
   * Constructor where all parameters can be set
   * @param initialCapacity - the initial size of the underlying array
   * @param factor - the factor at which the size should grow (or shrink).
   */
  public LongArrayList(int initialCapacity,double factor){
    this.factor=factor;
    currArray=new long[initialCapacity];
    currLength=0;
  }

  /**
   * Constructor that uses the standard growing factor of 1.1
   * @param initialCapacity  - the initial size of the underlying array
   */
  public LongArrayList(int initialCapacity){
    this(initialCapacity,stdFactor);
  }

  /**
   * Constructtor that uses the standard initial capacity, 10.
   * @param factor - the factor at which the size should grow (or shrink).
   */
  public LongArrayList(double factor){
    this(stdInitialCapacity,factor);
  }

  /**
   * Constructor with standard values for initial capacity(10) and
```

```
 * factor(1.1).
 */
public LongArrayList(){
  this(stdInitialCapacity,stdFactor);
}

/**
 * Append d to the end of the list. Extend the list if necesseary.
 * Note that n calls to this method will give a running time of O(n).
 * The amortized complexity is thus O(1).
 * @param d - the long that should be appended.
 */
public void add(long d){
  if(currLength==currArray.length) extend();
  currArray[currLength++]=d;
}

private void extend(){
  int newSize=(int)(currArray.length*factor);
  long[] newCurr = new long[newSize];
  for(int i=0;i<currLength;i++) newCurr[i]=currArray[i];
  currArray=newCurr;
}

private void extend(int index){
  int newSize=(int)(currArray.length*factor);
  long[] newCurr = new long[newSize];
  for(int i=0;i<index;i++) newCurr[i]=currArray[i];
  for(int i=index;i<currLength;i++)newCurr[i+1]=currArray[i];
  currArray=newCurr;
}

/**
 * Add a long at a specified position (the list size will grow with 1).
 * @param d - the long to add.
 * @param index - the index at which the long should be added.
 */
public void add(long d, int index){
  if(currLength==currArray.length){
    extend(index);
    currArray[index]=d;
  }
  else{
    long tmp=currArray[index];
    long tmp2=0;
    for(int i=index+1;i<currLength;i++){
      tmp2=currArray[i];
      currArray[i]=tmp;
      tmp=tmp2;
    }
  }

}

/**
 * Just as set in ArrayList, but I think replace is a better name.
 * @param d - the long that should replace the current
 * @param index - at the given position.
 */
public void replace(long d,int index){
  currArray[index]=d;
}

private void shrink(int index){
  int newSize=(int)(currArray.length/factor);
  long[] newCurr = new long[newSize];
  for(int i=0;i<index;i++)newCurr[i]=currArray[i];
  for(int i=index+1;i<currLength;i++)newCurr[i-1]=currArray[i];
  currArray=newCurr;
```

```
    currLength--;
  }

  /**
   * Remove a long at the given index. Warning: This is not a very efficient
   * method
   * @param index
   * @return
   */
  public long remove(int index){
    long ret = currArray[index];
    int minsize = (int)(((double)currArray.length)/(factor*factor));
    if(currLength<minsize && currArray.length>10) shrink(index);
    else{
      for(int i=index;i<currLength-1;i++) currArray[i]=currArray[i+1];
      currArray[--currLength]=0;
    }
    return ret;
  }

  /**
   * The size of the list
   * @return number of longs in list
   */
  public int size(){
    return currLength;
  }

  /**
   * Method to make sure that the given index can be accessed
   * @param index - the index we will ensure we can access.
   */
  public synchronized void ensureSize(int index){
    while(currLength<index+1) add(0);
  }

  /**
   * Getting the long at the given position
   * @param index - the index at which we want to get the long.
   * @return - the long at index index.
   */
  public long get(int index){
    return currArray[index];
  }
}
```

C.8.16 brille.utils.NoMoreDocsException

```
package brille.utils;

/**
 * Exception thrown when the correct amount of terms is added to the index.
 *
 * @author Truls A. Ã¸Bjrklund
 */
public class NoMoreDocsException extends Exception{

  private static final long serialVersionUID = -4236037276752271499L;

  public NoMoreDocsException(){
    super();
  }

}
```

C.9 Extra code for getting I/O-statistics on FreeBSD

C.9.1 Header-file

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class brille_utils_FreeBSDLibAccesser */

#ifndef _Included_brille_utils_FreeBSDLibAccesser
#define _Included_brille_utils_FreeBSDLibAccesser
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     brille_utils_FreeBSDLibAccesser
 * Method:    get_stats
 * Signature: ([Ljava/lang/String;)[Ljava/lang/String;
 */
JNIEXPORT jobjectArray JNICALL Java_brille_utils_FreeBSDLibAccesser_get_1stats
  (JNIEnv *, jobject, jobjectArray);

/*
 * Class:     brille_utils_FreeBSDLibAccesser
 * Method:    get_numdevs
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_brille_utils_FreeBSDLibAccesser_get_1numdevs
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

C.9.2 Implementation

```
#include "brille_utils_FreeBSDLibAccesser.h"

#include <sys/resource.h>

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <devstat.h>

/**
 * The implementation of the method to get the number of disk devices
 * on this system. It is essentially a call to sysctlbyname(), with
 * kern.devstat.numdevs as an argument.
 */

JNIEXPORT jint JNICALL Java_brille_utils_FreeBSDLibAccesser_get_1numdevs
  (JNIEnv * env, jobject obj){
  int numdevs=0;
  size_t numdevsize;
  numdevsize = sizeof(int);
  if(sysctlbyname("kern.devstat.numdevs", &numdevs, &numdevsize, NULL, 0)==-1){
    return(-1);
  }
  return(numdevs);
}

/**
 * The method to retrieve the current statistics about all disks.
```

```
 * This is essentially a call to sysctlbyname with the argument
 * kern.devstat.all. There is also some memory alloction. The
 * inspiration for how to make this method was found in kernel source
 * files under FreeBSD. The most important files are iostat.c and
 * devstat.c
 *
 */
JNIEXPORT jobjectArray JNICALL Java_brille_utils_FreeBSDLibAccesser_get_1stats
  (JNIEnv *env, jobject obj, jobjectArray arr){
  int numdevs=0,i=0;
  size_t numdevsize;
  size_t dssize;
  char buf[100];
  jstring tmp;
  struct devinfo *dinfo;

  numdevsize = sizeof(int);
  dinfo = malloc(sizeof(struct devinfo));
  bzero(dinfo,sizeof(struct devinfo));

  if(sysctlbyname("kern.devstat.numdevs", &numdevs, &numdevsize, NULL, 0)==-1){
    return(arr);
  }
  dssize= (numdevs * sizeof(struct devstat))+sizeof(long);
  dinfo->numdevs = numdevs;
  dinfo->mem_ptr = (u_int8_t *)malloc(dssize);
  if(sysctlbyname("kern.devstat.all",dinfo->mem_ptr, &dssize, NULL, 0)<0){
    return(arr);
  }
  dinfo->generation = *(long *)dinfo->mem_ptr;
  dinfo->devices = (struct devstat *)(dinfo->mem_ptr + sizeof(long));
  for(i=0;i<dinfo->numdevs;i++){
    sprintf(buf, "%s%d",dinfo->devices[i].device_name, dinfo->devices[i].unit_number);
    tmp = (*env)->NewStringUTF(env, buf);
    (*env)->SetObjectArrayElement(env,arr,9*i,tmp);
    sprintf(buf, "%ld",dinfo->devices[i].bytes[DEVSTAT_READ]);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+1,tmp);
    sprintf(buf, "%ld",dinfo->devices[i].bytes[DEVSTAT_WRITE]);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+2,tmp);
    sprintf(buf, "%ld",dinfo->devices[i].bytes[DEVSTAT_FREE]);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+3,tmp);
    sprintf(buf, "%ld",dinfo->devices[i].operations[DEVSTAT_READ]);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+4,tmp);
    sprintf(buf, "%ld",dinfo->devices[i].operations[DEVSTAT_WRITE]);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+5,tmp);
    sprintf(buf, "%ld",dinfo->devices[i].operations[DEVSTAT_NO_DATA]);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+6,tmp);
    sprintf(buf, "%ld",dinfo->devices[i].operations[DEVSTAT_FREE]);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+7,tmp);
    sprintf(buf, "%d",dinfo->devices[i].block_size);
    tmp = (*env)->NewStringUTF(env,buf);
    (*env)->SetObjectArrayElement(env,arr,9*i+8,tmp);
  }
  return(arr);
}
```