Distributed Inverted Indexes

Simon Jonassen

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

Abstract

This document presents and compares different organizations for distributed inverted indexes for large document collections. Two main schemes are known as Global and Local Inverted Index organizations, or partitioning by the term ID's and partitioning by the document ID's. There are also a number of modifications and hybrids formed from these two organizations presented and discussed in this document.

This report provides a brief background overview for a distributed full-text retrieval system using an inverted index. The processing principles and the algorithms for query search in a stored index using the different partitioning schemes is presented and discussed in the background part. All the relevant previous work, some of the most important papers published in the last two decades and the work performed earlier by the author, are discussed with the purpose to explain differences in the results as an outcome of the assumptions and the techniques that have been used. The main intention for this is to discover at which circumstances, such as the system architecture, disk and network specific characteristics, document and query set specific features, ranking methods, etc., a particular partitioning approach can be superior.

A new simulation model together with a semi-mathematical description are provided to compare the performance of the methods presented. The main difference from the previous simulation model developed by the author is a higher simplicity of the process model itself, at the same time as it provides more realistic metrics and a higher number of different partitioning and retrieving methods. The metrics provided in the new simulation model are based on the tests and benchmarks performed by the author.

A great number of simulation experiments based on a real document collection and a real query set show the advantage of the local indexing over the global indexing in form of a higher query rate and smaller wait times. The local indexing can also provide an even better performance, if the associated disk-bottleneck issues can be resolved.

On the other hand, for a conjunctive query model, a variation of the global indexing known as the pipelined indexing may provide an even better performance. This indexing method provides also a better scalability with both the number of nodes and the number of the concurrent queries in the system.

The results obtained from the experiments partly support the results known from the previously published papers. All the differences between the results from the most recent publications and the results obtained from the simulation experiments can be explained by the limitations of a simulation model which fails to reproduce the complexity and all

the small details of a real processing system.

Three of the issues partly discussed but not tested in this master thesis are finally proposed as relevant topics for a further research.

Preface

This is a master thesis for the Master in Computer Technology program at the Department of Computer and Information Science at the Norwegian University of Science and Technology. The assignment was given by the Information Access Disruptions research group and carried out during five months of spring 2008.

The author would like to thank his supervisors, Svein Erik Bratsberg and Øystein Torbjørnsen, for valuable feedback and help; Ph.D student Truls A. Bjørklund for providing the source code of Brille and a dictionary dump of TREC GOV2 document collection, and master student Asjbørn A. Fellinghaug for providing a reference to the Terabyte TRACK 05 query set.

Simon Jonassen
Trondheim, 2nd June 2008

Contents

1 Introduction **1**

2 Background **3**
2.1 An Introduction to Search Engines . . . 3
2.2 Inverted Indexes . . . 4
2.2.1 The Vocabulary File . . . 4
2.2.2 Inverted Lists . . . 4
2.3 Inverted Index Creation . . . 6
2.4 Inverted Index Update . . . 6
2.5 Query Processing with an Inverted Index . . . 7
2.5.1 Query Processing Models . . . 7
2.5.2 Basic Algorithms . . . 9
2.5.3 Approximation Methods for the Vector Model . . . 11
2.6 Distributed Inverted Indexes . . . 14
2.7 Scalability of a Search Platform . . . 14
2.8 Partitioning Schemes for an Inverted Index . . . 14
2.8.1 Local Indexing . . . 16
2.8.2 Global Indexing . . . 19
2.8.3 Alternative Indexing Schemes . . . 23
2.9 Other Related Issues: Caching . . . 26

3 Previous Work and Results **27**
3.1 Tomasic and Garcia-Molina, 1992 . . . 27
3.1.1 Results . . . 29
3.1.2 Critics . . . 31
3.1.3 Relevance . . . 31
3.2 Jeong and Omiecinski, 1995 . . . 32
3.2.1 Results . . . 32
3.2.2 Critics . . . 33
3.2.3 Relevance . . . 33
3.3 Ribeiro-Neto and Barbosa, 1998 . . . 33
3.3.1 Results . . . 33
3.3.2 Critics . . . 34

3.3.3 Relevance . . . 34
3.4 MacFarlane, McCann and Robertson, 2000 . . . 34
3.4.1 Results . . . 35
3.4.2 Critics . . . 35
3.4.3 Relevance . . . 35
3.5 Badue, Ribeiro-Neto, Baeza-Yates, Zivani, 2001 . . . 36
3.5.1 Results . . . 36
3.5.2 Critics . . . 36
3.5.3 Relevance . . . 36
3.6 Xi, Sornil, Luo, Fox, 2002 . . . 37
3.6.1 Results . . . 37
3.6.2 Critics . . . 38
3.6.3 Relevance . . . 38
3.7 Badue, Ribeiro-Neto, Barbosa, Golgher, Zivani, 2005 . . . 38
3.8 Moffat, Webber, Zobel, Baeza-Yates, 2005 . . . 39
3.8.1 Results . . . 39
3.8.2 Critics . . . 41
3.8.3 Scientific Remarks . . . 42
3.8.4 Relevance . . . 42
3.9 Moffat, Webber, Zobel, 2006 . . . 43
3.9.1 Results . . . 43
3.9.2 Critics . . . 43
3.9.3 Relevance . . . 44
3.10 Jonassen, 2007 . . . 44
3.10.1 Results . . . 45
3.10.2 Critics . . . 47
3.10.3 Relevance . . . 48

4 State of the Art **49**
4.1 The Assignment Text and The Solution Approach . . . 49
4.2 The Approach . . . 49
4.3 The Previous Simulation Model . . . 50
4.3.1 Visualisation of the Simulation Results . . . 52
4.4 The Roadmap to a New Simulation Model . . . 52

5 Simulation Model **55**
5.1 Ideas and Decisions behind a New Simulation Model . . . 55
5.1.1 Framework . . . 56
5.1.2 Study of a Real Search Engine . . . 62
5.1.3 Processing Model . . . 64
5.1.4 Simulation of the Document Collection and the Query Set . . . 64
5.1.5 Algorithms and Metrics . . . 65
5.1.6 Visualisation Tool and Reporting . . . 75
5.2 Implementation of the New Simulation Model . . . 78

5.2.1 simulation.node . . . 78
5.2.2 simulation.query . . . 78
5.2.3 simulation.model . . . 80
5.2.4 simulation.process . . . 80
5.2.5 simulation.log . . . 82
5.2.6 simulation.micro . . . 83
5.3 Micro-Benchmarking and Parameter Estimation . . . 83
5.3.1 Network Characteristics . . . 84
5.3.2 Disk Characteristics . . . 84
5.3.3 CPU Characteristics . . . 85
5.3.4 Data Structures and Memory Requirements . . . 86
5.3.5 Document Collection Parameters and Characteristics . . . 87
5.3.6 Term Disjunction and Conjunction Frequency . . . 88

6 Simulation Experiments and Results **91**
6.1 Specifications for the Performance Evaluation . . . 91
6.2 The Plan . . . 92
6.3 The Schedule . . . 92
6.4 Experiment Results . . . 93
6.4.1 Baseline Experiments . . . 94
6.4.2 Node Number and Concurrency Level Experiments . . . 98
6.4.3 CPU configuration experiments . . . 101
6.4.4 Network configuration experiments - Bandwidth . . . 105
6.4.5 Disk configuration experiments . . . 107
6.4.6 Additional experiments . . . 112
6.4.7 Combination of the Experiment Results with the Previous Results 112

7 Conclusions and Further Work **113**
7.1 Further Work . . . 114
7.2 Interesting Topics Related to this Master Thesis . . . 114

Bibliography **117**

A Appendix **121**
A.1 Samples of Dictionary Data . . . 121
A.1.1 docstat . . . 121
A.1.2 querylog . . . 122
A.2 Network Microbenchmark . . . 124
A.2.1 clustis.c . . . 124
A.2.2 clustis.out . . . 126
A.3 An example experiment property file . . . 128
A.4 Trace Mode Simulation Reports for Baseline Experiments . . . 130

B Source code **143**
B.1 Simulation Model Source Code . . . 143
B.1.1 simulation.model.Config . . . 143
B.1.2 simulation.model.ModelS . . . 146
B.1.3 simulation.model.QueryLogReader . . . 149
B.1.4 simulation.processes.model.GeneratorProcess . . . 150
B.1.5 simulation.log.LogProcess . . . 151
B.1.6 simulation.log.LogWriter . . . 152
B.1.7 simulation.log.Statistics . . . 163
B.1.8 simulation.micro.HeapInterleaveTest . . . 165
B.1.9 simulation.micro.InterleaveTwoTest . . . 167
B.1.10 simulation.node.Node . . . 169
B.1.11 simulation.node.ResHandler . . . 174
B.1.12 simulation.processes.QueryProcess . . . 175
B.1.13 simulation.processes.QueryProcessGI . . . 177
B.1.14 simulation.processes.QueryProcessGIPM . . . 179
B.1.15 simulation.processes.QueryProcessHD . . . 181
B.1.16 simulation.processes.QueryProcessHDPM . . . 183
B.1.17 simulation.processes.QueryProcessLI . . . 185
B.1.18 simulation.processes.QueryProcessLIPM . . . 187
B.1.19 simulation.processes.QueryProcessPL . . . 189
B.1.20 simulation.processes.QueryProcessPLPM . . . 191
B.1.21 simulation.processes.SubQueryProcess . . . 194
B.1.22 simulation.processes.SubQueryProcessGI . . . 195
B.1.23 simulation.processes.SubQueryProcessGIPM . . . 196
B.1.24 simulation.processes.SubQueryProcessHD . . . 197
B.1.25 simulation.processes.SubQueryProcessHDPM . . . 199
B.1.26 simulation.processes.SubQueryProcessLI . . . 200
B.1.27 simulation.processes.SubQueryProcessLIPM . . . 202
B.1.28 simulation.query.IndexHitList . . . 203
B.1.29 simulation.query.IndexTools . . . 203
B.1.30 simulation.query.Query . . . 207
B.1.31 simulation.query.QueryResult . . . 210
B.1.32 simulation.query.SimpleIndexHit . . . 211
B.1.33 simulation.query.SimulatedIndexHitList . . . 211
B.1.34 simulation.query.SubQuery . . . 212
B.1.35 simulation.query.SubQueryResult . . . 214

List of Figures

2.1 Information Retrieval Process . . . 3
2.2 Partitioning by the document id . . . 15
2.3 Partitioning by the term id . . . 15
2.4 An Example of Inverted Index Partitioning for a Document Collection, [XSLF02] . . . 16

4.1 A class diagram for the simulation model used in [Jon07] . . . 50

5.1 Resource Handler Routine . . . 58
5.2 Process coopeartion using Method 2 . . . 59
5.3 Memory Handler Routine . . . 61
5.4 A class diagram for the simulation.query package . . . 79
5.5 A class diagram for some of the classes contained in the simulation.model, simulation.node and simulation.log packages . . . 81
5.6 A class diagram for the simulation.query package . . . 82
5.7 Joint frequency of two terms using the AND model obtained with the Join Method 1 . . . 88
5.8 Joint frequency of two terms using the OR model obtained with the Join Method 1 . . . 89
5.9 Joint frequency of two terms using the OR model obtained with the Join Method 2 . . . 90

6.1 The average QPS with a varied concurrency level using 4 nodes . . . 98
6.2 The average query response time with a varied concurrency level using 4 nodes . . . 99
6.3 The average QPS with a varied concurrency level using 8 nodes . . . 99
6.4 The average query response time with a varied concurrency level using 8 nodes . . . 100
6.5 The average CPU load with a varied number of CPUs . . . 101
6.6 The average QPS rate with a varied number of CPUs . . . 102
6.7 The average query response time with a varied number of CPUs . . . 102
6.8 The average CPU load with a varied CPU factor . . . 103
6.9 The average QPS with a varied CPU factor . . . 104
6.10 The average query response time with a varied CPU factor . . . 104

6.11 The average Ethernet load with a varied network bandwidth 105
6.12 The average QPS rate with a varied network bandwidth 106
6.13 The average query response time with a varied network bandwidth 106
6.14 The average CPU load with a varied number of disks 107
6.15 The average disk load with a varied number of disks 108
6.16 The average QPS with a varied number of disks 108
6.17 The average query response time with a varied number of disks 109
6.18 The average CPU load with a varied disk type 110
6.19 The average disk load with a varied disk type 110
6.20 The average QPS rate with a varied disk type 111
6.21 The average query response time with a varied disk type 111

A.1 Local Indexing - Node Status 131
A.2 Local Indexing - Process Status 132
A.3 Global Indexing - Node Status 133
A.4 Global Indexing - Process Status 134
A.5 Pipelined Indexing - Node Status 135
A.6 Pipelined Indexing - Process Status 136
A.7 Local Indexing - Node Status 137
A.8 Local Indexing - Process Status 138
A.9 Global Indexing - Node Status 139
A.10 Global Indexing - Process Status 140
A.11 Pipelined Indexing - Node Status 141
A.12 Pipelined Indexing - Process Status 142

List of Tables

3.1 Summary of previous work 28
3.2 The index size and the relative throughput of the system in [MWZB07] . . 42

5.1 Comparison between search engine alternatives 62
5.2 Variables to be used in the simulation model, part 1. 66
5.3 Variables to be used in the simulation model, part 2. 67

6.1 Results summary of the baseline experiments (50000ms, 4 nodes, MNQ 12) 94
6.2 Results summary of the baseline experiments (50000ms, 4 nodes, MNQ 12) 97

List of Algorithms

1 Basic query processing with the Boolean AND/OR model. 10
2 Basic query processing with the Vector model. 11
3 Query processing with the Vector model and a restricted number of accumulators. 12
4 A simple approximation algorithm using the vector model and the impact-ordered inverted lists 13
5 Query processing with the local indexing according to the boolean model . 17
6 Query processing with the local indexing according to the vector model. . . 18
7 Query processing with the global indexing according to the boolean model 20
8 Query processing with global indexing according to the boolean model . . . 21
9 Query processing with the global indexing according to the vector model . 22
10 Query processing with the pipelined indexing according to the vector model 24
11 LI query process, algorithm details 69
12 LI sub-query process, algorithm details 69
13 GI query process, algorithm details 70
14 GI sub-query process, algorithm details 70
15 PL query process, algorithm details 71
16 LIPM query process, algorithm details 73
17 LIPM sub-query process, algorithm details 74
18 GIPM query process, algorithm details 74
19 GIPM sub-query process, algorithm details 75
20 PLPM query process, algorithm details 76

Chapter 1

Introduction

Full-Text Retrieval Systems are a subset of Information Retrieval Systems which deal with the gathering, storage and access to textual information. Full-text retrieval is limited to provide a number of documents or document excepts from a document collection in return for a user-specified textual query.

The use of an index in the full-text retrieval systems was first introduced by Salton et al. [SM86]. An inverted index file allows to perform time and result effective search, even for a large document collection with a big number of document and query terms. On a single computer the index can be stored on a single disk or striped over a number of disks which will solve both the capacity and some of disk-access performance issues. However the use ofa distributed index and a number of computing nodes will improve the performance, not only for disk-access, but also for processing of the data itself.

Distributed Index Organizations include two main methods, replication and partitioning. *Replication* of an index allows a higher number of queries to be processed at the same time using a higher number of nodes. *Partitioning* of an index allows the same query to be partially processed on a number of nodes in parallel. A fully scalable search engine architecture can be viewed as a two-dimensional array of nodes, where one of the dimensions describes replication and the other one describes partitioning. To handle a different query rate the system can be scaled within the first dimension, and to handle a bigger document collection or to reduce the processing time for a single query, the system can be scaled within the second dimension.

A modern search engine, such as those developed by FAST and Google, provides scalability in the way described above. In fact, all of those systems split the document index into a number of partitions specified by the document ID range, maintaining a *Local Inverted Index* for a subset of the document collection on each node. An alternative approach is to split the document index by the term ID, maintaining a subset of a *Global Inverted Index* on each node. A number of hybrid approaches and improvements for these two fundamental methods were proposed. However the performance of these can be speculated when the underlying architecture and document collection features are taken into the account.

A number of papers aimed to describe and compare different index partitioning meth-

ods were published in the last two decades. The target for some of those papers was to establish whether one of the two principal approaches is the superior one, while the others supposed to describe some alternative approaches and demonstrate a performance improvement. All these papers have used different architectures, scientific methods and query and document collection models, and the results were also different.

The previous work performed by author of this master thesis supposed to evaluate previous study and, if it was possible, to find at which circumstances a global index organization can outperform a local index organizations or vice versa. As it turns out, the results depend on a careful choice of the system parameters and the implementation of the system model. The final answer is still unclear, and a better simulation model is required to provide more reliable results.

This document and all the related work are based on the work performed during the Specialization Project in Complex Computer Systems, autumn 2007. Some of the information presented in the background and previous study part is aimed to refine the information presented in the project report, [Jon07].

Organization of this document is as follows. Chapter 2 gives a complete introduction to inverted indexes, retrieving models, implementation concepts, approximation methods, index replication and partitioning. Chapter 3 will present previous work relevant for this master thesis and results from the work performed during the preceding project. Chapter 5 presents a new simulation model together with a detailed description of the ideas and the guidelines for simulation of system behavior and estimation of system performance. All the metrics needed for the simulation model are estimated using micro-benchmarking, both the methods and the results are presented in the same chapter. Chapter 6 will finally present and discuss the results gathered from the simulation. The final conclusions and the proposals for further work are given in the Chapter 7.

Chapter 2

Background

This chapter introduces text search engines which use an inverted index. All the information in this chapter is an improved version of the background part from [Jon07] and is based on the ideas and the theory described in [ZM06, WMB99, BR99, LM06].

2.1 An Introduction to Search Engines

A complete search engine can be viewed as a number of different processes used to gather and store the content of a document collection, together with a number of processes used to answer incoming queries using the stored information. The whole system can be illustrated by Figure 2.1.

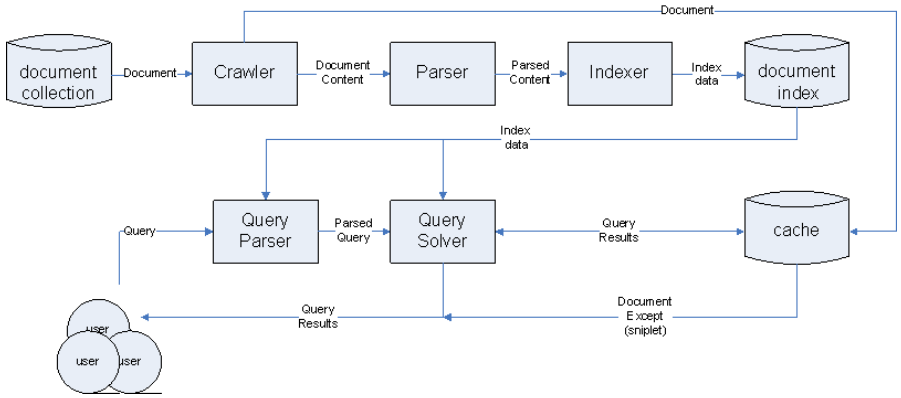

Figure 2.1: Information Retrieval Process

A Crawler Process explores the specified document collection and sends every document containing in it to a Parser Process. The Crawler Process can use cross-references between the documents and document locations to rank the results at the query time. The Parser Process in its turn first strips the document content from the document markup, which can also be used later for the results ranking. Then, all the letters are

folded to lower case and all the *stop words* (i.e. most frequent words such as *a, about, after, again, all, almost,* etc., which cannot be used to identify the document content uniquely) are removed. To limit the number of distinct words with the same roots and to allow more flexibility for the later search, all the words are *stemmed* by removing all the irrelevant prefixes, suffixes and endings.

The preprocessing operations presented are very important for later search, as they limit the amount of storage used and the work demand. On the other hand, these operations can induce enormous consequences for a later search. Phrases such as "the who" and "to be or not to be", and the words with common stems but different meaning must be processed carefully.

When all the documents are processed, all of the preserved words they contain, *terms*, are stored in a special way to be able to perform a time and result effective query search later. There are many ways to do this. The method considered in this master thesis is to use an Inverted Index, as it will be explained below. Other common methods are to use bitmaps, signatures, etc.

2.2 Inverted Indexes

The most effective and a relatively simple approach to represent and store the content of a document to be able perform a query search ever known is to use an Inverted Index. An inverted index consists of two parts, a vocabulary for the relevant terms (or their stems) and their occurrences. This structure is very similar to a book index presented in the end of any book ever known to the reader.

2.2.1 The Vocabulary File

The vocabulary file for an inverted index consists of a lexically sorted array of terms with pointers to the associated inverted lists. Different types of codings can be applied to save the storage space. If the resulting structure is too large to be stored in the main memory, it can be partially stored on the disk, while the most used part of it will be placed in the main memory. In this case it results in a two-level dictionary where the first level is stored in the main memory as an array and the second level is stored on the disk. The first level of the vocabulary can also be stored in a hash-map, which provides fast access to the second level pointers. A variation of this approach is to use a B-tree where all the leaf nodes stored on the disk, while the internal nodes are stored in the main memory. [MG] shows that with a such organization, an index with 1.6 million records consuming 320MB can be stored using 8KB blocks (400 records) by maintaining only 4000 tree-nodes in the memory, about 1MB of data in total.

2.2.2 Inverted Lists

An inverted list for each term contains all the term occurrences in the document collection and some additional information used for result ranking. Inverted lists are normally

placed on the disk, but a small fraction of the most frequently used posts can be cached in the main memory.

Granularity

Inverted lists may have many different representations. One of the most important issues is the granularity of the inverted index. At the first extreme it may contain only the IDs of the documents containing a given term, while at the second extreme it may contain both the document IDs and each single occurrence position for each single term and document pair. In the first case, the inverted lists are small, but it is impossible to match a phrase or expression. The second case is completely opposite to the first one. An alternative to both methods is to split each document in a number of blocks and use the block number as a position identifier. This approach is less storage-consuming than the one storing each single occurrence of each single term, while it is possible to determine whether the required terms occurs near each other or not. Any additional information about the term occurrence frequency in the document collection and each single document can be stored in the inverted lists. This information is usually used to calculate the similarity score, a metric describing how relevant a given document is for a given query.

Content

For the models which will be explained shortly a suitable inverted list representation is something like $[f_t :< d : f_{t,d} >]$. In this notation f_t is the frequency of a term t, d is a document id and $f_{t,d}$ is the frequency of the the term t within the document d. Alternatively, the number of occurrences can be used instead of the term frequency. In addition, all the occurence positions of a single term in each single document can be included to provide phrase matching. A simple compression method for a such inverted index is to use *d-gaps*, i.e. use the difference between two consecutive document IDs instead of IDs themself.

Although there are two other representations which can result in a better performance as it will be explained later. Since an inverted list contains only term frequencies and document numbers, its content can be ordered either by document IDs or by term frequencies. The approach demonstrated so far is known as the *document-ordered inverted lists* since it orders the content of a single inverted list by the document ID. Two other organizations are known as the *frequency-ordered inverted lists* and the *impact-ordered inverted lists*.

The content of an frequency-ordered list is something like $[f_t :< f_{t,*} : k :< d >>]$ ordered by decreasing $f_{t,*}$ value, which can be interpreted as a kind of a histogram for a given term. In this case the D-gap approach cannot be used, but a similar approach, perhaps *f-gaps*, can be used to reduce the size of an index file.

Impact-ordered lists have a syntax similar to frequency-ordered lists, but in addition they store an *impact value* for a block of inverted list records. The impact value is defined

by the query processing model.

Later organizations combined with a number of approximation methods result in a time and space effective processing as it will be explained later in Section 2.5.3.

Compression Issues

D-gaps and f-gaps can slightly reduce the size of an inverted index. Unary, β- or γ other parameterless codes applied carefully can be used to reduce the storage consumed by the position values. Parameter based codes such as Golumb codes can also be used. While it requires more effort to estimate the parameter values for each term. Other data-compression methods [WMB99] can be applied to reduce the size of inverted lists. This techniques can be either adaptive (dynamic) or non-adaptive (static). The latter results in a much better compression but the whole inverted list is then needed to be decoded before the required data can be used. A possible solution in this case is to divide the inverted list for a single term into blocks and compress them on their own.

In general, compression can reduce the size of an index entry from 48 to about 12 bits. A such data volume reduction would also improve the disk access time. Otherwise there is a trade-off between the time saved by reducing the size of the index and the additional time spent on decoding of the index data. Because of difficulties to estimate the processing demands, the compression methods will be never considered in the simulation model described later.

2.3 Inverted Index Creation

There are a number of different methods to build an inverted index. First, the indexing process can make a pass through the whole collection and count the number of document occurrences for each term, then allocate the required number of records on disk and make a second pass storing the right record to the right place. Alternatively, the indexer can perform only a single pass through the document collection and store a tuple for each term-document match, thereafter all the tuples can be sorted to create a term-ordered index. An even more efficient method is to create a large index is to build a number of small indexes in the memory and flush them to disk, then to merge them hierarchically until a single index is obtained.

2.4 Inverted Index Update

If some of the documents in the collection are removed or updated, or a number of new documents is added to the collection, the index is needed to be updated. This is a very important problem for the last decade. There are three general methods to handle updates. First, the whole index can be rebuilt once an update is performed. If the document collection is large and dynamic this method is very ineffective. But it can be used for small collections with a low update frequency. Another method is to create a small index for the updated part of the collection and then merge it with the rest of

the collection on an update. This method is also problematic for a large and dynamic collection, but it avoids to re-index unchanged documents. Third method is to create a partial index for the updated part of the document collection, but do not merge it with the rest of the index at once. The partial index can be also maintained in the memory. A background process can be scheduled to merge the partial index and the main index once in a while.

In the later presentation of a small search engine called Brille, an inverted index can be actually stored in a number of small indexes which are then flushed to the original one by a background process. A hierarchical index is a variation of an inverted index which maintains a number of inverted files of different size, instead of a single inverted file. Each of the index files maintained by a hierarchical index is usually by a factor bigger than the previous one, the content of a such file is moved to a larger one when the index file becomes full. Brille maintains a list of update-indexes and uses a background process to flush them to the main index.

2.5 Query Processing with an Inverted Index

2.5.1 Query Processing Models

There are three classic models used to perform a search in an inverted index and to rank the results: the Boolean model, the vector and the probabilistic model. All the models can be described in terms of a term t, a document d and the associated weight $w_{t,d}$. It is expected that the total number of the documents in the collection is N and the occurrence frequency of the term t is f_t for the whole collection and $f_{t,d}$ inside the document d. Result ranking returns the results according to a similarity value, $sim(d, q)$, and limits the number of the returned results to a predefined value r.

Boolean Model

The Boolean model is the simplest one, the $w_{t,d}$ is simply 1 if the term t occurs in the document d and 0 if not. The weighting function is described below. A Boolean query is a logical expression on a number of terms using operators such as AND, OR and NOT. A simple approach to process a such query is to transform it to a disjunctive normal form, which is a disjunction of conjunctions (an OR of ANDs), then to solve each of the conjunctions by taking only the documents containing every term in the given conjunction, finally merge the results from the previous stage. The similarity function is not relied on the weighting function, but it is ofter calculated afterwards using either one of the other models described here or an alternative model such as HITS/Page-Rank or Layout Based Boosting.

$$w_{t,d} = \begin{cases} 1 & \text{if } t \in d \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

Probabilistic Model

The probabilistic model uses the probability that a given document d is relevant and the probability that this document is irrelevant to estimate the its similarity value for a given query q.

$$sim(d,q) = \frac{P(R|d)}{P(\overline{R}|d)} \tag{2.2}$$

$$sim(d,q) = \frac{\prod_{t \in q} P(k_t|R) \times \prod_{t \notin q} P(\overline{k_t}, R)}{\prod_{t \in q} P(k_t|\overline{R}) \times \prod_{t \notin q} P(\overline{k}, \overline{R})} \tag{2.3}$$

Equation 2.3 shows the basic formula used to calculate the similarity value of a single document. R is the subset of the document collection containing the approved documents. Since the relevance scores for all the documents in the collection are expected to be independent, the Bayes' theorem and the chain rule can be used to calculate the total similarity value for any subset of the document collection. Usually an initial subset of the document collection is chosen at random, then some of the documents are excluded from the subset while some other documents are included into the subset in the way to maximize the total similarity value.

$$P(k_t|R) = \frac{S_t}{S} \tag{2.4}$$

$$P(k_t|\overline{R}) = \frac{N_t - S_t}{N - S} \tag{2.5}$$

The remaining problem is how to estimate the relevance probability of a single document itself. Usually these values are given by something like Equation 2.4 and Equation 2.5 for any set N. In these equations S is the cardinality of subset of the documents described as relevant, S_t is the number of documents in the subset containing the term t, N is the cardinality of the set itself and finally the N_t is the subset of S containing the term t. Now it depends only on which documents are considered as relevant, usually it is determined by the user feedback. This is why the probabilistic models are more often used in the interactive document content management rather than in the full-text retrieval. This is also the reason why the probabilistic model will never be used any later in this report.

Vector Model

The vector model is relatively simple and very flexible, and it provides a relatively high precision[1] together with a high recall[2]. The idea behind is to view the document d and the query q as two vectors of the associated weights $w_{t,d}$ and $w_{t,q}$. Then the similarity value is given by Equation 2.6.

[1]- the number of the relevant documents retrieved to the number of the documents in the retrieved

[2]- the number of the relevant documents retrieved to the number of the relevant documents in the collection

$$sim(d,q) = \frac{\sum_t (w_{t,d} \times w_{t,q})}{W_d \times W_q} \tag{2.6}$$

The $w_{t,d}$ and $w_{t,q}$ are usually estimated using Equation 2.7, where N is the number of the documents in the collection and f_t and $f_{t,d}$ is the term collection and document frequencies as it was mentioned earlier. This approach is also known as the TFxIDF scheme since it uses the term frequency in the document and the inverse term frequency for the whole collection.

$$\begin{array}{ll} w_{t,d} = 1 + \log_e f_{t,d} & w_{t,d} = \log_e(1 + N/f_t) \\ W_d = \sqrt{\sum_t w_{t,d}^2} & W_q = \sqrt{\sum_t w_{t,q}^2} \end{array} \tag{2.7}$$

Other Models

Other models that are out of scoop of this report are layout based model, web based models such as HITS and PageRank, Bayesian network based models, fuzzy set models, etc.

Okapi BM25 is a promising scheme based on the probabilistic model, while it uses the term frequency and the inverse document frequency which is not typical for a pure probabilistic model described above. Equations 2.8 - 2.9 show the formulas used to calculate the weighs and the similarity score for a document using Okapi BM25. W_A is the average document weight (average W_d), $f_{t,q}$ is the term frequency in the query and k_1, k_2 and b are system-defined parameters.

$$sim(d,q) = \sum_t (w_{t,d} w_{t,q}) \tag{2.8}$$

$$w(t,d) = \frac{(k_1+1) f_{t,d}}{k_1((1-b)+b\frac{W_d}{W_A})+f_{t,d}} \quad w(t,q) = \ln(\frac{N-f_t+0.5}{f_t+0.5})\frac{(k_2+1)f_{t,q}}{k_2+f_{t,q}} \tag{2.9}$$

Different sources such as [Bjø07] provide results which show the advantages of Okapi BM25 scheme compared to the TFxIDF. However this scheme is not discussed in this report any further.

2.5.2 Basic Algorithms

This part of the report will now present the basic algorithms for the Boolean and the Vector models. Note that many alternative, more sophisticated and effective processing schemes are possible, while this report presents only some of them at a high level. These algorithms will be later used to demonstrate the difference between the processing on a single-node system and the processing on a group of nodes.

Using the Boolean Model

The query processing algorithm for a single Boolean AND query is very simple. The inverted lists for all of the required terms are first fetched to the memory and merged using a number of pointers. Using the document-ordered lists the lowest document pointer can always be increased until all the pointers refer to the same document id, in this case there is an occurrence. All the other common occurrences can be obtained by traversing the rest of the inverted lists in this manner. More sophisticated Boolean queries can be executed by performing a number of AND queries and then interleaving the results just as it was described above. Finally, result ranking and result set extraction must be performed at the end. Algorithm 1 presents the whole processing for a simple version of the Boolean model.

```
1  Bring q to the disjunctive normal form;
2  foreach conjunction C in query q do
3      Let R_D be an empty result set;
4      Fetch all the inverted lists I_t for every non-negated term t in C;
5      Assign a pointer to the start of each of the inverted lists;
6      repeat
7          if all of the pointers refer to equal d values then
8              Add the referred value to the result set;
9              Increment all the pointers by one;
10         else
11             Increment the pointer referring to the lowest value;
12     until Any pointer runs over the end of its list ;
13 Let R_q be an empty result set;
14 Assign a pointer to the start of each result set R_D;
15 repeat
16     Add the lowest referred value v to R_q;
17     Increment all the pointers referring to the values equal v;
18 until All pointer run over the end its list ;
19 Calculate similarity scores for documents in R_q;
20 Extract r top-scored candidates;
```

Algorithm 1: Basic query processing with the Boolean AND/OR model.

Using The Vector Model

The processing algorithm for the Vector Model is actually easier than the one for the Boolean Model. It simply allocates an array of accumulators, one for every document in the collection. Then it uses the inverted list content of the query terms to increment the accumulator values. Since W_q is constant for a query it is never considered in the calculation of the resulting similarity score, as it is demonstrated by Algorithm 2. Note

that it is expected that the W_d values are stored on disk since they are constant for each document in the collection.

```
1  Let A be an array of accumulators, one for every document in the collection;
2  Set every A_d to 0;
3  foreach term t in q do
4  |   Fetch the inverted list I_t;
5  |   Calculate w_{t,q} based on f_i from the inverted list I;
6  |   foreach record d_i:f_{t,d} in I do
7  |   |   Set A_d ← A_d + w_{t,q} w_{t,d} calculated using f_{t,d};
8  Get W_d values;
9  Set S_d ← A_d / W_d;
10 Extract r top candidates;
```

Algorithm 2: Basic query processing with the Vector model.

Sorting and Extraction of Top-values

Extraction of the top candidates can be performed easily by using a Min-Heap. The trick is to initialize a heap on the first r elements of the accumulator list. Then each of the $|A| - r$ remaining elements is evaluated against the lowest value stored in the heap. If the value of an element is greater than the lowest heap value, then the lowest heap value is extracted and a new value is inserted. The total processing time for this stage is $O(N + 2N \log r)$.

Sniplet Generation

An important detail here is that the result for a query is usually more than just a document id. Usually it requires a *sniplet*, i.e. a fragment of the document, if possible containing and highlighting the searched terms. This part requires a cached copy of the document which can be used at the query time.

2.5.3 Approximation Methods for the Vector Model

The problem with the implementation of the vector model described above is that requires too much memory and processing. There are a number of tricks that can be applied to improve this.

First of all, the value of the most of the accumulators created by Algorithm 2 is either 0 or insignificantly low, so there is no need to maintain an accumulator for every document in the collection. Perhaps using $r = 1000$ and a collection of several millions of documents, a number of accumulators about 30000 may be large enough to provide good precision and recall. The problem is then to find when to create accumulators and what to do when the maximum number of the accumulators is reached. Two heuristics solving

the later issue is called *break* and *continue* . The break-heuristic stops the processing when the number of accumulators is exceeded, while the continue-heuristic restricts the creation of new accumulators, but allows to update existing accumulators. The *continue* heuristic is the most used one. [AdKM01] Algorithm 3 demonstrates the *continue* heuristic using up to L accumulators.

```
1  Let A be an empty set of accumulators;
2  foreach term t in q by decreasing w_{t,q} do
3      Fetch the inverted list I_t;
4      Calculate w_{t,q} based on f_t from the inverted list I;
5      foreach record d:f_{t,d} in I do
6          if |A| < L and A_d ∉ A then
7              Add a new accumulator A_d;
8          if A_d ∈ A then
9              Set A_d ← A_d + w_{t,q} w_{t,d};
10 Get W_d values;
11 Set S_d ← A_d / W_d;
12 Extract r top candidates;
```

Algorithm 3: Query processing with the Vector model and a restricted number of accumulators.

Since it is needed to maximize the value of allocated accumulators, the creation of the accumulators is usually performed by maximizing $w_{t,d}w_{t,q}$ for newly created accumulators. A further observation shows that the records with the lowest f_t and the highest $f_{t,d}$ values will normally result in the most optimal candidates. This is where *frequency-ordered inverted lists* become advantageous. Since the content of each list is organized by descending $f_{t,d}$ it requires just to process all the lists in parallel restricting $w_{t,d}w_{t,q} > S$, where S is chosen to be as high as possible. When the number of proved candidates reaches 0, S is recalculated and the whole process repeated until all the data is processed. The main advantage of this solution is that it limits not only the memory usage, but it may also limit the amount of the inverted lists to be fetched from disk. The processing time can be limited to a predefined value and accumulation for a query stops when the value exceeded. The drawback of this approach is that it requires a data structure which allows a fast accumulator look-up and insertion.

A further improvement of this approach can be obtained by using the *impact-ordered inverted lists* with the impact value defined by $w_{t,d}/W_d$, since the impact of a single inverted list record is not $w_{t,d}w_{t,q}$, but $w_{t,d}w_{t,q}/W_d$. In this case the impact value of a block of records is stored together with the inverted lists. At the query time the inverted lists are processed nearly as described above. The differences are that the minimal impact value is used instead of the minimal product of term weights, and that the accumulators are increased not by the real impact of a document, but by an approximated value. The approach is demonstrated by Algorithm 4. Note that the algorithm does not provide the

same result set as the original one, but [ZM06] ensures that the quality of the answer set is unchanged.

```
1  Let A be an empty set of accumulators;
2  Fetch the first blocks of inverted lists for every term t in q;
3  Let s_t be the impact values of fetched blocks;
4  repeat
5      Let b be the block with the highest value of C = s_t w_{t,q};
6      foreach d referenced in b do
7          if |A| < L and A_d ∉ A then
8              Add a new accumulator A_d;
9          if A_d ∈ A then
10             Set A_d ← A_d + C;
11     Fetch a new block if possible, update the value of s_t;
12 until processing time is exhausted ;
13 Extract r top candidates from A;
```

Algorithm 4: A simple approximation algorithm using the vector model and the impact-ordered inverted lists

2.6 Distributed Inverted Indexes

A simple inverted index on a single machine is proved to be an effective approach with good precision and recall characteristics. However a problem arises when the document collection becomes too large or the query rate becomes too high to be processed by a single computing node. In this case a number of computer nodes and a careful processing architecture are needed.

A *Distributed Inverted Index* is an inverted index stored on a number of computer nodes with a provided cooperation for index maintenance and query solving. Usually it means both *partitioning* and *replication*. Replication is usually obtained by storing either a full-copy of the index on a number of other machines, or storing only a small and frequently used fraction of the index. Partitioning is obtained by dividing the index into a number of disjoint subsets, where each subset is then assigned to one of the nodes. The main difference between these two is that replication does not require as much cooperation between the nodes as partitioning does. From this point this report looks at the partitioning methods for an inverted index, while the replication part is still an interesting issue to research and evaluate.

2.7 Scalability of a Search Platform

As it was mentioned in the Introduction chapter, a scalable search engine [Ris04] can be viewed as a two-dimensional array of processing nodes. With this organization, the columns replicate the index, while the rows split it into a number of partitions. A number of additional nodes is then needed to route a query through the processing node array. The advantage of this system is that processing time and delay for a single query remains nearly the same when either the size of the document collection or the query rate is increased, by adding a number of new rows or new columns. The only difference in the final performance is induced by an increase in the routing time, which is logarithmic to the array dimensions.

2.8 Partitioning Schemes for an Inverted Index

There are a number of different partitioning schemes and their variation presented during the last two decades, but this report looks only at the partitioning schemes designed for a multi-node system. Two essential approaches are known as *global indexing* and *local indexing*, or *partitioning by the term id* and *partitioning by the document id*.

The main difference between these is that for the first one all of the terms in the vocabulary are partitioned over a number of nodes, while for the second one all the documents in the collection are partitioned over a number of nodes. The difference in the partitioning concepts leads to a completely different index construction, processing algorithms and associated trade-offs and benefits. There are also a number of alternative approaches such as *pipelined indexing* which differs from the global indexing in the query processing, as it uses a pipeline rather than a scatter-gather/master-slave processing

node 1	$d_1 \times t_1$	$d_1 \times t_2$	...	$d_1 \times t_k$
	...	...	...	...
	$d_{n/m} \times t_1$	$d_{n/m} \times t_2$	...	$d_{n/m} \times t_k$
node 2	$d_{n/m+1} \times t_1$	$d_{n/m+1} \times t_2$	...	$d_{n/m+1} \times t_k$
	...	...	...	...
	$d_{2n/m} \times t_1$	$d_{2n/m} \times t_2$	...	$d_{2n/m} \times t_k$
...	...	...	...	...
node m	$d_{(m-1)n/m+1} \times t_1$	$d_{(m-1)n/m+1} \times t_2$	...	$d_{(m-1)n/m+1} \times t_k$
	...	...	...	...
	$d_{n/m} \times t_1$	$d_{n/m} \times t_2$	...	$d_{n/m} \times t_k$

Figure 2.2: Partitioning by the document id

node 1	*node* 2	...	*node* m
$d_1 \times t_1$ ... $d_1 \times t_{k/m}$	$d_1 \times t_{k/m+1}$ ... $d_1 \times t_{2k/m}$	...	$d_1 \times t_{(m-1)k/m+1}$ ... $d_1 \times t_k$
$d_2 \times t_1$ ... $d_2 \times t_{k/m}$	$d_2 \times t_{k/m+1}$ ... $d_1 \times t_{2k/m}$	...	$d_2 \times t_{(m-1)k/m+1}$ ... $d_2 \times t_k$
...	...	...	...
$d_n \times t_1$ ... $d_n \times t_{k/m}$	$d_n \times t_{k/m+1}$ ... $d_n \times t_{2k/m}$	...	$d_n \times t_{(m-1)k/m+1}$ ... $d_n \times t_k$

Figure 2.3: Partitioning by the term id

- d1: a b a c d, d2: a d e a, d3: b c a b; d4: b
- LI:
  N1: a:<d1:1>,<d1:3> b:<d1:2> c:<d1:4> d:<d1:5>
  N2: a:<d2:1>,<d2:4> d:<d2:2> e:<d2:5>
  N3: a:<d3:3> c:<d3:2> b:<d3:1>,<d3:4>
  N4: b:<d4:1>
- GI:
  N1: a:<d1:1>,<d1:3>,<d2:1>,<d2:4>,<d3:1>
  N2: b:<d1:2>,<d1:5>,<d3:1>,<d3:4>,<d4:1>
  N3: c:<d1:4>,<d3:2>
  N4: d:<d2:2> e:<d2:3>
- HD (chunk size 4):
  N1: a:<d1:1>,<d1:3>,<d2:1>,<d2:4>
  N2: b:<d1:2>,<d1:5>,<d3:1>,<d3:4>
  N3: a:<d3:1> c:<d1:4>,<d3:2>
  N4: b:<d4:1> d:<d2:2> e:<d2:3>

Figure 2.4: An Example of Inverted Index Partitioning for a Document Collection, [XSLF02]

model, or *hybrid indexing* which splits its inverted lists into a number of chunks which are then mapped to a number of nodes. Figures 2.2 - 2.4 visualizes the ideas.

2.8.1 Local Indexing

When the local indexing is used, the document collection is divided into a number of equally large subsets and each of the subsets is then assigned to a node. Since a node is responsible only for its own documents, a local index for its document subset is created. This can be performed by a single node on its own and no other nodes are needed to be involved. An index update such as an addition, deletion or modification of a single document requires to update the index on a single node which can be again performed by this node on its own.

The main advantage of the local indexing is that most part of the query processing can be performed by the node containing the required index data. The node responsible for solving of a query requires only to broadcast the query to all of the other nodes and combine the results returned. However, the processing is somewhat different for the boolean and the vector models. Query processing according to the vector model has also many possible optimizations which will be shortly presented.

As it will be explained later, it is expected that the length of the inverted list fetched by a single node is l_t/n. However it requires that each term in the collection has equally

many occurrences in each of the document subsets. The problem in this case is how to map the documents to the nodes. If the documents are mapped to the different nodes using a simple modulo-hash based on the document id and the number of nodes, in the worst case a single term can be contained only in a single partition. A mapping where all the nodes would have an equally high load is a NP-Complete problem.

Query Processing according to the Boolean Model

Query processing according to the boolean model requires basically nothing more than a broadcast of the query followed by a retrieval of the partial results. If the local results from the nodes do not provide any similarity score, upto r random results are chosen from the provided local results by the receptionist node (i.e. the node responsible for this query). Otherwise, upto r top-scored results are chosen based on the associated similarity score. Given that a node provides upto r' local results, the receptionist node can normally chose between upto $n \cdot r'$ results. Usually r' is chosen to be equal r, since in the worst case only the local results from a single node will be used in the final result set. Algorithm 5 illustrates the idea.

```
1 Broadcast the query to all the nodes;
2 foreach node in parallel do
3 |   Retrieve the inverted lists from the disk;
4 |   Process the local results according to the bolean model;
5 |   Send up to r' local results to the receptionist node;
6 Receive up to r' results from each of the nodes;
7 Return up to r results as the answer set;
8 [Opt: provide a sniplet for each of the results in the answer set];
```

Algorithm 5: Query processing with the local indexing according to the boolean model

Query Processing according to the Vector Model

Query processing using a local index according to the vector model is only slightly more complicated than the one using the boolean model. The major differences here are that (1) results returned from a node are always ranked and (2) results returned from the receptionist node must contain the r top-scored results chosen among the local results. A modified solution is given by Algorithm 6.

The algorithm provided describes the processing at a very high level. Processing of the local results using the vector model operations requires to increment the accumulator values according to the equations provided earlier or similar. In this case information about the global term frequency is needed. This implies that after constructing an index, each node requires to exchange the f_t-values for its own terms with the values stored by the other nodes. Alternatively, these values can be replicated. The final retrieval of the

```
1 Broadcast the query to all the nodes;
2 foreach node in parallel do
3     Retrieve the inverted lists from the disk;
4     Process the local results according to the vector model;
5     Send up to r' top-scored local results to the receptionist node;
6 Receive up to r' results from each of the nodes;
7 Return up to r top-scored results as the answer set;
8 [Opt: provide a sniplet for each of the results in the answer set];
```

Algorithm 6: Query processing with the local indexing according to the vector model.

top-scored local results can be implemented using a min-heap, just as it was described in the previous chapter.

In the final stage r top-scored must be chosen from up to $n \cdot r$ provided local results. This can be implemented either by using a min-heap which results in an asymptotic complexity $O(nr \log(r))$, or by using a multi-way merge which results in an asymptotic complexity $O(nr)$. Despite to a factor of $\log(r)$ the min-heap solution provides an advantage, since the multi-way merge requires for all partial results to be stored before the post-processing can begin. If the number of results required is low, the imbalance between the node load or index size is high or a higher multiprogramming level is provided, the min-heap solution is a better alternative.

The most important observation here is that for both the boolean and the vector model it requires to perform $|q|$ disk seeks on each node, $n \cdot |q|$ disk seeks in total. A number of papers describing the index partitioning suggest that the total volumes of the inverted lists fetched by each single node and are expected to be equal. Note that in the worst case it will be false. A term can be more common in the one of the sub-collections than in the others, it would induce more imbalance. On the other hand, when the number of documents and the number of words are very high, the real values are expected to be nearly equal to the average value, but not necessary.

Optimizations of the Processing Algorithms using the Vector Model

One of the most important advantages of the local indexing is that query processing for a subset of the document collection can be performed by a single node on its own. It results in that the approximation techniques described earlier can be used in the first stage of the query processing.

Sniplet Generation

Another advantage of the local indexing is in the sniplet generation. If the query processing requires to provide a document fragment along with the results returned, the receptionist node can extend the query processing with a stage when it retrieves stored

cache copies of the documents by given id's. The main advantage here is that the node responsible for storing index records for a given document can also store a cached copy of this document. It can be stored either explicitly by storing a full copy of the document or implicitly by using a high granularity index.

A short summary of the advantages and disadvantages

The main advantages and the disadvantages of the local indexing can be summed up to the following:

- The index is relatively easy to construct and update, but an additional information/information exchange is needed if the processing model requires any document collection related data, not only the collection subset related data.
- The query processing is relatively easy, requires only slightly more effort than the processing on a monolithic system.
- On a system with n nodes the local indexing requires n times as many disk seeks as a monolithic system, but these can be performed in parallel. On the other hand, the size of the inverted lists fetched by a single node and the amount of the work performed by a single node are expected to be n times smaller.

2.8.2 Global Indexing

Using the global indexing the vocabulary is divided into a number of equally large subsets and each of the subsets is then assigned to a node. This results in that a single node contains complete inverted lists for a number of terms. Since the different documents will occur on different nodes, the index construction and update are more complicated than for the local indexing.

The index construction can be simply performed by creating a local index for a subset of the document collection on each node, then sending a local inverted list for a term to the node responsible for this term. This node needs then to re-merge all the gathered inverted lists and calculate and store the document collection frequency for this term. An index update is complicated since it involves a number of nodes. A solution is to store a small in-memory index for new/updated terms before updating the stored index. But it may also be problematic since it requires a high level of cooperation and synchronization between the nodes.

The main advantage of the global indexing is that some of the nodes are not involved in the processing of a single query. During the processing of a typical query containing $|q| < n$ terms, at least $n - |q|$ nodes are expected to be idle. Therefore the global indexing provides a higher concurrency than the local indexing does. The drawback of this is that the query processing itself is much more complicated than for the local indexing.

Similar to the local indexing, a very critical issue which combines index construction and query processing is mapping of terms. The question is which terms should be assigned to which node. The easiest solution is to use a hash function which maps a term to a node

either lexicographically or in any other way without using any statistical information. The problem in this case is that it results in a high load imbalance since some more frequent terms can then occur on the different nodes.

Another solution is to use the term rank in the document collection to determine the hosting node. It requires additional processing and has a drawback - terms mapped to a single node can now be more frequent in the query set than the terms having a similar term rank, which will again lead to a high imbalance. So a better solution is to use both the term rank in the document collection and the expected term rank in the query set to determine the hosting node. This will not result in an optimal solution, but it should maximize the load balancing as much as possible. On the other hand, it is difficult to predict the query rank and all the small changes in the query term frequency will affect the load balancing. A final solution is to allow some of the most used inverted-lists to migrate to the less saturated nodes. This solution is even more complex, but it should both maximize the load balancing and automatically adapt to the changes in the query term frequency. Term mapping for the global indexing is a very difficult and interesting issue itself and there are many different techniques that can be researched and tested.

Query Processing according to the Boolean Model

Query processing according to the boolean model can be performed in two different ways. In the first case, the receptionists sends a number of sub-queries to the different nodes and receives the inverted lists for the terms involved in the query. Then it uses the received index data to perform the processing on its own. The only advantage in this case is a parallel disk access, the disadvantages are a high network and processing loads at the receptionist node. The idea behind is demonstrated by Algorithm 7.

```
1 Send a sub-query to each of the nodes containing the inverted lists for the relevant
  terms;
2 foreach active node in parallel do
3 |   Retrieve the inverted lists from the disk;
4 |   Send the retrieved term occurrences to the receptionist;
5 Receive the results from each of the nodes;
6 Process the retrieved data according to the boolean model;
7 Return up to r documents as the answer set;
8 [Opt: provide a sniplet for each of the results in the answer set];
```

Algorithm 7: Query processing with the global indexing according to the boolean model

A more sophisticated approach is demonstrated by Algorithm 8. In this case a query is first transformed to a disjunction of term conjunctions. For each of the conjunctions a number of sub-queries is transferred to the nodes, conjunctions for the terms contained on a single node can be processed by a single node on this own. The results are then needed to be transferred to the receptionist node. The receptionist needs then to interleave the local results for the different conjunctions. When all the conjunctions are processed, the

receptionist node needs to merge the results to provide a complete answer.

It is also possible to combine the terms from the different conjunctions in a single query transferred to a single node. But it would not reduce the post-processing time in the later stage while it would complicate the sub-query processing.

```
1  Transform the query into DNF;
2  foreach conjunction of terms do
3      Send sub-queries to the nodes containing the inverted lists for the relevant
       terms;
4      foreach active node in parallel do
5          Retrieve the inverted lists from the disk;
6          Generate results using boolean AND operations;
7          Send the retrieved document id's to the receptionist;
8      Interleave the local results;
9  Receive the results from each of the nodes;
10 Merge the partial results and return up to r top-scored documents as the answer
   set;
11 [Opt: provide a sniplet for each of the results in the answer set];
```

Algorithm 8: Query processing with global indexing according to the boolean model

Query Processing according to the Vector Model

The processing algorithm for the vector model looks to be easier than the one used for the Boolean model. Each of the nodes needs to process the inverted lists for the relevant terms and accumulate the similarity scores, just like for a monolithic systems. However, in the next stage the accumulated values are transferred to the receptionist node where they are combined/accumulated together. Finally the receptionist node can use a min-heap or similar to determine the top-ranked documents. Algorithm 9 demonstrates this

idea.

```
1 Send a number of sub-queries to the nodes containing the inverted lists for the
  relevant terms;
2 foreach active node in parallel do
3 |   Retrieve the inverted lists from the disk;
4 |   Accumulate the similarity values according to the vector model;
5 |   Send the accumulated values to the receptionist;
6 Receive the partial values accumulated by the nodes;
7 Accumulate the partial values together;
8 Return up to r top-ranked documents as the answer set;
9 [Opt: provide a sniplet for each of the results in the answer set];
```

Algorithm 9: Query processing with the global indexing according to the vector model

Optimizations of the Processing Algorithms using the Vector Model

The main disadvantage of the global indexing is that it induces a higher network and processing loads on the receptionist node. The problem lies in how many accumulators are needed to be returned to the receptionist node. In the worst case it requires an accumulator for each document in the collection, multiplied by the number of nodes involved in the query. This data must be accumulated and post-processed by the receptionist node.

Limiting the number of accumulators returned to the receptionist must be performed very carefully, since it can destroy the original ranking order. But there are at least two different solutions to the problem. First, each node can send the number of the accumulators and the highest accumulator value it has. The receptionist can then use this information to determine the fraction of the accumulators it needs from each node.

Another solution is to use a *query bundle*. The receptionist sends first a sub-query to the node containing either the term with highest document frequency or the highest number of terms. The number of accumulators returned from this node are expected to be minimal. The accumulator list returned to the receptionist node and can then be included into the later sub-queries. Alternatively the accumulator array can be transferred back and forth between the receptionist node and the other nodes according to increasing lowest document frequency, where each of the descending nodes will update the accumulator array. *Accumulator thresholding techniques* can now be applied. In the latter case the early termination technique can be used - after a time limit the receptionist stops sending/receiving sub-queries and moves to the post-processing stage.

Sniplet Generation

In contrast to the local indexing, with the global indexing a cached copy for a document cannot be stored implicitly using a high granularity index. Also the mapping of a document to a node has nothing to do with the term to node mapping. It implies that

some of the nodes not involved in the term processing can be later involved in the query processing if they are responsible for the sniplet generation for one or more results for this query.

A short summary of the advantages and disadvantages

- A global index is much more difficult to construct and update than a number of local indexes.
- The number of disk seeks performed by all of the nodes is only $|q|$, but in the worst case all the disk seeks would be performed by a single node. The size of the inverted lists fetched for each term is the same as the original size. Combining these two, the worst case the processing time for inverted lists will be equal or worse than the processing time on a monolithic system.
- The amount of data to be transferred to the receptionist node and the amount of work to be performed during the post-processing may be very high. This results in a potential bottleneck at the last stage of the query processing.
- It is expected that different nodes have dramatically different work load using the global indexing resulting in a high load imbalance. In addition the term mapping is a very critical issue which affects the load balancing characteristics.
- It is very difficult to apply known optimization techniques to the global indexing. Even if it possible, the performance advantage gained will be very difficult to predict.
- Due to the fact that some of the terms having a similar document frequency are more popular in the query set than the others, the load imbalance of the global indexing is expected to be high. However, a number of dynamic mapping strategies can be applied to improve the load balancing. The price for this is an additional amount of workload required to move the inverted lists and to update the dictionary structures.

2.8.3 Alternative Indexing Schemes

In addition to the local and the global indexing a number of alternative indexing methods is known. Two most interesting of these are *the hybrid indexing method* by Sornil et al. and *the pipelined indexing method* by Moffat et al..

Pipelined Indexing

The pipelined model can be viewed as an alternative to the *query bundle* optimization technique for the global indexing. The only difference is that, instead of sending an accumulator set/array between the nodes containing the relevant terms, a query bundle is routed through these nodes. In this case, the last node in the route takes over the task of

the receptionist node, it post-processes the results and returns the answers. Accumulator limiting techniques and early termination can be used during the processing. Algorithm 10 demonstrates the idea.

```
1  Generate a route over the nodes containing the terms involved in the query based
   on the statistical data;
2  Send the query structure and an empty accumulator set to the first node in the
   route;
3  foreach active node in turn do
4      Receive the accumulator set from the previous node;
5      Retrieve the inverted lists from the disk;
6      Accumulate the similarity values according to the vector model;
7      if The last node is reached or the time limit exceeded then
8          Return up to r top-ranked documents as the answer set;
9          [Opt: provide a sniplet for each of the results in the answer set];
10     else
11         Send the updated query and the accumulator structure to the next node in
           the route;
```

Algorithm 10: Query processing with the pipelined indexing according to the vector model

The most important issue here is how to choose the route for a query-bundle. The easiest approach is to use the increasing lowest f_t for each node. But [MWZB07] [MWZ06] show that it results in a low load balancing if used alone. Since the workload for a single term is a product of the term frequency and the document frequency, $L_t = Q_t \cdot B_t$, the document collection terms should be processed according to the decreasing L_t and mapped to the node having the lowest workload so far. The L_t value used for a such mapping can be calculated using the Q_t from the previous query history. This information must be stored on the receptionist node in addition to term mapping information itself.

The main advantage of this scheme is that it requires less work at the receptionist node. Further, it provides a higher concurrency, however a single node can easily become a bottleneck if too many queries are routed through it. The main disadvantage of this technique is that it induces a high network load and high imbalance. Because of the network transfers, the pipelined indexing cannot result in a better performance than a monolithic system can provide. But it provides an advantage in multiprogramming/concurrency and distributed storage.

Hybrid Indexing

Another variation of the global indexing designed by Sornil [Sor01] is known as the *hybrid indexing*. In this scheme a global inverted list for a single term is divided into a number of chunks which are then distributed between the nodes using $(term_{id} \oplus chunk_{id}) \mod n$ as the id of the destination node. The main advantage with this is that lists contained on a single node are shorter and the load balancing is supposed to be much higher. The

downside of this is that it introduces a higher number of disk accesses than with the global indexing.

The processing algorithm is almost similar to the one provided for the global indexing itself, the only difference is that the receptionist node needs to determine all the nodes that may contain a single term. This information can be estimated using the document frequency.

Index creation and update are even more problematic than for the global indexing since there are a greater number of small indexes to be stored. But an update of a single document requires to update only a number of small chunks instead of the modification of the whole index. On the other hand, all the global index information must to be updated, and it may be also difficult to determine which chunks must be updated and which are not.

Different chunk sizes will result in different performance. Too small chunks result in too many disk seeks, while too large chunks make the hybrid indexing to look more like the global indexing. Sornil [Sor01] suggest that a chunk size at 1024 entries provides the best results.

Finally, since the processing with the hybrid indexing is similar to the global indexing, most of the optimization techniques applicable for the global indexing suppose to work with the hybrid indexing.

2.9 Other Related Issues: Caching

The purpose of this master thesis is to analyze the performance issues of the different partitioning schemes presented so far. As it will be presented later, a simulation approach is chosen to perform this task. However, one important issue that is not taken care of by the simulation model is caching of the inverted lists. For a real search engine, caching can result in a great performance improvement if it is implemented properly.

For a real search engine, the submitted queries tend to be correlated over the time [3]. So the query answers themselves can be cached. On the other hand, users have a tendency to modify their queries [SWJS01] and submit them for a re-evaluation, or try different permutations of popular query phrases. Therefore caching of the inverted lists can improve the performance even better.

One of the key issues for the distributed inverted indexes and parallel processing in general is the processing imbalance. The imbalance is defined as a ratio between the longest execution time and the average execution time along the parallel processes. It means also that the imbalance is defined by the performance of the slowest process.

As it was demonstrated by Badue et al. [BBR+07], the sources of the imbalance among homogeneous[4] index servers in a web search system are the disk cache, the total size of the memory and the number of the servers in the cluster.

Surprising enough, the imbalance depends on the ratio between the query size and the query frequency. [BBR+07] shows that queries having this ratio between 0.25 and 4 are large lists and frequent, but they are cached. While all the other queries have either small lists and are cached or have long lists and are not cached. The former of these results in a much shorter processing time, while the latter results in a much longer processing time. Either of these results in a high performance imbalance. From the results presented in [BBR+07], long and infrequent queries correspond to about a half of a query set.

A further investigation by Badue et al. looks at three different scenarios for query processing: No caching; a best case, when all servers performs from cache, and a worst case, when at least one server has a long execution time while all the others use cache. The best case and no cache result in best performance. But a larger number of servers increases the chances for the worst case, and less memory has the same effect, since the cache region is much smaller.

To summary up, a high number of the query servers and a smaller cache size result in a much higher imbalance. The imbalance itself depends also not only on hardware characteristics, but also on the queries submitted by the public. For a later discussion of the simulation results it must be kept in mind that both the performance and the load balancing of a real search engine caching the inverted lists may be completely different.

[3] `http://www.google.com/trends` allows to see the chronological and geographical popularity of terms and phrases

[4] ie. all of the servers has same configuration, so it excludes the imbalance induced by differences in hardware or software implementation.

Chapter 3

Previous Work and Results

The first part of this chapter presents an improved version of the "Previous Work and Results" chapter from [Jon07]. The original text is modified to eliminate all the irrelevant information, while some additional information is added. The final version contains a complete overview of the performed work, results, critics and comments about how relevant the performed work is to this thesis and which facts can be used or questioned.

3.1 Tomasic and Garcia-Molina, 1992

In the paper *'Query Processing And Inverted Indices in Shared Nothing Text Document Information Retrieval Systems'* [TG93], Tomasic and Garcia-Molina look at the distributed index organization for text document retrieval systems and compare different index organizations using a probabilistic simulation model of the database and queries.

The inverted index organizations studied in this paper differ from those presented so far. The main difference is that these organizations adapt distribution at the disk level, not only at the node level.
The organizations discussed in this paper are:

- *System* - each disk keeps a part of the index file. This is similar to the global indexing at the disk level.
- *Host and I/O Bus* - each host or I/O Bus keeps an index for its own documents, but the index may be striped over a number of disks to reduce the internal fragmentation. This is similar to the local indexing at the node level, but hybrid indexing at the disk level.
- *Disk* - each disk keeps an index for its own documents. This is similar to the local indexing at the disk level.

There are also three optimization techniques for the *system* organization:

- Prefetch I - determine a query keyword k with the shortest interleaved list, then: (1) send a single sub-query containing k to the host that handles k, and (2) attach

Authors	Query Result Model Architecture	Document Collection Query Collection	Superior model
[TG93]	Boolean (AND-only) probabilistic + simulated, shared-nothing	- -	host (disk is worst)
[JO95]	Boolean analytical + simulation, shared-everything	synthetic uniform + Zipf synthetic	local
[RB98]	Vector shared-nothing	TREC3 50 real queries	global
[MMR00]	Probabilistic shared-nothing	base1 and base10 subsets of VLC2 50 short queries adopter from TREC-7	local
[BBRZ01]	Vector shared-nothing	TREC3 50 real and 2000 artificial queries	global
[XSLF02]	N/A shared-nothing	TREC9/10 100GB 200 real queries	hybrid (local is worst)
[MWZB07]	Boolean shared-noting	TREC Gov2 420GB pseudo-natural	local (global is worst)
[MWZ06]	Boolean shared-noting	TREC Gov2 420GB GOVQ	local (global is worst)

Table 3.1: Summary of previous work

a partial answer set from that node to all other sub-queries. This way the lists produced from nodes in (2) are expected to be very short.

- Prefetch II - analogous to Prefetch 1, but use a sub-query with the largest number of keywords.
- Prefetch III - use Prefetch II, when the choice is ambiguous - use Prefetch I.

Authors chose to look at the Boolean AND-only model which implies a simple estimation for the size of the result set. The term distribution for the document collection itself is modeled using an equation similar to Zipf's law. The query-term distribution is modeled using a uniform distribution which cuts off most infrequent words as it presented by Equation 3.1.

$$Q(t) = \begin{cases} 1/uT & \text{if } 1 \leq t \leq uT \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

The size of the answer set is then obtained by multiplying the total number of documents by the probability that a single document contains the required query terms. The expected length of the index list for a term in the number of entries is then given by Equation 3.2 and the number of scored documents is given by Equation 3.3.

$$|I_t| = Z(t)WD \tag{3.2}$$

$$|R_q| = D \prod_{t_i \in q}^{k} (1 - e^{-WZ(t_i)}) \tag{3.3}$$

The complete system behavior is described using the equations presented so far, a number of equations for disk, CPU and LAN access and a number of system variables. The whole system is then finally simulated using `DeNeT`.

3.1.1 Results

Authors expect that, as with the global indexing, the *system* organization has a potential bottleneck at the receptionist node, which interleaves inverted lists gathered from other nodes with its own, since it requires more network resources and a higher CPU-load. However, as with the local indexing, the *disk* approach, which merges inverted lists instead of interleaving them, needs a higher number of disk seeks. Also as it was mentioned earlier, regardless to the index organization, the same amount of data would be fetched from the disk using any of the organizations.

From the results presented, the *system* organization is the worst one in terms of both the query response time and the query throughput. The problem lies in a high network load, since all the inverted lists must be transferred to the receptionist node, which causes a bottleneck. Prefetch I, II and III result in a significantly better performance since the amount of data transferred later is dramatically reduced, but a single sub-query is needed

to be processed before any other sub-queries can be scheduled. Therefore the performance of an improved system organization is only two times better than the performance of the original one, while the network load is reduced by a factor of eight or more.

The *disk* organization provides a four times better performance than the system organization. The improvement is limited by a high disk utilization, just as it was expected. The I/O utilization is lowest with this organization. The host organization performs only slightly better than the disk organization, while the disk utilization is much lower than with the disk system organization and the network utilization is slightly lower than with the Prefetch I. But the I/O bus utilization is highest for the *I/O bus* organization.

The *I/O bus* organization shows the best average query response time and query throughput, nearly five times better than the disk organization. The disk utilization is high, but is slightly lower than with the disk organization. LAN utilization is nearly equal to the utilization with the Prefetch I and the I/O utilization is somewhat higher than for the disk organization, but lower than for the host organization. This organization shows the highest CPU load (60.9%), but it does not seem to be a problem.

An increase in the maximum keyword rank affects the system approach most dramatically, while the disk organization seems to perform only slightly better. The reason is that the disk organization has more load balancing among its disks. In a worst case, the query response time with the system organization is more than five times higher than with the disk organization. Because of reduction in network load the Prefetch I shows a good improvement for low maximum keyword rank, but still has a worst case query response time nearly 2.5 times higher than with the disk, I/O bus or host organizations.

On the other hand, an increase in disk seek time affects the disk approach mostly. Experiments show that, with a disk seek time at 0 milliseconds, the disk organization performs slightly better than the host organization does, but with a disk seek time at 100 milliseconds the query response time for the disk organization is just as large as for the system organization. The increase is by a factor of six. I/O bus has a somewhat similar behavior, but the increase observed has a factor of three. Other organizations are observed to be unaffected by the increased disk seek time.

With an increase in the multiprogramming level, the host organization outperforms the I/O bus organization. But the disk organization is getting worse with a higher multiprogramming level, because of a disk access bottleneck on each node. The system organization has no improvement with an increase in the multiprogramming level, while the query throughput for the Prefetch I is improved by a factor of three at 30 processes per host. This confirms the suspicions about the limitations in the scalability of the system organization caused by a network bottleneck.

With an increasing number of words in a single query the query response time increases most for the system organization and its optimizations, and least for the host and I/O bus organizations.

All the results show that the host and I/O bus organizations are the most suitable ones. While the system and disk organizations show both a low performance and a low scalability. Finally, the authors try to see what happens if the network bottleneck at the

receptionist is removed and the disk time is increased. This experiment shows that for 16 hosts with one I/O bus per host, a high network bandwidth and a high disk seek time (about 80 ms), the Prefetch I outperforms the alternative organizations.

3.1.2 Critics

The final observation from the experiments is very interesting since it shows that unrewarding to the earlier tests, the system organization and its modifications can have a superior performance on a high-speed network if the disk seek time is increased. However, it requires the disk seek time to be at least 65ms, which is at least ten times greater than an expected value for a real system. On the other hand, these results will be true if all the other system parameters would be improved by a factor of ten and the disk time will be unchanged.

This paper can be criticized for using a limited simulation model which considers only simple Boolean AND queries. However, it has a number of advantages. AND-only queries provide more simplicity and precision in the estimation of the total number of the results for a single query. A simulation model with synthetic document and query sets makes it easy to alter the system parameters and run many different tests in an easy and flexible way.

On the other hand, fetches from the disk was not modeled. Otherwise it would induce a higher disk load, if both the sniplet generation and the query execution were performed by the same set on nodes. Further, the simulation model has no pipelined resource access that would possibly provide a greater throughput for the system organization at a higher multiprogramming level. The algorithms implemented have no early termination, while the authors mention that it could be a great improvement. The simulation model is closed and high penalties for the system throughput are caused by large response times. Finally, the broadcast functions were not so well implemented, so the network issues are should be re-validated. These issues result in that the prefetch organizations could possibly obtain better results than they did under the tests presented so far.

3.1.3 Relevance

This paper was the first article about the distributed inverted indexes the author of this report has studied and it is the oldest one presented in this report. The organizations presented and tested by the authors differ from the organizations presented in the previous chapter, but the simulation model and the tests performed are very impressive and inspiring. In fact, the final observations from this paper show that, unrewarding to the current system setup, there can exist an alternative setup where an organization showing the best results can be equal or worse than the one showing the worst results. As it will be demonstrated later, this is also true for a later discussion about whether the global or the local indexing is the most suitable organization. This paper is the source for many interesting observations and arguments presented in the previous chapter. The equations presented in this paper can be reused by a future simulation models.

3.2 Jeong and Omiecinski, 1995

The article *'Inverted File Partitioning Schemes in Multiple Disk Systems'* [JO95] by Jeong and Omiecinski shows an analytical model for the basic inverted file partitioning schemes on a shared-everything multiple disk systems tested by a simulation on a synthetic document collection with a probability distribution based on the Zipf's law.

As in the previous paper, the authors use a Boolean query model on a simulated computer multi-node system. The partitioning methods described this time are similar to those presented in the previous chapter.
The article introduces also two alternative techniques for the term partitioning approach:

- Partition by Term 1 - the inverted lists are grouped into partitions of equal size
- Partition by Term 2 - the inverted lists are grouped according to access frequencies in addition to partition size

In addition to a uniform query term distribution model used by Tomasic and Garcia-Molina (Equation 3.1) the authors use a skewed query distribution-model (Equation 3.4).

$$Q(t) = \begin{cases} C * Z(t) & \text{if } 1 \leq t \leq uT \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

3.2.1 Results

The main problem appointed by the authors is the need for multiple I/O requests to the posting file for each term in a query with the partitioning by the document id, but the data transfer time is expected to be much shorter with this technique. The authors appoint also to that the partitioning by the term id can result in a poor query performance due to the I/O load imbalance among disks, especially if the term distribution frequencies are highly skewed.

The skew in the query term distributions has a solid impact on the simulation results. From the experiments, as the term distribution becomes more skewed, the partitioning by the document id becomes more advantageous. In a highly skewed environment the performance ratio of the partitioning by the document id increases together with the number of disks. But for the partitioning by the term id, heavy I/O loads is observed on the disks that keep the high frequency terms. The reason for a such performance degradation within the partitioning by the term id is the performance imbalance. But with a uniform distribution the partitiong by the term id is more effective, since it requires fewer I/O requests.

The partitioning by the document id provides more disk utilization in general. The authors say also that the partitioning by the document id gains a relatively high throughput increase with an increase of multiprogramming level, also when the data distribution is highly skewed. Optimization techniques for the partitioning by the term id provide some improvement, but the performance is not as good as with the partitioning by the document id. As the number of disks increases, the performance improvement with load

balancing decreases, if the skew is high. But if the skew is low, the performance of obtained with the optimization techniques increases. The partitioning by the term id is highly dependent on the data skew, and it can be advantageous only when the skew is low.

3.2.2 Critics

The authors of this paper use a shared-everything system, which is not so practical for a distributed search engine and has different memory and disk access issues than a shared-nothing system. Both the number of documents and the number of words per document in the simulation model are very low.

The most critical issue associated with this paper is that no experiments werer performed to see what happens when the network access or disk characteristics change. So the experiments performed show only how the system performs at a particular setup.

3.2.3 Relevance

This paper presents another simulation model which can be an inspiration source for a later simulation models. A new issue introduced by the authors is the skew in the query term distribution. Another important issues introduced are the number of disks and the disk site. This paper contains also a number of interesting conclusions and observations (presented above) which can be used in the later work. The most important one is that the partitioning by the document id is the method of choice when the query term distribution and the number of disks per node are both high.

3.3 Ribeiro-Neto and Barbosa, 1998

In their paper *'Query Performance for Tightly Coupled Distributed Digital Libraries'* [RB98], Ribeiro-Neto and Barbosa describe an analytical model for the performance analysis of an inverted index distributed over a network of workstations. The analytical model presented is tested with a small simulator using the Tipset/TREC3 document collection and 50 real queries. The system described in this paper uses disjunctive queries processed according to the vector model with the weighting function given by Equations 2.6 - 2.7, instead of conjunctive queries discussed by earlier papers.

3.3.1 Results

One of the most important arguments presented in this paper is that, for the global indexing, there are some machines that are not involved in the processing of a single query, and many of the query terms may be mapped to a single machine. Therefore a higher concurrency is essential for the global indexing. There are also some problems with the ranking using the global indexing, since only a small fraction of the query result information is available. Also using the local indexing, a query must be processed by every single node, but with the global indexes only a single sub-query may be sent.

The overall conclusion is that a global index organization can be quite advantageous in presence of fast communication channels. The authors also experiment with the number of machines used, however the number of queries and the size of the document collection are both constant. The global index outperforms the local index in provided results. The authors suggest that number of disk seeks performed locally drops as the number of the network machines gets higher. But a central broker may become a bottleneck. The authors state finally that a fast network and disk access are essential for the global indexing method.

3.3.2 Critics

The mathematical model presented in this paper seems to be too vague at the first glance. But a closer look makes this model to appear very flexible and informative. However, the model cannot predict the execution time for a single query in presence of other queries in the system.

The results from the simulation performed supposed to agree with the analytical model presented in this paper. However, it is difficult to see how many queries have been executed at any given time. I addition, such small number of executed queries makes the results to be difficult to generalize for a large scale system.

3.3.3 Relevance

This paper is excellent at presenting the basic algorithms for the global and the local inverted indexes and describing their pros and cons. It is also one of the earliest papers studied by the author of this report. A lot of the theory in the previous chapter is taken from this paper.

The most interesting idea presented by the authors is the fact that the local indexing exploits a *parallel* query processing while the global indexing exploits a *concurrent* processing.

The conclusions about the relationship between the global and local indexing using different disk seek time and the network bandwidth seem to be true from the simulation results performed by the author of this report during the preliminary work. But they need to be verified with a better model, as it will be described later.

Both the analytical model and the experiments performed are very inspiring and they will be used as a reference work for a later simulation model.

3.4 MacFarlane, McCann and Robertson, 2000

In their paper, *'Parallel Search using Partitioned Inverted Files'* [MMR00], MacFarlane, McCann and Robertson use BASE1 and BASE10 subsets of the 100GB VLC2 document collection together with the Okapi BM25 query processing model. The authors use 50 small queries adopted from the TREC-7 topic descriptions.

All the tests were performed using the PLIERS system running on a shared-nothing system consisting of eight Fujitsu AP3000 workstations with 167Mhz Ultra1 processors

and 128MB of main memory (running Solaris 2.5.1) and a 200MBps torus network. The probability of the code has been tested using an Alpha farm consisting of eight Digital Alpha 600 266Mhz workstations with 128Mb RAM. The network interconnects used are 155Mbps ATM LAN and a 10MBps Ethernet LAN. The results from the first installation were verified on the second one.

3.4.1 Results

There are two hypotheses tested by the authors:

1. Users tend to submit short queries and therefore to force the load imbalance when the term partitioning is used.

2. If all the documents are evenly distributed, a good load balance is provided when the document partitioning is used.

The results show that for the whole topic search, the partitioning by the document id provides two to four times faster query execution together with a five times greater speedup when the number of leaf processes increases. But at the same time, the document partitioning seems to have a slightly lower load imbalance and a much higher efficiency, while the efficiency of the term partitioning decreases by a factor as the number of leaf processes increases up to eight.

The alternative allocation schemes for the partitioning by the term id proposed in the paper result in a small improvement, as it demonstrated in the appendix of [MMR00]. But the term partitioning outperforms these by a significant factor even when the load imbalance ratios for the term and the document partitioning are equal.

The latter results show that the document partitioning is a much better alternative, since the term partitioning has a bottleneck at the top node. However, since the later allocation schemes result that the term partitioning gains a slightly lower load balancing than the document partitioning, the authors conclude that the term partitioning can be useful.

3.4.2 Critics

The main weakness of the work and the experiments performed by the authors is a too small number of queries. In addition, the article says nothing whatsoever about the possible improvements associated with the use of higher concurrency or multiprogramming levels.

3.4.3 Relevance

The most interesting contribution in this paper is the study on how a different term allocation scheme can be used to reduce the load imbalance in the term partitioning. But the scientific approach itself and the query processing model described in the paper differ completely from the approach and the model chosen for this report.

3.5 Badue, Ribeiro-Neto, Baeza-Yates, Zivani, 2001

An updated study by Badue, Ribeiro-Neto, Baeza-Yates, Zivani, *'Distributed Query Processing Using Partitioned Inverted Files'* [BBRZ01], shows the advantages of the global indexing over the local indexing. This paper can be viewed as a proceeding of [RB98] presented earlier. The authors use the vector space model, TREC-3 document collection and a shared-nothing architecture. This time the authors present some real results instead of a simulation model. The query set being used consists of 50 real queries and about 2000 artificial queries. All the tests were executed on five 500MHz AMD-k6-2 workstations with 256MB RAM and 30GB IDE disks (running Linux 2.2.14), interconnected with a 100Mbps Ethernet through a 16 port switch.

3.5.1 Results

The authors expect that the global indexing has a higher concurrency and less disk seeks, a lower imbalance, larger inverted lists and a larger local answer set. Also more data is to be returned to the receptionist. Despite to this fact, the results given by the authors show that the global indexing is a better technique than the local indexing in terms of the throughput, especially when the number of nodes exceeds the average query length.

The most exciting idea introduced in this work is the use of a filtering technique which considers only the documents with a high within-document frequency as the candidate answers. The preparatory study presented in the article says that filtering reduces the memory load to about 2 percent and the amount of the terms required to about 10 percent.

The results show a higher speedup for the global indexing, but also a higher growth in the load. Using 4 processors, the processing time associated with the global indexing is slightly better than with the local indexing. The processing time with the global indexing is less than 80% of the processing time using the local indexing on more than two processors and 50 real TREC-3 queries. Even better results are achieved using 2000 artificial queries.

3.5.2 Critics

Unfortunately, the authors say nothing about the fact that a higher query rate or a higher multiprogramming level may advocate the local indexing. The number of the real queries used in this paper is just as low as in the papers presented earlier. However, 2000 artificial queries provides more confidence for the results.

3.5.3 Relevance

Both the theory and the results provided by the authors are very interesting, especially those considering the approximation technique and the resulting processing time and load imbalance. The later simulation model and its results presented in this report will use this information as a reference.

Another important issue is the implementation details for the system presented by the authors. From the article, the receptionist process consists of an insertion thread, a merging thread, and a number of scheduling threads (one for each server process). There are also a number of scheduling queues shared by these processes and a result buffer. All the communication between processes is socked-based. The insertion thread is responsible for inserting a query/sub-query into the scheduling queue for the required server. The scheduling thread is responsible for taking a query out of the queue and sending it to its server, receiving an answer and storing the local results into the result buffer. The merging thread is responsible for merging the local results as soon as they arrive.

Some of the implementation details described by the authors are slightly difficult to understand, but the main idea is that it is possible to run a greater number of queries or sub-queries in the system without a requirement to run a greater number of threads. Thus the thread start-up and switching costs can be ignored.

Finally, the conclusions about the advantage of the global indexing when the average number of query terms is less than the number of processing nodes. It can be very interesting to see what happens for very long queries. Longer queries will probably involve all of the nodes when the local indexing is used. On the other hand, the number of disk seeks would increase when the global indexing is used. But if a conjunctive query model is used (a Boolean AND-only model), the volume of the data to be transferred to the receptionist node would decrease for both the local and the global indexing. Thus the disk access and the sub-query processing would be most critical issues. However, if the disjunctive queries are used, the data volume to be transferred and post-processed would increase dramatically, especially for the global indexing, making the network transfer and query post-processing to be the most critical issues.

3.6 Xi, Sornil, Luo, Fox, 2002

In their paper, *'Hybrid Partition Inverted Files: Experimental Validation'* [XSLF02], Xi, Sornil and Fox look at a hybrid partitioning approach for an inverted index distributed across the nodes of a parallel shared-nothing system. The document collection used is 100GB of TREC-9 and 10 data and the query set used consists of 200 real queries. All the tests were performed on a VT-PetaPlex-1 system consisting of 100 nodes with a 200Mhz CPU and 25GB disk at each node, and an IR server connected through a 100BaseT network.

3.6.1 Results

From the paper, the hybrid partitioning provides a much higher utilization than the distribution by the term id, at the same time there is some performance gain with a higher multiprogramming level, even more than with the distribution by the term id.

The results show that the hybrid approach achieves a better load balancing than the document partitioning does. Two very interesting, perhaps strange, features are: (1) the

load balancing gets better with a smaller chunk size, and (2) a higher multiprogramming level results in a higher throughput level. The experiments show also that the term partitioning is better than the document partitioning at a high multiprogramming level, but the hybrid partitioning with chunk size 1024 is better than the term partitioning.

3.6.2 Critics

The paper fails to explain why and how the hybrid approach is better than the others, and some other sources such as [MWZB07] say that the hybrid approach results in a greater number of disk accesses and has no any clear advantages.

3.6.3 Relevance

The hybrid partition has already been presented in the previous chapter. For the later simulation experiments it could be interesting to retest this scheme with different chunk sizes and to see how its performance compares to the other schemes. However as it will be presented later, the hybrid indexing does not allow to use conjunctive queries in a easy and flexible way. The reason for this is that the chunks containing same the document range will be probably placed on different nodes (see Figure2.4).

3.7 Badue, Ribeiro-Neto, Barbosa, Golgher, Zivani, 2005

In the paper named *'Basic Issues on the Processing of Web Queries'* [BBG+05], Badue, Barbose, Golgher, Ribeiro-Neto and Zivani explain a number of important advantages associated with the local indexing used on a shared-nothing system. Some of the advantages are the dominance of the disk utilization over the CPU utilization, the avoidance of a bottleneck at the broker and the elimination of the load imbalance provided by a random document distribution.

To provide enough evidence, a number of experiments with the ToroBR search engine on an eight node cluster were performed. The document collection during these experiments was at 80000 documents distributed over the nodes, resulting in 16GB of index data at each node. The query set used consists of 20000 real queries from a query log, where the first 10000 were used to *warm-up* the system, and the last 10000 were used to measure the results. The authors have also simulated a cluster with a high number of servers and seen what happens with the broker.

During the experiments it was observed that using 100 queries per second and 256 servers, the broker could spent up to 10ms on processing of a single query. Since the broker executes only simple operations, it cannot be a problem at this time. The report shows also that the query execution time for a query reduces with a higher number of queries, but it fails to explain whether it is an effect of a high concurrency or multiprogramming, or if it is a cache bi-effect. All the advantages named in this paper are very important for the evaluation of the local indexing and they are useful for the later discussion, but the authors fail to explain the reasons and results.

3.8 Moffat, Webber, Zobel, Baeza-Yates, 2005

A very impressive paper called *'A pipelined architecture for distributed text query evaluation'* [MWZB07] was published by Moffat, Weber, Zobel and Bayeza-Yates. In this paper the authors present the pipelined approach as an alternative method for the distribution by the term id. This approach retains the disk access benefits from the term partitioning, but suffers from load balancing problems. The main intention with this technique is to reduce the number of disk accesses at the same time as to reduce the bottleneck at the receptionist node. The experiments show that this approach is more scalable than the original term partitioning approach.

All the tests were performed using 426GB of the TREC GOV2 data collection and pseudo-realistic queries, synthetic query stream based on a real query log from the Excite97 with removed stop-words and all the words replaced to match the terms in the collection (all the frequencies and co-occurrences remain to be the same, while none of queries makes sense). The average query length used was about 2.15 terms per query. The authors experimented also with the size of the accumulator structure and claim that 100000 accumulators are a good choice, while 400000 accumulators are needed for an absolute retrieval (i.e. a retrieval with a highest possible precision).

A good solution for the performance analysis was to look at the normalized throughput rate $(q \times T)/(k \times s)$, that is the product of the number of queries and the size in Terabytes divided by the product of the number of machines used and the elapsed time. An increase in both the data volume and the number of nodes at the same time is expected to result in a constant normalized throughput rate. However a possible drawback here is that adding new machines will increase not only the amount of processor power, but also the size of the main memory available, and therefore result in a performance improvement.

The test system used consists of eight nodes with 2.8Ghz Pentium IV, 1GB main memory and 250GB Sata disks, interconnected by a 1Gbit Ethernet network. Finally a dual 2.8Gh Xeon with 2GB main memory, a 73GB SCSI disk and twelve 148GB SCSI disks (RAID-5). A modified version of Zettair system (`http://www.seg.rmit.edu.au/zettair/`) has been developed to evaluate the concepts.

3.8.1 Results

The results show that, because of the synchronization issues, the multi-threading results in 75% active load with the document partitioning. The authors experiment with the number of threads in the different organizations and come with the following results. For the document partitioning and the given document collection, the highest normalized throughput is using 32 threads. The same result yields also the term partitioning, but the difference in normalized throughput at four threads and 32 threads is only 4.5%. The most interesting result here is that the pipelined indexing shows the highest normalized throughput using 64 threads. Since this is only 1.5 % improvement from 32 threads, authors conclude that 32 is the optimal number. The connection between the number of threads and the query throughput is more clean using the pipelined approach, since the

number of threads is an number of queries in the system at the moment.

Here is a short summary over the results for each distributed index organization type discussed. The results stated below uses an approximation method with an accumulator target size at 100000, 32 threads, and the number of nodes varied between 1 and 8 unless otherwise is stated. The collection size is varied between 1 and 1/64 of the whole collection.

- **Monolithic** It was expected that the normalized throughput should halve as the size of the collection doubles. But the results show that the normalized throughput using 32 threads increments as the collection size doubles. On the other hand, the I/O wait load increments at the same time.

- **Document partitioning** The document partitioning method has been tested using 10000 queries and a varied number of machines. The number of documents gathered from each node was limited to $r' = 1000$ while the total number of documents in the result set was limited to $r = 1000$. The statistics needed for the document ranking were created locally and then gathered at the receptionist node.

  An interesting result observed with increasing of the data set and the number of nodes by a factor of two is that the degradation is very small, only about 4.8% at a change from one to eight nodes in average. However a twice as high number of nodes and the same amount of data result in a throughput degradation, in average 29.7% at a transition from one to eight nodes. It may be explained by the network overhead and the fact that the query response time is determined by the slowest machine.

- **Term partitioning** As it was mentioned earlier, the number of documents required in the result set is 1000, but using the term partitioning every node returns the accumulators for the whole inverted index for each term. As a result, the total normalized throughput degradation is observed to be about 60% for a constant data set, and about 50% when the data set was doubled at the same time as the number of nodes was doubled. The main reason for this is a bottleneck at the receptionist and a high network load. The network load on TB/01 has been recorded to be about 0.43Gbps using the term partitioning, while only some tens or hundreds of megabytes using the document partitioning.

- **Pipelined approach** The observed normalized throughput degradation is under 20% when both the data set and the number of nodes are doubled, and about 40% with the size of the dataset is remained constant However its value is only about 75% of the normalized throughput achieved by the document partitioning. But the network load during the test was only 0.1Gbps this time.

  The pipelined indexing results in a three times smaller amount of sector reads, distinct reads and I/O waits. But this does not help as much as expected. Communication between the nodes seems to be the most problematic part of the processing.

> The pipelined indexing fetches only 40% as much data as the document indexing and uses only 15% of the disk-read operations, spending only 25% on blocking for the I/O. But the load is unbalanced, some of the nodes have been busy for only 40% of the time, resulting in 60% of the average load, while the average load for the document partitioning is greater than 95%. The single node load and the commutative load are very varied with the pipelined system, between 38% and 73% of the communication load using the document distributed system. It looks like the load balancing and the communication itself are two weakest points of the pipelined approach.

The authors observed also that a greater number of accumulators results in a lower throughput rate. At a transition from 40000 to 400000 accumulators, the throughput falls by about 35% using the document partitioning, but it falls by 50% using either the pipelined approach or the term partitioning.

Finally, the overall results show that the pipelined indexing is a more efficient and scalable method than the term partitioning. On the other hand, it suffers from a poor workload distribution and therefore does not scale as well as the document partitioning. Authors claim that in a less skewed environment this approach is the method of choice, but the document partitioning is the best approach so far.

3.8.2 Critics

The queries used are not fully synthetic since they are based on the real queries. However changing query terms even if the co-occurrence frequency remains unchanged results in unrealistic queries that will result in completely different answer sets.

Another problem that can be mentioned here is the relationship between the network capacity and the memory size per node. The authors say that the network was close to be saturated using the global indexing. However, other articles presented so far looked at the network capacity versus disk access. So the memory and the disk performance characteristics have always been of an important consideration in the discussion. For the experiments presented each node has 1GB of memory. Say that each node stores 400000 accumulators for each of its 32 sub-queries and 8 query processes and some more memory to process other data. At least 3/4 of the memory for each single node is still empty, it means that at least 700MB can be used to cache most frequent indexes. A careful look at Table 3.2 reveals the transmission from TB/16, where the whole index can be placed in the memory of a single node (given $k \geq 2$), to a next case where disk access may be needed. After this point, the I/O load is a nearly constant fraction of the processing time for a query. A possible conclusion from this is the number of the disk accesses could be decreased by using an index or qurey result cache. Than the only issues considered in this paper are the network load and the load at the receptionist. The load balancing appointed by the authors to be the main problem of the global indexing may therefore be a side effect of the index/result cache use.

collection	TB/64	TB/32	TB16	TB/08	TB/04	TB/02	TB/01
Index size (GB)	0.3	0.7	1.2	2.4	4.5	8.7	16.6
Disk read (GB)	0.0	0.0	0.10	0.38	1.23	3.33	9.37
I/O wait load (%)	0.0	0.0	0.0	4.7	6.1	5.5	6.0
Throughput	3.18	4.25	5.15	5.41	5.75	6.67	6.83

Table 3.2: The index size and the relative throughput of the system in [MWZB07]

3.8.3 Scientific Remarks

Some other remarks have been published in the *'In Search of Reliable Retrieval Experiments'* [WM05] by Webber and Moffat, which describes the problems and the process behind the paper presented above. This paper gives a very interesting and informative description of a 2 year long scientific process. But it also mentions some important factors to influence the performance of an inverted index organization, most important factors are:

1. More imbalance will be induced for the term partitioning with a higher number of nodes using any approach.

2. More imbalance will be induced for the term partitioning when the query set is appropriate for the document collection.

3. k machines have k times as much memory as one machine, k times as much data can be located in the main memory.

4. Since the radial velocity of a hard disk is constant, the placement of the data on the disk is a very critical issue.

5. Some disks from the same series are slower than the others.

3.8.4 Relevance

A closer look at the paper reveals an important detail missing, namely that term to node mapping plays a very important role for the load balancing using the term partitioning. Authors of this paper say that a mapping based on the term frequency would improve the load balancing and therefore provide much better results.

Provided all the critics for the memory size and the disk access, it would be interesting to see how much will the final results alter if there is no disk access needed or if the memory cannot cache any index or result data. It could be also interesting to simulate the cache effects, but this is a very difficult task itself.

3.9 Moffat, Webber, Zobel, 2006

Another paper, *'Load Balancing for Term-Distributed Parallel Retrieval'* [MWZ06], by Moffat, Webber and Zobel tries to improve the results obtained in the previous work by using the pipelined approach together with a load based assertion method and replication of the most work-consuming terms. The system and the test configuration were almost the same as in the previous paper - a modified version of the Zettair search engine running on an eight node Beowulf cluster with 2.8GHz Pentium IV with 1GB RAM and 250GB local SATA disks, and a dual 2.8GHz Intel Xeon with 2GB RAM, a 73GB SCSI disk and twelve 146GB SCSI disk organized in a RAID-5 array. 426GB TREC GOV2 has been used as the document collection. An important difference from the previous paper was the query set, this time it is a modified version of GOVQ queries provided by the Microsoft Search. The total number of queries was 60000, 10000 in each of 6 batches.

To remind, in the original approach the route is chosen after an increasing lowest term frequency, F_t, for each relevant node. The performance using this method was not as good as it was expected because of a high work imbalance, where a small number of terms resulted in the enormously high loads. As the authors point out, it is not the most common terms that cause this problem, but the terms with the lowest product of both the term frequency in the query set and the frequency in the document collection. To eliminate this problem, the authors have introduced the definition of workload, L_t, presented earlier and made an alternative term to node assertion scheme where all the terms are processed after their decreasing workload and asserted to a node with the least total workload. Since the workload cannot be predicted exactly, the term frequency from the previous batch is used to predict the workload for the next batch. The query route itself is chosen by the increasing lowest term frequency for a sub-query mapped to a node.

Another trick proposed was to duplicate the inverted lists for the terms with the highest workload, it was observed that the replication of each inverted list is not better than the duplication of 100 most workload consuming terms.

3.9.1 Results

The modified version of the pipelined approach results in a 30% improvement in the query throughput, but it is not as good as the document partitioning. The scalability test shows that the normalized throughput using the modified pipelined approach and 32 active query threads degrades by 11% when both the number of nodes and the total collection size double eight times, and 11% when only the number of nodes is altered. At the same time, the normalized throughput for the document-distributed approach degrades by 45% in the first case, but only by 2% in the second.

3.9.2 Critics

The query set has similar problems as the one in the previous paper. Also just as in previous work, the authors use the normalized query throughput to measure the efficiency of the partitioning scheme. However, it does not tell anything about the query response

times which are more critical for the end-user. Finally, there are no mention of any possible improvements in the performance that could be achieved by using a faster disk, processing unit or network.

3.9.3 Relevance

The alternative approach presented is very interesting. Authors mention that this time the performance degradation is caused by the difference in the term workload in two different batches. Therefore, it could be useful to consider about an approach where the information from the previous batch together with some partial workload information for this batch could be used to re-map the inverted lists. But it would require access to real-time workload information and real-time data exchange between the nodes in the cluster. In addition, event this approach may provide a low performance when the term workload varies within the same batch.

In any matter, the ideas proposed in this paper and the associated ideas can be considered for a further investigation, but it would be difficult to compare the measured results provided in this paper against the results produced by a simulation model.

Due to the time limits for this master thesis, evaluation of alternative assertion methods would be a way to much. Thus, the impact of different assertion methods for the global indexing and its variations would be proposed as an alternative topic for a later master thesis assignment or a research project, along with the impact of the approximation methods.

3.10 Jonassen, 2007

The preparatory study for this master thesis was presented in a project report named "Global vs Local Indexing" [Jon07] by Simon Jonassen. A first half of the project report contains much of the information presented so far, but limiting it to only the standard local/global indexing. The second part of the project report states a number of hypotheses based on the results from the papers given as the previous study, describes a simple simulation model for a disjunctive query processing system used then to test these hypotheses and finally presents and discusses the simulation results.
There are seven **Truth/Myth** statements proposed:

1. *The partitioning by the term id is advantageous only when the skew in the query distribution is low.* [JO95]

2. *The partitioning by the document id does it significantly better with a higher number of disks and a higher multiprogramming level in a high skew environment.* [JO95]

3. *The global indexing organization is quite advantageous in presence of a fast communication channel.* [RB98]

4. *The global indexing outperforms the local indexing using the vector model and a number of approximation techniques in the terms of the processing time and scalability.* [BBRZ01]

5. *The load balancing is a critical issue for the global indexing using the vector model.* [BBRZ01]

6. *For the global indexing, the receptionist node's load is manageable at a defined query rate.* [BBG+05]

7. *The global indexing is significantly less advantageous than the local indexing due to a high imbalance together with a high network load and the workload at the receptionist.* [MWZB07] [MWZ06]

The simulation model implemented with the Desmo-J attempted to mimic a shared-nothing system with various network, disk and load characteristics. Both the document collection and the document were generated using the numbers and equations gathered from [ZS05] (query length) and [BBR+07] (term frequency), and a uniform and an exponential distribution for the query terms. Many of the ideas behind the simulation model are inspired by [TG93].

The model itself considers only the processing of a simple disjunctive query resulting in a number of document id's. Only two standard algorithms and no sniplet generation, no approximation techniques and no more sophisticated mechanisms were used.

3.10.1 Results

The report describes more than 24 test runs performed, divided between the baseline experiments with a standard configuration, high skew experiments using an exponential query term distribution, and experiments with a double number of nodes, disks or CPUs, experiments with ten times slower network or disk, or no disk access at all.

From the baseline experiments, the local indexing results in a much higher number of query and sub-query processes and up to nine times longer query processing times for a single query and almost twice as much time to process all the 50 queries used in tests.

Suggested explanation for this is that the disk is the bottleneck for the local indexing, a higher number of processes in the system would make it only worse. On the other hand, the global indexing shows worse disk access times for its sub-queries. The CPU load is much higher and denser for the global indexing, but the processing takes longer time in total. The local indexing uses much more memory and the disk usage is constantly at 100% which supports the expectations about the disk access bottleneck.

A careful analysis of the graphical reports shows that for the global indexing, the queries arrived first tend to be solved first, but it is less true for the local indexing.

More skew in the query term frequency results in a higher CPU and Disk utilization for both schemes. For the local indexing the CPU times are almost not observable in either queries or sub-queries, while the service wait times are up to four times longer

than for the global indexing. The global indexing does it worse than in the original test, but much better then the local indexing in the either of these two tests.

A double amount of nodes results in a lower CPU and memory usage, but the disk usage seems to be unaffected. There is more imbalance between the nodes when the global indexing is used. The worst case query and sub-query times drop by a half when the global indexing is used, but they remain unchanged when the local indexing is used. A double amount of CPUs results only in a lower CPU load, while the processing times remain unchanged. A double amount of disks has very interesting results. As the disk load reduces from being constantly 100% to variate between 0 and 100% for the local indexing, the service wait time reduces too, since more sub-queries can be served at the same time. As a result, the processing time for the local indexing is reduced by a factor of four when the number of disks per node is increased by a factor of two, but the total processing time for the global indexing is reduced only by a factor of two at the same time. The total processing times used by the local and the global indexing are nearly the same in this test.

A ten times slower disk results in a worse performance for the local indexing. Under the test for the global indexing the total time used on a single query is greater than 1000ms. On the other hand, a ten times slower network bandwidth shows dramatic results for the average query time when the global indexing is used. But even in this test the worst case query times are nearly the same as for the original experiments.

So-called Batch2 experiments show that 4 disks on each node and a slower network results in sub-query times different only by 30%, but the final query times are exactly the same. No disk access makes the local indexing to be a better alternative.

After a number of experiments it was observed that the relationship between the local and the global indexing depends on the parameters chosen for the system configuration. Corrected CPU metrics would result in a much worse performance for the global indexing, or even make the local indexing to be a better alternative. A faster disk access would result in the local indexing performing significantly better, especially for a higher number of nodes or processes. Finally, a higher skew in the query set would make the difference between the local and the global indexing even greater.
Despite to these observations, the truth and myth statements can be answered as follows:

1. False from the results presented, but it is possible to disprove the results.

2. From the results, a higher multiprogramming level and a high skew will result in a bottleneck. But a higher number of disks will reduce the effect or even change the relationship between the local and the global indexing if the network and/or CPU access are slow.

3. The results show that the global indexing is faster when a high speed communication channel is used. But it could be false if the other CPU costs were different.

4. No approximation techniques were tested during the project, but in the current situation the global indexing is more advantageous. However, any approximation techniques are easier to apply on the local indexing. On the other hand, it is

possible that with different system parameters the relationship between these two schemes could be different. This statement needs a further investigation.

5. No balancing problems were observed, but the imbalance will be higher for a higher number of nodes or a higher query rate. In this case at some point load balancing can be a critical issue.

6. In the current situation the imbalance is low.

7. False for the presented results, but the situation can be changed by adjusting the system variables.

The final conclusion in the report is that there are many different trade-offs which can either advocate or penalize the use of the local or the global indexing. Some of these trade-offs are the network bandwidth, disk bandwidth and disk delay, instruction costs, document collection and query set relevant characteristics. The testing approach, the assumptions made and the implementation itself play an important role as well.

3.10.2 Critics

First of all, the processing metrics used in the simulation model, such as the number of cycles to perform a single instruction, gathered by guessing were completely underestimated. A memory access instruction would probably take a hundred of clock cycles, not just 8 as it was proposed. This makes the statements about the processing times to be very uncertain. Benchmark based metrics would provide more realistic and valid results. Also in the current implementation of the simulation model, the average disk seek time and the rotation delay time are used as constant parameters, a normal distribution for these would result in a more realistic scenario, but the standard deviation of these variables needs to be estimated in this case. Next, the network transfers blocks the CPU. A real network adapter would rather use the DMA mode which allows the CPU to perform other tasks at the same time as a transfer occurs. Perhaps, whole the network implementation looks somewhat strange and unreliable. Finally, simulated CPU uses clock frequency to estimate the instruction costs. The problem with this is that a CPU with a different clock frequency would in addition use a different number of clock cycles to perform the same instruction. A better solution is to use an average instruction time rather than the number of clock cycles and the clock frequency. It solves also the problems associated with simulation of a pipelined or superscalar CPU.

The query set and the document collection itself are simulated using a set of very unrealistic equations which cannot be used to estimate the real-world processing times, even if the algorithms and the processing metrics would be corrected. A log from a real search engine or a real document collection and a real query set would provide more realistic results. As it was mentioned before, a synthetic query set allows more control over the query skew, but the correlation between the query and the document terms is completely wrong in this case. The query arrival distribution itself is a wrong approach to measure the maximum performance since the time between the arrivals is randomized

and the system implementation needs a lot of processing to dispatch and queue arrived queries (not to mention a bug in the dispatching process causing a high CPU usage). A better alternative could be to maintain either the number of queries and sub-queries or the total number of processes in the system constant and generate new queries when this number is too low. Then the number of queries processed during a time period could be used as a performance estimate.

The report refers to a number of errors contained in the code such as those with the dispatching processes and the memory allocation. From the report, there was an incident where the load distribution has suddenly changed and the results from the previous experiments could not be reproduced any further.

Another important issue is the term mapping. When the global indexing is used the simulation model maps a term t_i to the node $i \mod n$. In the particular implementation the term id is the rank of the term. Now, since the elements with nearly equal ranks are spread allover the nodes, the load balancing is expected to be very good. Instead, in a more realistic situation, a lexicographical mapping would result in a situation where many high frequent terms are assigned to a single node resulting in a higher imbalance. These two allocation schemes are discussed in [RB98] and [MWZ06].

The mapping used with the local indexing has also a kind of miscalculation. In the particular implementation a term contained in x documents would have $\lfloor x/n \rfloor$ documents on each of $n-1$ nodes and $x - n\lfloor x/n \rfloor$ documents on the last node. This is actually a hybrid partitioning with a dynamic chunk size, rather than a local partition. In a more realistic case it could end up with all of the term occurrences contained on a single node, but none on the other nodes. On the other hand, if there are x occurrences of a single term. The probability that a single document contains a particular term is x/D. Since a node contains about D/n documents, the expected number of the term occurrences on this node is x/n.

Approximation methods and caching effects were never considered in the simulation model presented, but it could be interesting to see if any of the techniques named earlier could be used to get a significant performance improvement.

3.10.3 Relevance

Despite to the critics presented, the work performed had an enormous impact on gathering new ideas for an improved simulation model. The simulation results were also helpful for a revision of the previous study and a better understanding of details and results in the papers presented. This report will now proceed with a further development of the simulation model aimed to evaluate the partitioning schemes for an inverted index.

Chapter 4

State of the Art

This chapter presents the essence of this master thesis. It begins with a free translation of the assignment text in Section 4.1, followed by a description of the scientific method chosen to perform this task. Section 4.3 will present the old simulation model produced during the preparatory work. Then Section 4.4 will present the transition to a new and better simulation model. The model itself, ideas behind and all the micro-benchmarking experiments performed to measure the required system parameters will be presented in the next chapter.

4.1 The Assignment Text and The Solution Approach

The assignment text for both the preparatory work and the master thesis itself, freely translated from Norwegian to English, sounds as follows:

For a search platform there are many different ways to organize a distributed inverted index. Perform a literature study to compare the inverted index partitioning by the term id and the document id. Another names for these two concepts are local and global inverted index files.

The work performed can be extended with a number of original experiments aimed to re-validate previous research, or to test different query and document collections

As the first part of the assignment was already presented, the second part aimed to verify or re-validate the previous research will now be presented. The next few sections will present the chosen approach to solve this task and a number of important decisions and choices.

4.2 The Approach

The approach chosen for the practical part of the assignment is a simulation model. Two alternative approaches discussed in the [Jon07] were an analytical model and a real implementation. The problems with an analytical model are a difficulty to describe a complex real-time system and to get any results that would be valid for the real

world. A real implementation, on other hand, requires a very high level of detail and provides almost no parameter flexibility or ability to easily redo all the experiments if it is required. As a compromise, a simulation model requires a low enough level of detail, while it provides a dynamic behavior similar to a real system. And the most important advantages that all the system parameters, all the small details, can be easily altered and all the test results can always be reproduced or re-validated.

4.3 The Previous Simulation Model

The simulation model programmed during the preparatory work [Jon07] for this master thesis was implemented using the Desmo-J, a Discrete Event Simulator for Java. The source code of the simulation model consists of 13 Java classes: `Config`, `QueryGenerator`, `ExperimentLog`, `SystemMonitor`, `Query`, `SubQuery`, `Node`, `QuerDispatchingProcess`, `SubQueryDispatchingProcess`, `QueryProcess`, `SubQueryProcess`, `ResultTransfer` and finally `Simulation`. Figure 4.1 demonstrates the class diagram.

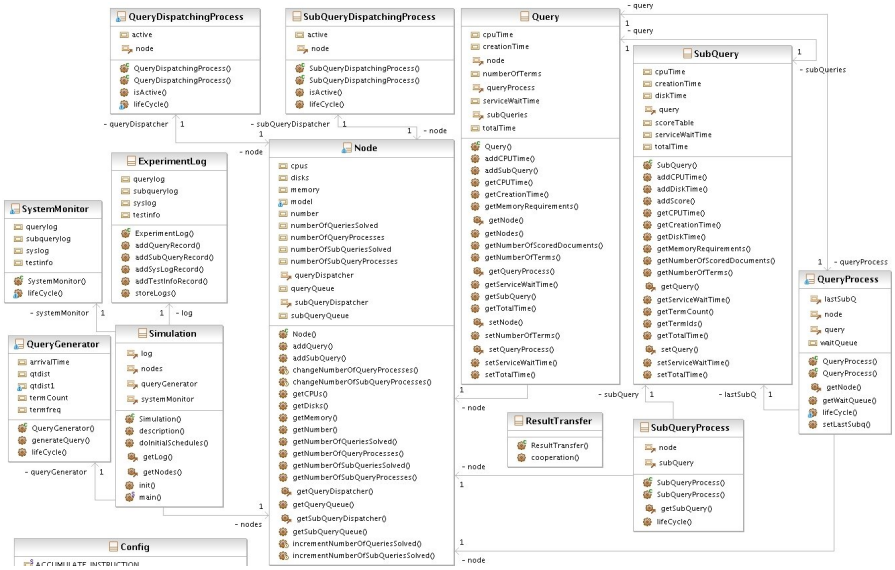

Figure 4.1: A class diagram for the simulation model used in [Jon07]

The `QueryGenerator` is a `SimProcess` which is responsible for generating queries and submitting them to a queue contained by a `Node`. The `QueryGenerator` uses an exponential distribution with a median at 20ms to determine the query arrival times, an empirical distribution based on the data from [ZS05] to determine the query length and either a uniform distribution or an exponential distribution with a median at 1000 to determine the rank of the query terms. The term document frequency is based on the normalized frequency from [BBR+07] (which is actually wrong since the cumulative

frequency of the terms contained in the query dictionary would be greater than 1.0). Queries are submitted to the nodes in a round-robin manner.

One important detail here also mentioned earlier is that the term id used is the same as the term rank. It results in that the distribution of the terms to a number of documents is very balanced across the nodes. A distribution based only on the term id, with no considerations to the term rank, would result in a less optimal case for the global indexing.

A `SubQueryProcess` simulates the disk accesses and sub-query processes by blocking the Node's disk and CPU resources. It tries also to simulate the memory allocation, but there are some miscalculations which lead to the fact that the total allocated amount of data is not large enough to perform all the required processing.

Each `Node` in the simulation model have an associated `QueryDispatchingProcess` which removes `Query` objects from its query queue and, if it has enough resources and if the number of the processes running on this node is small enough, starts a new `QueryProcess` for each incoming `Query`. A `QueryProcess` would then block a node's CPU for the time required to transfer all the required sub-queries and add the corresponding `SubQuery` objects to the queues of the destination nodes.

A `SubQueryDispatchingProcess` running on each node does the same task as the `QueryDispatchingProcess`, namely it checks the first `SubQuery` object in the queue and, if it is possible, starts a new `SubQueryProcess` for this one.

To limit the processing load, both `QueryDispatchingProcess` and the `SubQueryDispatchingProcess` passivate when there are no objects in the queue and they are automatically activated when a new object arrives. However, there is a bug in the implementation which causes the dispatching process to overload the CPU when the number of running processes is reached.

The equations for the processing estimates used in the project report will be not explained here. However, there are a number of important details to mention.

First, the number of scored documents for a single term was suggested to be the same as the global term frequency, f_t, multiplied by the number of the documents in the collection. That is $D \cdot f_t$. A more correct estimate based on the binomial distribution is 1 minus the probability that a given term does not occur in a document, also $D \cdot (1 - (1 - f_t)^W)$.

Second, the number of scored documents for a number of terms was estimated as the total number of the documents in the collection multiplied by the minimum between 1.0 and the summary term frequency for these terms. The resulting value is an overestimate. Two possible solutions to this problem will be presented and explained later.

Third, the processing costs are estimated as the number of bytes in the data volume to be processed multiplied by the number of the CPU cycles for a single processing instructions. First, the number of CPU cycles cannot be applied directly since it misses the benefits of a super-scalar/pipelined CPU architecture. Second, the values used in

the model were guessed without any idea about what they should be. No considerations about the memory access or on-chip cache benefits for large data volumes were applied.

When a sub-query is processed, its partial answer (an imaginary accumulator list) is moved back to the query node by blocking the CPU on both nodes for the time required for the transfer. This operation is realized by `ResultTransfer` which implements the `ProcessCoop` interface. The process cooperation allows to simulate a synchronized transaction. However, from the results obtained from the simulation model it is uncertain how effective it is. In other words, the sub-query result transfer implementation looks to be ugly and buggy.

In addition, blocking of the CPU to simulate a network transfer was inspired by a number of outdated simulation models. Modern computers however use a special processing unit contained at each Ethernet board which allows the data transfer to be performed in the DMA[1] mode. So a better idea would be to use a dedicated resource which would be responsible for the network transfer.

When the data is transferred back to the query node, all the required post-processing is applied and the query process is finished. All the relevant statistics and metrics are then stored using the methods provided by `ExperimentLog`. Some of a node's status data is measured during the simulation run by the `SystemMonitor` process and stored by a number of methods also provided by the `ExperimentLog`.

Finally, the `Simulation` class encapsulate the simulation model setup and control itself and the `Config` class is used to define the parameters and metrics used by the simulation model.

4.3.1 Visualisation of the Simulation Results

In addition to the Java code, a number of Gnuplot scripts were used to transform the textual simulation data into a number of visual diagrams. However, the results were not so effective as it was expected. The node utilization charts have too much detail and too many plots, so it is practically impossible to determine a node's state at a particular point. The query time and sub-query data provides a lot of insight into the distribution of processing and disk access times and the accumulated system wait time. But it does not explain when those different times take place or why a given query or sub-query has a given time value. To be more efficient, the graphical reports for the simulation runs should provide enough information to explain wait times. It would also simplify the debugging of the model.

4.4 The Roadmap to a New Simulation Model

The progression during this master thesis was not as straightforward as it can seems, and a lot of time was spent on thinking which way is the best way to go, analyzing which

[1] Direct Memory Access

alternative is best to choose, and sometimes going back and choosing another alternative.

As it started, the first task was to rewrite the previous study and the background chapters. The results obtained from the previous simulation model were very useful to get a better understanding of the issues and the results mentioned in the previously published papers. A number of errors and misunderstandings were also corrected.

As the previous simulation model had too many problems, a reasonable decision was to write a new simulation model using the experience from the old one in a back hand. However, as it will be mentioned, there were at least two different architectures to choose, to begin with. The easiest and most flexible one was chosen.

At the next stage, the simulation model framework was programmed and a number of different algorithms were designed. To begin with, the old simulation model had to support only OR-queries and had to use a sum of term frequencies as the resulting frequency. Keeping in mind that this would produce incorrect results, the simulation model was altered to use real arrays filled up with randomly generated data as inverted lists. The problem with this approach was a low scalability in both time and memory. So the simulation model was extended to support numerical only simulation. The disjunction frequency issue of inverted lists for distinct terms was solved by generating a look-up table for disjunction of two terms. This part was later replaced by a simple mathematical formula. Further it was observed that resource allocation can slowdown the Desmo-J performance. A numerical only simulation together with a small change in the source code had to improve the overall performance of the simulation model, so it would run fast enough even with a large number of nodes and a very high number of simulated documents.

At the same time with the work mentioned over an integrated visualization tool to plot the query processes and the node state data was programmed. The source code and ideas behind are based on the Bootchart, a Boot Process Performance Visualization tool for Linux by Ziga Mahkovec.

Another task performed at the same time was to find a search engine to use for the master thesis. But it was not quite clear whether a real search engine should be used along with the simulation model or used only to perform benchmarking for the system parameters, or just studied to see how a real implementation looks on the inside. A number of different open source search engines were studied and a small search engine developed by Truls A. Bjørklund called Brille was considered as the one to be used. The source code of the search engine was analyzed, but when the author of this master thesis started to index the TREC GOV2 document collection a number of problematic issues came up.

First, the main memory of the computer workstation used by the author was only 1GB while the system had to run a great number of user and system processes, just as any normal workstation does. To be effective it would require a dedicated workstation that would run only the search engine and a number of essential processes. Second, the document collection to be indexed was about 480GB, stored on a USB disk. The primary disk of the system was only 80GB, but the inverted index for the document

collection supposed to be about 60GB. However, since a merge based inversion algorithm was used, it would require about 120GB of disk storage to construct the index. Third, an alternative solution was to get a dumped dictionary for the TREC GOV2 collection indexed with Brille and use it instead of the search engine. Since the source code of the simulation model contained a lot of the processing implementation details, it would be an improvement anyway.

By doing this, a study of a real search engine resulted in a better understanding of the topic and a possible implementation. So it cannot be considered as a great amount of time wasted without any useful results. On the other hand, the system metrics were not estimated at this point. The master thesis by Truls A. Bjørklund [Bjø07] contains the performance model for the Brille. But the algorithm implemented in Brille reads a small part of an inverted list at a time, while the algorithms implemented in simulation model so far supposed to read a whole inverted list and to merge it with the data accumulated so far. An idea that came up here was to read all the inverted lists first and then to interleave/merge them together in a single heap-interleave/merge operation. So the algorithms implemented in the simulated model were extended by an alternative version for each one, resulting in a slightly different performance.

The system performance was finally measured by implementing a number of small code fragments doing partial processing tasks such as merging two inverted lists implemented as numerical arrays or extracting the top candidates from a result list. These results from these tests was used to describe the time characteristics of a CPU. The estimates for required network characteristics were gathered from a practical exercise performed by the author of this master thesis at the course 'TDT4200 Parallel Computations' at NTNU during the spring 2007. The disk performance characteristics were taken from a number of data sheets and a couple of memory metrics were taken from [Bjø07].

In addition to the dumped dictionary intended to simulate the document collection, a query log containing 50000 queries for the Terabyte track 05 was obtained to be used as the query set. However, the dumped dictionary containing an textual representation of the terms and associated number of documents was about 1.1GB which is a way too large to be contained in memory or to be processed effectively. But it was observed that the query set contains only 32000 words, which can be easily stored as a textual document to be read and kept in memory under a simulation experiment. The simulation model was now finally altered to support the query set and the dictionary data. Then a number of features to store a textual log for simulation tests and to read the configuration parameters aimed to automate the simulation process were added.

The rest of the project consisted of writing the report, performing simulation experiments, analyzing the results and making a final conclusion and suggestions for a further work. The next section would now explain some of the most important issues and decisions for the simulation model introduced above. The following chapter will present the implementation of the simulation model itself, ideas and decisions behind and all of the related micro-benchmark experiments and their results.

Chapter 5

Simulation Model

5.1 Ideas and Decisions behind a New Simulation Model

To summary up the for the two previous chapters, the most critical issues needed to be resolved by a new simulation model were:

1. A more realistic simulation of the network transfers, avoid to use CPU to perform a transfer.
2. An improved routine for dispatching and synchronization between a query process and its sub-query processes.
3. An improved memory usage model.
4. More realistic CPU estimates and model parameters.
5. A more realistic implementation of algorithms for global (GI), local (LI), hybrid (HD) and pipelined (PL) indexing.
6. A mapping for GI which is not based on the term rank.
7. A better estimation for the total number of scored documents.
8. An improved document collection and query set simulation.
9. Support for approximation and filtering methods to reduce the memory usage, disk, cpu and network loads.
10. An improved method to measure and analyze the performance of the simulated system.

This section will now proceed with an explanation about how the different issues were solved and how the final decisions were taken, and the next section will present the source code of the new simulation model.

5.1.1 Framework

As it was observed, the approach and its implementation used to execute, dispatch and synchronize query and sub-query processes implemented in the previous model had a bad architecture. However, there are at least two quite different methods to implement a better simulation model.

Method 1

A realistic implementation would use one single thread on each node to receive queries, one thread to perform query dispatching and sub-query scheduling, one to perform transfer of scheduled sub-queries, one to receive sub-queries, one to dispatch incoming sub-queries, one to perform disk access, one to combine results, one to transfer the results back, then one thread to receive partial answers, one to combine the answers and finally one more to transfer the complete answers back. The number of threads can be either reduced or increased by combining some of the described tasks or splitting them into a number of sub-tasks. But the main idea is to use a number of constantly running threads and a message based communication approach to co-operate between them.

This method is very realistic, and it can provide a lot of flexibility and efficiency by using priority queues to store tasks and messages. However, there are many difficulties and weaknesses here. First, the implementation is much more complicated than the one implemented in the previous simulation model. Second, it would be difficult to track a single query or sub-query, since now it would be just a message and a bunch of the associated data passed between the threads and nodes.

Method 2

An alternative method is to think of a query or a sub-query as a process itself. A *process* in this case means not a sequence of instructions performed by a single CPU, but a number of processing events performed for a single purpose. In this case a such process can just use the resources on the different nodes if they are available, or just wait if they are not. Now it would not provide any details on how the processing is implemented, whether there are a single thread to perform a similar task for all the queries (Method 1), or if there are a single thread to progress the whole query (previous simulation model).

For so far, this method is less realistic. But this one is more easy to implement, and it can easily provide information about the lifetime and the progression of a single query.

Decision and additional details

The final decision is to use the Method 2 in the new simulation model. In this case, each query process will consist of a query transfer to a node in the system, scheduling a number of sub-query processes, waiting for all of the processes to finish, partially performing post-processing of incoming sub-query answers, finally postprocessing the query results and performing a final transfer of the results.

An important advantage with this model, is that whole the processing can now be performed using a pull model instead of a push model. It means also that instead on maintaining a queue for incoming queries, the system can just fetch a new query and start a new query process when the number of executing queries falls below a given number.

This text will now proceed with some of the most important details for the chosen concept. Some of the details explained below may be quite difficult to understand, but the source code for the new simulation model, which will be introduced in the next section, provides a quite simple implementation of these.

(In the later discussion words like 'allocate', 'free' , 'process' , etc. refers to simulated events, and not the actually events performed by the program code.)

Synchronization There are two important issues here. The first one is how to synchronize query process and sub-query processes. The solution is to maintain a queue for sub-query results at each of the query processes. Then, when all of the sub-queries are scheduled, the query process would passivate itself. A sub-query process will put its results into the sub-query result queue maintained at its query process and re-activate the query process, then terminate. A newly re-activated query process can now serve all of the sub-query results stored in its queue, then either passivate again or, if all of the sub-queries have already finished, proceed to the post-processing of the results.

Network model Another issue is how to provide a realistic network model and resource control for the simulation model. A solution is to define a query or a sub-query process as a whole sequence of events from the point when a transfer of a query or a sub-query to a particular node is initiated to the point when the result is received by the receptionist node. Then the network can be simulated by keeping a network resource at each node, and when a transfer between two nodes occurs, a process needs to take the network resource from the destination and the source node, hold if for the time required to perform the transfer and finally return its resources.

The problem here is that, if an allocation for a number of resources is requested, some of resources can be taken, while some of them can be free. Doing it in a standard Desmo-J way by calling provide() would block if one of the resources was unavailable. So allocating a part and waiting for another can result in a dead lock.

A solution to this is to use a global resource handler structure with a number of associated routines where each pending process would check whether all of the resources it demands are available. If so, it would retrieve the resources right away. If not, the process will place itself in a queue and passivate. When some of the resources are returned through the resource handler routine, the returning process will reactivate all of the processes waiting for the resources which are now available in the system. Some of the reactivated processes will now get the required resources, while some of them would be placed back to the resource handler queue and passivated.

Figure 5.1 illustrates the resource handler idea and Figure 5.2 shows a complete lifeline of a query process and four sub-query processes on four different nodes with just

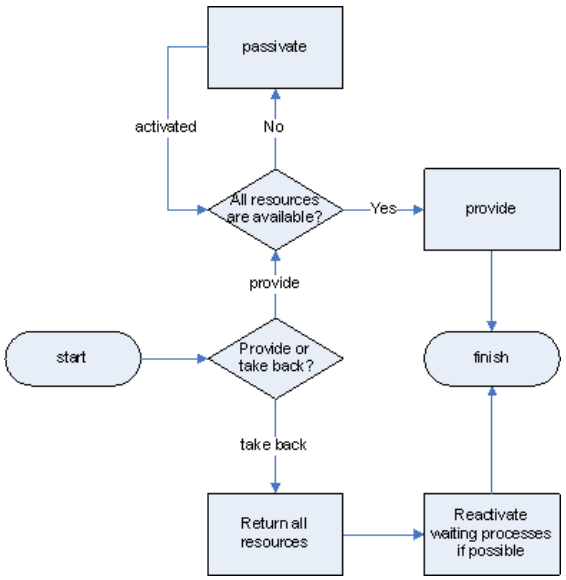

Figure 5.1: Resource Handler Routine

one disk access on each node. CPU access is ignored for simplicity.

One final note here is that the resource handler itself could also be implemented as a process rather than a structure. In this case it would maintain a queue over the pending requests. Then it would be activated once in a while to check if there are enough resources. If so, it would reactivate the process. Further the resource handler process would passivate when the queue becomes empty and be reactivate again on request. However, the simulation model which would be presented in the next chapter uses a resource handler implemented as a structure and not as a process. The process approach seems to be more elegant, but in this case it would be impossible to see which processes has been activated after which event and which resources has been used by which process.

CPU model The previous simulation model viewed a CPU as a resource with a capacity corresponding to the number of processing units and the allocation times based on working cycles. However, this approach is quite wrong since the model does not simulate super-scalar or pipelined CPU execution. Instead of using CPU cycles and estimates on based on a workload per byte, the new simulation model can estimate the CPU load based on a number of empirically estimated time costs per data element.

Another weakness with the previous model is that the processing of a whole inverted list is executed as a single processing phase on the CPU, where all other processes are required to block and wait. A real operating system will never work this way. A single thread or process is scheduled to the CPU for a small fraction of time, then moved to a queue and replaced by another thread or process. This allows a number of processes to perform concurrently.

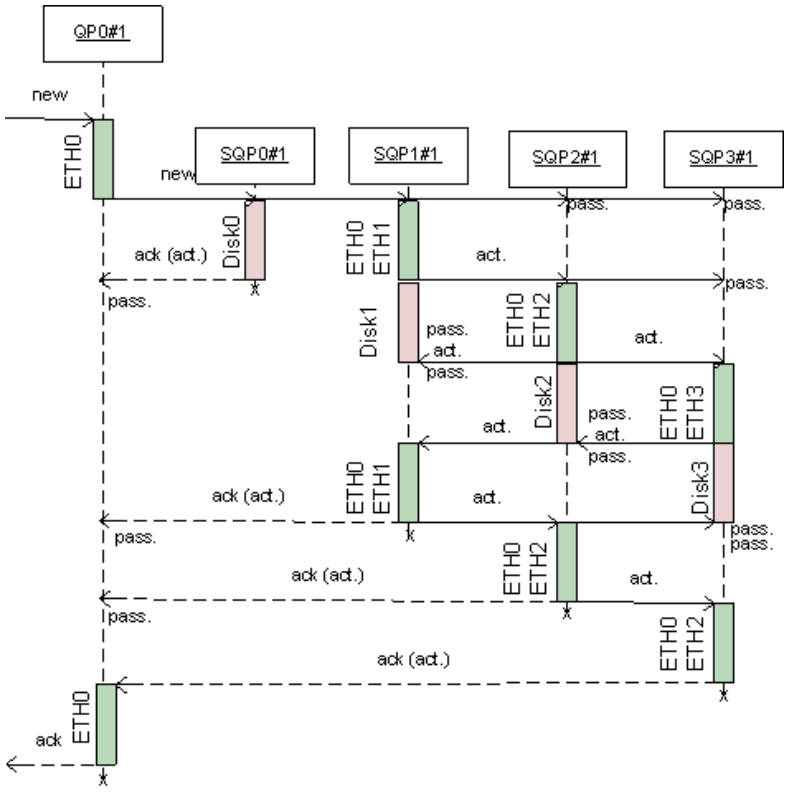

Figure 5.2: Process coopeartion using Method 2

A good thread multiplexer which supports interrupts, exceptions etc. may be quite difficult to implement (in fact, a popular Unix-like operating system has started as a thread multiplexer). But a poorer version which switches between processes on a round robin basis can be easily implemented by splitting the total time needed to perform a task into a number of small slices and then use a for loop to perform these. DesmoJ will now take care of queueing of the running processes etc.

Memory model Memory can be easily implemented just as in the previous model, using a dedicated resource with a capacity corresponding to the amount of the memory blocks in the system. However, as it was observed - DesmoJ will slow down when the capacity of a resource is too high. Therefore the capacity of the memory resources needs to be reduced. Using 1MB blocks instead of 10KB blocks reduces the number of resources by 100, but the problem is now that the blocks are a way too large and a single block can provide enough memory for a number of different requests. A reasonable solution is to keep track on the allocated memory (i.e. the provided amount of the memory resource) but unused memory and, if it is possible, to use it instead of allocating more memory. The problem now is that the number of memory resources provided and returned by a query or a sub-query process would be different and Desmo-J disallows a process to return more resources than it has allocated or to return a fractional number of resources.

To illustrate the problems, suppose it is only possible to allocate only a whole number of resources entities at once. Suppose now that a Process1 allocates 1.8 entities of the momory resource, then a Process2 allocates 0.1. Using the naive method, the total amount of memory allocated would be 3. By using allocated but unused memory, the total amount of the allocated memory can now be reduced to 2. But suppose now that Process1 finishes before the Process2. Process1 cannot return 2 since 0.1 is used by Process2 and it can return only a whole number, so it may return only 1 entity. When Process2 terminates it cannot return any data either since it had not allocated any resources to begin with, and in addition the floor value of 0.1 is 0. In this case when both of processes terminates one whole entity of the memory resource is lost.

A solution is to introduce a memory handler which is a process maintaining two request queues, one used to allocate resources and one to return them back. The reason for maintaining the second queue is that the resources allocated by a process must be returned by the same process, and in this case it is the memory handler that does both.

Now the number of the available (i.e. allocated but not used) resource entities can be maintained by the memory handler. A process requesting an amout of a resource will place a request in the request-queue, re-activate the memory handler and passivate itself. A process which returns an amount of a resource will place a request and re-activate the memory handler without passivating itself.

When the memory handler serves a request from the request-queue, it can either just increase the availiable amount of this resource, or allocate the amount missing to serve the request. If the total amount that can be allocated is too small, the process requesting these must be passivated. When a free-request is served, the returned amount is added to the available amount and as much as possible of the total amount is returned to the

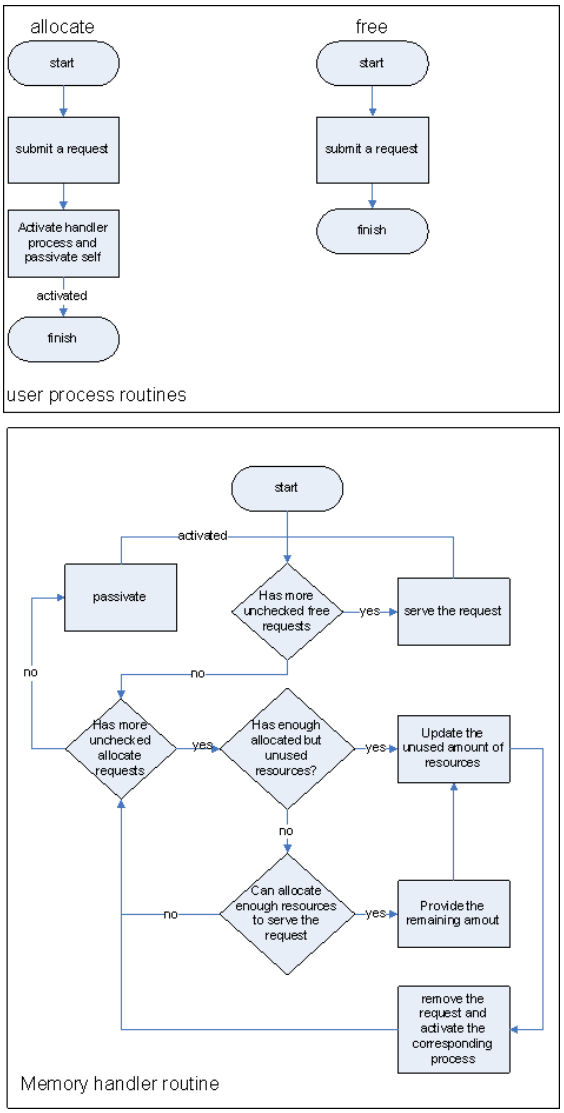

Figure 5.3: Memory Handler Routine

system. When the queue becomes empty or if it is impossible to serve any of the requests, the memory handler would passivate. Figure 5.3 illustrates the complete idea.

Disk model The disk access can be implemented just as in the previous simulation model. The number of disks per node can be specified by the capacity of the disk resource.

5.1.2 Study of a Real Search Engine

As it was mentioned earlier, it was expected to study an open source search engine and, if it could be possible, use it in some way to provide more realistic measurements and processing. Surprisingly there are many different search engines that can be used in terms of GPL, BSD or Apache licenses. Five of these - Brille, Lucene, MG4J, Zettair and MG - written in C and Java were considered as possible candidates. Table 5.1 compares different alternatives based on the number of lines of code, calculated using

```
find -name *.java | xargs cat | grep [\{\}\;] | wc -l
```

or similar.

After a short overview of the code structure and its content, Brille and Lucene were considered as two most interesting alternatives. Since Lucene does not provide a native support for the TREC GOV2 file format (this part however can be solved by adopting a parser from Brille or MG4J) and in addition there is almost no on-line documentation for Lucene, besides two presentations and an API which does not provide any useful information that could be used to get started.

There is no on-line documentation for Brille either, but for the author of this master thesis it was much easier to contact the author of Brille directly when it was needed. Also some of the implementation details and performance estimates for Brille are documented in [Bjø07]. In addition Brille is the smallest engine considered.

Name	Language	Version	LOC
Brille	Java	0.0.1	8958 (src)
Lucene	Java	2.3.1	27867 (src/java)
MG4J	Java	2.0.1	34979 (java/it)
Zettair	C	0.9.3	43688
MG	C	1.2.1	11603 (src/text)

Table 5.1: Comparison between search engine alternatives

Brille : a short introduction

This part explains shortly how the Brille Search Engine works. More details can be achieved by reading the source code (`http://www.idi.ntnu.no/~trulsamu/brille.tar.gz`).

Brille uses an index structure called Hierarchic Index. A hierarchic index consists of two lists of sorted listed dictionaries. The first one contains a number of searchable indexes of different sizes, and the second one contains a number of small dictionaries flushed by the thread adding new documents. At the search time, both of the lists are processed, the searchable dictionaries are processed first.

The content of a dictionary itself is usually stored on disk, it contains the term id, a pointer to the corresponding inverted list, the number of documents and the total number of occurrences. However, the lowest record of each buffer block is also stored in the memory. In this case it requires only one single disk access to perform the dictionary look-up. Look-up at each level is performed using a binary search. Finally it returns an inverted list iterator which refers to the beginning of each inverted list.

An inverted lists iterator takes the required sorted list dictionary entry as an argument, then it can iterate over the inverted list by pinning new buffers (ensuring that the current buffer is in memory) automatically when the old one ends. Note that in this case it will not fetch the whole inverted list at once, but fetch one buffer after another.

Inverted list iterators for all the terms are then accumulated into a single array list and passed to a search result merger which maintains a min-heap of the inverted list iterators based on the current document number. The search result merger will then calculate the score for each document, and the document id and an associated score will be passed to the result heap which may either accept it or discard it, depending on how many results are received so far and whether a new candidate's score is higher than the lowest known score or not.

There are two important aspects here. First, to calculate a final score a ranking value for the document (IDF) is needed. This values can be stored in memory using a custom-made array list. Second, when all of the top-ranking documents identified, an URI must be resolved. This can be performed by looking-up into a B-Tree, records of which are again stored on disk.

Finally note that the current implementation of Brille does support only single-word queries. However, multiple-word queries can be implemented by using a mechanism similar to the merging of different indexes for the same word. In other words, a number of iterators for each word can be processed using a min-heap. Values from the iterators can then be combined and inserted into the result heap. This method is known as *parallel merge* [CP97]. However the difference between the proposed method and the method described in [CP97], is that latter does not use a hierarchic index.

Practical Use for Brille

Unfortunately, as it was explained earlier, the Brille Search Engine itself was never practically used in the new simulation model. However, the document collection simulation part of the simulation model, as it will be presented, uses data from a dumped dictionary created with Brille. Otherwise, the algorithms to be presented will be based on the information gathered from the source code of Brille.

5.1.3 Processing Model

Most of the previous papers evaluate either a Boolean-AND or the Vector model. Therefore it could be interesting to look at both types of query processing models. The first one expects disjunctive queries, just as in the previous simulation model. However, the second one expects conjunctive queries, just as with a Boolean-AND model. Later these to models will be referred as the AND-model and the OR-model.

5.1.4 Simulation of the Document Collection and the Query Set

The previous simulation model had to use a synthetic document collection and a synthetic query set. The advantage with this was the flexibility to alter the skew of query terms. The drawback of this however was that all of the values and measurements were unrealistic and probably wrong. In addition, the assignment text states indirectly that it could be interesting to look at a real document collection. Also, both of the papers evaluating the pipelined indexing using a real document collection and a real query set.

Therefore a real document collection, TREC GOV2, and a real query set, Terabyte Track 05, could be used. But since it was impossible to use the Brille itself, just a dumped dictionary from the collection indexed with Brille could be used. On the other hand, there are two problems in this case.

Dictionary

First, the dictionary is stored as a text file have a total size at 1.1GB. The dictionary is a way too large to be stored into the main memory. However, the query set consisting of 50000 queries contains only about 32000 distinct words. A good solution is to store all of the words contained in the query set into a hash-map. Then go through the dictionary and perform a look-up for each single word. If a word occurs in both the dictionary and the query set, then store both the number of the scored documents and the term id. To simulate a larger query set it is important to store the right id.

Remind that the previous simulation model had a bug where the term id was expected to be the same as the term rank, resulting in a better load balancing for the global indexing. Using the dictionary position as the term id avoids this problem. For any of the words not occurring in the dictionary the term id is stored as a negative number and the number of the scored documents is 0. Those negative numbers must also be distinct to separate between non-existing terms, while they would not be used to assign a term to a node in any way.

(*Note that in following discussion, the 'probability of a term' means the probability that a term occurs in a given document, not that a randomly chosen word is the given word.*)

Joint Frequency

Second, a problem arise when a query contains a number of distinct terms. Namely how to find the joint probability of a number of terms. This problem has two solutions.

The first one is to generate inverted lists filled with random data. This solution will be very slow when the number of simulated documents increases. The second one is to use either a mathematical expression (Join Method 1) or a look-up table (Join Method 2) to estimate the joint probability for a number of terms. This solution will require the same amount of time when the size of the simulated document collection increases. However, as it will be explained later, it is quite difficult to estimate the joint frequency for the hybrid indexing using the OR-model. Therefore both of the solutions are needed to be implemented. The first solution is straight forward, but the other one can be explained in details.

Join Method 1 For the AND-model the join probability for a number of terms can be the same as the product of the probabilities for each single term, given that all of the words are statistically uncorrelated (which is not true for the real word). That is:

$$P(t_1 \cap t_2 \ldots \cap t_k) = P(t_1) \cdot P(t_2) \cdot \ldots \cdot P(t_k) \tag{5.1}$$

For the OR-model the total probability, given that all of the terms are independent, is 1 minus the probability that none of the words occurs, which is a product of 1 minus the probability of a single word. That is:

$$P(t_1 \cup t_2 \ldots \cup t_k) = 1 - (1 - P(t_1)) \cdot (1 - P(t_2)) \cdot \ldots \cdot (1 - P(t_k)) \tag{5.2}$$

Join Method 2 An alternative approach is to generate a table for join frequency $P(x, y)$ based on the randomly generated data. The problem here is that the information will be limited to the scale of the look-up table and it would require more precision for the low-frequent terms than for the high-frequent terms. A possible solution in this case is to use an $n \times n$ look-up table with a cell x,y containing the value of $P((x/n)^k, (y/n)^k)$. The value of k should be defined in a such way that $(1/n)^k$ would be the same as $(1/D)$, where D is the number of the documents in the collection. For example, having $D = 2500000$ and $n = 100$, the value of k is 4.

Since the latter approach limits precision and requires additional data, the first approach will be used. But the micro-benchmarking part will provide a comparison between the Join Method 1 and the Join Method 2 based on a graphical plot. The final implementation of the simulation model implements a real join based on the random data and the Join Method 1.

5.1.5 Algorithms and Metrics

This part is actually most important for the whole simulation model since it designs the implementation of the algorithms and sets the requirements for the system metrics and constants needed to be measured for the new simulation model. Some of the implementation details are based on the background information presented earlier, while others are based on the source code of Brille and related theory.

This section will now proceed with a semi-mathematical model for all of the algorithms to be implemented. To do so, it requires first to define a number of system variables and fundamental functions. The final implementation of the algorithms which can be found in Appendix B.1 is completely consistent with the following description.

System Variables

All the required variables and related descriptions are provided in Table 5.2-5.3.

D	total number of documents
B	size of a memory buffer block
n	number of nodes
r	maximum number of results (hits) required
f_t	document collection frequency of a term t
$\|q\|$	number of terms in a query q
$\|q'\|$	number of sub-queries to be constructed from a query q
r_q	total number of results (hits) for a query q
r'_q	total number of results (hits) for a query q accumulated so far
$\|sq\|$	number of terms in a sub-query sq
r_{sq}	total number of results (hits) for a sub-query sq
r'_{sq}	total number of results (hits) for a sub-query sq accumulated so far
t_{nd}	network delay time in ms.
t_{nt}	inverse network bandwidth in ms.
t_{rd}	disk rotation delay time in ms.
t_{rs}	disk seek time in ms.
t_{mc}	in-memory binary compute/compare and update time in ms.

Table 5.2: Variables to be used in the simulation model, part 1.

System Functions

To describe the algorithms in details, the simulation model requires to specify a number of fundamental system functions to provide memory management, in-memory look-up, disk fetch, network transfer and a number of CPU operations.

Network Transfer The time required to transfer b bytes of data is specified by Equation 5.3, where t_{nd} is the network delay and t_{nt} is the inverse network bandwidth. It is expected that the networking units on both the sending and the receiving node are busy during the transfer operation, thereafter the transferred data is automatically stored in the memory of the receiving node. It is expected also that the network transfer does not require any CPU access.

$$\tau(b) = t_{nd} + t_{nt} \cdot b \tag{5.3}$$

s_{idx}	size of an index entry in bytes
s_{acc}	size of an accumulator entry in bytes
s_{res}	size of a result entry in bytes
s_t	size of a query/sub-query entry in bytes
s_{qh}	size of a query header in bytes
s_{sqh}	size of a sub-query header in bytes
s_{qrh}	size of a query result header in bytes
s_{sqrh}	size of a sub-query result header in bytes
s_{chd}	number of index entries in a chunk used by hybrid partitioning
t_{lu}	time to perform a dictionary look-up in memory only
t_{hts}	constant factor to perform a determine a top result using a heap
t_{i2l}	constant factor to perform interleave/merge on two accumulator sets
t_{mm}	constant factor to perform a interleave/merge on a number of accumulator sets using a heap
t_c	time required to perform a compare and store operation on two elements
k	any particular number of elements
l	any particular number of accumulator lists/sets

Table 5.3: Variables to be used in the simulation model, part 2.

Disk Transfer The time required to read b bytes of data from disk is specified by Equation 5.3, where t_{ds} is the disk seek time, t_{rd} is the disk rotation delay and t_{dt} is the inverse disk bandwidth. Disk access is expected to be performed in the DMA mode, where disk is blocked for the time required by the transfer function, afterward the data is stored in the memory. As with the network transfer, it is expected that a disk transfer does not require any CPU access.

$$\varphi(b) = t_{ds} + t_{rd} + t_{dt} \cdot b \tag{5.4}$$

Dictionary Look-Up Further it requires to specify a number of CPU functions. The first one is then the time to look-up a number of terms in memory. A memory look-up requires to block the CPU for the time needed to search the B-tree defined by t_{lu}. Then a block of size B is needed to be transferred from disk. Finally it requires to perform a binary search on the dictionary block, but this time is also included in t_{lu}.

In addition, when a look-up for k terms is performed. it requires to sort the resolved terms by increasing document frequency. It can be performed by blocking the CPU for $n \cdot \log_2(n) \cdot t_c$.

All the operations described here performed on k terms will be referred as $\lambda(k)$.

Memory Allocation It is expected that the memory allocation does not require any CPU access. This is actually wrong for a real system. But this value is so small that it can probably be ignored.

Interleave Two Lists One of the most important operations performed during the search is an interleave operations on two accumulator lists of a total size k. This requires to block the CPU for $\zeta_{i2l}(k) = t_{i2l} \cdot k$ milliseconds.

Heap Top And Merge Next, when the results from a number of accumulator lists are combined, some of the processing algorithms will require to determine r top-scored results from any k candidates and sort these in a decreasing order. This can be normally performed using a min-heap of r elements. The total time complexity of this operation is $\zeta_{hts}(k, r) = t_{hts} \cdot (k + 2 \cdot r) \cdot \log_2(r)$ milliseconds.

Multi-way Merge One more operation which is performed when a number of accumulator lists are merged into a single accumulator list. The total time complexity of this operation is $\zeta_{mm}(k, l) = t_{mm} \cdot k \cdot \log_2(l)$ milliseconds.

Basic Algorithms

Local Indexing The basic implementation of the query processing with the local indexing is demonstrated by Algorithms 11 - 12. The algorithm describes different processing stages and time constrains. An important detail missing is the estimation of r_q, r'_q, r_{sq} and r'_{sq}. These values are either calculated from a randomly generated data set or calculated using the Join Method 1. In this case r'_{sq} can be calculated iteratively from its previous $r'_{sq}/(D/n)$ and the current f_t/n.

$$r'_{sq} \leftarrow (\frac{r'_{sq}}{D} \bowtie f_t/n) \cdot D \tag{5.5}$$

The final value of r'_{sq} is the r_{sq} itself.

Since the sub-query result sets do not overlap, the values of r'_q and r_q can be calculated as follows:

$$r'_q \leftarrow r'_q + r_{sq} \tag{5.6}$$

Global Indexing The basic implementation of the query processing with the global indexing is demonstrated by Algorithms 13 - 14. As with the local indexing, the values of r_q, r'_q, r_{sq} and r'_{sq} are calculated either from a randomly generated data set or calculated using the Join Method 1. The values of r'_{sq} and r_q are calculated using the current f_t.

$$r'_{sq} \leftarrow (\frac{r'_{sq}}{D} \bowtie f_t) \cdot D \tag{5.7}$$

. The final value of r'_{sq} is assigned to r_{sq}. As a difference from the local indexing, distinct sub-query result sets do overlap in this case. The values of r'_q and r_q can therefore be calculated as:

$$r'_q \leftarrow (\frac{r'_q}{D} \bowtie \frac{r_{sq}}{D}) \cdot D \tag{5.8}$$

1 Receive a query q from the gateway node, $\tau(s_{qh} + |q| \cdot s_t)$;
2 Allocate $s_{acc} \cdot r \cdot (|q'| + 1)$ bytes of memory;
3 Generate and schedule the sub-queries for this query;
4 Initialize an accumulator list;
5 **repeat**
6 | Wait for the next result, r_{sq};
7 **until** *All of the sub-query results are received* ;
8 Parallel merge the results to determine the top-scored results,
$\zeta_{mms}(\sum(\min(r_{sq}, r)), |q'|)$ with an additional cost of $\zeta_{hts}(\sum(\min(r_{sq}, r)), r)$;
9 Transfer the results back to the gateway node,
$\tau(s_{qrh} + \min(\sum(\min(r_{sq}, r)), r) \cdot s_{res})$;
10 Free the allocated memory;
11 Terminate;

Algorithm 11: LI query process, algorithm details

1 Receive a sub-query sq from the query node, $\tau(s_{sqh} + |sq| \cdot s_t)$;
2 Perform a dictionary look-up for the sub-query terms, $\lambda(|sq|)$;
3 **if** *at least one term is missing* **then**
4 | **if** *system is in AND-mode* **then**
5 | | Transfer 0 results to the gateway node, $\tau(s_{sqrh})$;
6 | | Terminate;
7 | **else**
8 | | Eliminate the non-existing terms;
9 Allocate $D \cdot (s_{idx}/n + s_{acc})$;
10 Initialize an accumulator list;
11 **for** *sub-query terms ordered by increasing* f_t **do**
12 | Fetch the inverted list for a term t, $\varphi(D/n \cdot f_t \cdot s_{idx})$;
13 | Merge the inverted list into the accumulator list, $\zeta_{i2l}(r'_{sq} + D/n \cdot f_t)$;
14 Determine and sort the top-scored results, $\zeta_{hts}(r_{sq}, r)$;
15 Transfer the results back to the gateway node, $\tau(s_{qrh} + \min(r_{sq}, r) \cdot s_{res})$;
16 Free the allocated memory;
17 Terminate;

Algorithm 12: LI sub-query process, algorithm details

```
1  Receive a query q from the gateway node, τ(s_qh + |q| · s_t);
2  Perform a dictionary look-up for the query terms, λ(|q|);
3  if at least one term is missing then
4      if system is in AND-mode then
5          Transfer 0 results to the gateway node, τ(s_qrh);
6          Terminate;
7      else
8          Eliminate the non-existing terms;
9  Allocate D · s_acc · |q'| bytes of memory;
10 Generate and schedule the sub-queries for this query;
11 Initialize an accumulator list;
12 repeat
13     Wait for the next result, r_sq;
14     Interleave the current list with the previous achieved result, ζ_i2l(r'_q + r_sq);
15 until All of the sub-query results are received ;
16 Determine and sort the top-scored results, ζ_hts(r_q, r);
17 Transfer the results back to the gateway node, τ(s_qrh + min(r_q, r) · s_res);
18 Free the allocated memory;
19 Terminate;
```

Algorithm 13: GI query process, algorithm details

```
1  Receive a sub-query sq from the query node, τ(s_sqh + |sq| · s_t) ;
2  Perform a dictionary look-up for the sub-query terms, λ(|sq|);
3  Allocate D · (s_idx + 2 · s_acc);
4  Initialize an accumulator list;
5  for sub-query terms ordered by increasing f_t do
6      Fetch the inverted list for a term t, φ(D · f_t · s_idx);
7      Merge the inverted list into the accumulator list, ζ_i2l(r'_sq + D · f_t) ;
8  Transfer the results back to the gateway node, τ(s_qrh + r_sq · s_res);
9  Free the allocated memory;
10 Terminate;
```

Algorithm 14: GI sub-query process, algorithm details

Pipelined Global Indexing The basic implementation of the query processing with the global indexing is demonstrated by Algorithm 15. The values of r_q, r'_q, r_{sq} and r'_{sq} are estimated using the same equations as for the global indexing.

```
1  Receive a query q from the gateway node, τ(s_qh + |q| · s_t);
2  Perform a dictionary look-up for the query terms, λ(|q|);
3  if at least one term is missing then
4      if system is in AND-mode then
5          Transfer 0 results to the gateway node, τ(s_qrh);
6          Terminate;
7      else
8          Eliminate the non-existing terms;
9  Allocate 2 · D · s_acc bytes of memory as Buffer1;
10 Generate the sub-queries for this query and create a query bundle with an empty
   accumulator list;
11 for All of the sub-queries ordered by increasing lowest f_t do
12     Transfer the query-bundle to the next node sq from the query node,
       τ(s_sqh · |q'||q| · s_t + r'_q · s_acc) ;
13     Free Buffer1 on the previous node;
14     Perform a dictionary look-up for the sub-query terms, λ(|sq|);
15     Allocate D · s_idx as Buffer2;
16     for sub-query terms ordered by increasing f_t do
17         Fetch the inverted list for a term t, φ(D · f_t · s_idx);
18         Merge the inverted list into the accumulator list, ζ_i2l(r'_q + D · f_t) ;
19     Free Buffer2;
20     if this sub-query is the last one then
21         Determine and sort the top-scored results, ζ_hts(r_q, r);
22         Transfer the results back to the gateway node, τ(s_qrh + min(r_q, r) · s_res);
23         Free Buffer1;
24         Terminate;
```

Algorithm 15: PL query process, algorithm details

Hybrid Indexing The simulation of the hybrid indexing is very problematic. As it turns out, the hybrid indexing does not provide an easy approach to perform AND-model query processing. Since in this case, the corresponding documents for the different terms are stored on different nodes. In this case, these cannot be merged at the sub-query processing stage, so all the inverted lists must be transferred to the receptionist node and then merged together. Since all the processing must be performed by the receptionist node, this approach has a bad performance.

An alternative approach is to, while processing a sub-query, create a number of lists

where a range of documents is hashed to a node and then to transfer those records to the different nodes in a kind of all-to-all exchange operation. When these records arrives on the destination nodes, each node will now receive the document id's corresponding to the same range, now the inverted indexes can be interleaved to solve the sub-queries. In a final stage, the sub-query results are needed to be transferred to the receptionist node and finally combined using a multi-way merge operation.

This approach will in practice never be implemented in the new simulation model, but its performance seems to be similar or worse than the performance of the local indexing.

The hybrid indexing can be implemented for the OR-queries. The processing and the calculation itself will be the same as for the GI, except from an important detail. In the calculations for a sub-query, a partial term frequency f'_t will be used instead of the actual term frequency f_t. Since a chunk contains s_{chd} entries, the term frequency impact of a chunk is either $\frac{s_{chd}}{D}$, for all the chunks except from the last one, or $\frac{(D \cdot f_t) \mod s_{chd}}{D}$, for the last chunk. The total number of chunks is $\lceil f_t \cdot D / s_{chd} \rceil$.

However, since the records for the same document are probably mapped to a number of different nodes, the total number of the sub-query results cannot be estimated using the Join Method 1. The only way to simulate the hybrid indexing for the OR-model is therefore to generate the inverted lists, divide them and then combine them back. Because of the limited scalability, this method cannot be used for a large simulated document collection.

Parallel Merge

An interesting observation is as follows: since the standard algorithms tend to merge two and two lists, it requires to merge each new list into the list combined so far. For j inverted lists of size l, the worst case number processed elements would be $O(j)$ times larger than the total number of elements. This is because of the fact that, the number of processed elements would be $2l + 3l + 4l + \ldots + jl = O(lj^2)$. But the total number of the elements to be processed is only lj.

A parallel merge version, which retrieves all of the inverted lists and merges them using a min-heap would in this case result in $O(lj \log(j))$ processed elements. The number of processed element in this case is $O(\log(j))$ times faster than the first version.

Note that the parallel merge concept is not the same to the one described in [CP97] or implemented in Brille, since it retrieves the whole inverted list for each term and stores it into the main memory before it starts to process the data. As a result it would require much more memory than the original approach, but reduce the CPU load. The original parallel merge in the other hand, would take more processing time but reduce the memory load. Ironically, both of the methods perform a merge operation on a number of inverted lists in parallel, hence the name.

Local Indexing A modified version of the processing algorithms for the local indexing using a parallel merge is demonstrated by Algorithms 16 - 17. The only difference is the memory requirements and the processing of the inverted lists for sub-query.

1 Receive a query q from the gateway node, $\tau(s_{qh} + |q| \cdot s_t)$;
2 Allocate $s_{acc} \cdot r \cdot (|q'| + 1)$ bytes of memory;
3 Generate and schedule the subqueries for this query;
4 Initialize an accumulator list;
5 **repeat**
6 | Wait for the next result, r_{sq};
7 **until** *All of the sub-query results are received* ;
8 Parallel merge the results to determine the top-results, $\zeta_{mms}(\sum(\min(r_{sq}, r)), |q'|)$ with an additional cost of $\zeta_{hts}(\sum(\min(r_{sq}, r)), r)$;
9 Transfer the results back to the gateway node, $\tau(s_{qrh} + \min(\sum(\min(r_{sq}, r)), r) \cdot s_{res})$;
10 Free the allocated memory;
11 Terminate;

Algorithm 16: LIPM query process, algorithm details

Global Indexing A parallel merge version of the basic global indexing implementation is demonstrated by Algorithms 18-19. This version differs from the previous one in the memory requirements and in how the partial data is processed.

```
1  Receive a sub-query sq from the query node, τ(s_sqh + |sq| · s_t) ;
2  Perform a dictionary look-up for the sub-query terms, λ(|sq|);
3  if at least one term is missing then
4      if system is in AND-mode then
5          Transfer 0 results to the gateway node, τ(s_sqrh);
6          Terminate;
7      else
8          Eliminate the non-existing terms;
9  Allocate D/n · (s_idx · |sq| + s_acc);
10 Initialize an accumulator list;
11 for sub-query terms ordered by increasing f_t do
12     Fetch the inverted list for a term t, φ(D/n · f_t · s_idx);
13 Merge the inverted lists into the accumulator list, ζ_mm(∑(D/n · f_t), |sq|) ;
14 Determine and sort the top-scored results, ζ_hts(r_sq, r);
15 Transfer the results back to the gateway node, τ(s_qrh + min(r_sq, r) · s_res);
16 Free the allocated memory;
17 Terminate;
```

Algorithm 17: LIPM sub-query process, algorithm details

```
1  Receive a query q from the gateway node, τ(s_qh + |q| · s_t);
2  Perform a dictionary look-up for the query terms, λ(|q|);
3  if at least one term is missing then
4      if system is in AND-mode then
5          Transfer 0 results to the gateway node, τ(s_qrh);
6          Terminate;
7      else
8          Eliminate the non-existing terms;
9  Allocate D · s_acc · (|q'| + 1) bytes of memory;
10 Generate and schedule sub-queries for this query;
11 Initialize an accumulator list;
12 repeat
13     Wait for the next result, r_sq;
14 until All of the sub-query results are received ;
15 Interleave the accumulator lists into a single accumulator list, ζ_mm(∑(r_sq), |q'|);
16 Determine and sort the top-scored results, ζ_hts(r_q, r);
17 Transfer the results back to the gateway node, τ(s_qrh + min(r_q, r) · s_res);
18 Free the allocated memory;
19 Terminate;
```

Algorithm 18: GIPM query process, algorithm details

1 Receive a sub-query sq from the query node, $\tau(s_{sqh} + |sq| \cdot s_t)$;
2 Perform a dictionary look-up for the sub-query terms, $\lambda(|sq|)$;
3 Allocate $D \cdot (s_{idx} \cdot |sq| + s_{acc})$;
4 Initialize an accumulator list;
5 **for** *sub-query terms ordered by increasing* f_t **do**
6 ⌊ Fetch the inverted list for a term t, $\varphi(D \cdot f_t \cdot s_{idx})$;
7 Merge the inverted lists into the accumulator list, $\zeta_{mm}(\sum(D \cdot f_t), |sq|)$;
8 Transfer the results back to the gateway node, $\tau(s_{qrh} + r_{sq} \cdot s_{res})$;
9 Free the allocated memory;
10 Terminate;

Algorithm 19: GIPM sub-query process, algorithm details

Pipelined Global Indexing A parallel merge version of the pipelined indexing is demonstrated by Algorithm 20. The basic algorithm is modified in the memory requirements and the processing of the inverted lists.

Hybrid Indexing A parallel merge version of the hybrid indexing can be implemented by replacing the sequential merge of the inverted index chunks with a parallel one. However, the problems associated with the AND-model and Join Method 1 estimation presented previously are still unresolved. The implementation of the HDPM will therefore be also limited. Otherwise, the algorithm itself is similar to the one for the global indexing, except from the f_t.

Approximation Techniques and Filtering Methods

Due to the time limits of this master thesis, none of the filtering methods or approximation techniques were implemented.

5.1.6 Visualisation Tool and Reporting

The final task is to provide enough information so it would be possible to analyze and understand the performance issues under the experiments. For this purpose, the experiment reports can be presented in two different versions. In the first one, a graphical history for each node showing the information about the disk, Ethernet, memory and CPU loads during the whole experiment can be stored. The disk and the network load information must in this case provide not only the resource load, but also the effective transfer bandwidth. In addition to the node state information it is interesting to provide a graphical representation of all of the processes in the system. This information can be used to determine problems and errors, and to explain long delays. For the textual reports it is interesting to measure the amount of the data read from disk, sent and received by a node. The query and term evaluation metrics such as the query response

```
1  Receive a query q from the gateway node, τ(s_qh + |q| · s_t);
2  Perform a dictionary look-up for the query terms, λ(|q|);
3  if at least one term is missing then
4      if system is in AND-mode then
5          Transfer 0 results to the gateway node, τ(s_qrh);
6          Terminate;
7      else
8          Eliminate the non-existing terms;
9  Allocate 2 · D · s_acc bytes of memory as Buffer1;
10 Generate the sub-queries for this query and create a query bundle with an empty
   accumulator list;
11 for all of the sub-queries ordered by increasing lowest f_t do
12     Transfer the query-bundle to the next node sq from the query node,
       τ(s_sqh · |q'||q| · s_t + r'_q · s_acc) ;
13     Free Buffer1 on the previous node;
14     Perform a dictionary look-up for sub-query terms, λ(|sq|);
15     Allocate D · s_idx · |sq| as Buffer2;
16     for sub-query terms ordered by increasing f_t do
17         Fetch the inverted list for a term t, φ(D · f_t · s_idx);
18     Merge the inverted lists and the accumulator list into a new accumulator list,
       ζ_mm(r'_q + Σ(D · f_t), |sq| + 1) ;
19     Free Buffer2;
20     if this sub-query is the last one then
21         Determine and sort the top-scored results, ζ_hts(r_q, r);
22         Transfer the results back to the gateway node, τ(s_qrh + min(r_q, r) · s_res);
23         Free Buffer1;
24         Terminate;
```

Algorithm 20: PLPM query process, algorithm details

time, the total number of terms and results and the total index size are of the highest interest. DesmoJ allows to create Histograms for such data in an easy way.

The simulation model to be presented, supports both textual and graphical reports. All the graphical experiments are referred later as the *trace mode*. In this case it will be impossible to specify a number of warm-up queries. The trace mode is not suitable for long runs, the non-trace mode in other hand can use a number of queries to warm-up. Since it does not log any node states or process events it runs slightly faster. However, in both modes, it is possible to use the built-in tracing mechanism and the reporting and debugging functionality provided by the DesmoJ.

Finally, the most important result metrics to be used in later discussion of the experiment results are the average number of queries per second (QPS) and the average query response time. But because of the multiprogramming, there is a kind of a trade of between these two.

5.2 Implementation of the New Simulation Model

The code base of the simulation model consists of 37 classes divided between 6 packages: `simulation.model`, `simulation.node`, `simulation.query`, `simulation.process`, `simulation.log` and `simulation.micro`. Most of the ideas and concepts of the simulation model have been already explained. This section will therefore give only a short overview over how the code and interaction between the classes is organized. More details can be achieved either from the previous section of directly from the source code provided in Appendix B.1.

5.2.1 simulation.node

The node package provides one of the most fundamental parts of the simulation model, namely it creates a `Node` abstraction for the later simulation model by a number of methods such as `malloc`, `free`, `fetch`, `transfer`, `lookupTerms`, `holdCPU`, `heapTopAndSort`, etc., which can later be used by any query or sub-query process without going into calculation details. The network transfer, disk access and CPU times and memory requirements are based on the equations specified in the previous section. The `Node` class tracks also the total amount of the data read from disk, sent to and from this node, and the total number of disks seeks performed.

The `Node` encapsulates a `MemHandler` process which implements the memory handler idea described in the previous section. The second class in this package, `ResHandler`, implements in its turn the previously described idea with the resource handler.

5.2.2 simulation.query

The second fundamental part of the simulation model is contained in the query package which consists of 9 different classes: `Query`, `QueryResult`, `IntexTools`, `IndexHitList`, `SimpleIndexHitList`, `SimpleIndexHit`, `SimulatedIndexHitList`, `SubQueryResult` and finally `QueryResult`.

The `Query` is an `Entity` class used to store a number of terms specified by a term id and an `IndexHitList`. This class provides also a number of methods to retrieve the term ids, total index size, eliminate non-existing terms and, most important, to partition a query into a number of sub-queries. There are three different methods to partition a query into a number of sub-queries - one for the global indexing, one for the local indexing and one for the hybrid indexing. And all of the methods use different scenarios whenever the document collection is simulated using the Join Method 1 for the approximation, or if it actually uses a kind of inverted lists filled with random data.

The `SubQuery` is also an `Entity` class similar to the `Query` itself. It stores a number of term ids associated with a number of `IndexHitLists`. This class provides a number of methods to retrieve the number of scored elements, find which term has the least number of scored document, retrieve a sorted list of the terms based on the number of scored documents, etc.

The `IndexHitList` is just an interface implemented by `SimpleIndexHitList` and `SimulatedIndexHitList`. The difference between these two is that the first one maintains an array list of `SimpleIndexHit`, which is a document id and a corresponding number of documents, but the other one uses only a fractional number to specify the term frequency.

Hit lists implement both the accumulator and the inverted index concepts. When a query is scheduled it contains one single hit list for each of its terms. When a query is partitioned into a number of sub-queries, the corresponding hit lists are partitioned according to the indexing scheme. During the processing the inverted litsts are combined into a number of larger inverted lists till a single inverted list for the whole sub-query/query is obtained.

The `IndexTools` class provides four static methods to **interleave** or **merge** hit lists, either two and two or all at the same time. The implemented methods use also different scenarios whether the simulated-only mode is used or not and whether the AND-query mode is used or not.

The `SubQueryResult` is an `Entity` class used to represent the result set for a sub-query. It can be initiated either with a single `IndexHitList` or with an array list of `IndexHitList`s. There are only two non-constructor methods provided in this class: one to get the total number of the documents (hits) and one to get the resulting hit list.

The `QueryResult` is another `Entity` class used to represent the final result set for a query. The class contains a hit list which can be obtained either by assigning it using a public method `setHitList`, or by adding a number of `QueryResult` and combining them using the methods provided by `IndexTools`.

Figure 5.4 demonstrates the structure of the query package.

Figure 5.4: A class diagram for the simulation.query package

5.2.3 simulation.model

The model package consist of four classes: `Config`, `ModelS`, `GeneratorProcess` and `QueryLogReader`. The first one defines a number of the system constants, such as simulation type, indexing mode, experiment duration, AND/OR mode, CPU factor, CPU time constants. `Config` provides also a method to load the system parameters from a `.property` file. An example configuration file is provided in Appendix 1.3.

`QueryLogReader` is a class which reads the `docstat` file and the `querylog` file stored in the directory specified by the `DATAPATH` constant and provides a method to retrieve a next query resolved as an array of integer values, where an element $2 * i$ specifies a term id and an element $2 * i + 1$ specifies the corresponding number of scored documents.

`ModelS` is the main simulation class which initializes and schedules the simulation experiment. The most important methods provided in this class are `shceduleQuery`, which is used by a `GeneratorProcess` (see `simulation.process`) to start a new query process on a given node, when the number of query processes falls below the specified number, and `querySolved` which is used by a query process to notify the simulation model about its termination and store the query statistics.

The interaction here is follows, the `GeneratorProcess` would start up-to a given number of queries and passivate. When a query process finishes, it calls `querySolved` on the simulation model. The latter in its turn re-activates the generator process which submits a number of new queries to the different nodes using the `scheduleQuery` method. The query content data used by the `GeneratorProcess` is retrieved from the `QueryLogReader` using the `getNextQuery` method.

Note that the model creates one more Node that it is specified in the configuration. The last node is just a gateway to receive queries and transfer results.

Figure 5.4 shows how the simulation classes cooperate, the diagram does not include any classes from the query or the process packages.

5.2.4 simulation.process

The process package combines all the other classes into a complete model. The package contains 16 classes including two abstract classes, `QueryProcess` and `SubQueryProcess` and a number of classes implementing the indexing algorithms described in the previous section.

Each class extending the `QueryProcess` contains a reference to a `Node`, a `Query`, a `Queue` to store the incoming `SubQueryResult`s and finally a `QueryResult`. It saves also its starting time which is used to calculate the query response time. Two most important functions implemented in the `Query` class are `checkQueue` and `ackQuery`, which implements the ideas described in the previous section: `ackQuery` is meant to be called by a sub-query process to post its results into the sub-query result queue. In response for this event the corresponding query process is scheduled to be activated. When a process extending the `QueryProcess` is activated, it needs to perform its own implementation of the `checkQueue` function, which usually does the post-processing and the removal of the sub-query results from the result queue.

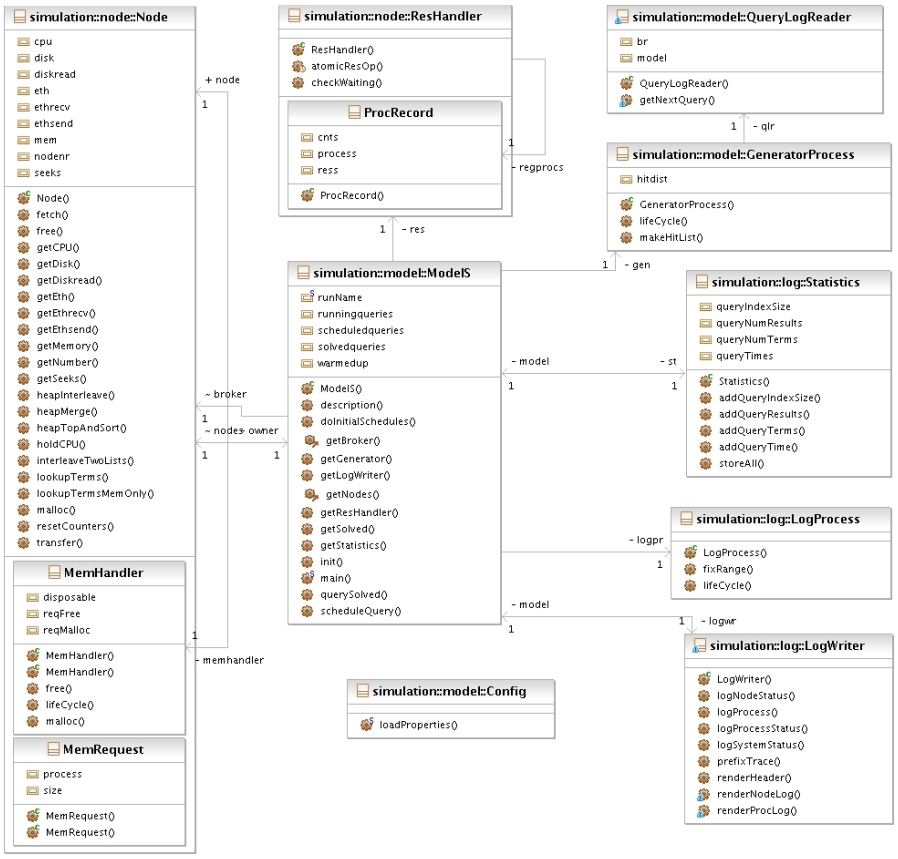

Figure 5.5: A class diagram for some of the classes contained in the simulation.model, simulation.node and simulation.log packages

The `SubQueryProcess` class provides a framework for the sub-query processes. The class contains the referrences to its sub-query, node and query process.

The rest of the package extends these two classes used to implement the previously presented algorithms by using the methods provided by `simulation.node`, `simulation.query` and eventually stores the progression using the methods provided by `simulation.log`.

Figure 5.6 illustrates the class diagram for query package.

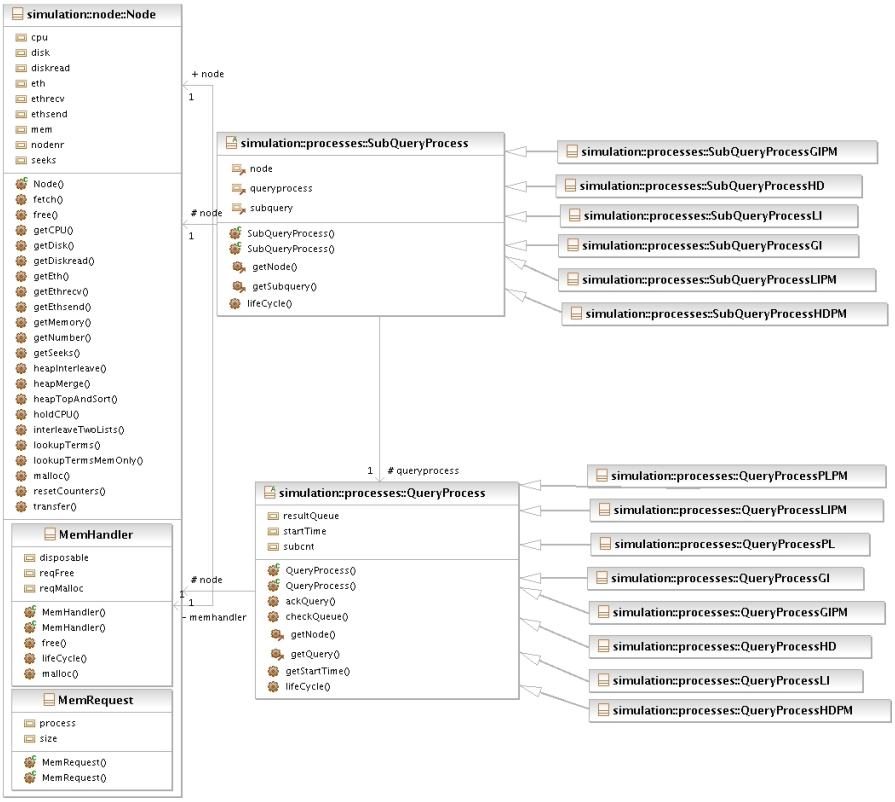

Figure 5.6: A class diagram for the simulation.query package

5.2.5 simulation.log

This package provides three classes used to log the process and the node state information and to report the final statistics for an experiment. `LogProcess` is a `SimProcess` which samples all the information such as the average CPU, Disk, Memory and Ethernet usage, th amount of the data transferred from and to a node, and the amount of the data transferred from disk. This process is used only in the *trace mode* and all of the sampled information is then passed to an instance of the `LogWriter`.

The `LogWriter` is the class responsible for creation and storage of a graphical report for the experiment performed in the trace mode. It is also the largest class in the simu-

lation model. The graphical report files have always the same prefix as the experiment name, but the suffix is either '_node.png' or '_process.png' corresponding to the node or process state log. The maximum size of the report diagram is specified by two constant in the source code, the current size is 2000 by 5000 pixels. For longer runs the images will be scaled down to satisfy these constraints, therefore the trace mode is best to be used on the experiments with a duration between 1000 and 30 000 ms and a number of nodes between 2 and 8.

`Statistics` is an other class used to measure the experiment results. This class encapsulates a number of histograms for the number of the query terms, the maximum possible number of the query results, the total index size and the query response time. Results from all of the histograms are automatically stored in the basic DesmoJ report, but the average, minimum and maximum values for these are also stored in the graphical and the textual report.

Finally, the `Statistics` is used to store a short textual summary for the experiment performed in the non-demo mode. This information stored is stored in a '.log' file with the experiment name.

5.2.6 simulation.micro

The final package provides three classes used to estimate the CPU metrics: `InterleaveTwoTest`, `HeapInterleaveTest` and `HeapTopAndSortTest`. The content and the results of these will be explained in the next section.

5.3 Micro-Benchmarking and Parameter Estimation

This section provides both the information and the estimated values for system constants used in the simulation model presented in previous chapter. Some of these metrics are obtained from a number of small code fragments executed on the author's workstation, while some of these metrics are obtained from the external sources, such as data sheets, practical exercisese in earlier courses, other master thesis reports or Internet reviews. Two final subsections of this chapter reviews also some of the features and the characteristics of the document collection, query set and calculation methods used to estimate the joint probability of a combination of two different terms.

Network, Disk and CPU characteristics are discussed in subsections 5.3.1, 5.3.2, refmicrocpu respectively. Subsection 5.3.4 explains how metrics used to define sizes of the data structures were chosen. Finally Subsection 5.3.5 reviews some of the aspects of the combination of the query set and the document collection and Subsection 5.3.6 reviews the joint probability estimates and compares it to the real world.

The next chapter will definine the goals for the experiments to be performed and then present and discuss the obatined results.

5.3.1 Network Characteristics

As it was stated earlier, the disk performance is estimated using Equation 5.3, where t_{nd} is the network delay and t_{nt} is the inverse bandwidth. For a Gigabit network t_{nt} is therefore 8e-6ms. However, the network delay will vary from a system to another system. Fortunately, the author of this master thesis had taken a course in parallel programming where one of the practical assignments was to measure the network bandwidth and the network delay for a cluster of DELL 750 (3.4Ghz Pentium IV, 1GB RAM) interconnected with a gigabit Ethernet switch. The source code for the assignment is provided in the Appendix 1.2.

The measured t_{nd} is somewhere between 6.5775e-05s and 8.26295e-05s, so the estimated value to be used is assumed to be 0.08ms.

5.3.2 Disk Characteristics

As a reminder, the disk transfer time is usually estimated by Equation 5.4. In this equation t_{ds} is the average disk seek time, t_{rd} is the average disk rotation delay and finally t_{dt} is the average inverse disk transfer bandwidth.

Magnetic Hard Drive

The previously used disk characteristics were taken from the data sheet for the Barracuda ES hard-drive [LLC08b] using t_{ds} at 8.5ms, t_{rd} at 4.16ms and t_{dt} at 1.25e-5ms. The same bandwidth, somewhere about 80Mbps, is measured by [Bjø07]. However an updated version of this hard-drive, Barracuda ES 2 [LLC08a], provides a sustained bandwidth at 105MBps instead of only 78MBps resulting in t_{dt} at 0.95e-5ms. Even more, Seagate provides also two other enterprise disk branches, Cheetah [LLC08c] and Savio [LLC08d]. Cheetah NS provides disk read seek time at 3.9ms, rotation delay at 2.98ms, and sustained transfer rate at 97Mbps (t_{dt} at 1.03e-5ms). The total capacity of a Cheetah NS can be up to 300GB. Savio 15K is a 2.5 inch disk with the total capacity up to 73GB, providing a disk read seek time at 2.9ms, and a rotation delay at 2.0ms. The transfer rate varies between 112 and 79MBps depending on the track position on the disk.

Flash Hard Drive

An interesting alternative for a classic hard drive is a flash hard drive. Today there are a number of such products having quite different characteristics. As an example, a Samsung 64GB SSD disk reviewed by an Internet hardware web-site called `slashgear.com` provides a measured transfer rate at 59.3MBps and an average disk access time at 0.3ms. There are also a number of other flash drives providing much better performance, but the performance of a Samsung SSD disk is a great improvement compared to the performance of a hard drive such as Barracuda ES.

For the later expirements it can be interesting to evaluate the performance of a system using a simulated flash disk. This can be easily integrated into the current

implementation of the model by setting the average seek time to the value of the random access time and the average rotation delay to 0.

5.3.3 CPU Characteristics

To estimate the values for the CPU parameters used in the simulation model the code base for the simulation model were extended with three new classes: `HeapInterleaveTest`, `InterleaveTwoTest` and `HeapTopAndSort`, which can be found in the package `simulation.micro` of the source code provided in Appendix B.1.

All the following tests were performed on a workstation with two Intel Pentium 4 3.0GHz CPUs and 1GB RAM, running Java 1.6.0_03.

Interleave Two Lists

`InterleaveTwoTest` performs simply a test consisting of the user-defined number of runs, where each run generates two integer arrays (suggested default value is 200000) filled with random data. Each $2i$th element is suggested to be a document id, and each $2i + 1$th element is suggested to be the corresponding score. These two lists is then interleaved to produce a combined list and the time used to perform this operation is measured. The average value to the number of elements (a half of the resulting array size) over all of the runs is then printed out as the test result.

From the tests performed it was observed that the resulting value depends on both the size of the array used and the number of runs. Two possible reasons for this are the speculative prefetching of the memory data and the run-time compiling in Java. Long arrays processed linearly can be effectively stored in the processor cache, so it would require less time then to access the required data stored in the main memory. Since a single method will be performed over and over, the run-time compiling provided by the Java VM significantly improves the final performance for the longer runs.

Anyway, under the latest performed experiments consisting of 5 consecutive tests of 1000 runs each, the resulting values were 2.4594515e-5, 2.4065914e-5, 2.4076205e-5, 2.4136100e-5 and 2.4219670e-5 milliseconds for each element. This results in estimated t_{i2l} at 2.421848e-05.

Finally, the comparison instruction time, t_c, used currently as a scaled time estimate for sorting of the resolved terms is expected to be the same as the interleave time for a single element or better.

Heap Extraction

`HeapTopAndSortTest` contains a simple implementation of a binary heap which can be used to extract and sort the top-scored documents from a hit list. Just as in the previous case, the hit list itself is simulated using an integer array where the document ids is then used as the heap values and their scores as the corresponding keys. A heap of a given size (with 1024 used as the default value) is then initiated at the beginning of the array. Then all the remaining entries are evaluated against the root element (the smallest element

contained in the heap). If the current element is smaller than the root element, the root element is replaced and the heap is updated. The code does not perform any sorting itself, but this task could be performed easily by replacing the root element with the rightmost leave element and updating the heap. This would be equivalent to an in-place $O(n \log(n))$ sorting.

All the tests are processed in runs, and the total processing time for each run is then divided by the number of the entries and the binary logarithm to the size of the heap. The average value is then finally returned.

An experiment with 5 tests containing 1000 runs each for a list of 100000 entries and a heap of 1024 elements resulted in 5.999120e-7, 5.885690e-7, 5.847940e-7, 5.880140e-7 and 5.869340e-7 milliseconds for each element. The average t_{hts} measured is then 5.896446e-7 milliseconds. This result is very suspicious, but the program code seems to be free from errors or miscalculations.

Heap-Interleave Lists

`HeapInterleaveTest` provides the code used to simulate and evaluate the time required by an interleave operation of a number of lists using a heap.

The code generates a number of hit lists using a number of integer arrays just as in the last case, then it initializes a number of pointers to a position inside each of the lists. These pointers are contained in a heap which allows to extract a pointer referring to the hit list with the lowest current document id. The lists are then processed in turn by extracting the lowest pointer, combining this result with the data extracted so far (which results in either an increment of the partial score or in a write back of the partial score and reseting it to the current value) and finally incrementing the pointer and updating the heap. The resulting time value used to perform extraction is divided by the total number of the elements and the logarithm base two to the number of lists, and then averaged over a number of runs.

An experiments with 5 consecutive tests with 100 runs each, using 4 lists with 40000 elements resulted in a average cost at 5.405691e-5, 4.874002e-5, 4.819033e-5, 4.806042e-5 and 4.823812e-5 milliseconds for each element. The average t_{mm} value is then 4.945716e-05 milliseconds.

Look-up

The time needed to look-up a term in the dictionary contained in memory was never measured by the author of this report, but it was provided along with the efficiency model of Brille in [Bjø07]. The expected value of t_{lu} is therefore chosen to be 0.0105 milliseconds.

5.3.4 Data Structures and Memory Requirements

The sizes of the data structures were actually never measured, but they are expected to be reasonably large. The data structures presented in [Bjø07] support these assumptions.

The header lengths for a query, query result, sub-query or a sub-query result are expected to be about 32 bytes.

An index location entry, which is an entry in an inverted list describing a single occurrence of a term, contains normally the position number and a single byte description resulting in 5 bytes of data. An index document entry, which describes a single document containing a term, contains normally the document number and the number of occurrences, 8 bytes in total.

It is also expected that an accumulator entry consists of the document id and a partial score, resulting in a 8 bytes of data in total. The same value is also used for a result entry. But a query entry representing a term can consist of a single term id, instead of the actually word representation, this results in only 4 bytes of data.

The total memory size for a basic configuration is chosen to be 4GB. To be effective in Desmo-J the memory is then simulated as a number of 1MB blocks (1048576 bytes), resulting in 4096 blocks in total.

The basic chunk size used by a hybrid index is expected to be 1024 entries as it suggested by Sornil [Sor01].

Finally, the dictionary look-up for a single word requires a single disk access to fetch a block of dictionary entries and then perform a binary search within the block. The question is how much data is needed to be fetched from the disk in the case. For Brille this value is defined by the `BUFFER_SIZE` system constant, which has a default value at 131072 bytes.

5.3.5 Document Collection Parameters and Characteristics

Since the query set and the document set used by the simulation model are based on the real data, no estimation of the set characteristics is required. But it can be interesting to know some of the characteristics of the used data.
From the data contained in the dictionary, query log and stat-files:

- Document Collection
  - Trec GOV2
  - 32 801 629 words
  - 25 205 179 documents
- Query Set
  - Terabyte track 05
  - 50 000 queries
  - 31 220 words are contained in the document collection
  - 3912 words are not contained in the document collection

Other characteristics such as the term frequencies and the average query length for the whole collection were not measured, but the simulation model includes the observed query length and load distribution in both the graphical and the textual reports.

A sample of the first 50 queries from the query log are provided in Appendix 1.1.2. The resulting frequency for each word in the query phrase is given in the square brackets.

5.3.6 Term Disjunction and Conjunction Frequency

This part describes not a benchmark used to estimate some of the simulator parameters, but a validation of the methods chosen to calculate the joint frequency of two terms. As a reminder, the joint frequency of two terms can be estimated either as a function of the term frequencies or as a result of a number of table look-ups for each two and two terms. The resulting frequency distributions for both the AND and the OR models using the Join Method 1 are illustrated by Figures 5.7-5.8.

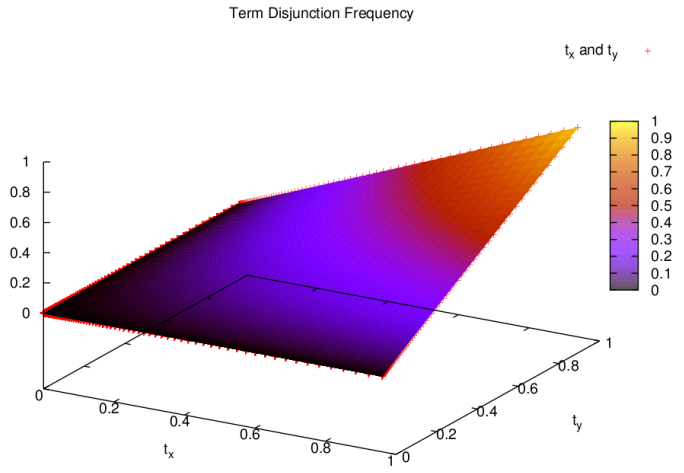

Figure 5.7: Joint frequency of two terms using the AND model obtained with the Join Method 1

The results from the diagrams can be explained by two different cases. In the first case one of the frequencies is significantly higher or lower than another, while in the other case the frequencies are almost equal. When the frequencies are quite different, the AND model would reduce the resulting frequency to be a fraction of the lowest one, while the OR model would make it to be larger than the highest one, reducing the effect by an inverse of the frequency. But when the frequencies are mostly equal, the resulting frequency would be squared for the AND model, while it would be higher for the OR model (again, much higher when both of the frequencies are low, and only slightly higher when the both of them are high).

A practical example is as follows. Suppose the word 'katy' occurs in 8664 out of

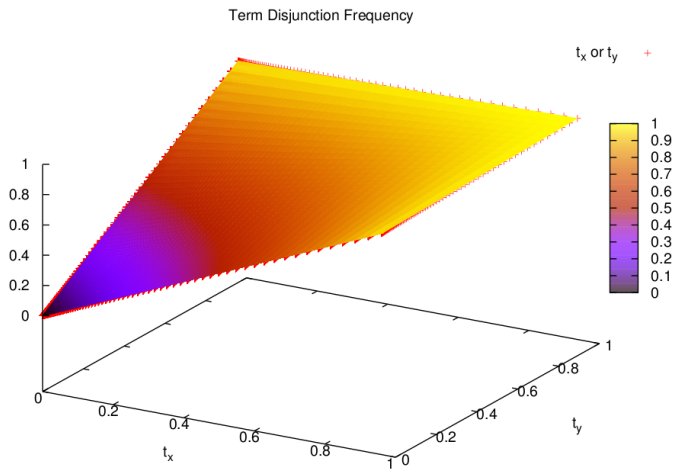

Figure 5.8: Joint frequency of two terms using the OR model obtained with the Join Method 1

25205179 documents and the word 'augustus' occurs in 4905 documents. The corresponding document frequencies of these words are 3.437e-4 and 1.946e-4. Now suppose the query phrase is 'katy augustus'. The AND model would result in 1 document in the joint result set, while the OR model results in 12376 documents. To compare this result to a real world model, the word katy results in 18 800 000 documents if searched at Google, and the word augustus occurs in 38 500 000 documents. As a result, both of the words occur in 96 800, but either or both occur in 53 500 000 documents.

However, the problem with the both of the the models is that all of the terms are assumed to be statistically independent, while this is not true for a real document collection. Suppose Katy Augustus was a rock star, the search on the phrase 'Katy Augustus' would result in a dozen of millions of results, and probably just as many as for 'Katy Or Augustus'. Another consequence here is that a query consisting of a high number of medium frequent terms would probably result in 0 common occurrences using the AND model. If the co-occurrences were taken into the calculation, the resulting number of the documents in this case would be higher.

Finally, for the discussion of the Join Method 1 and Join Method 2. It took several hours to generate a look-up table for an OR-join of two terms for a document collection encounting 25 millions documents. However, the resulting distributions differs only slightly from the distribution obtained with the Join Method 1, so the Join Method 1 is almost as good as Join Method 2. Figure 5.9 illustrates the results for the OR-model and Join Method 2.

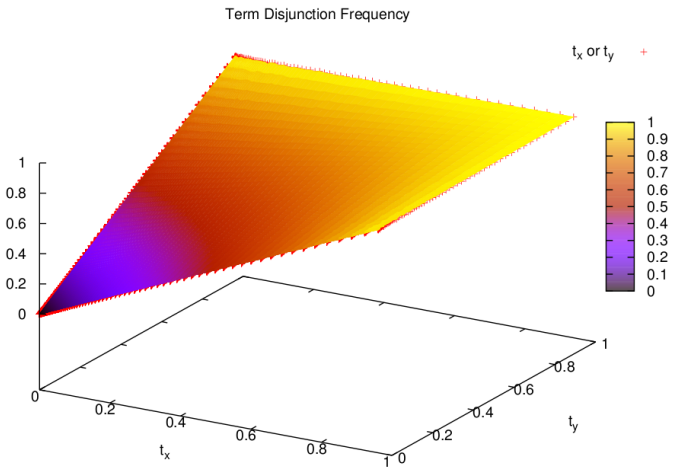

Figure 5.9: Joint frequency of two terms using the OR model obtained with the Join Method 2

Chapter 6

Simulation Experiments and Results

The main idea for this part was to perform a number of experiments similar to those performed for the previous version of the simulation model. However the simulation model has too many variables that can be varied for each single experiment. For example, for each single combination of the node performance related characteristics, both the number of nodes and the number of concurrent queries in the system can be varied. For this reason a good idea is to perform a single experiment that compares the performance at different concurrency levels for a number of different cluster sizes.

In addition, the simulation model provides a sequential and a parallel merge version of each single algorithm. Therefore it can be of a practical interest to compare those two in a single experiment and then to use only one of them in the later experiments.

The simulation model provides a quite different performance behavior for conjunctive and disjunctive queries. Therefore it can be useful to test both types in the later tests.

Finally, it can be interesting to observe the changes in the system performance when different CPU, network a disk characteristics are used.

This chapter starts with a specification for the system performance evaluation in Section 6.1. The experiment plan will be given in Section 6.2 and the experiment schedule in Section 6.3. The experiment results will be finally presented and discussed in Section 6.4.

6.1 Specifications for the Performance Evaluation

The most important system performance metrics are the average number of queries per second (QPS) and the average query response time. These two metrics are also correlated. To begin with, a higher number of concurrent queries results in a higher QPS, but also in a longer average response time for a single query. On the other hand, since the average response time does not evaluate the unfinished queries, a slightly better system performance may show a lower average response time, just because the system can process a higher number of longer queries.

Next, it can be interesting to evaluate the system load and the imbalance. The system load alone cannot be used, since if some of the resources are overloaded while all the others are idle, the average load is moderate, but the total system performance is very low.

Finally, to get a more complete picture of the system performance and its behavior, it can be a good idea to look at both the numerical results for long experiment runs and the graphical process and node status charts for short runs.

6.2 The Plan

The final experiment plan to perform a number of experiments divided into the following groups:

- **Baseline Experiments** - a small number of experiments aimed to compare different indexing schemes and processing implementations (sequential vs parallel merge).
- **Node Number and Concurrency Level Experiments** - a large number of experiments aimed to compare the system performance for a varied number of queries
- **CPU configuration experiments** - a number of experiments aimed to compare the system performance using a varied number of CPUs and different CPU factors.
- **Network configuration experiments** - a number of experiments aimed to compare the system performance using a varied network bandwidth.
- **Disk configuration experiments** - a number of experiments aimed to compare the system performance using a varied number of disks and different disk characteristics.
- **Additional experiments** : HD - a number of experiments with generated data aimed to compare the performance of the Hybrid Indexing to the other methods for the OR model.

6.3 The Schedule

All of the experiments besides the node number and the concurrency level experiments were be performed using 4 nodes and up to 12 concurrent queries. For each of these experiments it was provided a corresponding benchmark run for 50000 milliseconds and a trace run for 5000 milliseconds.

To organize the simulation process a directory hierarchy is created. In this hierarchy each subdirectory name describes either an experiment class (cpu, cpu_demo, disk, disk_demo, etc), a model type (and, or) or an indexing scheme (LI, GI, PL, etc). Each leaf directory in the hierarchy contains a number of property files with a name describing

their special features (PL_2CPU for an experiment aimed to test the pipelined indexing for a system with 2 CPUs, or GI_CPUd100 for an experiment aimed to test the global indexing for a system with a 100 times slower CPU).

There are many reasons to maintain a configuration hierarchy. First, it provides a great structure and an easy and quick way to find the required results (experiment type → query mode → indexing scheme → test name). Second, it is very easy to add new experiments by making a copy of some part of the hierarchy, then to modify some of the configuration parameters and rename the files. A modification of a system parameter in all of the configuration files in a sub-directory can be easily performed by the following one-line program:

```
find . -name *.property | xargs perl -pi -w -e 's/METRIC_NAME = oldval/METRIC_NAME
    = newval/g;'
```

The final version of the simulation model encounters more than 332 experiment configuration files, running for more than 14 hours. To automate the execution a simple BASH script is used to crawl through the hierarchy, run of all the experiments and store the results in the directory containing the corresponding property file:

```
#!/bin/bash
export run='java -Xmx800m -Xms200m -classpath /home/simonj/workspace/Sim2/bin:/
    home/simonj/desmoj_2.1.1.jar simulation.model.ModelS'
for i in $(find . \( ! -regex '.*/\..*' \) -type d); do
    cd $i;
    for j in $(ls | grep .property | sed -e 's/.property//g' ); do
        $run $j
    done;
    cd -;
done;
```

Another advantage of this approach is that in the previous simulation model all of the experiments were executed from the Eclipse by modifying the source code, recompiling and scheduling the resulting code to execution. This approach resulted in a quite different system behavior in the most recent experiments. The approach used this time avoids to recompile the source code.

Finally, all of the experiments were performed using Java(TM) SE Runtime Environment (build 1.6.0_03-b05), Java HotSpot(TM) Client VM (build 1.6.0_03-b05, mixed mode, sharing) on a Dual Pentium IV 3.2 Ghz /w 1GB RAM, running Ubuntu Linux 7.10 (i686, GNU/Linux 2.6.22-14-generic).

The experiment configuration hierarchy and its results are provided along with the digital version for this master thesis (either as a compressed archive on the Internet or as a CD-ROM).

6.4 Experiment Results

This section provides the experiment results and the result discussion. All of the performance metrics is and data is taken from the log-files and manually post-processed by the author. All of the source files for plots and the Gnuplot scripts used can be found along with the digital version of this master thesis. Only a small number of the trace charts are

provided in the appendix, while all of them can be found along with the digital version. Finally, not all of the experiment results will be discussed, but only the most important ones.

6.4.1 Baseline Experiments

GI, LI and PL - Impact of the query processing model

Table 6.1 shows the general results for the local, global and pipelined indexing on a simulated system using a default configuration with 4 nodes and th maximum number of concurrent queries set to 12. Each simulated experiment lasted for 50000 milliseconds, with no warm-up queries. The observed query length was between 1 and 8 terms, 2.7-2.8 terms in average.

	LI or	GI or	PL or	LI and	GI and	PL and
QPS	4.56605	2.98764	3.64826	4.62639	3.68944	4.73072
Avg. query response time	2589.2	3102.4	3117.1	2542.9	2987.7	2452.7
Avg. CPU load	0.26709	0.28589	0.26113	0.16420	0.19519	0.16741
Tot. avg. CPU imb.	1.00000	1.26076	1.23165	1.00055	1.36631	1.17353
Avg. disk load	1.00000	0.55546	0.66923	1.00000	0.66284	0.82174
Tot. avg. disk imb.	1.00000	1.35075	1.18793	1.00000	1.32525	1.15859
Avg. eth. load	0.00151	0.39689	0.36311	0.00134	0.18873	0.00597
Tot. avg. eth. imb.	1.00000	1.09310	1.22522	1.00746	1.36793	1.17923
Avg. mem. load	0.17067	0.41655	0.28384	0.16675	0.46903	0.27931
Tot. avg. mem. imb.	1.00205	1.21476	1.19955	1.00222	1.26860	1.23515
Tot. disk seek	5060	1283	1561	5084	1568	1977

Table 6.1: Results summary of the baseline experiments (50000ms, 4 nodes, MNQ 12)

The OR model From the results, when the OR model is used, both the highest QPS rate and the lowest average query response time are provided by the LI. The PL provides only a slightly higher QPS rate than the GI, but also a slightly longer average response time.

The CPU load and the CPU imbalance are highest for the GI. Remarkably, the LI has no disk, Ethernet or CPU imbalances, and only a low memory imbalance, while the resource imbalances for the GI are between 21% and 35%. Most notable that the Ethernet load is 260 times higher for the GI than for the LI. This may explain the differences in the QPS rate and the average query wait time. It also confirms all the suspicions about the imbalance problems resulting in a low performance in a system using the global indexing. Keeping in mind that the tested system has no cache simulation, the imbalance in a real system would be much higher.

However, for the local indexing the disk is utilized by 100% already with a concurrency level at 12 queries. The reason for this is, just as it was expected, the total number of the disk seeks performed by the system. This confirms the previous statements about a disk access bottleneck in a system using the local indexing. Even if the global indexing shows the best performance for the OR model during this experiment, it is a question if it will be the superior approach on a system with a greater concurrency level.

For the OR model, the pipelined indexing has a slightly better CPU, disk and memory load balancing than the global indexing, and as a result it solves a higher number of queries. But the consequence of this is a much higher network load imbalance. A high network load and imbalance explains why the pipelined indexing performs worse than the local indexing during the experiments for the OR model.

The AND model The interesting part of the results is what happens when the system uses the AND model instead of the OR model. The LI performs only slightly better, 1% improvement in the QPS rate and 2% improvement in the average query response time. The CPU load drops by 40%, but the disk load is still a problem. The GI improves its QPS rate by whole 23% and reduces the average query response time by about 4%. There is also a decrease in the average CPU load, but it results in a 10% higher load imbalance as a consequence. Notable that the Ethernet load is decreased by 50%, but the imbalance is increased by 25%.

The pipelined indexing provides some remarkable results for the AND model - its QPS is increased by 29% and the average query response time is decreased by 32%. The CPU load is reduced just as with the other indexing schemes, but since it does more work, the disk utilization is increased by 23%, resulting actually in a lower disk imbalance than when using the OR model. Finally, the Ethernet load is decreased by 98% (ninety eight percent). As a result, the difference in the Ethernet load between the LI and the PL is only three times in favor of the LI, but the disk load using the PL is lower than using the LI.

The conclusion from the results is that, because of a great decrease in the Ethernet load and the data volumes in the later processing stages, the pipelined indexing provides the highest utilization of the resources which results in a better system performance in terms of the highest QPS rate and the lowest average query response time.

Since none of the resources are critical for the pipelined indexing when the AND model is used, an interesting question is what happens with a higher concurrency level. From the current situation, because of the disk load at 100%, the performance of the LI cannot be expected to be improved by using a higher concurrency level, while the performance of the PL and the GI can be at least slightly increased by increasing the system load. From the experiment results, there is also an observation that a higher average resource load results normally in a lower load imbalance [1].

[1] otherwise it can be stated that a higher resource imbalance results in a lower average resource load, which is also true from the experiment results

Trace Mode All the information presented above and in the standard reports is very interesting, but it does not explain what does actually happens in the system when the different partitioning methods and processing algorithms are used. For this reason Appendix 1.4 provides a number of process diagrams and node state diagrams for all of the algorithms described.

The process diagram for the global indexing using the OR model shows a number of long periods where some of the processes wait for the network access over a long period of time, more than 1000ms. However, this problem could be solved if the network implementation in the simulation model could provide a concurrent access, just as the cpu implementation for a single node. There is also a period where a number of processes wait for memory over a thousand of milliseconds. Because of the CPU implementation, there are no visible CPU wait times, but all of the processes running on the same node share the CPU load. Anyway, the diagram shows that 70% of total query times are the disk wait times and the process report shows that the average disk wait time on a single node is as high as 330.19152ms for a single disk access.

The process diagram for the local indexing shows more disk wait times induced by a high number of disk seeks, but no Ethernet wait times or memory problems. The sub-query processing times are much shorter due to shorter disk access and post-processing times.

The pipelined indexing, from the process diagram, has much shorter disk wait times in average, but many long disk, cpu and network accesses resulting sometimes in long network and disk wait periods. This could again be improved if the simulation model could provide concurrent disk/network access. But the main point here is that, because of a smaller number of simulated processes, the pipelined indexing seems to use the system resources in a more effective way than the global or the local indexing alone.

The only difference between the processing with the OR and the AND models for the local indexing is shorter processing times and as a result shorter periods with a high CPU load. For the global indexing, on the other hand, the difference is in both the reduced memory, network and CPU loads. The memory and the network wait times named above are eliminated when the AND model is used. Finally, for the pipelined indexing the is a significant decrease in both the Ethernet transfer volume and the post-processing times, which explains the significant performance improvement observed.

Serial vs. Parallel Merge A number of the experiments in the trace mode show that the Parallel Merge approach for processing of the inverted lists and accumulators does not improve the system performance. The results presented in Table 6.2 show a decrease in the QPS rate by 5% in average and a higher average query response time, especially for the global indexing using the OR model. The reason for this is a more bursted CPU load.

The explanation for these results is as follows. In the standard version of the sub-query processing algorithms, all the disk accesses are followed by a single CPU access to merge the inverted list with the partial results achieved so far, and a single CPU access to post-process the partial results at the end. But in the modified version all the algorithms

	LI or	GI or	PL or	LI and	GI and	PL and
QPS change	-11%	-11%	-5%	-11%	-5%	-0%
Wait time change	+0%	+36%	+5%	+0%	+3%	+8%

Table 6.2: Results summary of the baseline experiments (50000ms, 4 nodes, MNQ 12)

require a number of disk accesses then a long CPU access to process the index data. The outcome is that resource load over the time is less balanced and it requires longer wait times in the second case, therefore the total performance is also reduced.

This observations support also the prognoses about a potential improvement in the load balancing and the total system performance that can be achieved by providing a concurrent disk and network access.

6.4.2 Node Number and Concurrency Level Experiments

Because of the trade-off between the number of concurrent queries, the resulting QPS rate and query wait times, a large number of experiments were performed with the only purpose to see what happens when the maximum number of concurrently executing queries increases. Figures 6.1 - 6.4 show the results for a change from 2 to 24 queries for a system using 4 nodes, and from 2 to 34 for a system with 8 nodes. These charts provide also information about the scalability of the system when the number of nodes is increased.

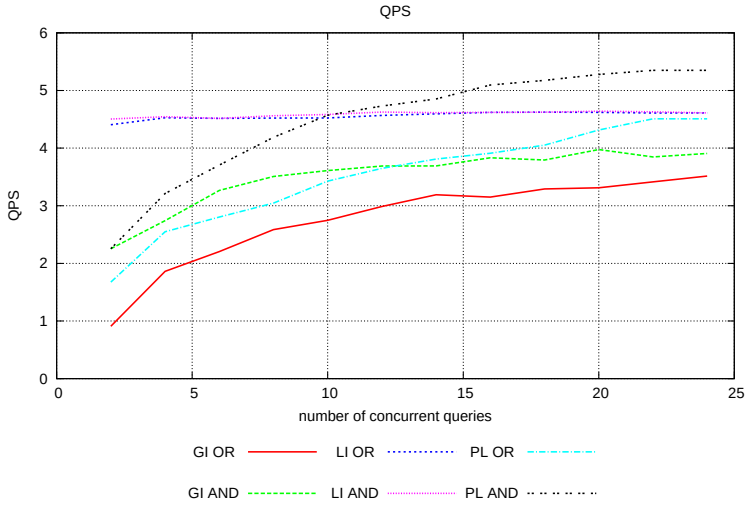

Figure 6.1: The average QPS with a varied concurrency level using 4 nodes

The results show that the local indexing provides the worst scalability since there is no notable gain in the QPS rate when the number of queries is increased, but the average query response time increases linearly. The same situation is for both of the systems and it is notable that by using twice as many nodes the gain in the QPS rate is about 44% and there is almost no change in the average query response time.

The the results look very interesting for as the concurrency level gets higher. At a level higher than 10 queries the pipelined indexing outperforms the local indexing in both the QPS rate and the average query wait time when the AND model is used. On a system with 4 nodes the pipelined indexing is almost as good as the local indexing when the OR model is used, and for the same model on a system with 8 nodes the pipelined indexing outperforms the local indexing when the concurrency level is greater than 22 queries. This looks very exciting as it shows that, because of the provided scalability, the pipelined indexing can is the superior approach for both the AND and the OR model if

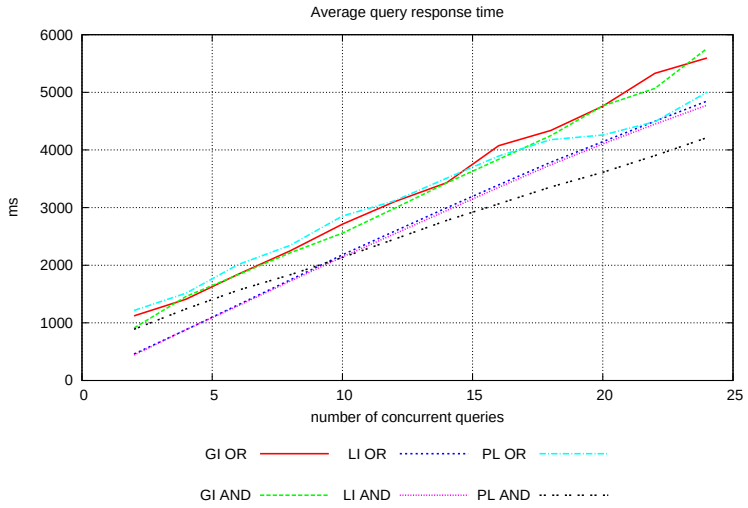

Figure 6.2: The average query response time with a varied concurrency level using 4 nodes

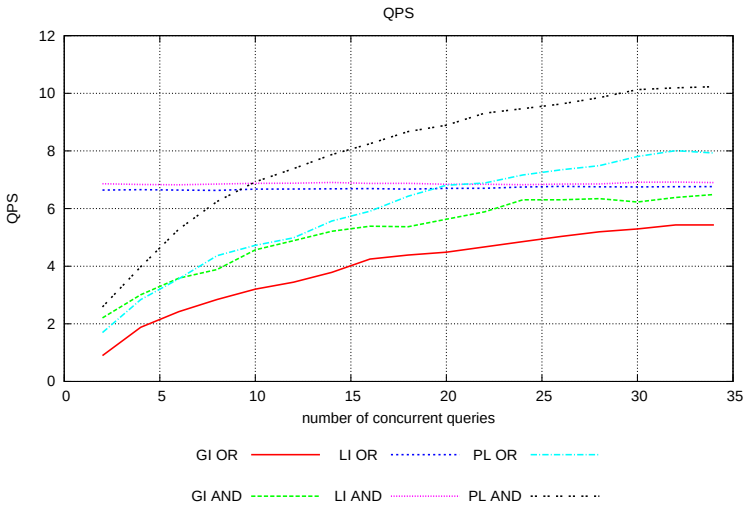

Figure 6.3: The average QPS with a varied concurrency level using 8 nodes

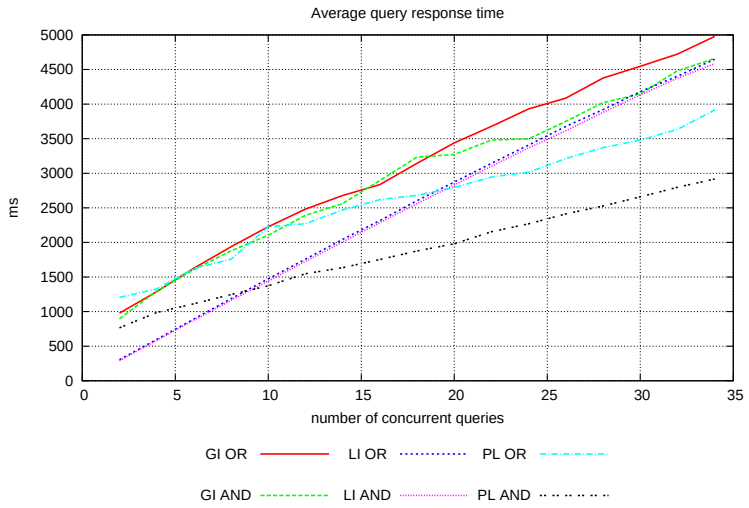

Figure 6.4: The average query response time with a varied concurrency level using 8 nodes

the number of nodes and the concurrency level are high enough. In addition, the QPS rate seems to increase logarithmically when the query wait times increases linearly, but with a smaller constant factor than for the local indexing.

6.4.3 CPU configuration experiments

There are two types of the CPU experiments that were performed. The first type experiments with the number of CPUs per node, and the second type experiments with the CPU factor, if it is assumed that a CPU with a factor f can perform all of the CPU operations f times faster.

Number of CPUs

The experiments with the number of CPUs show that an increase in the number of CPUs results in a corresponding decrease in the CPU load, but no additional gain in form of a higher QPS rate or a shorter average respons time. The reason for this is that the CPU is not a critical resource, but if it would be, there would be a notable improvement.

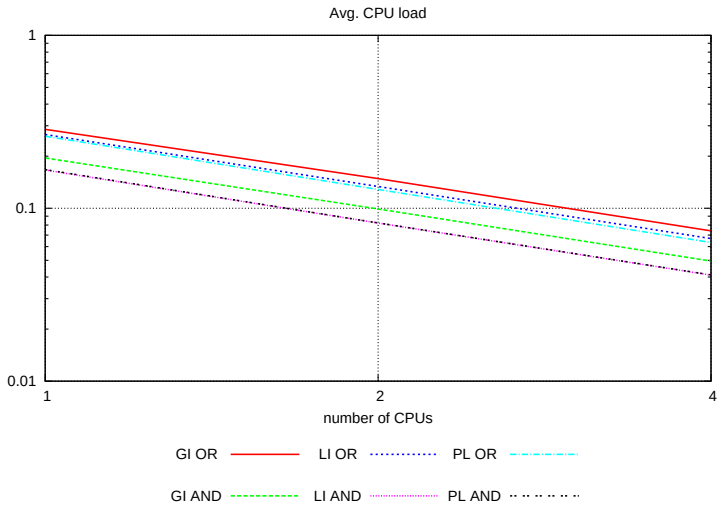

Figure 6.5: The average CPU load with a varied number of CPUs

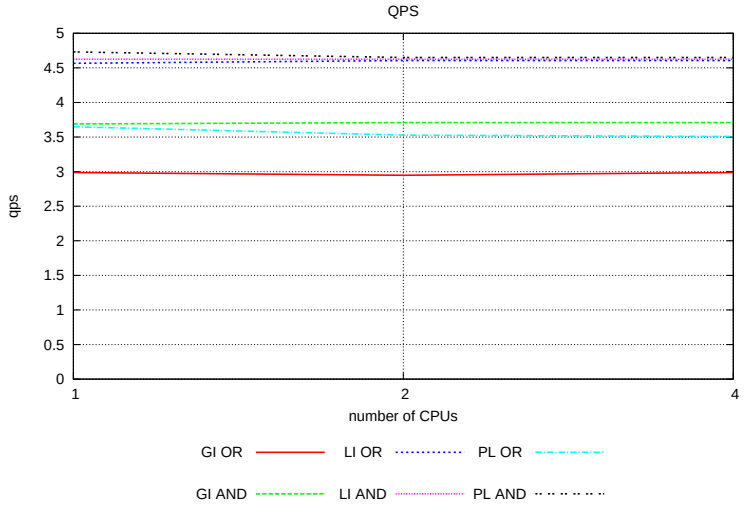

Figure 6.6: The average QPS rate with a varied number of CPUs

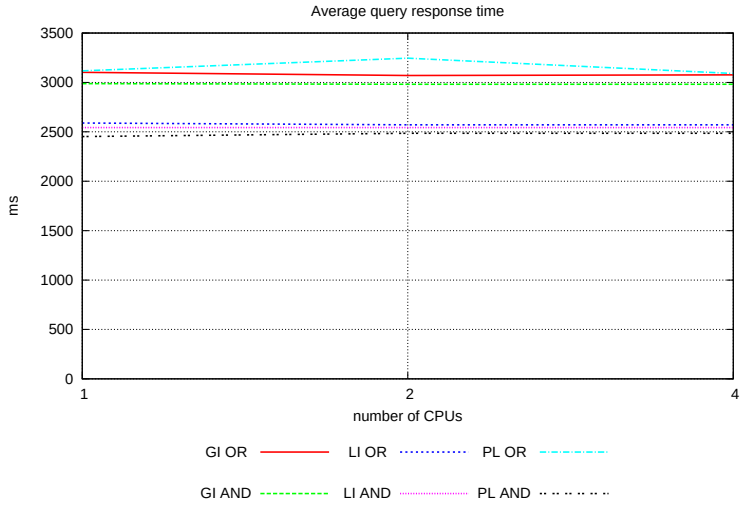

Figure 6.7: The average query response time with a varied number of CPUs

CPU factor

The results from the experiments with a varied CPU factor are very interesting as they show that for a ten times faster CPU there is almost no difference in the QPS rate and the average query response time between the local and the pipelined indexing. But for a 100 times slower CPU, the local indexing may offer a slightly higher QPS rate, while the pipelined indexing will provide a shorter average query response time.

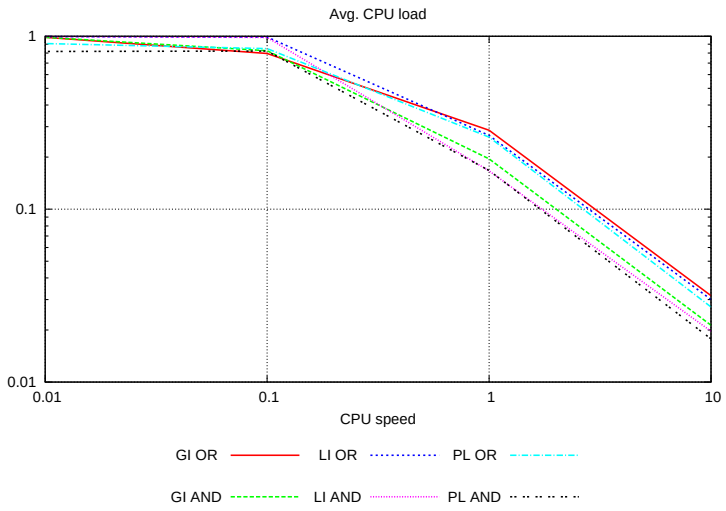

Figure 6.8: The average CPU load with a varied CPU factor

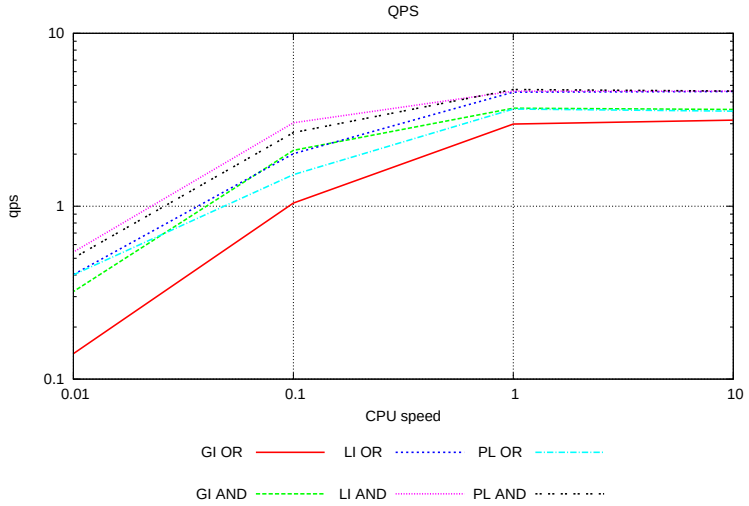

Figure 6.9: The average QPS with a varied CPU factor

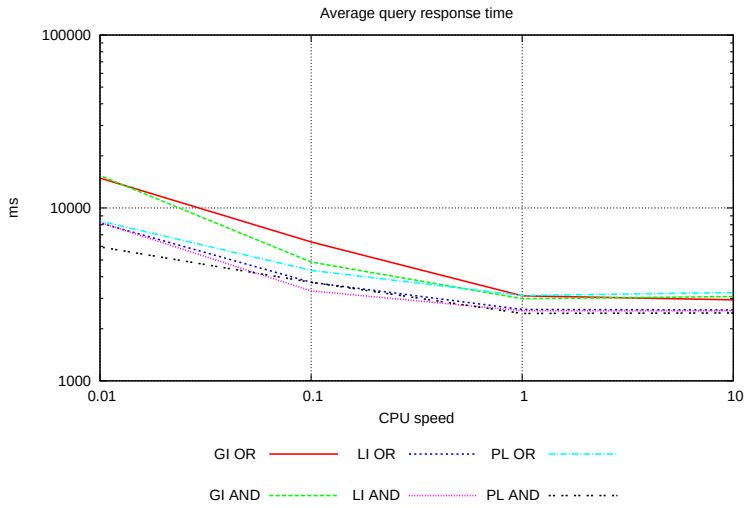

Figure 6.10: The average query response time with a varied CPU factor

6.4.4 Network configuration experiments - Bandwidth

The network experiments performed were aimed to demonstrate what happens when the network bandwidth is changed. The results in Figures 6.11-6.13 show that the local indexing is not affected by a decrease in the bandwidth, even if the decrease is 100 times. Both the QPS rate and the average query response time for the local indexing remain constant, and the Ethernet load increases just slightly (it can be suggested a logarithmic increase in the Ethernet load when a linear decrease in the bandwidth is applied).

The pipelined indexing using the AND model provides a slightly lower QPS rate and a slightly longer average query response time, both are by about three times when the network bandwidth is reduced from 0.1Gbps to 0.01Gbps.

One interesting detail here is that the average query response time for the pipelined indexing is much shorter when the network bandwidth is decreased by a factor of 100. It can be very misleading, but a true reason for this is that the system can perform only a few short queries while all the longer queries are timed out and therefore are not taken into the calculation. A closer look at the experiment report shows that on two of the nodes the Ethernet load is 99%, so all the queries involved in those two tests are probably timed out.

The final conclusion from this is that the pipelined indexing using the OR-model is the worst one when a very slow network is used, and the local indexing is the best one for both the AND and the OR models.

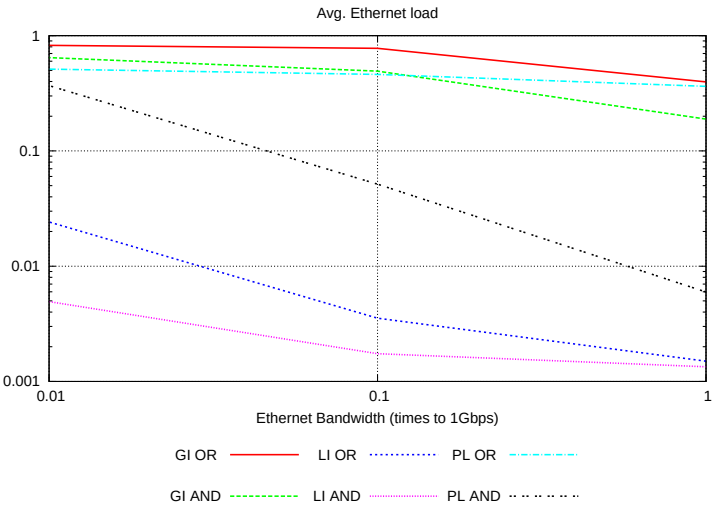

Figure 6.11: The average Ethernet load with a varied network bandwidth

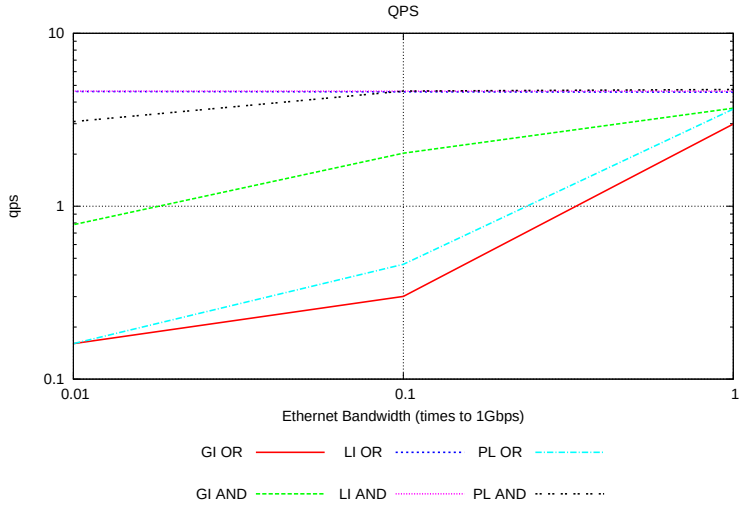

Figure 6.12: The average QPS rate with a varied network bandwidth

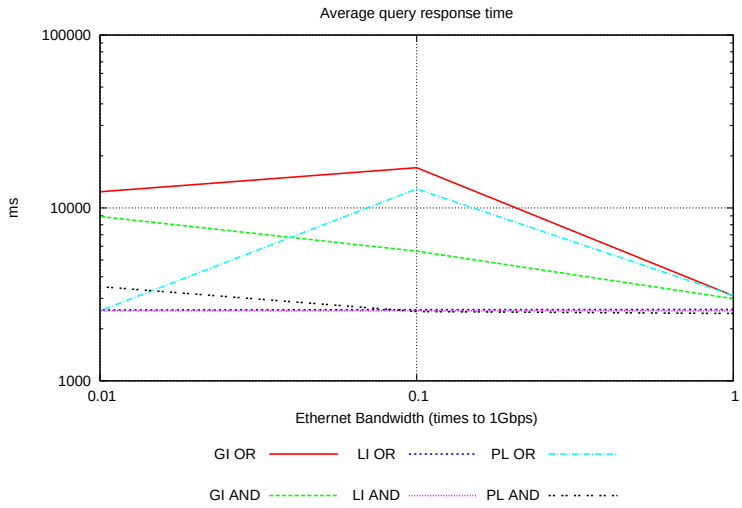

Figure 6.13: The average query response time with a varied network bandwidth

6.4.5 Disk configuration experiments

There are two kinds of the disk experiments that have been performed. The first one supposed to show what happens when the number of disks on a single node increases, in which case more disk accesses can be performed simultaneously. The second one, supposed to show how a better disk seek time can improve the system performance.

Number of Disks

The results for the experiments with the number of disks are presented in Figures 6.14-6.17. As it was expected, the disk seek time is the most critical issue for the performance of the local indexing. The results show that the QPS rate increases linearly when the number of disks increases, and the local indexing with 4 disks per node provides a QPS rate as high as 18.8 queries per second.

As it was also expected, the performance of the global indexing is least influenced by the increasing number of disks. Surprisingly, the performance of the pipelined indexing improves when the number of disks is increased. A possible explanation for this is that the disk access on a single node may become a bottleneck when a number of query bundles are routed through this node. So a higher number of disks allows a higher number of queries 'route' through a single node and proceed on the other nodes.

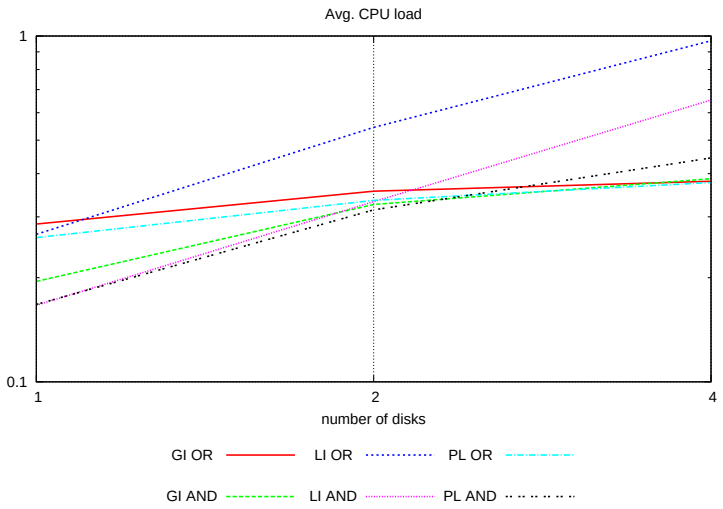

Figure 6.14: The average CPU load with a varied number of disks

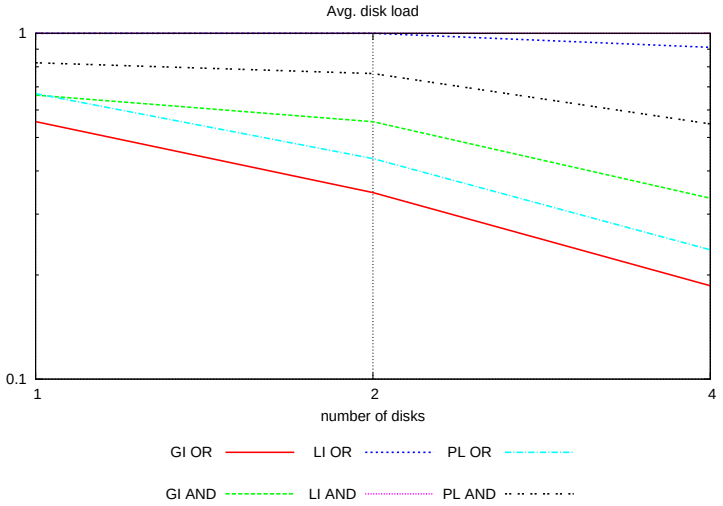

Figure 6.15: The average disk load with a varied number of disks

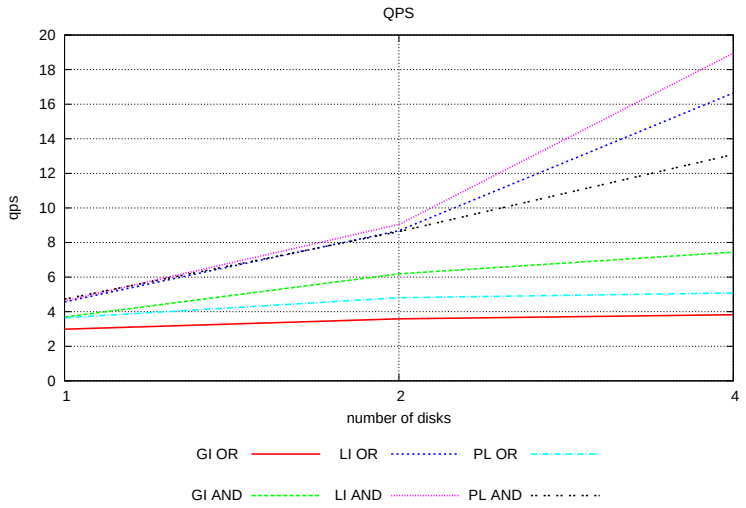

Figure 6.16: The average QPS with a varied number of disks

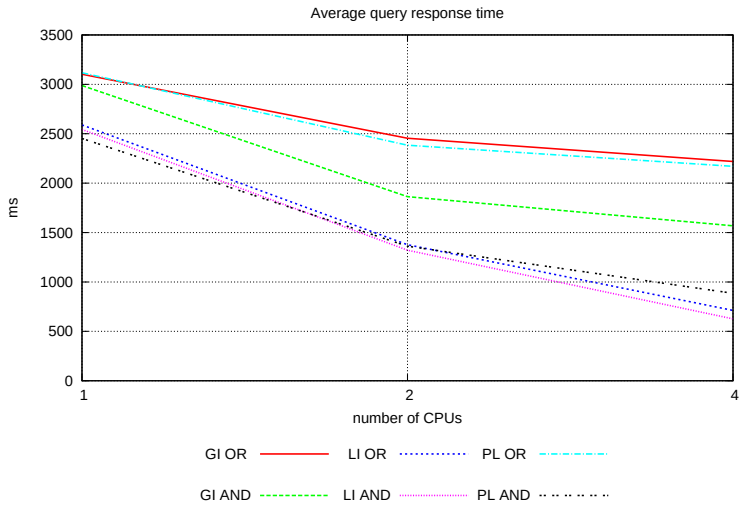

Figure 6.17: The average query response time with a varied number of disks

Disks Characteristics

The results for the experiments with the disk seek times are very surprising. As Figures 6.18-6.21 show, a high-end disk improves the system performance, but a flash disk provides actually a lower performance than the original one. The reason for this is that the high-end disk provides the same bandwidth and a shorter seek-time and rotation delay, while the flash disk provides a better access time, but a 26% lower bandwidth.

Under the experiments simulating a flash disk the disk seek time was actually defined as 0.03ms, when 0.3ms was the suggested value for the disk random access time. But even a ten times shorter disk access time had no remarkable positive effect.

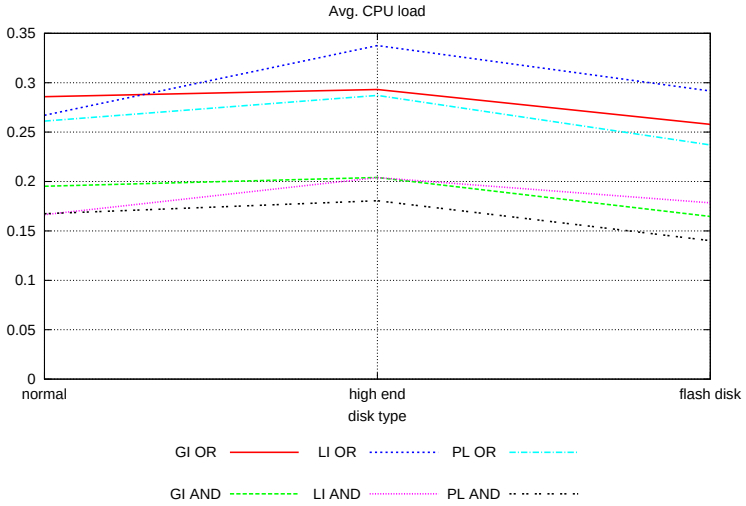

Figure 6.18: The average CPU load with a varied disk type

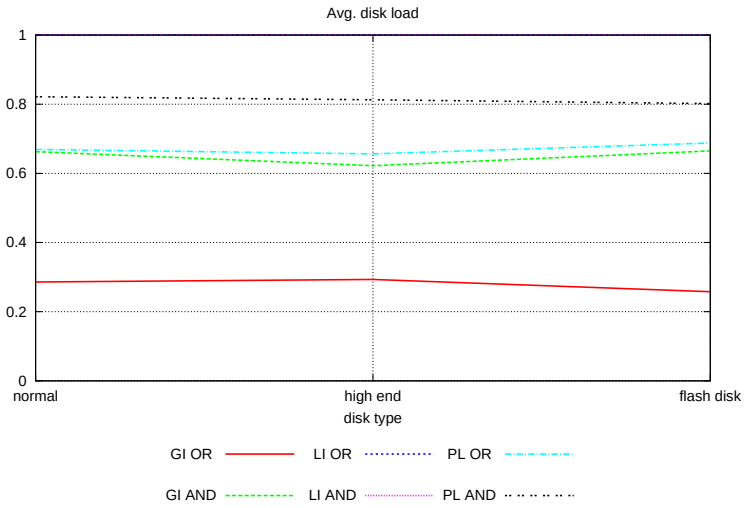

Figure 6.19: The average disk load with a varied disk type

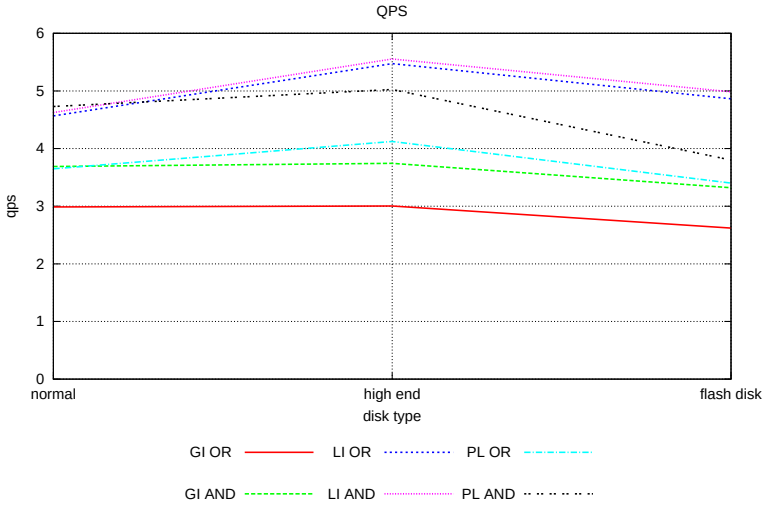

Figure 6.20: The average QPS rate with a varied disk type

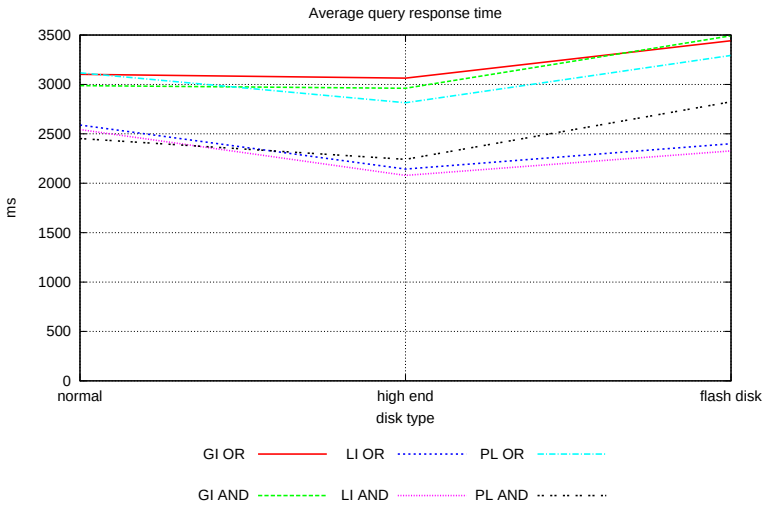

Figure 6.21: The average query response time with a varied disk type

6.4.6 Additional experiments

To validate the performance of the hybrid indexing a number of experiments with generated inverted lists were performed. Because of the limited main memory capacity, the document collection size in these experiments was only 250000, 100 times smaller than in the original experiments. From the experiment results, the hybrid indexing has a three times lower QPS rate and two times longer average query response time than the global indexing. Performance of the pipelined indexing is better than the global indexing, while none of the tests for the local indexing have completed.

6.4.7 Combination of the Experiment Results with the Previous Results

An interesting question is how do this results compare to the previously achieved results. First of all, all of the statements about all of the pros and cons of the indexing methods seem to be true. Second, rthe esults from the current simulation model disagree with the results obtained from the previous simulation model. However, the current model improves all of the weaknesses and errors done with the previous model. Therefore is the difference.

Third, a superior performance of the local indexing was appointed by 4 out of 7 studied papers. The most important of these are the most recent papers by Moffat et al., [AdKM01] and [MWZ06]. From the results in these papers, the pipelined indexing provides a better performance than the global indexing, but not a good as the local indexing may provide.

The difference in the conclusions about the performance can be explained as follows. The simulation model expects that for all of the terms the occurrence of a single term in a single document has no impact for the occurrences of the other terms in the same document. So the number of the final results (hits) using the conjunctive model is much smaller than it would be in the real world. If this number would be greater, the performance of the pipelined indexing would be lower. But the local indexing seems to have only a small difference in the performance for the conjunctive query model compared to the disjunctive query model.

The search engine used in [MWZ06] and [MWZB07], on the other hand, is a real implementation which provides a much greater number of the final results. At the same time, the search engine may use filtering techniques to improve the algorithm performance. Finally, a real search engine may also get an performance improvement by using an index/result cache, but the system load imbalance would be also much greater in this case.

Chapter 7

Conclusions and Further Work

This master thesis has presented a number of different approaches to perform query processing on an inverted index distributed over a number of nodes. The fundamental part of the master thesis has presented and analyzed the most important papers in this field published over the last twenty years. In addition, a simulation model was created to compare the search performance of the different distribution schemes. A dictionary for a real document collection, TREC GOV2, and a real query set, Terabyte Track 05, were used to perform the simulation experiments.

The results from the experiments with the simulation model show that the local indexing provides a better performance than the global indexing because of a better load balancing and shorter disk access, network access, network wait, memory wait and post-processing times. But the price for this is a very high disk load resulting in longer disk wait times. The disk load is actually the bottleneck for the local indexing, and by improving this part by either providing more disks or by improving the disk characteristics, the performance advantage of the local indexing can be even greater. The load imbalance is the most important issue with the global indexing, even when the observed average disk load is significantly lower using the global indexing than using the local indexing, the difference between the average disk load on two differen nodes in the same system can be as high as 8 times.

However, the most interesting results are provided by the pipelined indexing. The pipelined indexing can be viewed as an improvement technique for the global indexing, and its performance on a disjunctive query model is only slightly better than the performance of the global indexing. The improvement is a result of a slightly lower system load and a better load balancing. But for the conjunctive query model, the pipelined indexing provides an even better performance than the local indexing may provide. The main reason for this is a great decrease in the data volume in the later pipeline stages, resulting in a significant decrease in the network load and a better utilization of the other resources.

The advantage of the pipelined indexing is even greater for a higher number of nodes and a higher concurrency level. On a system with 8 nodes and a concurrency level greater than 22 queries the pipelined indexing is the superior approach in terms of the QPS rate

for both the AND and the OR query models, while the local indexing does not seems to scale at all.

On the other hand, both the experiments with a 100 times slower CPU and the experiments with a 100 times slower network make the local indexing to be the method of choice.

The experiments with a higher number of disks show a linear increase in the QPS rate for the local indexing when the number of disks increases. There is also a significant improvement for the local indexing when a simulated high end disk is used, but only a small improvement for a simulated flash disk with a insignificantly short disk access time (0.03ms). The conclusion from this is that the number of concurrent disk accesses and the combination of the disk characteristics, but not the disk seek time on its own, are the most critical issues for the local indexing.

Finally, the simulation results do agree with the previous study and all the statement about the performance issues for the different inverted indexing methods. However, the obtained results do not agree with the previous results by 100%. But the difference in the final results can be explained by a smaller number of the query results, no approximation methods or filtering techniques applied and finally no simulation of the index/result cache.

7.1 Further Work

The simulation model, the algorithms described and the final performance can be slightly improved by providing a concurrent network access, where a number of transfers to or from the same node can be performed simultaneously. The performance of the pipelined indexing will be probably much better in this case.

Next, it could be interesting to implement the original version of the parallel merge approach described earlier, the one aimed to perform a higher number of disk accesses to reduce the memory consumption. Here it could be also interesting to see if a concurrent disk access can improve the final performance.

Finally, there was a problem with the estimation of the average query response time. The problem was that none of the timed out queries were include into the calculation of the query response time. This problem could be solved by running a large number of warm-up queries over a long-period, and then running a long period measuring the response times for the queries that terminates during its time bounds. If the performance is stable, the characteristics of the queries started outside and finished inside the time bounds and the queries started inside and finished outside suppose to be similar. Therefore the resulting value would be a better estimate.

7.2 Interesting Topics Related to this Master Thesis

The use of filtering techniques was partly discussed in the background part but never implemented for the simulation model. The reason for this is that the approximation

algorithms described in the background chapter require a data structure that would provide a fast look-up operation to insert or update the accumulator values and a quick sort based either on the accumulator score or on the accumulator id. Otherwise it could be interesting to see if there are any alternative ways to perform an approximated query processing for the global indexing.

Another important issue named but not tested by the simulation model is the construction of the index itself. Even if the global indexing or its variations would provide a better performance than the local indexing, it is much more difficult to generate a distributed global index than a local index. An even more difficult task is to perform updates on a global index. Therefore it is an important argument in the discussion about whether the global or the local indexing is a better alternative.

A third issue closely related to the first two, and also discussed in this master thesis, is the term mapping used for the global indexing. As it was suggested, neither a lexicographical or a rank based mapping would ever provide a perfect load balancing, since some of the terms having similar rank may be more popular in the query set than the other ones, resulting in a lower load balancing. The alternatives proposed in [MWZ06] was to redistribute the terms after the end of each batch and to replicate the most work demanding terms. Another method proposed but not tested in this thesis was to move the most work demanding term to the other nodes in real-time. Of course, the last method would require a higher synchronization level and result in a higher network load, but the question is if it would result in a workload decrease that can compensate the work performed.

Bibliography

[AdKM01] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 35–42, New York, NY, USA, 2001. ACM.

[BBG+05] Claudine Badue, Ramurti Barbosa, Paulo Golgher, Berthier Ribeiro-Neto, and Nivio Ziviani. Basic issues on the processing of web queries. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 577–578, New York, NY, USA, 2005. ACM Press.

[BBR+07] C. S. Badue, R. Baeza-Yates, B. Ribeiro-Neto, A. Ziviani, and N. Ziviani. Analyzing imbalance among homogeneous index servers in a web search system. *Inf. Process. Manage.*, 43(3):592–608, 2007.

[BBRZ01] Claudine Santos Badue, Ricardo A. Baeza-Yates, Berthier A. Ribeiro-Neto, and Nivio Ziviani. Distributed query processing using partitioned inverted files. In *SPIRE*, pages 10–20, 2001.

[Bjø07] Truls A. Bjørklund. Experimentation with inverted indexes for dynamic document collections, 2007.

[BR99] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[CP97] Douglas R. Cutting and Jan O. Pedersen. Space optimizations for total ranking. In *RAIO Proceedings*, 1997.

[JO95] Byeong-Soo Jeong and Edward Omiecinski. Inverted file partitioning schemes in multiple disk systems. *IEEE Trans. Parallel Distrib. Syst.*, 6(2):142–153, 1995.

[Jon07] Simon Jonassen. Global vs. Local inverted indexes. Report in TDT4590 Complex Computer Systems, Specialization Project, December 2007.

[LLC08a] Seagate Technology LLC. Seagate Barracuda ES 2 Data Sheet. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es_2.pdf`, 2008.

[LLC08b] Seagate Technology LLC. Seagate Barracuda ES Data Sheet. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_barracuda_es.pdf`, 2008.

[LLC08c] Seagate Technology LLC. Seagate Cheetah NS Data Sheet. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_cheetah_ns.pdf`, 2008.

[LLC08d] Seagate Technology LLC. Seagate Savio 15K Data Sheet. `http://www.seagate.com/docs/pdf/datasheet/disc/ds_savio_15k.pdf`, 2008.

[LM06] Amy N. Langville and Carl D. Meyer. *Goooogle's PageRank and Beyond: The Science of Search Engine Rankings.* Princeton University Press, Princeton, NJ, USA, 2006.

[MMR00] A. MacFarlane, J. A. McCann, and S. E. Robertson. Parallel search using partitioned inverted files. In *SPIRE '00: Proceedings of the Seventh International Symposium on String Processing Information Retrieval (SPIRE'00)*, page 209, Washington, DC, USA, 2000. IEEE Computer Society.

[MWZ06] Alistair Moffat, William Webber, and Justin Zobel. Load balancing for term-distributed parallel retrieval. In *SIGIR '06: Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 348–355, New York, NY, USA, 2006. ACM Press.

[MWZB07] Alistair Moffat, William Webber, Justin Zobel, and Ricardo Baeza-Yates. A pipelined architecture for distributed text query evaluation. *Inf. Retr.*, 10(3):205–231, 2007.

[RB98] Berthier A. Ribeiro-Neto and Ramurti A. Barbosa. Query performance for tightly coupled distributed digital libraries. In *DL '98: Proceedings of the third ACM conference on Digital libraries*, pages 182–190, New York, NY, USA, 1998. ACM Press.

[Ris04] Knut Magne Risvik. *Scaling Internet Search Engines - Methods and Analysis.* PhD thesis, 2004. Chair-Edward A. Fox.

[SM86] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill, Inc., New York, NY, USA, 1986.

[Sor01] Ohm Sornil. *Parallel inverted index for large-scale, dynamic digital libraries.* PhD thesis, 2001. Chair-Edward A. Fox.

[SWJS01] Amanda Spink, Dietmar Wolfram, Major B. J. Jansen, and Tefko Saracevic. Searching the web: the public and their queries. *J. Am. Soc. Inf. Sci. Technol.*, 52(3):226–234, 2001.

[TG93] Anthony Tomasic and Hector Garcia-Molina. Query processing and inverted indices in shared nothing text document information retrieval systems. *The VLDB Journal*, 2(3):243–276, 1993.

[WM05] William Webber and Alistair Moffat. In search of reliable experiments. In *10th Australasian Document Computing Symposium*, pages 26–33. University of Syndey, 2005.

[WMB99] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[XSLF02] Wensi Xi, Ohm Sornil, Ming Luo, and Edward A. Fox. Hybrid partition inverted files: Experimental validation. In *ECDL '02: Proceedings of the 6th European Conference on Research and Advanced Technology for Digital Libraries*, pages 422–431, London, UK, 2002. Springer-Verlag.

[ZM06] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2):6, 2006.

[ZS05] Jiangong Zhang and Torsten Suel. Efficient query evaluation on large textual collections in a peer-to-peer environment. In *P2P '05: Proceedings of the Fifth IEEE International Conference on Peer-to-Peer Computing (P2P'05)*, pages 225–233, Washington, DC, USA, 2005. IEEE Computer Society.

Appendix A

Appendix

A.1 Dictionary Data

A.1.1 docstat

Here are 50 sample entries from the resolved dictionary, each line starts with a dictionary term followed by the term id and the associated number of occurrences. Non-existing terms are represented with a negative number as the term id.

```
consumtionjuction -958 0
contac 15513992 1275
contact 15514061 8088813
contacting 15514601 172339
contacts 15514975 1111228
contadora 15515311 95
container 15515887 152450
containers 15516120 126371
contempary 15519022 5
contemporary 15519321 63015
contender 15519547 1470
contenential 15519678 1
content 15519798 2483799
contential 15520087 8
contentinal 15520107 1
contessa 15520866 374
contest 15520874 81221
contestants 15520928 3624
contests 15521098 23572
contienentl -959 0
contigo 15526738 227
continenalairlines -960 0
continental 15527293 90702
continentalairlines -961 0
continential 15527422 161
continuing 15529050 566719
contos 15531309 179
contour 15531332 38040
contract 15532176 1308870
contracted 15532484 148725
contracters 15532549 19
contractors 15533244 647536
contracts 15533603 816571
```

```
34 contracture 15534037 2707
35 contractures 15534058 866
36 contradictory 15534295 15880
37 contrast 15535199 203274
38 contributions 15536907 423689
39 control 15537999 2484208
40 controled 15538479 520
41 controling 15538687 419
42 controllato 15538795 62
43 controlled 15538827 456845
44 controller 15539026 84276
45 controlling 15539237 203067
46 controls 15539693 406455
47 contromatics 15540156 5
48 contruction 15540668 1425
49 contusion 15541375 1681
50 convayor 15542781 1
```

A.1.2 querylog

Here are 50 sample entries from the query log, each line starts contains a single query. The sample is extended with the resolved frequency for each query term.

```
1  pierson s twin lakes marina [3.6853537124255294E-4 0.564495812547096
       0.00512466108651718 0.028598686008141423 0.00155579137128921 ]
2  nurseries in woodbridge new jersey [6.573252266924984E-4 0.6285635186324208
       3.1870434246866487E-4 0.3192352651016682 0.028013290443206135 ]
3  miami white pages [0.01596394137887297 0.06149037862417085 0.040436729292817164 ]
4  delta air lines [0.010095028486010751 0.05969118489497734 0.05331987525262169 ]
5  hsn [3.217592701880832E-5 ]
6  ironman ivan stewart s super off road [1.011696842145021E-5 8.9311010249123814E-4
       0.004465352140526358 0.564495812547096 0.0034602015720658044
       0.08134475061653004 0.05042459726233248 ]
7  pajaro carpintero [1.9983988211311652E-4 4.284833684378912E-6 ]
8  kitchen canister sets [0.002957051009239014 3.992830203665683E-4
       0.041297147701272026 ]
9  buy pills online [0.008666671242445849 4.7930625686094114E-4 0.09294871502400359 ]
10 hotel meistertrunk [0.005740090161629084 3.967438596647142E-8 ]
11 cingular [6.141594947609775E-5 ]
12 katy augustus [3.4373888001350836E-4 1.946028631655423E-4 ]
13 what is the sales tax in columbus ohio [0.1446839159523525 0.49777012097394746
       0.7599695284846023 0.046713970966046306 0.06045162385079669 0.6285635186324208
        0.0096314729603785 0.029383683408874026 ]
14 notes on the green mile [0.06307481490212785 0.5207393686829203 0.7599695284846023
        0.058345548746152526 0.011601980688175235 ]
15 frazier road villa rica georgia [7.089416028348777E-4 0.05042459726233248
       7.485763144153827E-4 0.0019401568225323852 0.02511721896519759 ]
16 free picnic table centerpieces crafts [0.07545675434401795 0.0015915776674309673
       0.098766527307741 2.297146947458695E-5 0.0011018370470608442 ]
17 louisiana technical school [0.02708578264808197 0.06805097476197253
       0.06806394828618356 ]
18 auctions unlimited in ri [7.447675733626013E-4 0.005182228620554529
       0.6285635186324208 0.005357033965122803 ]
19 yahoo [0.0019292860407775 72 ]
20 tyson fight [3.8333391720804 68E-4 0.004957949316686067 ]
21 land rover series 11a [0.061289348510478736 4.889471326507937E-4
       0.08000831892524946 9.419492716159643E-4 ]
22 soaringeaglecasino [0.0 ]
23 beatles [3.6302063159321346E-5 ]
24 alltheweb be [8.220532772252876E-5 0.35002127935691313 ]
```

```
25 modesto bee [0.001784117462526253 0.001911194520776861 ]
26 wild plum jelly [0.010023336870569338 7.348489768709835E-4 4.695860322991557E-4 ]
27 kaiser [0.0016067332828701593 ]
28 kingsway financial services [2.920034807132296E-5 0.04246159092938796
       0.24414280096959437 ]
29 british airways [0.0047237910907119525 7.195346638879256E-4 ]
30 chevrolet [3.7980289685703085E-4 ]
31 hud [0.004283881499115717 ]
32 rattlesnake trail head camping [6.7529772353531E-4 0.025018905836772673
       0.02428965094832296 0.020904830709593453 ]
33 youve got pictures [1.4877894737426781E-5 0.007956301361716177
       0.006302831652177515 ]
34 thirteen buddy icons [0.00215451752990923 6.128502400240839E-4
       0.0038691254682222255 ]
35 star wars [0.008968037878247166 0.0013508334933864188 ]
36 hooker furniture company [6.151513544101392E-4 0.0064313766627088825
       0.06517228066501729 ]
37 new york grandparents rights [0.3192352651016682 0.06894741751288495
       0.0012895762414541 87 0.07837540054764142 ]
38 nike id [1.4754904140930718E-4 0.052050334576080576 ]
39 real estate forms [0.03689872624987111 0.017312434083487364 0.058699087199499754 ]
40 cn8 [5.0783214037083414E-6 ]
41 er cast [0.0073626138501138 99 0.007343887539937725 ]
42 touch kirby [0.0045633875482495 09 7.014828182731811E-4 ]
43 business loans [0.12252374799639391 0.008088893159616126 ]
44 water958 [0.0 ]
45 fare tracker [0.0010462532323218176 0.0048283727721195715 ]
46 real estate zumstein ave cincinnati [0.03689872624987111 0.017312434083487364
       2.8565557895859417E-6 0.012445061389962753 0.006867675885182168 ]
47 obi won [5.697241824785295E-5 0.0077451939539885835 ]
48 model girls [0.05785922012297552 0.0025792715060662733 ]
49 ticonderoga [8.609341754724297E-5 ]
50 noir chloe [5.9233858247941825E-5 1.3326626246137747E-4 ]
```

A.2 Network Microbenchmark

A.2.1 clustis.c

```
1  /*
2   * All rights for the original code belongs to TDT4200 staff,
3   * Thorvald Natvig and Jan Christian Meyer, 2007-2008
4   *
5   * This program will measure the pingpong bandwidth of the cluster.
6   *
7   * As Clustis2 uses gigabit ethernet, we expect 125MB of transfer to take
8   * about 1 second.
9   *
10  */
11
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <unistd.h>
15 #include <mpi.h>
16
17 int rank, size;
18
19 /*
20  * Function to perform a pingpong test
21  */
22
23 double pingtime(int buffsize, int ntimes)
24 {
25     int i;
26     double start, stop;
27     unsigned char *buffer;
28     MPI_Status status;
29
30     /* Allocate dynamic memory for buffer and initialize */
31     buffer = malloc(buffsize);
32     for (i = 0; i < buffsize; i++)
33     buffer[i] = i;
34
35     /* Synchronize all nodes */
36     MPI_Barrier(MPI_COMM_WORLD);
37     start = MPI_Wtime();
38
39     /* Rank 0 should send, then recevie.
40      * Rank 1 should receive, then send.
41      * Everyone else should do nothing.
42      */
43
44     if (rank == 0) {
45         for (i = 0; i < ntimes; i++) {
46             MPI_Send(buffer, buffsize, MPI_UNSIGNED_CHAR, 1, 0, MPI_COMM_WORLD);
47             MPI_Recv(buffer, buffsize, MPI_UNSIGNED_CHAR, 1, 0, MPI_COMM_WORLD, &
                   status);
48         }
49     } else if (rank == 1) {
50         for (i = 0; i < ntimes; i++) {
51             MPI_Recv(buffer, buffsize, MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD, &
                   status);
52             MPI_Send(buffer, buffsize, MPI_UNSIGNED_CHAR, 0, 0, MPI_COMM_WORLD);
53         }
54     }
55
```

```
56      stop = MPI_Wtime();
57
58      /* Free (deallocate) dynamic memory */
59      free(buffer);
60
61      /* Return average time spent. We did ntimes*2 transfers */
62      return (stop - start) / (ntimes * 2.0L);
63  }
64
65  double reducetest(int els, int ntimes, int all){
66      int i;
67      double start, stop;
68      MPI_Status status;
69
70      int *isend=(int*)malloc(els*sizeof(int));
71      int *irecv=(int*)malloc(els*sizeof(int));
72
73      for (i=0;i<els;i++){
74          isend[i]=i;
75      }
76
77
78      MPI_Barrier(MPI_COMM_WORLD);
79      start = MPI_Wtime();
80
81      if (all)
82          for (i=0;i<ntimes;i++)
83              MPI_Allreduce ( isend, irecv, els, MPI_INT, MPI_SUM, MPI_COMM_WORLD );
84      else
85          for (i=0;i<ntimes;i++)
86              MPI_Reduce ( isend, irecv, els, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
87
88      stop = MPI_Wtime();
89
90      free(isend);
91      free(irecv);
92
93      return (stop - start) / ntimes;
94  }
95
96  double alltoalltest(int els, int ntimes, int pros){
97      int i;
98      double start, stop;
99      MPI_Status status;
100
101     int *isend=(int*)malloc(els*sizeof(int));
102     int *irecv=(int*)malloc(els*sizeof(int));
103
104     for (i=0;i<els;i++){
105         isend[i]=i;
106     }
107
108     MPI_Barrier(MPI_COMM_WORLD);
109     start = MPI_Wtime();
110
111     for (i=0;i<ntimes;i++)
112         MPI_Alltoall (isend, els/pros, MPI_INTEGER, irecv, els/pros, MPI_INTEGER,
                MPI_COMM_WORLD );
113
114     stop = MPI_Wtime();
115     free(isend);
116     free(irecv);
```

```
117     return (stop - start) / ntimes;
118 }
119 int main(int argc, char **argv){
120     double ts, beta;
121     double tests[]={0,0,0,0,0,0,0,0,0};
122 
123     /* Initialize MPI */
124     MPI_Init(&argc, &argv);
125     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
126     MPI_Comm_size(MPI_COMM_WORLD, &size);
127 
128     /* Find Ts and beta */
129     ts = pingtime(0, 1000);
130     beta = pingtime(10000000, 10) / 10000000.0;
131 
132     /* Broadcast this to all nodes */
133     MPI_Bcast(&ts, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
134     MPI_Bcast(&beta, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
135 
136     /* Print out results */
137     if (rank == 0) {
138         printf("Time: Tick %g, Ts %g, Beta %g\n", MPI_Wtick(), ts, beta);
139         printf("Sanity test: 125 MB in %fs\n", 125000000.0 * beta);
140     }
141 
142     /* Your code goes here.
143      * You should measure the time it takes for a MPI_Reduce,
144      * MPI_Allreduce and MPI_Alltoall, all with varying
145      * buffer sizes and see how that matches up to
146      * expectations.
147      */
148     tests[0]=reducetest(0,100,0);
149     tests[1]=reducetest(1000,100,0);
150     tests[2]=reducetest(1000000,100,0);
151     tests[3]=reducetest(0,100,1);
152     tests[4]=reducetest(1000,100,1);
153     tests[5]=reducetest(1000000,100,1);
154     tests[6]=alltoalltest(0,100,size);
155     tests[7]=alltoalltest(1000,100,size);
156     tests[8]=alltoalltest(1000000,100,size);
157     //MPI_Bcast(&tests, 9, MPI_DOUBLE, 0, MPI_COMM_WORLD);
158     if (rank == 0) {
159         printf("Reduce tests:\n");
160         printf("### %f with 0 bytes\n",tests[0]);
161         printf("### %f with %d bytes\n",tests[1],1000*sizeof(int));
162         printf("### %f with %d bytes\n",tests[2],1000000*sizeof(int));
163         printf("Allreduce tests:\n");
164         printf("### %f with 0 bytes\n",tests[3]);
165         printf("### %f with %d bytes\n",tests[4],1000*sizeof(int));
166         printf("### %f with %d bytes\n",tests[5],1000000*sizeof(int));
167         printf("Alltoall tests:\n");
168         printf("### %f with 0 bytes\n",tests[6]);
169         printf("### %f with %d bytes\n",tests[7],1000*sizeof(int));
170         printf("### %f with %d bytes\n",tests[8],1000000*sizeof(int));
171     }
172 
173     MPI_Finalize();
174 }
```

A.2.2 clustis.out

```
1 Clustis_test_info: 27. januar 2007, 18:54
```

```
2
3  Time: Tick 1e-06, Ts 8.26295e-05, Beta 2.3626e-08
4  Sanity test: 125 MB in 2.953253s
5  Reduce tests:
6  ### 0.000000 with 0 bytes
7  ### 0.000872 with 4000 bytes
8  ### 0.168039 with 4000000 bytes
9  Allreduce tests:
10 ### 0.000000 with 0 bytes
11 ### 0.000919 with 4000 bytes
12 ### 0.272745 with 4000000 bytes
13 Alltoall tests:
14 ### 0.000000 with 0 bytes
15 ### 0.000552 with 4000 bytes
16 ### 0.138842 with 4000000 bytes
17 =>> PBS: job killed: walltime 99 exceeded limit 90
18
19 Time: Tick 1e-06, Ts 6.5775e-05, Beta 2.63078e-08
20 Sanity test: 125 MB in 3.288473s
21 Reduce tests:
22 ### 0.000000 with 0 bytes
23 ### 0.000562 with 4000 bytes
24 ### 0.100720 with 4000000 bytes
25 Allreduce tests:
26 ### 0.000000 with 0 bytes
27 ### 0.000355 with 4000 bytes
28 ### 0.171127 with 4000000 bytes
29 Alltoall tests:
30 ### 0.000000 with 0 bytes
31 ### 0.000253 with 4000 bytes
32 ### 0.088699 with 4000000 bytes
33 =>> PBS: job killed: walltime 127 exceeded limit 90
34
35 Time: Tick 1e-06, Ts 7.92665e-05, Beta 2.33988e-08
36 Sanity test: 125 MB in 2.924849s
37 Reduce tests:
38 ### 0.000000 with 0 bytes
39 ### 0.000889 with 4000 bytes
40 ### 0.206218 with 4000000 bytes
41 Allreduce tests:
42 ### 0.000000 with 0 bytes
43 ### 0.001710 with 4000 bytes
44 ### 0.359614 with 4000000 bytes
45 Alltoall tests:
46 ### 0.000000 with 0 bytes
47 ### 0.001719 with 4000 bytes
48 ### 0.176309 with 4000000 bytes
49 =>> PBS: job killed: walltime 121 exceeded limit 90
```

A.3 An example experiment property file

```
1  //main test parameters, those are stored into
2  SIMULATED_ONLY = true
3  USE_AND_QUERIES = true
4  DEMO_MODE = true
5  //LI 0, GI 1, PL 2, HD 3, LIPM 4, GIPM 5, PLPM 6, HDPM 7
6  INDEXING_MODE = 2
7  NUMBER_OF_NODES = 4
8  NUMBER_OF_CPUS_PER_NODE = 1
9  NUMBER_OF_DISKS_PER_NODE = 1
10 MAX_NUMBER_OF_QUERY_PROCESSES = 12
11 CPU_FACTOR = 1.0
12
13 //number of queries to process before the non-demo benchmark
14 NUMBER_OF_WARMUP_QUERIES = 0
15 //maximum length of the experiment
16 SIM_DURATION = 5000
17 //log sample time for demo run
18 LOG_SAMPLE_TIME = 10
19
20 //document collection constants
21 NUMBER_OF_RESULTS_REQUIRED = 1000
22 NUMBER_OF_DOCUMENTS = 25000000
23 NUMBER_OF_DOCUMENTS_STAT = 25205179
24 //used only for the microbenchmarking
25 NUMBER_OF_WORDS_PER_DOCUMENT = 10000
26
27 //memory constants
28 //in BLOCKS
29 SIZE_OF_MEMORY_PER_NODE = 4096
30 //4096
31 //in bytes
32 MEMORY_BLOCK_SIZE = 1048576
33 BUFFER_SIZE = 131072
34
35 //network performance constants
36 NETWORK_INVERSE_BANDWIDTH = 8E-6
37 NETWORK_OVERHEAD = 0.08
38
39 //disk performance constants
40 DISK_INVERSE_BANDWIDTH = 1.25E-5
41 DISK_SEEK_TIME = 8.5
42 DISK_ROTATION_DELAY = 4.16
43
44 //cpu processing constants, to be divided by CPU_FACTOR
45 COMPARSION_INSTRUCTION_TIME = 1.75E-5
46 INTERLEAVE_TWO_LISTS_INSTRUCTION_TIME = 2.421848E-5
47 HEAP_INSTRUCTION_TIME = 5.896446E-7
48 HEAP_MULTIWAY_INTERLEAVEMERGE_TIME = 4.945716E-5
49 LOOKUP_TIME = 0.0105
50
51 //data structure constants
52 QUERY_HEADER_LENGTH = 32
53 SUBQUERY_HEADER_LENGTH = 32
54 SUBQUERY_RESULT_HEADER_LENGTH = 32
55 QUERY_RESULT_HEADER_LENGTH = 32
56 //loc + 1byte descr
57 SIZE_OF_INDEX_LOCATION_ENTRY = 5
58  //docnum + occs
59 SIZE_OF_INDEX_DOCUMENT_ENTRY = 8
```

```
60  SIZE_OF_ACC_ENTRY = 8
61  SIZE_OF_QUERY_ENTRY = 4
62  SIZE_OF_RESULT_ENTRY = 4
63  //number of entries per chunk using the hybrid scheme
64  HYBRID_CHUNK_SIZE = 1024
65
66  //cpu time slice for each process
67  CPU_SLICE_SIZE = 0.1
```

A.4 Trace Mode Simulation Reports for Baseline Experiments

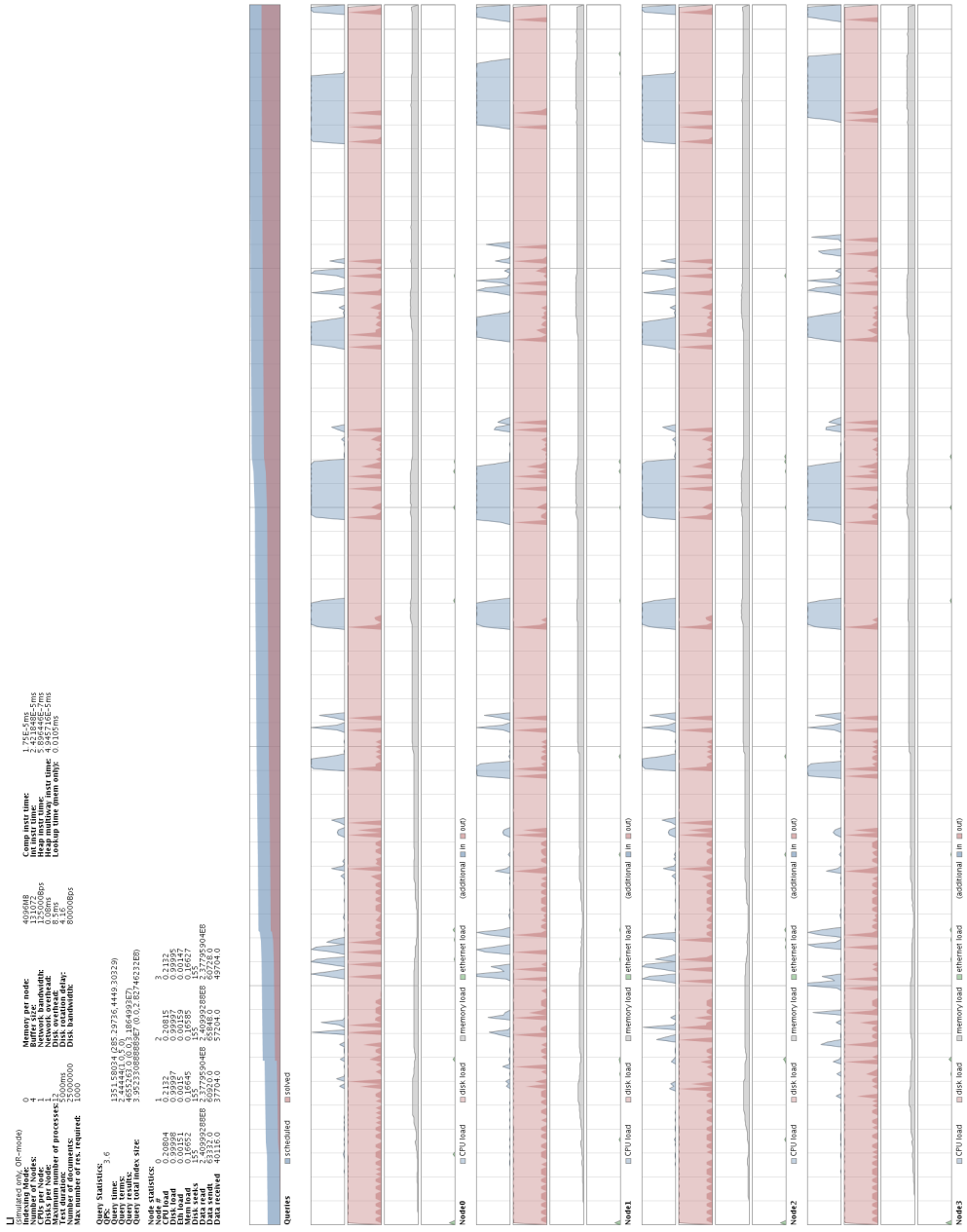

Figure A.1: Local Indexing - Node Status

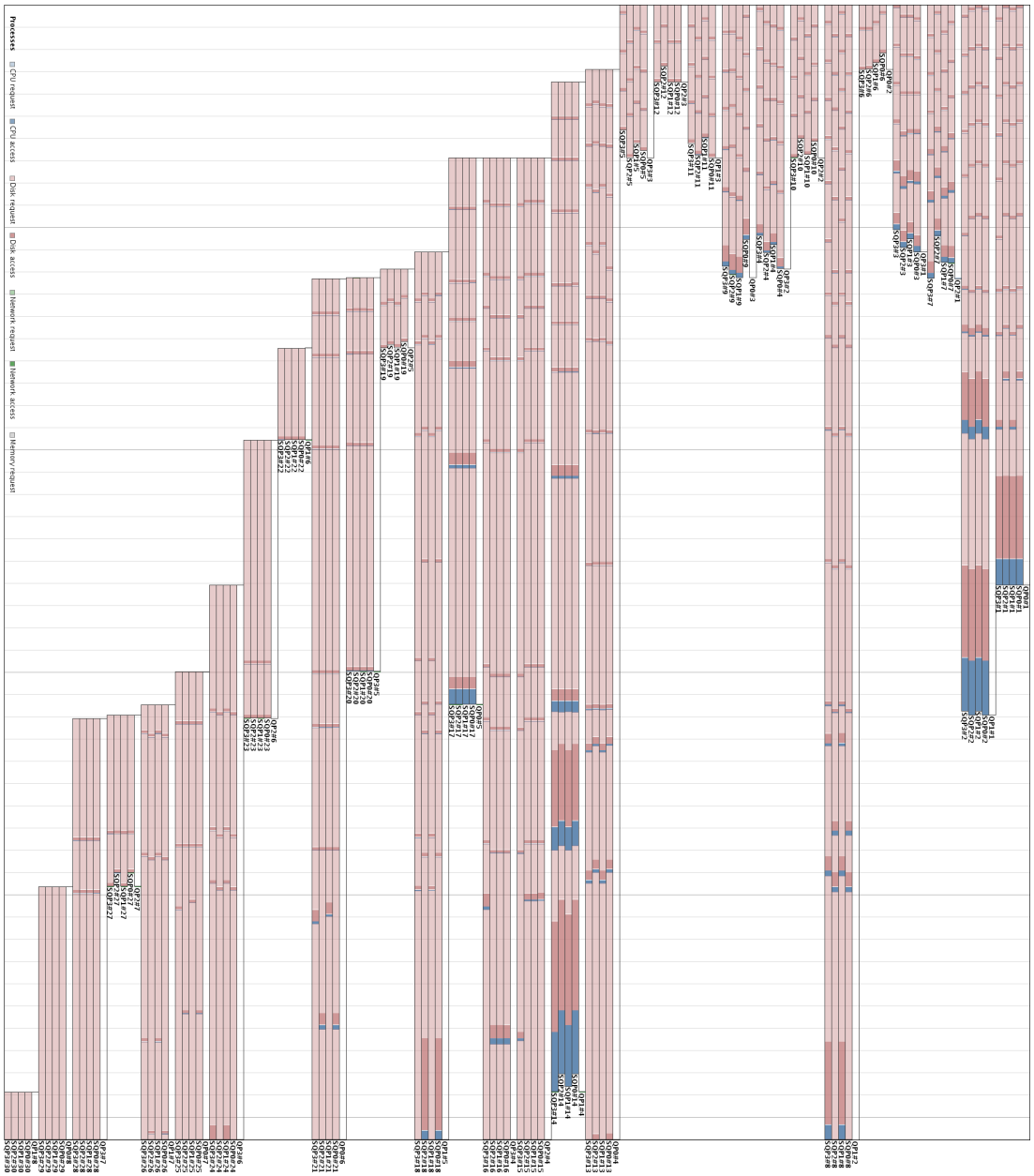

Figure A.2: Local Indexing - Process Status

Figure A.3: Global Indexing - Node Status

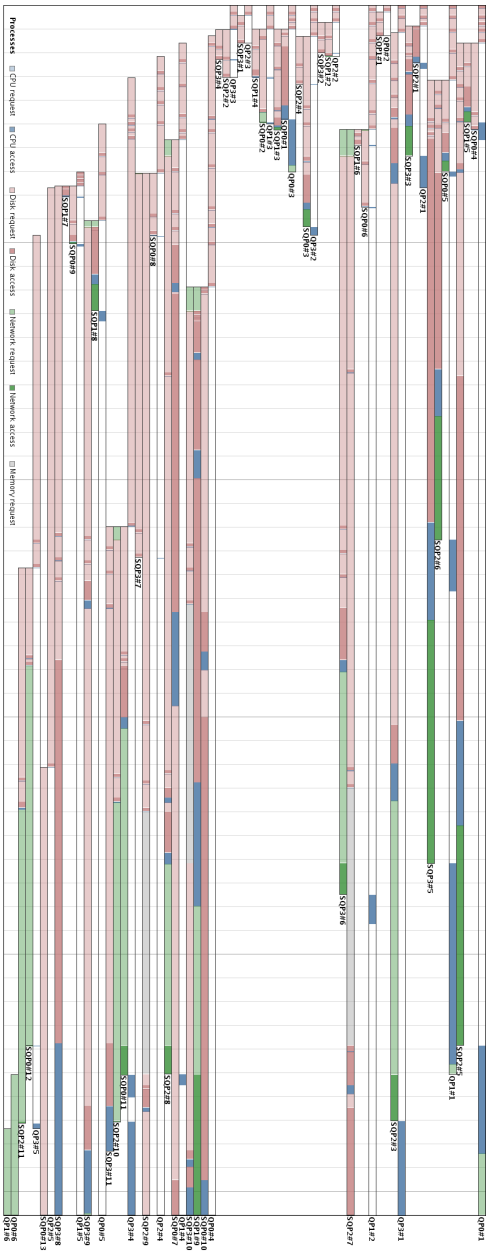

Figure A.4: Global Indexing - Process Status

Figure A.5: Pipelined Indexing - Node Status

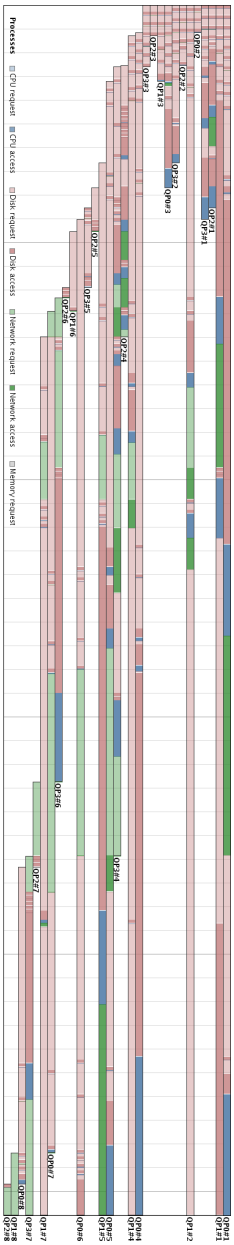

Figure A.6: Pipelined Indexing - Process Status

Figure A.7: Local Indexing - Node Status

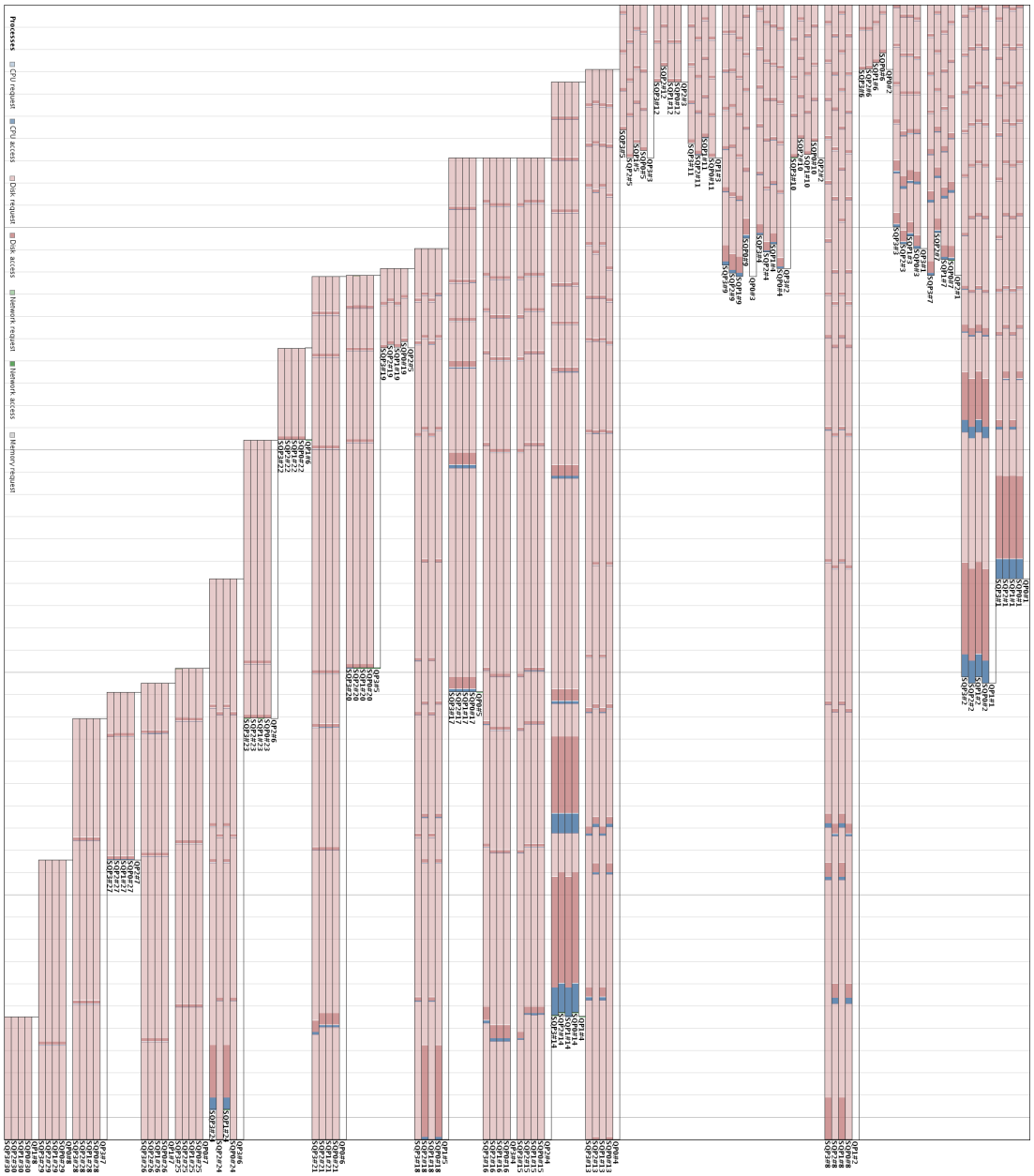

Figure A.8: Local Indexing - Process Status

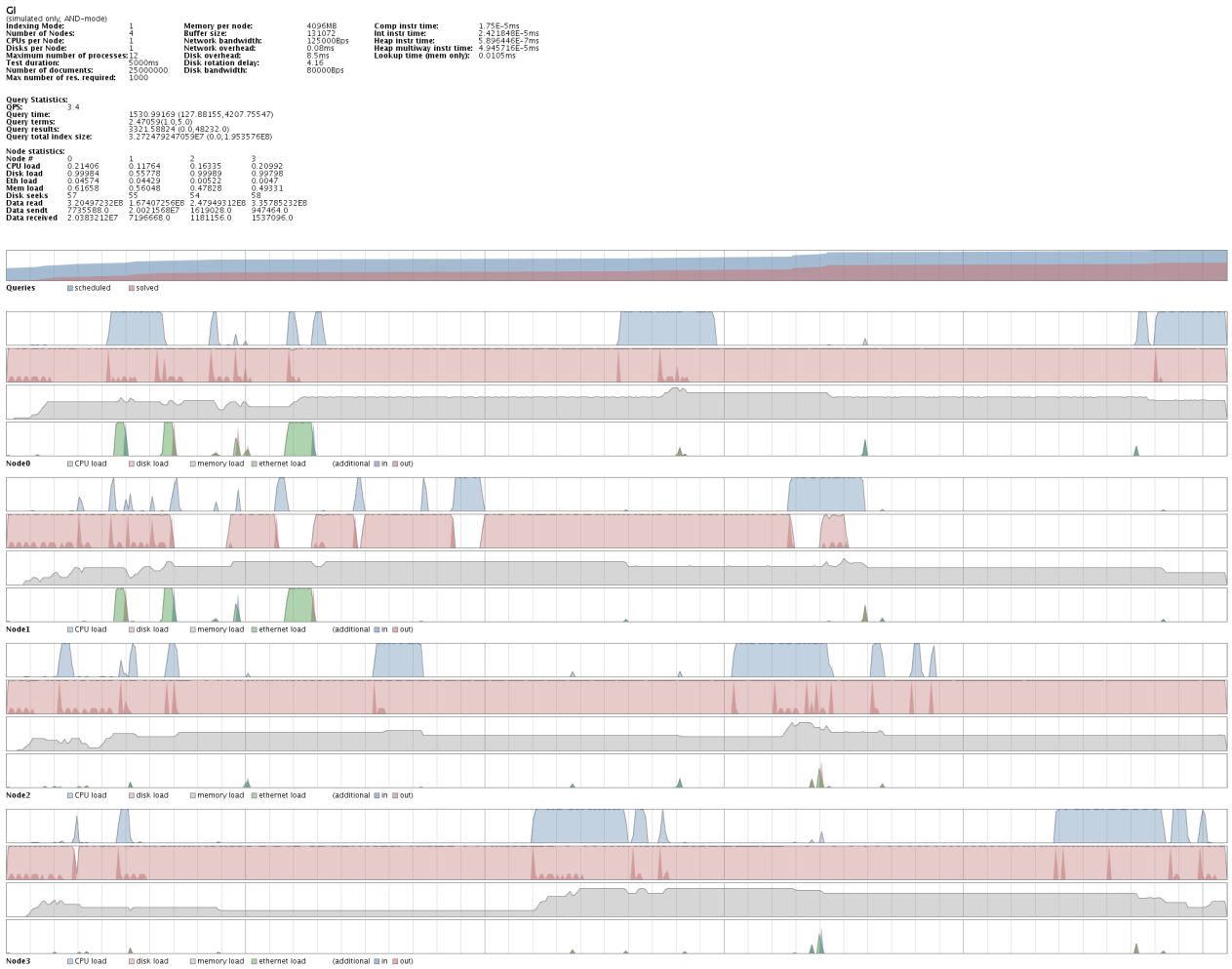

Figure A.9: Global Indexing - Node Status

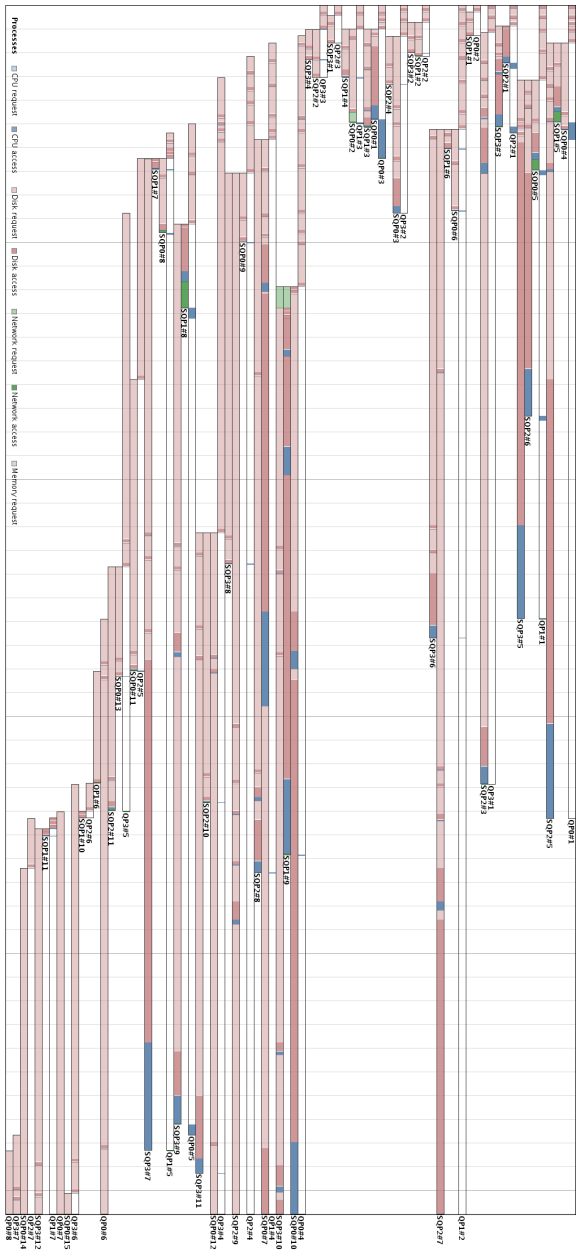

Figure A.10: Global Indexing - Process Status

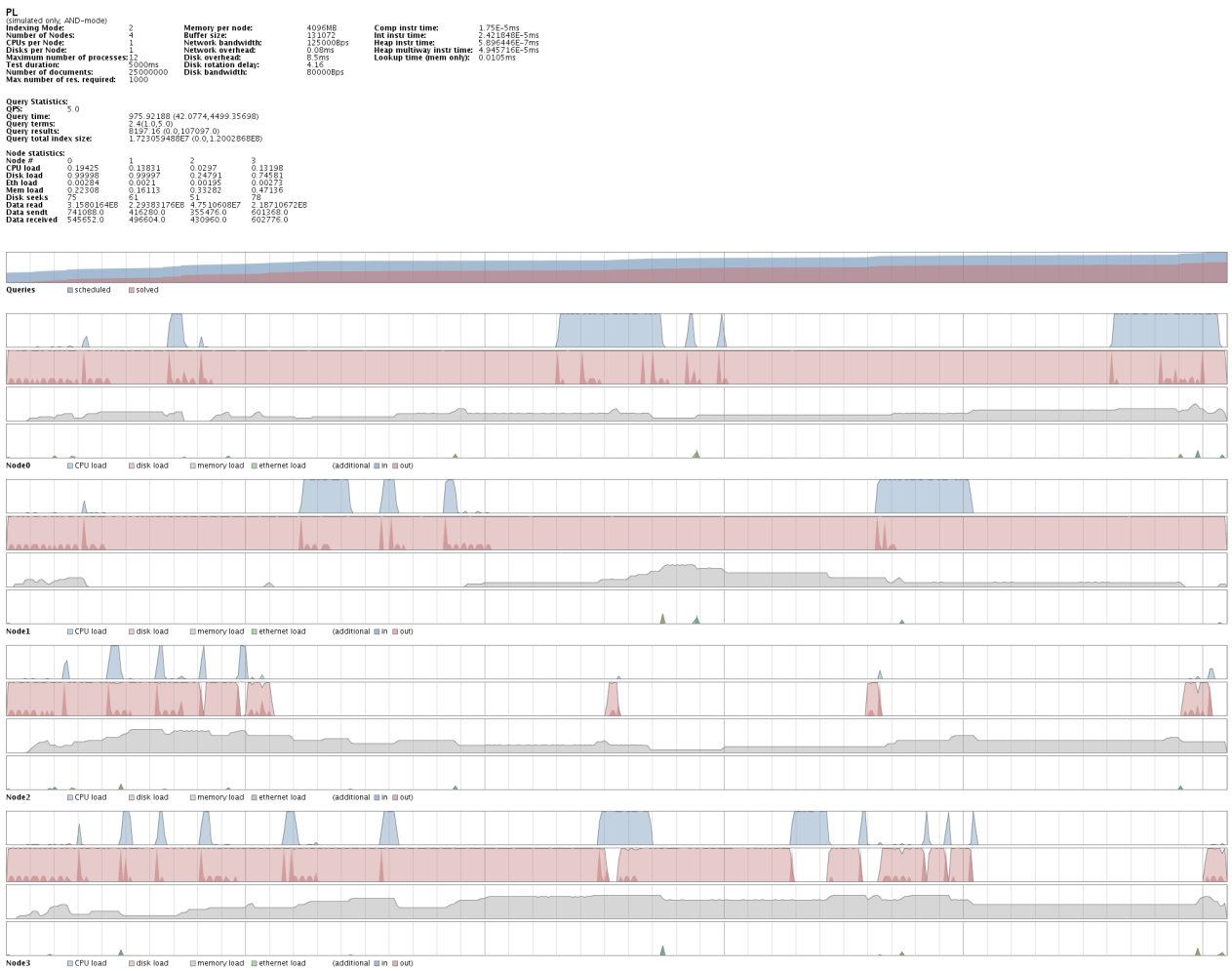

Figure A.11: Pipelined Indexing - Node Status

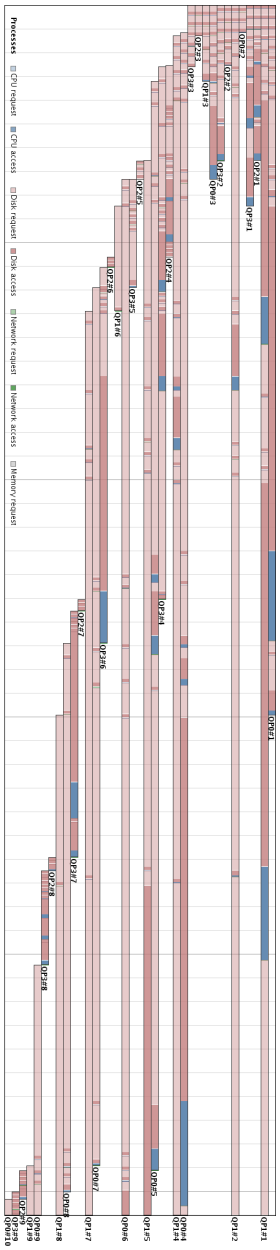

Figure A.12: Pipelined Indexing - Process Status

Appendix B

Source code

B.1 Simulation Model Source Code

B.1.1 simulation.model.Config

```
1  package simulation.model;
2  import java.util.*;
3  import java.io.*;
4
5  public class Config {
6      public static String DATAPATH = "/home/simonj/workspace/Sim2/data/";
7
8      //algorithm choices
9      public static final int LOCAL_INDEXING = 0;
10     public static final int GLOBAL_INDEXING = 1;
11     public static final int GLOBAL_PIPELINED_INDEXING = 2;
12     public static final int HYBRID_INDEXING = 3;
13     public static final int LOCAL_INDEXING_PARALLEL_MERGE = 4;
14     public static final int GLOBAL_INDEXING_PARALLEL_MERGE = 5;
15     public static final int GLOBAL_PIPELINED_INDEXING_PARALLEL_MERGE = 6;
16     public static final int HYBRID_INDEXING_PARALLEL_MERGE = 7;
17
18     //main test parameters, those are stored into
19     public static boolean SIMULATED_ONLY = true;
20     public static boolean USE_AND_QUERIES = true;
21     public static boolean DEMO_MODE = true;
22     public static int INDEXING_MODE = -1;
23     public static int NUMBER_OF_NODES = 4;
24     public static int NUMBER_OF_CPUS_PER_NODE = 1;
25     public static int NUMBER_OF_DISKS_PER_NODE = 1;
26     public static int MAX_NUMBER_OF_QUERY_PROCESSES = 12;
27     public static double CPU_FACTOR = 1.0;
28     //number of queries to process before the non-demo benchmark
29     public static int NUMBER_OF_WARMUP_QUERIES = 0;
30     //maximum length of the experiment
31     public static int SIM_DURATION = 10000; //
32     //log sample time for demo run
33     public static int LOG_SAMPLE_TIME = 10;
34
35     //document collection constants
36     public static int NUMBER_OF_RESULTS_REQUIRED = 1000;
37     public static int NUMBER_OF_DOCUMENTS = 25000000;
38     public static int NUMBER_OF_DOCUMENTS_STAT = 25205179;
```

```
39      public static int NUMBER_OF_WORDS_PER_DOCUMENT = 10000; //used only for the
            microbenchmarking
40
41      //memory contants
42      public static int SIZE_OF_MEMORY_PER_NODE = 4096; //in BLOCKS
43      public static int MEMORY_BLOCK_SIZE = 1048576; //in bytes
44      public static int BUFFER_SIZE = 131072; //in bytes
45
46      //network performance constants
47      public static double NETWORK_INVERSE_BANDWIDTH = 8E-6;
48      public static double NETWORK_OVERHEAD = 0.08;
49
50      //disk performance constants
51      public static double DISK_INVERSE_BANDWIDTH = 1.25E-5;
52      public static double DISK_SEEK_TIME = 8.5;
53      public static double DISK_ROTATION_DELAY = 4.16;
54
55      //cpu processing constants
56
57      public static double COMPARSION_INSTRUCTION_TIME = 2.421848E-05 / CPU_FACTOR;
58      public static double INTERLEAVE_TWO_LISTS_INSTRUCTION_TIME = 2.421848E-05 /
            CPU_FACTOR;
59      public static double HEAP_INSTRUCTION_TIME = 5.896446E-07 / CPU_FACTOR;
60      public static double HEAP_MULTIWAY_INTERLEAVEMERGE_TIME = 4.945716E-05 /
            CPU_FACTOR;
61      public static double LOOKUP_TIME = 0.0105 /CPU_FACTOR;
62
63      //data structure constants
64      public static int QUERY_HEADER_LENGTH = 32;
65      public static int SUBQUERY_HEADER_LENGTH = 32;
66      public static int SUBQUERY_RESULT_HEADER_LENGTH = 32;
67      public static int QUERY_RESULT_HEADER_LENGTH = 32;
68      public static int SIZE_OF_INDEX_LOCATION_ENTRY = 5; //loc + 1byte descr
69      public static int SIZE_OF_INDEX_DOCUMENT_ENTRY = 8; //docnum + occs
70      public static int SIZE_OF_ACC_ENTRY = 8;
71      public static int SIZE_OF_QUERY_ENTRY = 4;
72      public static int SIZE_OF_RESULT_ENTRY = 8;
73      //number of entries per chunk using the hybrid scheme
74      public static int HYBRID_CHUNK_SIZE = 1024;
75
76      //cpu time slice for each process
77      public static double CPU_SLICE_SIZE = 0.1;
78
79      public static void loadProperties(String filename){
80          try {
81              Properties prop = new Properties();
82              prop.load(new FileInputStream(filename));
83              //DATAPATH = prop.getProperty("DATAPATH");
84              SIMULATED_ONLY = Boolean.parseBoolean(prop.getProperty("SIMULATED_ONLY
                    "));
85              USE_AND_QUERIES = Boolean.parseBoolean(prop.getProperty("
                    USE_AND_QUERIES"));
86              DEMO_MODE = Boolean.parseBoolean(prop.getProperty("DEMO_MODE"));
87              INDEXING_MODE = Integer.parseInt(prop.getProperty("INDEXING_MODE"));
88              NUMBER_OF_NODES = Integer.parseInt(prop.getProperty("NUMBER_OF_NODES")
                    );
89              NUMBER_OF_CPUS_PER_NODE = Integer.parseInt(prop.getProperty("
                    NUMBER_OF_CPUS_PER_NODE"));
90              NUMBER_OF_DISKS_PER_NODE = Integer.parseInt(prop.getProperty("
                    NUMBER_OF_DISKS_PER_NODE"));
91              MAX_NUMBER_OF_QUERY_PROCESSES = Integer.parseInt(prop.getProperty("
                    MAX_NUMBER_OF_QUERY_PROCESSES"));
```

```
92              CPU_FACTOR = Double.parseDouble(prop.getProperty("CPU_FACTOR"));
93              NUMBER_OF_WARMUP_QUERIES = Integer.parseInt(prop.getProperty("
                    NUMBER_OF_WARMUP_QUERIES"));
94              SIM_DURATION = Integer.parseInt(prop.getProperty("SIM_DURATION"));
95              LOG_SAMPLE_TIME = Integer.parseInt(prop.getProperty("LOG_SAMPLE_TIME")
                    );
96              NUMBER_OF_RESULTS_REQUIRED = Integer.parseInt(prop.getProperty("
                    NUMBER_OF_RESULTS_REQUIRED"));
97              NUMBER_OF_DOCUMENTS = Integer.parseInt(prop.getProperty("
                    NUMBER_OF_DOCUMENTS"));
98              NUMBER_OF_DOCUMENTS_STAT = Integer.parseInt(prop.getProperty("
                    NUMBER_OF_DOCUMENTS_STAT"));
99              NUMBER_OF_WORDS_PER_DOCUMENT = Integer.parseInt(prop.getProperty("
                    NUMBER_OF_WORDS_PER_DOCUMENT"));
100             SIZE_OF_MEMORY_PER_NODE = Integer.parseInt(prop.getProperty("
                    SIZE_OF_MEMORY_PER_NODE"));
101             MEMORY_BLOCK_SIZE = Integer.parseInt(prop.getProperty("
                    MEMORY_BLOCK_SIZE"));
102             BUFFER_SIZE = Integer.parseInt(prop.getProperty("BUFFER_SIZE"));
103             NETWORK_INVERSE_BANDWIDTH = Double.parseDouble(prop.getProperty("
                    NETWORK_INVERSE_BANDWIDTH"));
104             NETWORK_OVERHEAD =  Double.parseDouble(prop.getProperty("
                    NETWORK_OVERHEAD"));
105             DISK_INVERSE_BANDWIDTH =  Double.parseDouble(prop.getProperty("
                    DISK_INVERSE_BANDWIDTH"));
106             DISK_SEEK_TIME =  Double.parseDouble(prop.getProperty("DISK_SEEK_TIME"
                    ));
107             DISK_ROTATION_DELAY =  Double.parseDouble(prop.getProperty("
                    DISK_ROTATION_DELAY"));
108             COMPARSION_INSTRUCTION_TIME =  Double.parseDouble(prop.getProperty("
                    COMPARSION_INSTRUCTION_TIME"))/CPU_FACTOR;
109             INTERLEAVE_TWO_LISTS_INSTRUCTION_TIME =  Double.parseDouble(prop.
                    getProperty("INTERLEAVE_TWO_LISTS_INSTRUCTION_TIME"))/CPU_FACTOR;
110             HEAP_INSTRUCTION_TIME = Double.parseDouble(prop.getProperty("
                    HEAP_INSTRUCTION_TIME")) /CPU_FACTOR;
111             HEAP_MULTIWAY_INTERLEAVEMERGE_TIME =Double.parseDouble(prop.
                    getProperty("HEAP_MULTIWAY_INTERLEAVEMERGE_TIME")) /CPU_FACTOR;
112             LOOKUP_TIME = Double.parseDouble(prop.getProperty("LOOKUP_TIME"))/
                    CPU_FACTOR;
113             QUERY_HEADER_LENGTH = Integer.parseInt(prop.getProperty("
                    QUERY_HEADER_LENGTH"));
114             SUBQUERY_HEADER_LENGTH = Integer.parseInt(prop.getProperty("
                    SUBQUERY_HEADER_LENGTH"));
115             SUBQUERY_RESULT_HEADER_LENGTH = Integer.parseInt(prop.getProperty("
                    SUBQUERY_RESULT_HEADER_LENGTH"));
116             QUERY_RESULT_HEADER_LENGTH =Integer.parseInt(prop.getProperty("
                    QUERY_RESULT_HEADER_LENGTH"));
117             SIZE_OF_INDEX_LOCATION_ENTRY = Integer.parseInt(prop.getProperty("
                    SIZE_OF_INDEX_LOCATION_ENTRY"));
118             SIZE_OF_INDEX_DOCUMENT_ENTRY = Integer.parseInt(prop.getProperty("
                    SIZE_OF_INDEX_DOCUMENT_ENTRY"));
119             SIZE_OF_ACC_ENTRY = Integer.parseInt(prop.getProperty("
                    SIZE_OF_ACC_ENTRY"));
120             SIZE_OF_QUERY_ENTRY = Integer.parseInt(prop.getProperty("
                    SIZE_OF_QUERY_ENTRY"));
121             SIZE_OF_RESULT_ENTRY = Integer.parseInt(prop.getProperty("
                    SIZE_OF_RESULT_ENTRY"));
122             HYBRID_CHUNK_SIZE = Integer.parseInt(prop.getProperty("
                    HYBRID_CHUNK_SIZE"));
123             CPU_SLICE_SIZE = Double.parseDouble(prop.getProperty("CPU_SLICE_SIZE")
                    );
124         } catch (IOException e){
```

```
125             e.printStackTrace();
126         }
127     }
128 }
```

B.1.2 simulation.model.ModelS

```
1  package simulation.model;
2  import java.util.ArrayList;
3
4  import simulation.log.*;
5  import simulation.node.*;
6  import simulation.processes.*;
7  import simulation.query.*;
8  import desmoj.core.simulator.*;
9
10 public class ModelS extends Model {
11     ArrayList<Node> nodes;
12     Node broker;
13
14     private LogWriter logwr;
15     private GeneratorProcess gen;
16     private ResHandler res;
17     private LogProcess logpr;
18     private Statistics st;
19     private boolean warmedup = false;
20     private int scheduledqueries, solvedqueries, runningqueries;
21
22     public static String runName;
23
24
25     public ModelS(Model owner, String name, boolean showInReport, boolean
           showIntrace) {
26         super(owner, name, showInReport, showIntrace);
27         runningqueries=0;
28         scheduledqueries=0;
29         solvedqueries=0;
30     }
31
32     public ArrayList<Node> getNodes(){
33         return nodes;
34     }
35     public LogWriter getLogWriter(){
36         return logwr;
37     }
38
39     public Node getBroker(){
40         return broker;
41     }
42
43     @Override
44     public String description() {
45         return "\\,,/␣o.O␣\\,,/";
46     }
47
48     @Override
49     public void doInitialSchedules() {
50     }
51
52     public ResHandler getResHandler(){
53         return res;
54     }
```

```
55
56      public GeneratorProcess getGenerator(){
57          return gen;
58      }
59      public boolean scheduleQuery(Query q, int nodenr){
60          if (runningqueries<Config.MAX_NUMBER_OF_QUERY_PROCESSES){
61              runningqueries++;
62              scheduledqueries++;
63              QueryProcess newqp;
64              switch (Config.INDEXING_MODE){
65                  case (Config.HYBRID_INDEXING):
66                      newqp = new QueryProcessHD(this, "QP"+nodenr, true, q, nodes.
                            get(nodenr));
67                  break;
68                  case (Config.HYBRID_INDEXING_PARALLEL_MERGE):
69                      newqp = new QueryProcessHDPM(this, "QP"+nodenr, true, q, nodes
                            .get(nodenr));
70                  break;
71                  case (Config.GLOBAL_PIPELINED_INDEXING):
72                      newqp = new QueryProcessPL(this, "QP"+nodenr, true, q, nodes.
                            get(nodenr));
73                  break;
74                  case (Config.GLOBAL_PIPELINED_INDEXING_PARALLEL_MERGE):
75                      newqp = new QueryProcessPLPM(this, "QP"+nodenr, true, q, nodes
                            .get(nodenr));
76                  break;
77                  case (Config.GLOBAL_INDEXING):
78                      newqp = new QueryProcessGI(this, "QP"+nodenr, true, q, nodes.
                            get(nodenr));
79                  break;
80                  case (Config.GLOBAL_INDEXING_PARALLEL_MERGE):
81                      newqp = new QueryProcessGIPM(this, "QP"+nodenr, true, q, nodes
                            .get(nodenr));
82                  break;
83                  case (Config.LOCAL_INDEXING):
84                  default:
85                      newqp = new QueryProcessLI(this, "QP"+nodenr, true, q, nodes.
                            get(nodenr));
86                  break;
87                  case (Config.LOCAL_INDEXING_PARALLEL_MERGE):
88                      newqp = new QueryProcessLIPM(this, "QP"+nodenr, true, q, nodes
                            .get(nodenr));
89                  break;
90
91              }
92
93              newqp.activate(new SimTime(0.0));
94              if (Config.DEMO_MODE)
95                  logwr.logSystemStatus(currentTime().getTimeValue(),
                        scheduledqueries, solvedqueries);
96              return true;
97          } else {
98              if (Config.DEMO_MODE)
99                  logwr.logSystemStatus(currentTime().getTimeValue(),
                        scheduledqueries, solvedqueries);
100             return false;
101         }
102     }
103
104     public void querySolved(QueryProcess qp, Query q, QueryResult qr){
105
106         st.addQueryTerms(q.getNumberOfTerms());
```

```
107         st.addQueryIndexSize(q.getTotalIndexSize());
108         st.addQueryResults(qr.getNumberOfDocuments());
109         st.addQueryTime(SimTime.diff(currentTime(), qp.getStartTime()));
110 
111         if (!Config.DEMO_MODE && solvedqueries == Config.NUMBER_OF_WARMUP_QUERIES
                && !warmedup){
112             //resetAllCounters
113             solvedqueries=0;
114             scheduledqueries=runningqueries;
115             reset();
116             for (Node node: nodes){
117                 node.resetCounters();
118             }
119             warmedup=true;
120 
121         }
122 
123         runningqueries--;
124         solvedqueries++;
125         if (runningqueries<Config.MAX_NUMBER_OF_QUERY_PROCESSES){
126             gen.activate(new SimTime(0.0));
127         }
128     }
129 
130     public Statistics getStatistics(){
131         return st;
132     }
133 
134     public int getSolved(){
135         return solvedqueries;
136     }
137 
138     @Override
139     public void init() {
140         st = new Statistics(this);
141         nodes=new ArrayList<Node>();
142         res=new ResHandler();
143         for (int i=0;i<Config.NUMBER_OF_NODES;i++){
144             nodes.add(new Node(this,i, Config.NUMBER_OF_DISKS_PER_NODE,
145                     Config.NUMBER_OF_CPUS_PER_NODE, Config.SIZE_OF_MEMORY_PER_NODE
                        ));
146         }
147         if (Config.DEMO_MODE) {
148             logwr = new LogWriter(this);
149             logpr = new LogProcess(this, "Logger", false);
150             logpr.activate(new SimTime(0.0));
151         }
152         broker = new Node(this, Config.NUMBER_OF_NODES+1, 1, 1, 51200);
153         gen = new GeneratorProcess(this,"Generator",true);
154         gen.activate(new SimTime(0.0));
155     }
156 
157     public static void main(String args[]){
158         if (args.length>0) runName = args[0];
159         else runName = "default";
160 
161         Config.loadProperties(runName+".property");
162 
163         ModelS model = new ModelS(null, runName, true, true);
164         Experiment exp = new Experiment(runName);
165         model.connectToExperiment(exp);
166 
```

```
167         exp.setShowProgressBar(false);
168
169         //exp.getReportManager().
170         exp.stop(new SimTime(Config.SIM_DURATION));
171         exp.tracePeriod(new SimTime(0.0), new SimTime(1000.0));
172         exp.debugPeriod(new SimTime(0.0), new SimTime(0.0));
173         exp.start();
174
175         exp.report();
176         exp.finish();
177
178         if (Config.DEMO_MODE){
179             model.logwr.renderNodeLog();
180             model.logwr.renderProcLog();
181         } else {
182             model.st.storeAll();
183         }
184
185         System.out.println("results:"+Config.MAX_NUMBER_OF_QUERY_PROCESSES+"\t"+
                Config.NUMBER_OF_NODES+"\t"+(model.getSolved() * 1000 /  SimTime.diff(
                model.currentTime(), model.resetAt()).getTimeValue()) + "\t" +model.st
                .queryTimes.getMean());
186     }
187 }
```

B.1.3 simulation.model.QueryLogReader

```
1  package simulation.model;
2
3  import java.util.*;
4  import java.io.*;
5  import simulation.*;
6  import simulation.query.Query;
7  import simulation.query.SimpleIndexHitList;
8  import simulation.query.SimulatedIndexHitList;
9  import desmoj.core.simulator.*;
10
11 public class QueryLogReader {
12     private BufferedReader br;
13     private HashMap<String, int[]> dict;
14     private Model model;
15
16     public QueryLogReader(Model model){
17         this.model = model;
18         dict = new HashMap<String, int[]>(31220);
19
20         try {
21             br = new BufferedReader(new FileReader(Config.DATAPATH + "docstat"));
22             String line, tmp[];
23             while ( (line = br.readLine()) != null ){
24                 tmp = line.split("␣");
25                 dict.put(tmp[0], new int[]{Integer.parseInt(tmp[1]), Integer.
                        parseInt(tmp[2])});
26             }
27         } catch (FileNotFoundException fnfe){
28             System.out.println("Query␣statistics␣cannot␣be␣found!");
29         } catch (IOException e){
30             System.out.println("Input␣error!");
31         }
32
33         try {
34             br = new BufferedReader(new FileReader(Config.DATAPATH + "querylog"));
```

```
35          }catch (FileNotFoundException fnfe){
36              System.out.println("Query log cannot be found!");
37          }
38      }
39
40      public int[] getNextQuery(){
41          Query q = new Query(model,  "Query", false);
42          try {
43              String line = br.readLine();
44              if (line==null) return null;
45              //System.out.print(line+" [");
46              String tmp[] = line.split(" ");
47              int ans[] = new int[tmp.length * 2];
48              for (int i=0; i<tmp.length; i++){
49                  int data[] = dict.get(tmp[i]);
50                  ans[2*i] = data[0];
51                  ans[2*i+1] = data[1];
52                  //System.out.print(((double) data[1]) / Config.
                        NUMBER_OF_DOCUMENTS_STAT+" ");
53              }
54              //System.out.println("]");
55              return ans;
56          }catch (IOException e){
57              System.out.println("Input error!");
58              return null;
59          }
60      }
61  }
```

B.1.4 simulation.processes.model.GeneratorProcess

```
1   package simulation.model;
2   import java.util.*;
3   import simulation.*;
4   import simulation.node.*;
5   import simulation.query.*;
6   import desmoj.core.dist.*;
7   import desmoj.core.simulator.*;
8
9
10  public class GeneratorProcess extends SimProcess{
11      private IntDistUniform hitdist;
12      private QueryLogReader qlr;
13
14
15      public GeneratorProcess(Model owner, String name, boolean showInTrace) {
16          super(owner, name, showInTrace);
17          qlr = new QueryLogReader(owner);
18          hitdist = new IntDistUniform(owner,"",0,Config.NUMBER_OF_DOCUMENTS-1,false
                ,false);
19      }
20
21      @Override
22      public void lifeCycle() {
23          ModelS model = (ModelS) this.getModel();
24          ArrayList<Node> nodes = model.getNodes();
25
26          while (true){
27              for (int i=0; i<nodes.size(); i++){
28                  Query q= new Query(model, "Query", false);
29                  int data[] = qlr.getNextQuery();
30                  int nterms= data.length / 2;
```

```
31                  for (int j=0; j<nterms; j++){
32                      int termid = data[2*j];
33                      double freq = ((double)data[2*j+1])/Config.
                            NUMBER_OF_DOCUMENTS_STAT;
34                      if (Config.SIMULATED_ONLY)
35                          q.addTerm(termid, new SimulatedIndexHitList(freq));
36                      else
37                          q.addTerm(termid, makeHitList(freq));
38                  }
39
40                  while(!model.scheduleQuery(q, i)){
41                      passivate();
42                  }
43
44              }
45          }
46      }
47
48      public SimpleIndexHitList makeHitList(double freq){
49          int numscored = (int)(freq * Config.NUMBER_OF_DOCUMENTS);
50          int hits[] = new int[Config.NUMBER_OF_DOCUMENTS];
51          for (int k=0; k< Config.NUMBER_OF_DOCUMENTS; k++ ){
52              hits[k]=0;
53          }
54          for (int k=0; k< numscored; k++ ){
55              int t=(int) hitdist.sample();
56              while (hits[t]>0){
57                  t=(t+1) % hits.length;
58              }
59              hits[t]=1;
60          }
61          return new SimpleIndexHitList(hits);
62      }
63  }
```

B.1.5 simulation.log.LogProcess

```
1   package simulation.log;
2   import java.util.ArrayList;
3
4   import simulation.model.Config;
5   import simulation.model.ModelS;
6   import simulation.node.Node;
7
8
9
10
11  import desmoj.core.simulator.*;
12
13
14  public class LogProcess extends SimProcess {
15
16      public LogProcess(Model owner, String name, boolean showInTrace) {
17          super(owner, name, showInTrace);
18      }
19
20      private double fixRange(double i){
21          if (i<0) return 0;
22          else if (i>1) return 1;
23          else return i;
24      }
25
```

```
26      @Override
27      public void lifeCycle() {
28          ModelS model = (ModelS) getModel();
29          ArrayList<Node> nodes = model.getNodes();
30
31          double lastTime = currentTime().getTimeValue();
32          double currTime;
33          double lastCPU[] = new double[nodes.size()];
34          double lastMem[] = new double[nodes.size()];
35          double lastDisk[] = new double[nodes.size()];
36          double lastEth[] = new double[nodes.size()];
37          double lastread[] =  new double[nodes.size()];
38          double lastrecv[] = new double[nodes.size()];
39          double lastsend[] = new double[nodes.size()];
40
41          double cpu, mem, disk, eth, dtime, dcpu, dmem, ddisk, deth, rdbw, sdbw,
                rvbw, send, recv, read;
42          while(true){
43              hold(new SimTime(Config.LOG_SAMPLE_TIME));
44              currTime=currentTime().getTimeValue();
45              dtime=currTime-lastTime;
46              for (int i=0; i< nodes.size(); i++){
47                  Node node=nodes.get(i);
48
49                  cpu = node.getCPU().avgUsage();
50                  mem = node.getMemory().avgUsage();
51                  disk = node.getDisk().avgUsage();
52                  eth = node.getEth().avgUsage();
53                  read = node.getDiskread();
54                  recv = node.getEthrecv();
55                  send = node.getEthsend();
56
57                  dcpu = fixRange((cpu*currTime-lastCPU[i]*lastTime)/dtime);
58                  dmem = fixRange((mem*currTime-lastMem[i]*lastTime)/dtime);
59                  ddisk = fixRange((disk*currTime-lastDisk[i]*lastTime)/dtime);
60                  deth = fixRange((eth*currTime-lastEth[i]*lastTime)/dtime);
61                  rdbw = fixRange((read-lastread[i])/dtime*Config.
                        DISK_INVERSE_BANDWIDTH);
62                  rvbw = fixRange((recv-lastrecv[i])/dtime*Config.
                        NETWORK_INVERSE_BANDWIDTH);
63                  sdbw = fixRange((send-lastsend[i])/dtime*Config.
                        NETWORK_INVERSE_BANDWIDTH);
64
65                  lastCPU[i] = cpu;
66                  lastDisk[i] = disk;
67                  lastMem[i] = mem;
68                  lastEth[i] = eth;
69                  lastsend[i] = send;
70                  lastread[i] = read;
71                  lastrecv[i] = recv;
72
73                  model.getLogWriter().logNodeStatus(i, currTime, dcpu, ddisk, dmem,
                         deth, rdbw, sdbw, rvbw);
74              }
75              lastTime=currTime;
76          }
77      }
78
79  }
```

B.1.6 simulation.log.LogWriter

```
1  package simulation.log;
2  import java.awt.*;
3  import java.awt.image.*;
4  import java.util.*;
5  import java.io.*;
6
7  import javax.imageio.ImageIO;
8
9  import desmoj.core.simulator.SimTime;
10
11 import simulation.model.Config;
12 import simulation.model.ModelS;
13 import simulation.node.Node;
14
15 /**
16  * All the graphic-related code is based on the ImageRender.java from Bootchart -
        Boot Process Visualization
17  * Copyright (C) 2004  Ziga Mahkovec <ziga.mahkovec@klika.si>
18  *
19  * Modified by Simon Jonassen, 2006
20  */
21
22 public class LogWriter {
23     public static final int START = 0;
24     public static final int WAIT = 1;
25     public static final int GRANT = 2;
26     public static final int RELEASE = 3;
27     public static final int FINISH = 4;
28
29     public static final int CPU=0;
30     public static final int DISK=1;
31     public static final int ETH=2;
32     public static final int MEM=3;
33     public static final int OTHER=4;
34
35     private ArrayList<SystemStatus> sstatus;
36     private ArrayList<ArrayList<NodeStatus>> nstatus;
37     private HashMap<String,ArrayList<ProcessStatus>> pstatus;
38     private HashMap<String,ArrayList<String>> ptree;
39     private ModelS model;
40
41     protected Graphics g = null;
42     protected BufferedImage img = null;
43
44     private class SystemStatus{
45
46         double time;
47         public int solved, scheduled;
48         public SystemStatus(double time, int scheduled, int solved){
49             this.time = time;
50             this.scheduled = scheduled;
51             this.solved = solved;
52         }
53     }
54
55     private class NodeStatus{
56         public double time, cpu, disk, mem, eth, read, send, recv;
57
58         public NodeStatus(double time, double cpu, double disk, double mem, double
              eth, double read, double send, double recv){
59             this.time=time;
60             this.cpu=cpu;
```

```
61              this.disk=disk;
62              this.mem=mem;
63              this.eth=eth;
64              this.read=read;
65              this.send=send;
66              this.recv=recv;
67          }
68      }
69
70      private class ProcessStatus{
71          public double time;
72          public int op;
73          public int dev;
74          private ProcessStatus(double time, int op, int dev){
75              this.time=time;
76              this.op=op;
77              this.dev=dev;
78          }
79      }
80
81      public LogWriter(ModelS model){
82          this.model=model;
83          sstatus = new ArrayList<SystemStatus>();
84          pstatus=new HashMap<String, ArrayList<ProcessStatus>>();
85          nstatus=new ArrayList<ArrayList<NodeStatus>>();
86          ptree = new HashMap<String, ArrayList<String>>();
87          ptree.put(null, new ArrayList<String>());
88          for (int i=0; i<Config.NUMBER_OF_NODES; i++){
89              nstatus.add(new ArrayList<NodeStatus>());
90          }
91      }
92
93      public void logSystemStatus(double time, int running, int solved){
94          sstatus.add(new SystemStatus(time, running, solved));
95      }
96
97      public void logProcess(String procName, String parentProcName, double time){
98          if (ptree.get(parentProcName)==null){
99              ptree.put(parentProcName, new ArrayList<String>());
100         }
101         ptree.get(parentProcName).add(procName);
102         pstatus.put(procName, new ArrayList<ProcessStatus>());
103     }
104
105     public void logNodeStatus(int nodenr, double time, double cpu, double disk,
            double mem, double eth, double read, double send, double recv){
106         nstatus.get(nodenr).add(new NodeStatus(time, cpu, disk, mem, eth, read,
                send, recv));
107     }
108
109     public void logProcessStatus(String procName, double time, int op, int dev){
110         if (pstatus.get(procName)==null) pstatus.put(procName, new ArrayList<
                ProcessStatus>());
111         pstatus.get(procName).add(new ProcessStatus(time, op, dev));
112     }
113
114     private final int MAXW = 2000;
115     private final int MAXH = 5000;
116     private final Color TICK_COLOR = new Color(220, 220, 220, 255);
117     private final Color TICK_COLOR_BOLD = new Color(170, 170, 170, 255);
118     private final Color CPU_COLOR = new Color(102, 140, 178, 100);
119     private final Color DISK_COLOR = new Color(194, 122, 122, 100);
```

```
120     private final Color ETH_COLOR = new Color(50, 140, 50, 100);
121     private final Color MEM_COLOR = new Color(150, 150, 150, 100);
122     private final Color CPU_COLOR_NT = new Color(102, 140, 178, 200);
123     private final Color DISK_COLOR_NT = new Color(194, 122, 122, 200);
124     private final Color MEM_COLOR_NT = new Color(150, 150, 150, 200);
125     private final Color ETH_COLOR_NT = new Color(50, 140, 50, 200);
126
127     private final Color IN_COLOR = new Color(102, 140, 178, 150);
128     private final Color OUT_COLOR =  new Color(194, 122, 122, 150);
129 //  private final Color NO_COLOR =  new Color(0, 0, 0, 0);
130
131     public void renderHeader(Graphics g, int rectX, int rectY){
132         g.setColor(Color.BLACK);
133         g.setFont(new Font("SansSerif", Font.BOLD, 16));
134         g.drawString(ModelS.runName, rectX, rectY+16);
135         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
136
137         g.drawString(
138                 "("+(Config.SIMULATED_ONLY ? "simulated only" : "") +", "+
139                 (Config.USE_AND_QUERIES ? "AND-mode" : "OR-mode") +") "
140         , rectX, rectY+28);
141
142         g.setFont(new Font("SansSerif", Font.BOLD, 12));
143         g.drawString("Indexing Mode:",rectX, rectY+40);
144         g.drawString("Number of Nodes:",rectX, rectY+52);
145         g.drawString("CPUs per Node:",rectX, rectY+64);
146         g.drawString("Disks per Node:",rectX, rectY+76);
147         g.drawString("Maximum number of processes:",rectX, rectY+88);
148         g.drawString("Test duration: ",rectX, rectY+100);
149         g.drawString("Number of documents:",rectX, rectY+112);
150         g.drawString("Max number of res. required:",rectX, rectY+124);
151
152         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
153         g.drawString(Config.INDEXING_MODE+"",rectX+200, rectY+40);
154         g.drawString(Config.NUMBER_OF_NODES+"",rectX+200, rectY+52);
155         g.drawString(Config.NUMBER_OF_CPUS_PER_NODE+"",rectX+200, rectY+64);
156         g.drawString(Config.NUMBER_OF_DISKS_PER_NODE+"",rectX+200, rectY+76);
157         g.drawString(Config.MAX_NUMBER_OF_QUERY_PROCESSES+"",rectX+200, rectY+88);
158         g.drawString(Config.SIM_DURATION+"ms",rectX+200, rectY+100);
159         g.drawString(Config.NUMBER_OF_DOCUMENTS+"",rectX+200, rectY+112);
160         g.drawString(Config.NUMBER_OF_RESULTS_REQUIRED+"",rectX+200, rectY+124);
161
162         g.setFont(new Font("SansSerif", Font.BOLD, 12));
163         g.drawString("Memory per node:",rectX+290, rectY+40);
164         g.drawString("Buffer size:",rectX+290, rectY+52);
165         g.drawString("Network bandwidth:",rectX+290, rectY+64);
166         g.drawString("Network overhead:",rectX+290, rectY+76);
167         g.drawString("Disk overhead:",rectX+290, rectY+88);
168         g.drawString("Disk rotation delay: ",rectX+290, rectY+100);
169         g.drawString("Disk bandwidth:",rectX+290, rectY+112);
170
171         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
172         //g.drawString((*Config.MEMORY_BLOCK_SIZE)+"B",rectX+500, rectY+40);
173         g.drawString((int)((double)Config.MEMORY_BLOCK_SIZE/1048576*Config.
                SIZE_OF_MEMORY_PER_NODE)+"MB",rectX+490, rectY+40);
174         g.drawString(Config.BUFFER_SIZE+"",rectX+490, rectY+52);
175         g.drawString((int)(1.0/Config.NETWORK_INVERSE_BANDWIDTH)+"Bps",rectX+490,
                rectY+64);
176         g.drawString(Config.NETWORK_OVERHEAD+"ms",rectX+490, rectY+76);
177         g.drawString(Config.DISK_SEEK_TIME+"ms",rectX+490, rectY+88);
178         g.drawString(Config.DISK_ROTATION_DELAY+"",rectX+490, rectY+100);
```

```
179         g.drawString((int)(1.0/Config.DISK_INVERSE_BANDWIDTH)+"Bps",rectX+490,
                rectY+112);
180
181         g.setFont(new Font("SansSerif", Font.BOLD, 12));
182         g.drawString("Comp instr time:",rectX+600, rectY+40);
183         g.drawString("Int instr time:",rectX+600, rectY+52);
184         g.drawString("Heap instr time:",rectX+600, rectY+64);
185         g.drawString("Heap multiway instr time:",rectX+600, rectY+76);
186         g.drawString("Lookup time (mem only):",rectX+600, rectY+88);
187
188         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
189         g.drawString(Config.COMPARSION_INSTRUCTION_TIME+"ms",rectX+770, rectY+40);
190         g.drawString(Config.INTERLEAVE_TWO_LISTS_INSTRUCTION_TIME+"ms",rectX+770,
                rectY+52);
191         g.drawString(Config.HEAP_INSTRUCTION_TIME+"ms",rectX+770, rectY+64);
192         g.drawString(Config.HEAP_MULTIWAY_INTERLEAVEMERGE_TIME+"ms",rectX+770,
                rectY+76);
193         g.drawString(Config.LOOKUP_TIME+"ms",rectX+770, rectY+88);
194
195         double duration = Config.SIM_DURATION;
196         g.setFont(new Font("SansSerif", Font.BOLD, 12));
197         //g.drawString("Results",rectX, rectY+148);
198         g.drawString("Query Statistics:",rectX, rectY+160);
199         g.drawString("QPS:",rectX, rectY+172);
200         g.drawString("Query time:",rectX, rectY+184);
201         g.drawString("Query terms:",rectX, rectY+196);
202         g.drawString("Query results:",rectX, rectY+208);
203         g.drawString("Query total index size:",rectX, rectY+220);
204
205         Statistics st = model.getStatistics();
206
207         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
208         g.drawString(model.getSolved() * 1000 / duration+"",rectX+100, rectY+172);
209         g.drawString(st.queryTimes.getMean()+" ("+st.queryTimes.getMinimum()+","+
                st.queryTimes.getMaximum()+")",rectX+200, rectY+184);
210         g.drawString(st.queryNumTerms.getMean()+"("+st.queryNumTerms.getMinimum()+
                ","+st.queryNumTerms.getMaximum()+")",rectX+200, rectY+196);
211         g.drawString(st.queryNumResults.getMean()+" ("+st.queryNumResults.
                getMinimum()+","+st.queryNumResults.getMaximum()+")",rectX+200, rectY
                +208);
212         g.drawString(st.queryIndexSize.getMean()+" ("+st.queryIndexSize.getMinimum
                ()+","+st.queryIndexSize.getMaximum()+")",rectX+200, rectY+220);
213
214         ArrayList<Node> nodes = model.getNodes();
215
216         g.setFont(new Font("SansSerif", Font.BOLD, 12));
217         g.drawString("Node statistics:",rectX, rectY+244);
218         g.drawString("Node #",rectX, rectY+256);
219         g.drawString("CPU load",rectX, rectY+268);
220         g.drawString("Disk load",rectX, rectY+280);
221         g.drawString("Eth load",rectX, rectY+292);
222         g.drawString("Mem load",rectX, rectY+304);
223         g.drawString("Disk seeks",rectX, rectY+316);
224         g.drawString("Data read",rectX, rectY+328);
225         g.drawString("Data sendt",rectX, rectY+340);
226         g.drawString("Data received",rectX, rectY+352);
227
228         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
229         int i=0;
230         for (Node node : nodes) {
231             g.drawString((i++)+"",rectX+100*i, rectY+256);
232             g.drawString(node.getCPU().avgUsage()+"",rectX+100*i, rectY+268);
```

```
233             g.drawString(node.getDisk().avgUsage()+"",rectX+100*i, rectY+280);
234             g.drawString(node.getEth().avgUsage()+"",rectX+100*i, rectY+292);
235             g.drawString(node.getMemory().avgUsage()+"",rectX+100*i, rectY+304);
236             g.drawString(node.getSeeks()+"",rectX+100*i, rectY+316);
237             g.drawString(node.getDiskread()+"",rectX+100*i, rectY+328);
238             g.drawString(node.getEthsend()+"",rectX+100*i, rectY+340);
239             g.drawString(node.getEthrecv()+"",rectX+100*i, rectY+352);
240 
241         }
242 
243     }
244 
245     public void renderNodeLog(){
246         int barh = 55;
247         int barw = 50;
248         int offx = 5;
249         int offy = 5;
250         int sw = 1000;
251         int w = (int) (Config.SIM_DURATION * sw / 1000) + 2 * offx;
252         int hh = 500;
253         int h = hh + (barh * 4+50) * Config.NUMBER_OF_NODES  + 2 * offy;
254         if (w > MAXW){
255             w = MAXW;
256             sw = (w - 2 * offx) * 1000 / Config.SIM_DURATION;
257             barw=sw/10;
258             //System.out.println(barw);
259         }
260 
261         //for (int i=0;)
262         //g.drawString(uname, offX, headerY + (hoff++) * (TEXT_FONT.getSize() + 2)
                ;
263         //System.out.println(w+" "+h);
264 
265         img = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
266         g = img.createGraphics();
267         if (g instanceof Graphics2D) {
268             Map renderHints = new HashMap();
269             renderHints.put(RenderingHints.KEY_ANTIALIASING, RenderingHints.
                    VALUE_ANTIALIAS_ON);
270             renderHints.put(RenderingHints.KEY_COLOR_RENDERING, RenderingHints.
                    VALUE_COLOR_RENDER_QUALITY);
271             renderHints.put(RenderingHints.KEY_DITHERING, RenderingHints.
                    VALUE_DITHER_DISABLE);
272             renderHints.put(RenderingHints.KEY_RENDERING, RenderingHints.
                    VALUE_RENDER_QUALITY);
273             renderHints.put(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.
                    VALUE_TEXT_ANTIALIAS_ON);
274             ((Graphics2D)g).addRenderingHints(renderHints);
275         }
276 
277         g.setColor(Color.WHITE);
278         g.fillRect(0, 0, w, h);
279         g.setColor(Color.BLACK);
280         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
281 
282         renderHeader(g, offx, offy);
283         int rectX, rectY,rectW,rectH;
284 
285 
286         rectX = offx;
287         rectY = offy+400;
288         rectW = w - 2 * offx;
```

```
289         rectH = 50;
290
291
292         {//do local scoop :)
293             int xpoints[] = new int[sstatus.size()+3];
294             int ypoints1[] = new int[sstatus.size()+3];
295             int ypoints2[] = new int[sstatus.size()+3];
296
297
298             int max = sstatus.get(sstatus.size()-1).scheduled;
299             int b=0;
300             for (int j = 0; j < rectW; j += sw/10, b++) {
301                 g.setColor(b % 10==0 ? TICK_COLOR_BOLD: TICK_COLOR);
302                 g.drawLine(rectX+j, rectY, rectX + j, rectY+rectH);
303             }
304
305             int i=1;
306             for (SystemStatus systemStatus : sstatus) {
307                 xpoints[i] = rectX + (int) (systemStatus.time * rectW / Config.
                        SIM_DURATION);
308                 ypoints1[i] = rectY+(50 - (int)( (double)systemStatus.scheduled /
                        max * 50));
309                 ypoints2[i] = rectY+(50 - (int)( (double)systemStatus.solved / max
                         * 50));
310                 //System.out.println(systemStatus.solved);
311                 i++;
312             }
313             xpoints[0]=rectX;
314             xpoints[xpoints.length-2]=rectX+rectW;
315             xpoints[xpoints.length-1]=rectX+rectW;
316             ypoints1[0] = ypoints2[0] = rectY+50;
317             ypoints1[ypoints1.length-2] =  ypoints1[ypoints1.length-3];
318             ypoints2[ypoints1.length-2] =  ypoints2[ypoints2.length-3];
319             ypoints1[ypoints1.length-1] =  rectY+50;
320             ypoints2[ypoints1.length-1] =  rectY+50;
321             g.setColor(IN_COLOR);
322             g.fillPolygon(xpoints, ypoints1, sstatus.size()+3);
323             g.setColor(OUT_COLOR);
324             g.fillPolygon(xpoints, ypoints2, sstatus.size()+3);
325
326             g.setColor(Color.GRAY);
327             g.drawRect(rectX, rectY, rectW, rectH);
328         }
329
330         g.setColor(Color.BLACK);
331         g.setFont(new Font("SansSerif", Font.BOLD, 12));
332         g.drawString("Queries", rectX, rectY+66);
333         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
334
335         g.drawString("scheduled", rectX+112, rectY+66);
336         g.drawString("solved", rectX+212, rectY+66);
337
338         g.setColor(IN_COLOR);
339         g.fillRect(rectX+100, rectY+58, 8, 8);
340         g.setColor(OUT_COLOR);
341         g.fillRect(rectX+200, rectY+58, 8, 8);
342         g.setColor(Color.GRAY);
343         g.drawRect(rectX+100, rectY+58, 8, 8); g.drawRect(rectX+200, rectY+58, 8,
                8);
344
345         rectX = offx;
346         rectY = offy+hh;
```

```
347         rectW = w - 2 * offx;
348         rectH = h - 2 * offy;
349
350         for (int i=0; i<Config.NUMBER_OF_NODES; i++){
351             ArrayList<NodeStatus> ns = nstatus.get(i);
352             int[] xpoints = new int[ns.size()+2];
353             int[][] ypoints = new int[7][ns.size()+2];
354             int pi=1;
355
356             xpoints[0]=rectX;
357             xpoints[ns.size()+1]=rectX+rectW;
358             for (int k=0; k<4; k++){
359                 int b=0;
360                 for (int j = 0; j < rectW; j += sw/10, b++) {
361                     g.setColor(b % 10==0 ? TICK_COLOR_BOLD: TICK_COLOR);
362                     g.drawLine(rectX+j, rectY+(barh+5)*k+(barh*4+50)*i, rectX + j,
                            rectY+(barh+5)*k+(barh*4+50)*i+barh);
363                 }
364
365                 g.setColor(Color.GRAY);
366                 g.drawRect(rectX, rectY+(barh+5)*k+(barh*4+50)*i, rectW, barh);
367                 ypoints[k][0]=ypoints[k][ns.size()+1] = rectY+(barh+5)*k+(barh
                        *4+50)*i+barh;
368             }
369
370             ypoints[4][0]=ypoints[4][ns.size()+1] = rectY+(barh+5)+(barh*4+50)*i+
                    barh;
371             ypoints[5][0]=ypoints[5][ns.size()+1]=ypoints[6][0]=ypoints[6][ns.size
                    ()+1]=
372                 rectY+(barh+5)*3+(barh*4+50)*i+barh;
373
374             g.setColor(Color.GRAY);
375             for (NodeStatus nodeStatus : ns) {
376                 xpoints[pi] = rectX + (int) (nodeStatus.time * rectW / Config.
                        SIM_DURATION);
377                 ypoints[0][pi] = rectY+(barh*4+50)*i+barh - (int)( nodeStatus.cpu
                        * barh);                //cpu
378                 ypoints[1][pi] = rectY+(barh+5)+(barh*4+50)*i+barh - (int)(
                        nodeStatus.disk * barh);    //disk
379                 ypoints[2][pi] = rectY+(barh+5)*2+(barh*4+50)*i+barh - (int)(
                        nodeStatus.mem * barh);   //memory
380                 ypoints[3][pi] = rectY+(barh+5)*3+(barh*4+50)*i+barh - (int)(
                        nodeStatus.eth * barh);   //eth
381                 ypoints[4][pi] = rectY+(barh+5)+(barh*4+50)*i+barh - (int)(
                        nodeStatus.read * barh);    //disk
382                 ypoints[5][pi] = rectY+(barh+5)*3+(barh*4+50)*i+barh - (int)(
                        nodeStatus.recv * barh);  //eth
383                 ypoints[6][pi] = rectY+(barh+5)*3+(barh*4+50)*i+barh - (int)(
                        nodeStatus.send* barh);   //eth
384                 g.setColor(Color.GRAY);
385                 for(int k=0; k<4; k++){
386                     if (pi>0) g.drawLine(xpoints[pi-1], ypoints[k][pi-1], xpoints[
                            pi], ypoints[k][pi]);
387                 }
388
389                 pi++;
390             }
391
392             g.setColor(Color.GRAY);
393             for(int k=0; k<7; k++){
394                 g.drawLine(xpoints[ns.size()],ypoints[k][ns.size()], xpoints[ns.
                        size()+1], ypoints[k][ns.size()+1]);
```

```
395             }
396
397             g.setColor(OUT_COLOR);
398             g.fillPolygon(xpoints, ypoints[4], ns.size()+2);
399             g.fillPolygon(xpoints, ypoints[6], ns.size()+2);
400             //g.setColor(NO_COLOR);
401             g.setColor(IN_COLOR);
402             g.fillPolygon(xpoints, ypoints[5], ns.size()+2);
403
404             g.setColor(CPU_COLOR);
405             g.fillPolygon(xpoints, ypoints[0], ns.size()+2);
406             g.setColor(DISK_COLOR);
407             g.fillPolygon(xpoints, ypoints[1], ns.size()+2);
408             g.setColor(MEM_COLOR);
409             g.fillPolygon(xpoints, ypoints[2], ns.size()+2);
410             g.setColor(ETH_COLOR);
411             g.fillPolygon(xpoints, ypoints[3], ns.size()+2);
412
413             g.setColor(Color.BLACK);
414             int ly = rectY+(barh+5)*4 + (barh*4+50)*i+12;
415             int lx = rectX;
416             g.setFont(new Font("SansSerif", Font.BOLD, 12));
417             g.drawString("Node"+i, lx, ly);
418
419             g.setFont(new Font("SansSerif", Font.PLAIN, 12));
420             g.drawString("CPU␣load", lx+112, ly);
421             g.drawString("disk␣load", lx+212, ly);
422             g.drawString("memory␣load", lx+312, ly);
423             g.drawString("ethernet␣load", lx+412, ly);
424             g.drawString("(additional", lx+532, ly);
425             g.drawString("in", lx+612, ly);
426             g.drawString("out)", lx+642, ly);
427
428             g.setColor(CPU_COLOR);
429             g.fillRect(lx+100, ly-8, 8, 8);
430             g.setColor(DISK_COLOR);
431             g.fillRect(lx+200, ly-8, 8, 8);
432             g.setColor(MEM_COLOR);
433             g.fillRect(lx+300, ly-8, 8, 8);
434             g.setColor(ETH_COLOR);
435             g.fillRect(lx+400, ly-8, 8, 8);
436             g.setColor(IN_COLOR);
437             g.fillRect(lx+600, ly-8, 8, 8);
438             g.setColor(OUT_COLOR);
439             g.fillRect(lx+630, ly-8, 8, 8);
440             g.setColor(Color.GRAY);
441             g.drawRect(lx+100, ly-8, 8, 8); g.drawRect(lx+200, ly-8, 8, 8);
442             g.drawRect(lx+300, ly-8, 8, 8); g.drawRect(lx+400, ly-8, 8, 8);
443             g.drawRect(lx+600, ly-8, 8, 8); g.drawRect(lx+630, ly-8, 8, 8);
444         }
445
446         try {
447             ImageIO.write(img, "png", new File(ModelS.runName+"_node.png"));
448         } catch (IOException e) {
449             e.printStackTrace();
450         }
451     }
452
453     public ArrayList<String> prefixTrace(String parentProcName){
454         ArrayList<String> ans= new ArrayList<String>();
455         ArrayList<String> tmp=ptree.get(parentProcName);
456         if (tmp == null) return ans;
```

```
457
458         for (String name : tmp) {
459             ans.add(name);
460             ans.addAll(prefixTrace(name));
461         }
462
463         return ans;
464     }
465
466
467     public void renderProcLog() {
468
469         ArrayList<String> names = prefixTrace(null);
470
471         int barh = 12;
472         int barw = 50;
473         int hh = 0;
474         int offx = 5;
475         int offy = 5;
476
477         int sw = 1000;
478
479         int w = (int) (Config.SIM_DURATION * sw / 1000) + 2 * offx + 70 ;
480         int h = barh * names.size() + hh + 2 * offy;
481
482         if (w > MAXW+70){
483             w = MAXW+70;
484             sw = (w - 2 * offx - 70) * 1000 / Config.SIM_DURATION;
485             barw=sw/10;
486         }
487
488         if (h > MAXH){
489             h = MAXH;
490             barh = (h - 2 * offy - hh) / names.size();
491         }
492
493         img = new BufferedImage(w, h, BufferedImage.TYPE_INT_RGB);
494         g = img.createGraphics();
495
496         if (g instanceof Graphics2D) {
497             // set best quality rendering
498             Map renderHints = new HashMap();
499             renderHints.put(RenderingHints.KEY_ANTIALIASING, RenderingHints.
                    VALUE_ANTIALIAS_ON);
500             renderHints.put(RenderingHints.KEY_COLOR_RENDERING, RenderingHints.
                    VALUE_COLOR_RENDER_QUALITY);
501             renderHints.put(RenderingHints.KEY_DITHERING, RenderingHints.
                    VALUE_DITHER_DISABLE);
502             renderHints.put(RenderingHints.KEY_RENDERING, RenderingHints.
                    VALUE_RENDER_QUALITY);
503             renderHints.put(RenderingHints.KEY_TEXT_ANTIALIASING, RenderingHints.
                    VALUE_TEXT_ANTIALIAS_ON);
504             ((Graphics2D)g).addRenderingHints(renderHints);
505         }
506
507         g.setColor(Color.WHITE);
508         g.fillRect(0, 0, w, h);
509         g.setColor(Color.BLACK);
510         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
511
512         int rectX = offx;
513         int rectY = offy+hh;
```

```
514         int rectW = w - 2 * offx-70;
515         int rectH = h - 2 * offy;
516         double scale = (double)(rectW)/Config.SIM_DURATION;
517 
518         int b=0;
519         for (int j = 0; j < rectW; j += sw/10, b++) {
520             g.setColor(b % 10==0 ? TICK_COLOR_BOLD: TICK_COLOR);
521             g.drawLine(rectX+j, rectY, rectX + j, rectY+rectH);
522         }
523 
524         for (int i=0; i<names.size(); i++){
525             String procName = names.get(i);
526             ArrayList<ProcessStatus> st = pstatus.get(procName);
527             int cnt = st.size();
528 
529             ProcessStatus tmp = st.get(0);
530             double starttime = (tmp.op==START) ? tmp.time : 0.0;
531 
532             tmp = st.get(cnt-1);
533             double endtime = (tmp.op==FINISH) ? tmp.time : Config.SIM_DURATION;
534 
535             double lasttime, nexttime;
536             for (int j=0; j<cnt; j++){
537                 tmp=st.get(j);
538                 lasttime=tmp.time;
539                 nexttime = ((j+1<cnt) ? st.get(j+1).time : Config.SIM_DURATION);
540 
541                 if (tmp.op == WAIT || tmp.op == GRANT){
542                     if (tmp.dev==CPU){
543                         if (tmp.op==WAIT) g.setColor(CPU_COLOR);
544                         else g.setColor(CPU_COLOR_NT);
545                     } else if (tmp.dev==DISK){
546                         if (tmp.op==WAIT) g.setColor(DISK_COLOR);
547                         else g.setColor(DISK_COLOR_NT);
548                     } else  if (tmp.dev==ETH){
549                         if (tmp.op==WAIT) g.setColor(ETH_COLOR);
550                         else g.setColor(ETH_COLOR_NT);
551                     } else  if (tmp.dev==MEM){
552                         if (tmp.op==WAIT) g.setColor(MEM_COLOR);
553                         else continue; //nothing!
554                     }
555 
556                     g.fillRect((int)(rectX+lasttime*scale), rectY+i*barh, Math.max
                            ((int)((nexttime-lasttime)*scale),1), barh);
557                 }
558             }
559             g.setColor(Color.DARK_GRAY);
560             g.drawRect((int)(rectX+starttime*scale), rectY+i*barh, (int)((endtime-
                    starttime)*scale), barh);
561             g.setColor(Color.BLACK);
562             g.setFont(new Font("SansSerif", Font.BOLD, barh));
563             g.drawString(procName, (int)(rectX+2+endtime*scale), rectY+(i+1)*barh)
                    ;
564         }
565         g.drawRect(rectX, rectY, rectW, rectH);
566 
567 
568         int ly = offy+rectH-10;
569 
570         g.setColor(Color.BLACK);
571         g.setFont(new Font("SansSerif", Font.BOLD, 12));
572         g.drawString("Processes", rectX+20, ly);
```

```
573
574         g.setFont(new Font("SansSerif", Font.PLAIN, 12));
575         g.drawString("CPU request", rectX+112, ly);
576         g.drawString("CPU access", rectX+212, ly);
577         g.drawString("Disk request", rectX+312, ly);
578         g.drawString("Disk access", rectX+412, ly);
579         g.drawString("Network request", rectX+512, ly);
580         g.drawString("Network access", rectX+637, ly);
581         g.drawString("Memory request", rectX+762, ly);
582
583         g.setColor(CPU_COLOR);
584         g.fillRect(rectX+100, ly-8, 8, 8);
585         g.setColor(CPU_COLOR_NT);
586         g.fillRect(rectX+200, ly-8, 8, 8);
587         g.setColor(DISK_COLOR);
588         g.fillRect(rectX+300, ly-8, 8, 8);
589         g.setColor(DISK_COLOR_NT);
590         g.fillRect(rectX+400, ly-8, 8, 8);
591         g.setColor(ETH_COLOR);
592         g.fillRect(rectX+500, ly-8, 8, 8);
593         g.setColor(ETH_COLOR_NT);
594         g.fillRect(rectX+625, ly-8, 8, 8);
595         g.setColor(MEM_COLOR);
596         g.fillRect(rectX+750, ly-8, 8, 8);
597
598         g.setColor(Color.GRAY);
599         g.drawRect(rectX+100, ly-8, 8, 8);
600         g.drawRect(rectX+200, ly-8, 8, 8);
601         g.drawRect(rectX+300, ly-8, 8, 8);
602         g.drawRect(rectX+400, ly-8, 8, 8);
603         g.drawRect(rectX+500, ly-8, 8, 8);
604         g.drawRect(rectX+625, ly-8, 8, 8);
605         g.drawRect(rectX+750, ly-8, 8, 8);
606
607         try {
608             ImageIO.write(img, "png", new File(ModelS.runName+"_process.png"));
609         } catch (IOException e) {
610             e.printStackTrace();
611         }
612     }
613
614
615 }
```

B.1.7 simulation.log.Statistics

```
1  package simulation.log;
2  import java.io.FileWriter;
3  import java.io.PrintWriter;
4  import java.util.ArrayList;
5
6  import desmoj.core.advancedModellingFeatures.Bin;
7  import desmoj.core.simulator.SimTime;
8  import desmoj.core.statistic.*;
9
10 import simulation.model.Config;
11 import simulation.model.ModelS;
12 import simulation.node.Node;
13 import simulation.query.QueryResult;
14
15 public class Statistics {
16     private ModelS model;
```

```
17
18      public Histogram queryNumTerms;
19      public Histogram queryNumResults;
20      public Histogram queryIndexSize;
21      public Histogram queryTimes;
22
23      public Statistics(ModelS model){
24          this.model=model;
25          queryNumTerms = new Histogram(model,"Number␣of␣terms␣per␣query"
                ,1.0,10.0,10,true,false);
26          queryNumResults = new Histogram(model,"Number␣of␣results␣per␣query"
                ,0.0,1000000.0,20,true,false);
27          queryIndexSize = new Histogram(model,"Total␣index␣size␣per␣query",0.0,
28                  Config.NUMBER_OF_DOCUMENTS*Config.SIZE_OF_INDEX_DOCUMENT_ENTRY,20,
                        true,false);
29          queryTimes = new Histogram(model,"Query␣Times",0.0, Config.SIM_DURATION,
                10, true,false);
30      }
31
32      public void addQueryTerms(int terms){
33          queryNumTerms.update(terms);
34      }
35
36      public void addQueryResults(int results){
37          queryNumResults.update(results);
38      }
39
40      public void addQueryIndexSize(double sz){
41          queryIndexSize.update(sz);
42      }
43
44      public void addQueryTime(SimTime t){
45          queryTimes.update(t.getTimeValue());
46      }
47
48      public void storeAll(){
49          try {
50              PrintWriter pwr = new PrintWriter(new FileWriter(ModelS.runName+".log"
                    ));
51              double duration = SimTime.diff(model.currentTime(), model.resetAt()).
                    getTimeValue();
52              pwr.println(ModelS.runName+"␣:␣experiment␣results");
53              pwr.println("Test␣Duration");
54              pwr.println(duration);
55              pwr.println("");
56              pwr.println("Query␣Statistics");
57              pwr.println("QPS␣\t" + model.getSolved() * 1000 / duration+"\n");
58              pwr.println("Query␣time␣(ms)\t"+queryTimes.getMean()+"␣"+queryTimes.
                    getMinimum()+"␣"+queryTimes.getMaximum());
59              pwr.println("Query␣terms␣\t"+queryNumTerms.getMean()+"␣"+queryNumTerms
                    .getMinimum()+"␣"+queryNumTerms.getMaximum());
60              pwr.println("Query␣results␣\t"+queryNumResults.getMean()+"␣"+
                    queryNumResults.getMinimum()+"␣"+queryNumResults.getMaximum());
61              pwr.println("Query␣index␣size␣\t"+queryIndexSize.getMean()+"␣"+
                    queryIndexSize.getMinimum()+"␣"+queryIndexSize.getMaximum());
62              pwr.println("");
63
64              ArrayList<Node> nodes = model.getNodes();
65              pwr.println("Node␣Statistics:␣\t␣CPU␣load␣\t␣Disk␣load␣\t␣Eth␣load␣\t␣
                    Mem␣load␣\t␣Disk␣seeks␣\t␣Data␣read␣\t␣Data␣sendt␣\t␣Data␣received
                    ");
66              int i=0;
```

```
67              for (Node node : nodes) {
68                  pwr.println((i++)+"\t"+
69                          node.getCPU().avgUsage()+"\t"+
70                          node.getDisk().avgUsage()+"\t"+
71                          node.getEth().avgUsage()+"\t"+
72                          node.getMemory().avgUsage()+"\t"+
73                          node.getSeeks()+"\t"+
74                          node.getDiskread()+"\t"+
75                          node.getEthsend()+"\t"+
76                          node.getEthrecv()
77                  );
78              }
79              pwr.close();
80          } catch (Exception e) {
81              e.printStackTrace();
82          }
83      }
84  }
```

B.1.8 simulation.micro.HeapInterleaveTest

```
1   package simulation.micro;
2   import java.util.Random;
3
4   import simulation.model.Config;
5
6   public class HeapInterleaveTest {
7
8       private static class PHeap{
9           private int[] data;
10          private int last, heapsize;
11
12
13          public PHeap(int[] data){
14              this.data = data;
15              this.last = data.length-2;
16              this.heapsize=data.length/2;
17              buildHeap();
18          }
19
20          private void buildHeap(){
21              for (int i=(heapsize-1)/2; i>=0; i--) heapify(i);
22          }
23
24          private void heapify(int p){
25              int l = p*2+1;
26              int r = p*2+2;
27
28              if ( l >= heapsize) return;
29
30              int n = (r >= heapsize) ? l : ((data[l*2] < data[r*2]) ? l : r);
31
32              if (data[p*2] > data[n*2]) {
33                  int t1 = data[p*2];
34                  int t2 = data[p*2+1];
35
36                  data[p*2] = data[n*2];
37                  data[p*2+1] = data[n*2+1];
38                  data[n*2] = t1;
39                  data[n*2+1] = t2;
40
41                  heapify(n);
```

```
42              }
43          }
44
45          public int getFirstValue(){
46              /*
47              for (int i=0;i<heapsize; i++){
48                  System.out.print(data[i*2]+":"+data[i*2+1]+" ");
49              }
50              System.out.println();
51              */
52              return data[1];
53          }
54
55          public void setFirst(int key, int value){
56              data[0] = key;
57              data[1] = value;
58              heapify(0);
59          }
60
61          public void removeFirst(){
62              heapsize--;
63              data[0] = data[heapsize*2];
64              data[1] = data[heapsize*2+1];
65              heapify(0);
66          }
67      }
68
69      public static void test(int runs){
70          double time=0.0;
71          int listnum = 4;
72
73          for (int j=0; j<runs;j++){
74              int num = 200000;
75              Random r = new Random(42);
76              int lists[][] = new int[listnum][num];
77
78              int last[]={0,0,0,0,0,0,0,0};;
79              for (int k=0; k<listnum; k++){
80                  for (int i=0; i<num; i+=2){
81                      last[k]+=r.nextInt(10);
82                      lists[k][i] = last[k];
83                      lists[k][i+1] = (int) ( r.nextDouble() * Config.
                            NUMBER_OF_WORDS_PER_DOCUMENT);
84                  }
85              }
86
87              Runtime.getRuntime().freeMemory();
88              long start = System.nanoTime();
89              int reslist[] = new int[num*listnum];
90              int p[] = {0,0,0,0,0,0,0,0};
91              int pr = 0;
92
93              int done = 0;
94
95              int heap[] = new int[listnum*2];
96
97              for (int i=0; i<listnum; i++){
98                  heap[i*2] = lists[i][0];
99                  heap[i*2+1] = i;
100             }
101
102             PHeap pheap = new PHeap(heap);
```

```
103
104             int lastdoc=-1;
105             int acc = 0;
106             while (done<listnum){
107                 int listnr = pheap.getFirstValue();
108
109                 int docnr = lists[listnr][p[listnr]];
110                 int val = lists[listnr][p[listnr]+1];
111
112                 if (docnr==lastdoc){
113                     acc+=val;
114                 } else{
115                     if (lastdoc!=-1){
116                         reslist[pr++] = lastdoc;
117                         reslist[pr++] = acc;
118                     }
119                     lastdoc=docnr;
120                     acc=val;
121                 }
122                 p[listnr]+=2;
123
124                 if (p[listnr]<num){
125                     pheap.setFirst(lists[listnr][p[listnr]], listnr);
126                 } else {
127                     pheap.removeFirst();
128                     done++;
129                 }
130             }
131
132             /*
133             for (int i=0; i<pr; i+=2){
134                 System.out.println(reslist[i] + ":" + reslist[i+1]+"+");
135             }
136             */
137
138             long end = System.nanoTime();
139             time+=((double)(end-start))/(num*listnum/2) / (Math.log(listnum)/Math.
                    log(2));
140         }
141         System.out.println(time/runs / 1000000 );
142     }
143
144     public static void main(String args[]){
145         test(100);
146         test(100);
147         test(100);
148         test(100);
149         test(100);
150     }
151 }
```

B.1.9 simulation.micro.InterleaveTwoTest

```
1  package simulation.micro;
2  import java.util.Random;
3
4  import simulation.model.Config;
5  import simulation.query.IndexTools;
6  import simulation.query.SimpleIndexHit;
7  import simulation.query.SimpleIndexHitList;
8
9
```

```
10 public class InterleaveTwoTest {
11     public static void test(int runs){
12         double time=0.0;
13 
14         for (int j=0; j<runs;j++){
15             int num = 200000;
16             Random r = new Random(42);
17             int lista[] = new int[num];
18             int listb[] = new int[num];
19 
20 
21             int lasta=0;
22             int lastb=0;
23             for (int i=0; i<num; i+=2){
24                 lasta+=r.nextInt(10);
25                 lastb+=r.nextInt(10);
26 
27                 lista[i] = lasta;
28                 lista[i+1] = (int) ( r.nextDouble() * Config.
                       NUMBER_OF_WORDS_PER_DOCUMENT);
29                 listb[i] = lastb;
30                 listb[i+1] = (int) ( r.nextDouble() * Config.
                       NUMBER_OF_WORDS_PER_DOCUMENT);
31             }
32 
33             Runtime.getRuntime().freeMemory();
34             long start = System.nanoTime();
35             int listc[] = new int[num*2];
36             int a=0, b=0, c=0;
37             while (a<num && b<num){
38                 if (lista[a] < listb[b]){
39                     listc[c]= lista[a];
40                     listc[c+1] = lista[a+1];
41                     a+=2;
42                     c+=2;
43                 } else if (lista[a] > listb[b]){
44                     listc[c]= lista[b];
45                     listc[c+1] = lista[b+1];
46                     b+=2;
47                     c+=2;
48                 } else {
49                     listc[c]= lista[a];
50                     listc[c+1] = lista[a+1]+lista[b+1];
51                     a+=2;
52                     b+=2;
53                     c+=2;
54                 }
55             }
56 
57             while (a<num){
58                 listc[c]= lista[a];
59                 listc[c+1] = lista[a+1];
60                 a+=2;
61                 c+=2;
62             }
63 
64             while (b<num){
65                 listc[c]= lista[b];
66                 listc[c+1] = lista[b+1];
67                 b+=2;
68                 c+=2;
69             }
```

```
70
71              long end = System.nanoTime();
72              time+=((double)(end-start))/(num);
73          }
74          System.out.println(time/runs / 1000000);
75      }
76
77      public static void main(String args[]){
78          test(1000);
79          test(1000);
80          test(1000);
81          test(1000);
82          test(1000);
83      }
84
85  }
```

B.1.10 simulation.node.Node

```
1   package simulation.node;
2
3   import simulation.log.LogWriter;
4   import simulation.model.Config;
5   import simulation.model.ModelS;
6   import desmoj.core.advancedModellingFeatures.*;
7   import desmoj.core.simulator.*;
8
9   public class Node {
10      private ModelS owner;
11      private Res cpu;
12      private Res disk;
13      private Res eth;
14      private Res mem;
15      private int nodenr;
16
17      private int seeks=0;
18      private double diskread=0;
19      private double ethrecv=0;
20      private double ethsend=0;
21
22      //private double allocatedmemfree=0;
23      private MemHandler memhandler;
24
25      public Node(ModelS owner, int nodenr, int disks, int cpus, int mem){
26          this.owner=owner;
27          this.nodenr = nodenr;
28          this.cpu = new Res(owner,"CPU"+nodenr,cpus,true,true);
29          this.eth = new Res(owner,"ETH"+nodenr,1,true,true);
30          this.disk = new Res(owner,"DISK"+nodenr,disks,true,true);
31          this.mem = new Res(owner,"MEM"+nodenr,mem,true,true);
32          memhandler = new MemHandler(this, owner, "memhandler"+nodenr, false);
33          memhandler.activate(new SimTime(0.0));
34      }
35
36      public void resetCounters(){
37          seeks=0;
38          diskread=0;
39          ethrecv=0;
40          ethsend=0;
41      }
42
43      public Res getCPU(){
```

```
44          return cpu;
45      }
46
47      public Res getDisk(){
48          return disk;
49      }
50
51      public Res getMemory(){
52          return mem;
53      }
54
55      public Res getEth(){
56          return eth;
57      }
58
59      public int getNumber(){
60          return nodenr;
61      }
62
63      public SimTime lookupTermsMemOnly(SimProcess process, int numberofterms){
64          SimTime totalTime = new SimTime(0.0);
65          for (int i=0; i<numberofterms; i++){
66              SimTime lookupTime = new SimTime(Config.LOOKUP_TIME);
67              totalTime = SimTime.add(totalTime, lookupTime);
68              holdCPU(process, Config.LOOKUP_TIME);
69          }
70
71          if (numberofterms>1)
72              holdCPU(process, numberofterms * Math.log(numberofterms)/ Math.log(2)
                    *
73                      Config.COMPARSION_INSTRUCTION_TIME);
74          return totalTime;
75      }
76
77      public SimTime lookupTerms(SimProcess process, int numberofterms){
78          SimTime totalTime = new SimTime(0.0);
79          for (int i=0; i<numberofterms; i++){
80              totalTime = SimTime.add(totalTime, fetch(process, Config.BUFFER_SIZE))
                    ;
81              SimTime lookupTime = new SimTime(Config.LOOKUP_TIME);
82              totalTime = SimTime.add(totalTime, lookupTime);
83              holdCPU(process, Config.LOOKUP_TIME);
84
85          }
86
87          if (numberofterms>1)
88              holdCPU(process, numberofterms * Math.log(numberofterms)/ Math.log(2)
                    *
89                      Config.COMPARSION_INSTRUCTION_TIME);
90          return totalTime;
91      }
92
93      public SimTime fetch(SimProcess process, double size){
94          SimTime transferTime = new SimTime(Config.DISK_SEEK_TIME + Config.
                DISK_ROTATION_DELAY +
95                  Config.DISK_INVERSE_BANDWIDTH* size);
96
97          if (Config.DEMO_MODE)
98              owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(),
99                  LogWriter.WAIT, LogWriter.DISK);
100         disk.provide(1);
```

```
101         if (Config.DEMO_MODE)
102             owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(),
103                 LogWriter.GRANT, LogWriter.DISK);
104         process.hold(transferTime);
105         disk.takeBack(1);
106         if (Config.DEMO_MODE)
107             owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(),
108                 LogWriter.RELEASE, LogWriter.DISK);
109 
110         diskread+=size;
111         seeks++;
112         return transferTime;
113     }
114 
115     public SimTime transfer(Node src, Node dest, SimProcess process, double size){
116         if (src==dest) return new SimTime(0.0);
117         SimTime transferTime = new SimTime(Config.NETWORK_OVERHEAD + Config.
                NETWORK_INVERSE_BANDWIDTH * size);
118         ResHandler rh = ((ModelS) owner).getResHandler();
119 
120         if (Config.DEMO_MODE)
121             owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(), LogWriter.WAIT, LogWriter.ETH);
122         while (!rh.atomicResOp(process, new Res[]{src.getEth(),dest.getEth()}, new
                 int[]{1,1}, true)){
123                 process.passivate();
124         }
125         if (Config.DEMO_MODE)
126             owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(), LogWriter.GRANT, LogWriter.ETH);
127         process.hold(transferTime);
128         rh.atomicResOp(process, new Res[]{src.getEth(), dest.getEth()}, new int
                []{1,1}, false);
129         if (Config.DEMO_MODE)
130             owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(), LogWriter.RELEASE, LogWriter.ETH);
131 
132         src.ethsend+=size;
133         dest.ethrecv+=size;
134 
135         return transferTime;
136     }
137 
138     public SimTime heapInterleave(SimProcess process, int elements, int lists){
139         double totaltime = elements * Config.HEAP_MULTIWAY_INTERLEAVEMERGE_TIME *
                Math.log(lists) / Math.log(2);
140         holdCPU(process, totaltime);
141         return new SimTime(totaltime);
142     }
143 
144     public SimTime interleaveTwoLists(SimProcess process, int elements){
145         double totaltime = elements * Config.INTERLEAVE_TWO_LISTS_INSTRUCTION_TIME
                ;
146         holdCPU(process, totaltime);
147         return new SimTime(totaltime);
148     }
149 
150     public SimTime heapMerge(SimProcess process, int elements, int lists){
151         double totaltime = elements * Config.HEAP_MULTIWAY_INTERLEAVEMERGE_TIME *
152                 Math.log(lists) / Math.log(2);
```

```
153         holdCPU(process, totaltime);
154         return new SimTime(totaltime);
155     }
156 
157     public SimTime heapTopAndSort(SimProcess process, int elements, int results){
158         double totaltime = (elements + 2 * results) *
159         Config.HEAP_INSTRUCTION_TIME * Math.log(results) / Math.log(2) ;
160         holdCPU(process, totaltime);
161         return new SimTime(totaltime);
162     }
163 
164     public void holdCPU(SimProcess process, double totaltime){
165         int slices = (int) (totaltime/Config.CPU_SLICE_SIZE);
166         double last = totaltime - slices * Config.CPU_SLICE_SIZE;
167         for (int i=0; i<=slices+1; i++){
168             if (Config.DEMO_MODE)
169                 owner.getLogWriter().logProcessStatus(process.getName(), process.
                        currentTime().getTimeValue(),
170                         LogWriter.WAIT, LogWriter.CPU);
171             cpu.provide(1);
172             if (Config.DEMO_MODE)
173                 owner.getLogWriter().logProcessStatus(process.getName(), process.
                        currentTime().getTimeValue(),
174                         LogWriter.GRANT, LogWriter.CPU);
175             process.hold(new SimTime(i==slices?last:Config.CPU_SLICE_SIZE));
176             cpu.takeBack(1);
177             if (Config.DEMO_MODE)
178                 owner.getLogWriter().logProcessStatus(process.getName(), process.
                        currentTime().getTimeValue(),
179                         LogWriter.RELEASE, LogWriter.CPU);
180         }
181     }
182 
183     public void malloc(SimProcess process, double size){
184         /*System.out.println(size/Config.MEMORY_BLOCK_SIZE);
185         if (allocatedmemfree < size){
186 
187 
188             mem.provide((int)Math.ceil(size/Config.MEMORY_BLOCK_SIZE));
189 
190 
191             allocatedmemfree +=  Math.ceil(size/Config.MEMORY_BLOCK_SIZE);
192         }
193         allocatedmemfree -=size;
194         */
195         if (Config.DEMO_MODE)
196             owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(),
197                     LogWriter.WAIT, LogWriter.MEM);
198         memhandler.malloc(process, size);
199         if (Config.DEMO_MODE)
200             owner.getLogWriter().logProcessStatus(process.getName(), process.
                    currentTime().getTimeValue(),
201                     LogWriter.GRANT, LogWriter.MEM);
202 
203     }
204 
205     public void free(double size){
206         /*
207         allocatedmemfree += ((int) size) % Config.MEMORY_BLOCK_SIZE;
208         mem.takeBack((int)(size/Config.MEMORY_BLOCK_SIZE));
209         */
```

```
210         memhandler.free(size);
211     }
212 
213     public double getEthrecv() {
214         return ethrecv;
215     }
216 
217     public double getEthsend() {
218         return ethsend;
219     }
220 
221     public double getDiskread() {
222         return diskread;
223     }
224 
225     public int getSeeks(){
226         return seeks;
227     }
228 
229     private class MemRequest extends Entity{
230         public SimProcess process;
231         public double size;
232 
233         public MemRequest(Model owner, String name, boolean showInTrace) {
234             super(owner, name, showInTrace);
235         }
236 
237         public MemRequest(SimProcess process, double size, Model owner, String
                name, boolean showInTrace) {
238             super(owner, name, showInTrace);
239             this.process = process;
240             this.size = size;
241         }
242     }
243 
244     private class MemHandler extends SimProcess {
245         public Queue reqFree;
246         public Queue reqMalloc;
247         public double disposable = 0;
248         public Node node;
249 
250 
251         public MemHandler(Model owner, String name, boolean showInTrace) {
252             super(owner, name, showInTrace);
253         }
254 
255         public MemHandler(Node node, Model owner, String name, boolean showInTrace
                ) {
256             super(owner, name, showInTrace);
257             this.node = node;
258             reqFree = new Queue(owner, "free", false, false);
259             reqMalloc = new Queue(owner, "malloc", false, false);
260         }
261 
262         public void free(double size){
263             reqFree.insert(new MemRequest(null, size,getModel(),"",false));
264             this.activate(new SimTime(0.0));
265         }
266 
267         public void malloc(SimProcess process, double size){
268             reqMalloc.insert(new MemRequest(process,size,getModel(),"",false));
269             this.activate(new SimTime(0.0));
```

```
270             process.passivate();
271
272         }
273
274         @Override
275         public void lifeCycle() {
276             while (true){
277                 //System.out.println("asdas");
278                 while (reqFree.first() != null){
279                     MemRequest req = (MemRequest) reqFree.first();
280                     double size = req.size;
281                     disposable += size;
282                     int blocks = (int) disposable / Config.MEMORY_BLOCK_SIZE;
283                     disposable %= Config.MEMORY_BLOCK_SIZE;
284                     if (blocks>0) node.mem.takeBack(blocks);
285
286                     reqFree.remove(req);
287                 }
288
289                 MemRequest req = (MemRequest) reqMalloc.first(), next;
290                 while (req != null){
291                     next = (MemRequest) reqMalloc.succ(req);
292                     double size = req.size;
293                     if (disposable >= size){
294                         disposable -= size;
295                         reqMalloc.remove(req);
296                         req.process.activate(new SimTime(0.0));
297
298                     } else {
299                         int blocks = (int) Math.ceil(size / Config.
                                MEMORY_BLOCK_SIZE);
300                         if (blocks < node.mem.getAvail()){
301                             node.mem.provide(blocks);
302                             disposable = size % Config.MEMORY_BLOCK_SIZE;
303                             reqMalloc.remove(req);
304                             req.process.activate(new SimTime(0.0));
305                         }
306                     }
307                     req = next;
308                 }
309                 this.passivate();
310             }
311         }
312     }
313 }
```

B.1.11 simulation.node.ResHandler

```
1  package simulation.node;
2  import java.util.*;
3
4  import desmoj.core.advancedModellingFeatures.Res;
5  import desmoj.core.simulator.*;
6
7
8  public class ResHandler {
9      private ArrayList<ProcRecord> regprocs;
10     public ResHandler(){
11         regprocs = new ArrayList<ProcRecord>();
12     }
13
14     public void checkWaiting(){
```

```
15          for (int i=0; i<regprocs.size(); i++) {
16              ProcRecord rec = regprocs.get(i);
17              boolean ok=true;
18              for (int j=0; j<rec.ress.length; j++){
19                  if ( rec.ress[j].getAvail() < rec.cnts[j]){
20                      ok = false;
21                  }
22              }
23              if (ok){
24                  rec.process.activate(new SimTime(0.0));
25                  regprocs.remove(i--);
26              }
27          }
28      }
29
30      public synchronized boolean atomicResOp(SimProcess proc, Res[] ress, int[]
            cnts, boolean provide){
31          if (provide){
32              boolean ok= true;
33              for (int i=0; i<ress.length; i++){
34                  if ( ress[i].getAvail() < cnts[i]){
35                      ok = false;
36                  }
37              }
38              if (ok){
39                  for (int i=0; i<ress.length; i++){
40                      ress[i].provide(cnts[i]);
41                  }
42                  return true;
43              } else {
44                  regprocs.add(new ProcRecord(proc,ress,cnts));
45                  return false;
46              }
47          } else {
48              for (int i=0; i<ress.length; i++){
49                  ress[i].takeBack(cnts[i]);
50              }
51              checkWaiting();
52              return true;
53          }
54      }
55
56      private class ProcRecord{
57          public SimProcess process;
58          public Res[] ress;
59          public int[] cnts;
60          public ProcRecord(SimProcess sp, Res[] r, int[] c){
61              process=sp;
62              ress=r;
63              cnts=c;
64          }
65      }
66  }
```

B.1.12 simulation.processes.QueryProcess

```
1  package simulation.processes;
2  import java.util.ArrayList;
3
4  import simulation.model.Config;
5  import simulation.model.ModelS;
6  import simulation.node.Node;
```

```
7  import simulation.query.Query;
8  import simulation.query.QueryResult;
9  import simulation.query.SubQueryResult;
10
11
12 import desmoj.core.simulator.*;
13
14
15 public abstract class QueryProcess extends SimProcess{
16     protected Query query;
17     protected Node node;
18     protected int subcnt;
19     protected Queue resultQueue;
20     protected QueryResult queryResult;
21     protected SimTime startTime;
22
23     public QueryProcess(Model owner, String name, boolean showInTrace) {
24         super(owner, name, showInTrace);
25     }
26
27     public QueryProcess(Model owner, String name, boolean showInTrace, Query query
           , Node node) {
28         super(owner, name, showInTrace);
29         this.query=query;
30         this.node=node;
31         resultQueue = new Queue(owner, "RQ", false, false);
32         queryResult = new QueryResult(owner, "QueryResult", false);
33         startTime=currentTime();
34         if (Config.DEMO_MODE)
35             ((ModelS) owner).getLogWriter().logProcess(getName(), null, startTime.
                   getTimeValue());
36     }
37
38     public void checkQueue(){
39         SubQueryResult sqr;
40         while ((sqr = (SubQueryResult)resultQueue.first())!=null){
41             resultQueue.remove(sqr);
42             queryResult.addResult(sqr);
43             subcnt--;
44         }
45     }
46
47     protected void ackQuery(SimProcess ackpro, SubQueryResult sqr){
48         if (sqr!=null) resultQueue.insert(sqr);
49         else subcnt--;
50
51         if (this.isScheduled()){
52             if ( SimTime.isSmallerOrEqual(this.currentTime(), this.scheduledAt()))
                       {
53                 this.reActivate(new SimTime(0.0));
54             } else {
55                 System.out.println("bugbug");
56             }
57         } else if ( this.isBlocked()){
58             //do nothing? got lucky?
59         } else {
60             this.activate(new SimTime(0.0));
61         }
62     }
63
64     public Query getQuery() {
65         return query;
```

```
66      }
67
68      public Node getNode() {
69          return node;
70      }
71
72      @Override
73      public void lifeCycle() {
74      }
75
76      public SimTime getStartTime(){
77          return startTime;
78      }
79  }
```

B.1.13 simulation.processes.QueryProcessGI

```
1   package simulation.processes;
2   import java.util.*;
3
4   import simulation.log.LogWriter;
5   import simulation.model.Config;
6   import simulation.model.ModelS;
7   import simulation.node.Node;
8   import simulation.query.IndexHitList;
9   import simulation.query.IndexTools;
10  import simulation.query.Query;
11  import simulation.query.SubQuery;
12  import simulation.query.SubQueryResult;
13
14
15
16  import desmoj.core.simulator.*;
17
18
19  public class QueryProcessGI extends QueryProcess {
20
21
22      public QueryProcessGI(Model owner, String name, boolean showInTrace) {
23          super(owner, name, showInTrace);
24      }
25
26      public QueryProcessGI(Model owner, String name, boolean showInTrace, Query
            query, Node node) {
27          super(owner, name, showInTrace, query, node);
28      }
29
30      @Override
31      public void checkQueue(){
32          SubQueryResult sqr;
33
34          IndexHitList reslist= queryResult.getHitList();
35
36          while ((sqr = (SubQueryResult)resultQueue.first())!=null){
37              resultQueue.remove(sqr);
38              queryResult.addResult(sqr);
39
40              IndexHitList subreslist= sqr.getHitList();
41
42              node.interleaveTwoLists(this, subreslist.getNumDocs()+(reslist!=null?
                    reslist.getNumDocs():0));
43
```

```
44              if (reslist == null)
45                  reslist = subreslist;
46              else
47                  reslist = IndexTools.interleaveEntries(reslist, subreslist);
48
49              subcnt--;
50          }
51          queryResult.setHitList(reslist);
52      }
53
54      @Override
55      public void lifeCycle() {
56          ModelS ms = (ModelS) getModel();
57          if (Config.DEMO_MODE)
58              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
59                  LogWriter.OTHER);
60
61          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms() *
62                  Config.SIZE_OF_QUERY_ENTRY);
63
64          //node.lookupTerms(this, query.getNumberOfTerms());
65          node.lookupTerms(this, query.getNumberOfTerms());
66
67          //no results for and query
68          if ( query.getTerms().get(0) < 0){
69              if (Config.USE_AND_QUERIES) {
70                  node.transfer(node, ms.getBroker(), this, Config.
                        QUERY_RESULT_HEADER_LENGTH);
71                  if (Config.DEMO_MODE)
72                      ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                            ().getTimeValue(),
73                              LogWriter.FINISH, LogWriter.OTHER);
74                  ms.querySolved(this, query, queryResult);
75                  //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                        getNumberOfDocuments());
76                  return;
77              } else {
78                  query.eliminateNonExistingTerms();
79              }
80          }
81
82          ArrayList<Node> nodes = ms.getNodes();
83          ArrayList<SubQuery> subs = query.partitionByTerm();
84          subcnt = subs.size();
85
86          //  allocate enough memory
87          int memreq = Config.NUMBER_OF_DOCUMENTS * Config.SIZE_OF_ACC_ENTRY *
                subcnt;
88          node.malloc(this,memreq);
89
90          for (SubQuery subQuery : subs) {
91              int destNode = subQuery.getNodeNumber();
92              SubQueryProcessGI sp = new SubQueryProcessGI(ms,"SQP"+destNode,true,
93                      this, subQuery , nodes.get(destNode));
94              sp.activate(new SimTime(0.0));
95          }
96
97          //result trick
98          do{
99              passivate();
```

```
100             checkQueue();
101         } while (subcnt>0);
102 
103         //postprocess
104         node.heapTopAndSort(this, queryResult.getNumberOfDocuments(), Config.
                NUMBER_OF_RESULTS_REQUIRED);
105 
106         node.transfer(node, ms.getBroker(), this, Config.
                QUERY_RESULT_HEADER_LENGTH +
107                     Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, queryResult.
                            getNumberOfDocuments()) *
108                     Config.SIZE_OF_RESULT_ENTRY);
109         node.free(memreq);
110 
111         if (Config.DEMO_MODE)
112             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                        getTimeValue(), LogWriter.FINISH,
113                     LogWriter.OTHER);
114         ms.querySolved(this, query, queryResult);
115         //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                getNumberOfDocuments());
116     }
117 }
```

B.1.14 simulation.processes.QueryProcessGIPM

```
1  package simulation.processes;
2  import java.util.*;
3  
4  import simulation.log.LogWriter;
5  import simulation.model.Config;
6  import simulation.model.ModelS;
7  import simulation.node.Node;
8  import simulation.query.IndexHitList;
9  import simulation.query.IndexTools;
10 import simulation.query.Query;
11 import simulation.query.SubQuery;
12 import simulation.query.SubQueryResult;
13 
14 
15 
16 import desmoj.core.simulator.*;
17 
18 
19 public class QueryProcessGIPM extends QueryProcess {
20     private int totalentries = 0;
21     private ArrayList<IndexHitList> hitlists = new ArrayList<IndexHitList>();
22 
23     public QueryProcessGIPM(Model owner, String name, boolean showInTrace) {
24         super(owner, name, showInTrace);
25     }
26 
27     public QueryProcessGIPM(Model owner, String name, boolean showInTrace, Query
            query, Node node) {
28         super(owner, name, showInTrace, query, node);
29     }
30 
31     @Override
32     public void checkQueue(){
33         SubQueryResult sqr;
34 
35         while ((sqr = (SubQueryResult)resultQueue.first())!=null){
```

```
36              resultQueue.remove(sqr);
37              queryResult.addResult(sqr);
38
39              IndexHitList subreslist= sqr.getHitList();
40              totalentries += subreslist.getNumDocs();
41              hitlists.add(subreslist);
42
43              subcnt--;
44          }
45      }
46
47      @Override
48      public void lifeCycle() {
49          ModelS ms = (ModelS) getModel();
50          if (Config.DEMO_MODE)
51              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
52                  LogWriter.OTHER);
53
54          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms() *
55                  Config.SIZE_OF_QUERY_ENTRY);
56
57          //node.lookupTerms(this, query.getNumberOfTerms());
58          node.lookupTerms(this, query.getNumberOfTerms());
59
60          //no results for and query
61          if ( query.getTerms().get(0) < 0){
62              if (Config.USE_AND_QUERIES) {
63                  node.transfer(node, ms.getBroker(), this, Config.
                        QUERY_RESULT_HEADER_LENGTH);
64                  if (Config.DEMO_MODE)
65                      ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                            ().getTimeValue(),
66                          LogWriter.FINISH, LogWriter.OTHER);
67                  ms.querySolved(this, query, queryResult);
68                  //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                        getNumberOfDocuments());
69                  return;
70              } else {
71                  query.eliminateNonExistingTerms();
72              }
73          }
74          ArrayList<Node> nodes = ms.getNodes();
75          ArrayList<SubQuery> subs = query.partitionByTerm();
76          subcnt = subs.size();
77
78          //allocate enough memory
79          int memreq = Config.NUMBER_OF_DOCUMENTS * Config.SIZE_OF_ACC_ENTRY * (
                subcnt+1);
80          node.malloc(this,memreq);
81
82          for (SubQuery subQuery : subs) {
83              int destNode = subQuery.getNodeNumber();
84              SubQueryProcessGIPM sp = new SubQueryProcessGIPM(ms,"SQP"+destNode,
                    true,
85                      this, subQuery , nodes.get(destNode));
86              sp.activate(new SimTime(0.0));
87          }
88
89          //result trick
90          do{
```

```
91              passivate();
92              checkQueue();
93          } while (subcnt>0);
94
95          //postprocess
96          queryResult.setHitList(IndexTools.interleaveEntries(hitlists));
97
98          node.heapInterleave(this, totalentries, hitlists.size());
99          node.heapTopAndSort(this, queryResult.getNumberOfDocuments(), Config.
                NUMBER_OF_RESULTS_REQUIRED);
100
101         node.transfer(node, ms.getBroker(), this, Config.
                QUERY_RESULT_HEADER_LENGTH +
102                 Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, queryResult.
                        getNumberOfDocuments()) *
103                 Config.SIZE_OF_RESULT_ENTRY);
104         node.free(memreq);
105         if (Config.DEMO_MODE)
106             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
107                 LogWriter.OTHER);
108         ms.querySolved(this, query, queryResult);
109         //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                getNumberOfDocuments());
110     }
111 }
```

B.1.15 simulation.processes.QueryProcessHD

```
1  package simulation.processes;
2  import java.util.*;
3
4  import simulation.log.LogWriter;
5  import simulation.model.Config;
6  import simulation.model.ModelS;
7  import simulation.node.Node;
8  import simulation.query.IndexHitList;
9  import simulation.query.IndexTools;
10 import simulation.query.Query;
11 import simulation.query.SubQuery;
12 import simulation.query.SubQueryResult;
13
14
15 import desmoj.core.simulator.*;
16
17
18 public class QueryProcessHD extends QueryProcess {
19
20     public QueryProcessHD(Model owner, String name, boolean showInTrace) {
21         super(owner, name, showInTrace);
22     }
23
24     public QueryProcessHD(Model owner, String name, boolean showInTrace, Query
           query, Node node) {
25         super(owner, name, showInTrace, query, node);
26     }
27
28     @Override
29     public void checkQueue(){
30
31         SubQueryResult sqr;
32         IndexHitList reslist= queryResult.getHitList();
```

```
33
34          while ((sqr = (SubQueryResult)resultQueue.first())!=null){
35              resultQueue.remove(sqr);
36              queryResult.addResult(sqr);
37
38              IndexHitList subreslist= sqr.getHitList();
39
40              node.interleaveTwoLists(this, subreslist.getNumDocs()+(reslist!=null?
                    reslist.getNumDocs():0));
41
42              if (reslist==null)
43                  reslist = subreslist;
44              else
45                  reslist = IndexTools.interleaveEntries(reslist, subreslist);
46
47              subcnt--;
48          }
49          queryResult.setHitList(reslist);
50      }
51
52      @Override
53      public void lifeCycle() {
54
55          ModelS ms = (ModelS) getModel();
56          if (Config.DEMO_MODE)
57              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
58                  LogWriter.OTHER);
59
60          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms() *
61                  Config.SIZE_OF_QUERY_ENTRY);
62
63          //node.lookupTerms(this, query.getNumberOfTerms());
64          node.lookupTerms(this, query.getNumberOfTerms());
65
66          //no results for and query
67          if ( query.getTerms().get(0) < 0){
68              if (Config.USE_AND_QUERIES) {
69                  node.transfer(node, ms.getBroker(), this, Config.
                        QUERY_RESULT_HEADER_LENGTH);
70                  if (Config.DEMO_MODE)
71                      ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                            ().getTimeValue(),
72                              LogWriter.FINISH, LogWriter.OTHER);
73                  ms.querySolved(this, query, queryResult);
74                  //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                        getNumberOfDocuments());
75                  return;
76              } else {
77                  query.eliminateNonExistingTerms();
78              }
79          }
80
81          ArrayList<Node> nodes = ms.getNodes();
82          ArrayList<SubQuery> subs = query.partitionHybrid();
83          subcnt=subs.size();
84
85          //allocate enough memory
86          int memreq = Config.NUMBER_OF_DOCUMENTS * Config.SIZE_OF_ACC_ENTRY * (
                subcnt + 2);
87          node.malloc(this, memreq);
```

```
88
89          for (SubQuery subQuery : subs) {
90              int destNumber = subQuery.getNodeNumber();
91              SubQueryProcessHD sp = new SubQueryProcessHD(ms,"SQP"+destNumber,true,
92                          this, subQuery, nodes.get(destNumber));
93              sp.activate(new SimTime(0.0));
94
95          }
96
97          //result trick
98          do{
99              passivate();
100             checkQueue();
101         } while (subcnt>0);
102
103         //postprocess
104         node.heapTopAndSort(this, queryResult.getNumberOfDocuments(), Config.
                NUMBER_OF_RESULTS_REQUIRED);
105
106
107         node.transfer(node, ms.getBroker(), this, Config.
                QUERY_RESULT_HEADER_LENGTH +
108                 Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, queryResult.
                        getNumberOfDocuments()) *
109                 Config.SIZE_OF_RESULT_ENTRY);
110
111         node.free(memreq);
112         if (Config.DEMO_MODE)
113             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
114                 LogWriter.OTHER);
115         ms.querySolved(this, query, queryResult);
116         //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                getNumberOfDocuments());
117     }
118 }
```

B.1.16 simulation.processes.QueryProcessHDPM

```
1  package simulation.processes;
2  import java.util.*;
3
4  import simulation.log.LogWriter;
5  import simulation.model.Config;
6  import simulation.model.ModelS;
7  import simulation.node.Node;
8  import simulation.query.IndexHitList;
9  import simulation.query.IndexTools;
10 import simulation.query.Query;
11 import simulation.query.SubQuery;
12 import simulation.query.SubQueryResult;
13
14
15
16 import desmoj.core.simulator.*;
17
18
19 public class QueryProcessHDPM extends QueryProcess {
20     private int totalentries = 0;
21     private ArrayList<IndexHitList> hitlists = new ArrayList<IndexHitList>();
22
23     public QueryProcessHDPM(Model owner, String name, boolean showInTrace) {
```

```
24          super(owner, name, showInTrace);
25      }
26
27      public QueryProcessHDPM(Model owner, String name, boolean showInTrace, Query
            query, Node node) {
28          super(owner, name, showInTrace, query, node);
29      }
30
31      @Override
32      public void checkQueue(){
33          SubQueryResult sqr;
34
35          while ((sqr = (SubQueryResult)resultQueue.first())!=null){
36              resultQueue.remove(sqr);
37              queryResult.addResult(sqr);
38
39              IndexHitList subreslist= sqr.getHitList();
40              totalentries += subreslist.getNumDocs();
41              hitlists.add(subreslist);
42
43              subcnt--;
44          }
45      }
46
47      @Override
48      public void lifeCycle() {
49
50          ModelS ms = (ModelS) getModel();
51          if (Config.DEMO_MODE)
52              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
53                  LogWriter.OTHER);
54
55          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms() *
56                  Config.SIZE_OF_QUERY_ENTRY);
57
58          //node.lookupTerms(this, query.getNumberOfTerms());
59          node.lookupTerms(this, query.getNumberOfTerms());
60
61          //no results for and query
62          if ( query.getTerms().get(0) < 0){
63              if (Config.USE_AND_QUERIES) {
64                  node.transfer(node, ms.getBroker(), this, Config.
                        QUERY_RESULT_HEADER_LENGTH);
65                  if (Config.DEMO_MODE)
66                      ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                            ().getTimeValue(),
67                              LogWriter.FINISH, LogWriter.OTHER);
68                  ms.querySolved(this, query, queryResult);
69                  System.out.println(query.getNumberOfTerms()+":"+queryResult.
                        getNumberOfDocuments());
70                  return;
71              } else {
72                  query.eliminateNonExistingTerms();
73              }
74          }
75
76          ArrayList<Node> nodes = ms.getNodes();
77          ArrayList<SubQuery> subs = query.partitionHybrid();
78          subcnt=subs.size();
79
```

```
80          //allocate enough memory
81          int memreq = Config.NUMBER_OF_DOCUMENTS * Config.SIZE_OF_ACC_ENTRY * (
                subcnt + 2);
82          node.malloc(this, memreq);
83
84          for (SubQuery subQuery : subs) {
85              int destNumber = subQuery.getNodeNumber();
86              SubQueryProcess sp = new SubQueryProcessHDPM(ms,"SQP"+destNumber,true,
87                          this, subQuery, nodes.get(destNumber));
88              sp.activate(new SimTime(0.0));
89
90          }
91
92          //result trick
93          do{
94              passivate();
95              checkQueue();
96          } while (subcnt>0);
97
98          //postprocess
99          queryResult.setHitList(IndexTools.interleaveEntries(hitlists));
100         node.heapInterleave(this, totalentries, hitlists.size());
101         node.heapTopAndSort(this, queryResult.getNumberOfDocuments(), Config.
                NUMBER_OF_RESULTS_REQUIRED);
102
103
104         node.transfer(node, ms.getBroker(), this, Config.
                QUERY_RESULT_HEADER_LENGTH +
105                 Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, queryResult.
                        getNumberOfDocuments()) *
106                 Config.SIZE_OF_RESULT_ENTRY);
107
108         node.free(memreq);
109         if (Config.DEMO_MODE)
110             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
111                     LogWriter.OTHER);
112         ms.querySolved(this, query, queryResult);
113     }
114 }
```

B.1.17 simulation.processes.QueryProcessLI

```
1  package simulation.processes;
2  import java.util.ArrayList;
3
4  import simulation.log.LogWriter;
5  import simulation.model.Config;
6  import simulation.model.ModelS;
7  import simulation.node.Node;
8  import simulation.query.Query;
9  import simulation.query.SubQuery;
10 import simulation.query.SubQueryResult;
11
12
13 import desmoj.core.simulator.*;
14
15
16 public class QueryProcessLI extends QueryProcess {
17     private double numres = 0;
18
19     public QueryProcessLI(Model owner, String name, boolean showInTrace) {
```

```
20          super(owner, name, showInTrace);
21      }
22
23      public QueryProcessLI(Model owner, String name, boolean showInTrace, Query
            query, Node node) {
24          super(owner, name, showInTrace, query, node);
25      }
26
27      @Override
28      public void checkQueue(){
29          SubQueryResult sqr;
30          while ((sqr = (SubQueryResult) resultQueue.first())!=null){
31              resultQueue.remove(sqr);
32              queryResult.addResult(sqr);
33              numres+=Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, sqr.
                    getNumberOfDocuments());
34              subcnt--;
35          }
36      }
37
38      @Override
39      public void lifeCycle() {
40          ModelS ms = (ModelS) getModel();
41
42          if (Config.DEMO_MODE)
43              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
44                      LogWriter.OTHER);
45
46          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms() *
47                      Config.SIZE_OF_QUERY_ENTRY);
48
49          ArrayList<Node> nodes = ms.getNodes();
50          ArrayList<SubQuery> subs = query.partitionByDoc();
51          subcnt = subs.size();
52
53          //allocate enough memory
54          int memreq = Config.NUMBER_OF_RESULTS_REQUIRED * subcnt * Config.
                SIZE_OF_RESULT_ENTRY;
55          node.malloc(this, memreq);
56
57
58          for (SubQuery subQuery : subs) {
59              int destNumber = subQuery.getNodeNumber();
60              SubQueryProcess newsqp = new SubQueryProcessLI(ms,"SQP"+destNumber,
                    true,
61                          this, subQuery , nodes.get(destNumber));
62              newsqp.activate(new SimTime(0.0));
63          }
64
65          do{
66              passivate();
67              checkQueue();
68          } while (subcnt>0);
69
70
71          queryResult.combineResults();
72
73          //FIXME: there is an alternative here! progressive heapmerge
74
```

```
75          node.heapMerge(this, (int) Math.min(Config.NUMBER_OF_RESULTS_REQUIRED,
                numres), Config.NUMBER_OF_NODES);
76
77          node.heapTopAndSort(this, (int) Math.min(Config.NUMBER_OF_RESULTS_REQUIRED
                , numres),
78                  Config.NUMBER_OF_RESULTS_REQUIRED);
79
80          node.transfer(node, ms.getBroker(), this, Config.
                QUERY_RESULT_HEADER_LENGTH +
81                  Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, queryResult.
                        getNumberOfDocuments()) *
82                  Config.SIZE_OF_RESULT_ENTRY);
83
84          node.free(memreq);
85          if (Config.DEMO_MODE)
86              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
87                      LogWriter.OTHER);
88
89          ms.querySolved(this, query, queryResult);
90          //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                getNumberOfDocuments());
91      }
92  }
```

B.1.18 simulation.processes.QueryProcessLIPM

```
1   package simulation.processes;
2   import java.util.ArrayList;
3
4   import simulation.log.LogWriter;
5   import simulation.model.Config;
6   import simulation.model.ModelS;
7   import simulation.node.Node;
8   import simulation.query.Query;
9   import simulation.query.SubQuery;
10  import simulation.query.SubQueryResult;
11
12
13  import desmoj.core.simulator.*;
14
15
16
17  public class QueryProcessLIPM extends QueryProcess {
18      private double numres = 0;
19
20      public QueryProcessLIPM(Model owner, String name, boolean showInTrace) {
21          super(owner, name, showInTrace);
22      }
23
24      public QueryProcessLIPM(Model owner, String name, boolean showInTrace, Query
            query, Node node) {
25          super(owner, name, showInTrace, query, node);
26      }
27
28      @Override
29      public void checkQueue(){
30          SubQueryResult sqr;
31          while ((sqr = (SubQueryResult) resultQueue.first())!=null){
32              resultQueue.remove(sqr);
33              queryResult.addResult(sqr);
```

```
34              numres+=Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, sqr.
                    getNumberOfDocuments());
35              subcnt--;
36          }
37      }
38
39      @Override
40      public void lifeCycle() {
41          ModelS ms = (ModelS) getModel();
42          if (Config.DEMO_MODE)
43              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
44                  LogWriter.OTHER);
45
46          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms() *
47                  Config.SIZE_OF_QUERY_ENTRY);
48
49          //dont need to lookup, just broadcast
50          //node.lookupTerms(this, query.getNumberOfTerms());
51          ModelS model = (ModelS) getModel();
52          ArrayList<Node> nodes = model.getNodes();
53          ArrayList<SubQuery> subs = query.partitionByDoc();
54          subcnt = subs.size();
55
56          //allocate enough memory
57          int memreq = Config.NUMBER_OF_RESULTS_REQUIRED * (subcnt + 1)* Config.
                SIZE_OF_RESULT_ENTRY;
58          node.malloc(this, memreq);
59
60
61          for (SubQuery subQuery : subs) {
62              int destNumber = subQuery.getNodeNumber();
63              SubQueryProcess newsqp = new SubQueryProcessLIPM(model,"SQP"+
                    destNumber,true,
64                      this, subQuery , nodes.get(destNumber));
65              newsqp.activate(new SimTime(0.0));
66          }
67
68          do{
69              passivate();
70              checkQueue();
71          } while (subcnt>0);
72
73
74          queryResult.combineResults();
75
76          node.heapMerge(this, (int) Math.min(Config.NUMBER_OF_RESULTS_REQUIRED,
                numres),
77                  Config.NUMBER_OF_NODES);
78          node.heapTopAndSort(this, (int) Math.min(Config.NUMBER_OF_RESULTS_REQUIRED
                , numres),
79                  Config.NUMBER_OF_RESULTS_REQUIRED);
80
81          node.transfer(node, ms.getBroker(), this, Config.
                QUERY_RESULT_HEADER_LENGTH +
82                  Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, queryResult.
                        getNumberOfDocuments()) *
83                  Config.SIZE_OF_RESULT_ENTRY);
84
85          node.free(memreq);
86          if (Config.DEMO_MODE)
```

```
87              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
88                  LogWriter.OTHER);
89
90          model.querySolved(this, query, queryResult);
91          //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                getNumberOfDocuments());
92      }
93  }
```

B.1.19 simulation.processes.QueryProcessPL

```
1   package simulation.processes;
2   import java.util.*;
3
4   import simulation.log.LogWriter;
5   import simulation.model.Config;
6   import simulation.model.ModelS;
7   import simulation.node.Node;
8   import simulation.query.IndexHitList;
9   import simulation.query.IndexTools;
10  import simulation.query.Query;
11  import simulation.query.SubQuery;
12
13
14
15
16  import desmoj.core.simulator.*;
17
18
19  public class QueryProcessPL extends QueryProcess {
20
21      public QueryProcessPL(Model owner, String name, boolean showInTrace) {
22          super(owner, name, showInTrace);
23      }
24
25      public QueryProcessPL(Model owner, String name, boolean showInTrace, Query
            query, Node node) {
26          super(owner, name, showInTrace, query, node);
27      }
28
29      @Override
30      public void checkQueue(){}
31
32      @Override
33      public void lifeCycle() {
34
35          ModelS ms = (ModelS) getModel();
36          if (Config.DEMO_MODE)
37              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
38                  LogWriter.OTHER);
39
40          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms()
41                  * Config.SIZE_OF_QUERY_ENTRY);
42
43          node.lookupTerms(this, query.getNumberOfTerms());
44          if ( query.getTerms().get(0) < 0){
45              if (Config.USE_AND_QUERIES) {
46                  node.transfer(node, ms.getBroker(), this, Config.
                        QUERY_RESULT_HEADER_LENGTH);
```

```
47                  if (Config.DEMO_MODE)
48                      ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                            ().getTimeValue(),
49                          LogWriter.FINISH, LogWriter.OTHER);
50                  ms.querySolved(this, query, queryResult);
51                  //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                        getNumberOfDocuments());
52                  return;
53              } else {
54                  query.eliminateNonExistingTerms();
55              }
56          }
57
58          ArrayList<Node> nodes = ms.getNodes();
59          ArrayList<SubQuery> subs = query.partitionByTerm();
60          subcnt=subs.size();
61
62          int subcntc = subcnt;
63          Node prev = null;
64          Node curr = node;
65          int memreq = 2*Config.NUMBER_OF_DOCUMENTS*Config.SIZE_OF_ACC_ENTRY;
66
67
68          curr.malloc(this, memreq);
69          IndexHitList reslist=null;
70
71          if (subcnt==0){
72              queryResult.setHitList(reslist);
73              curr.transfer(curr, ms.getBroker(), this, Config.
                    QUERY_RESULT_HEADER_LENGTH);
74          } else {
75              while(true){
76                  int bestcand=0;
77                  for (int i=1; i < subcnt; i++){
78                      //TODO: fix the heuristic part here!
79                      if (subs.get(bestcand).getLowestScore()>subs.get(i).
                            getLowestScore()){
80                          bestcand=i;
81                      }
82                  }
83                  prev=curr;
84                  curr=nodes.get(subs.get(bestcand).getNodeNumber());
85
86                  SubQuery subquery=subs.get(bestcand);
87
88                  if (prev!=curr){
89                      prev.transfer(curr, prev, this, Config.SUBQUERY_HEADER_LENGTH*
                            subcntc + query.getNumberOfTerms() *
90                          Config.SIZE_OF_QUERY_ENTRY+(reslist!=null?reslist.
                                getNumDocs()*Config.SIZE_OF_ACC_ENTRY:0));
91                      prev.free(memreq);
92                      curr.malloc(this, memreq);
93                  }
94
95                  node.lookupTerms(this, subquery.getNumberOfTerms());
96                  int memreq2 = Config.NUMBER_OF_DOCUMENTS * Config.
                        SIZE_OF_ACC_ENTRY;
97                  curr.malloc(this, memreq2);
98
99                  for (Integer termid : subquery.getSortedTermList()) {
100                     IndexHitList termlist=subquery.getHitList(termid);
```

```
101                     node.fetch(this, subquery.getNumberOfDocuments(termid) *
                            Config.SIZE_OF_INDEX_DOCUMENT_ENTRY);
102
103                     node.interleaveTwoLists(this, termlist.getNumDocs()+(reslist!=
                            null?reslist.getNumDocs():0));
104                     if (reslist==null)
105                         reslist = termlist;
106                     else
107                         reslist = IndexTools.interleaveEntries(reslist, termlist);
108                 }
109
110                 curr.free(memreq2);
111                 subs.remove(bestcand);
112
113                 if (--subcnt==0){
114                     //postprocess
115                     queryResult.setHitList(reslist);
116
117                     curr.heapTopAndSort(this, queryResult.getNumberOfDocuments(),
                            Config.NUMBER_OF_RESULTS_REQUIRED);
118                     curr.transfer(curr, ms.getBroker(), this, Config.
                            QUERY_RESULT_HEADER_LENGTH +
119                             Math.min(Config.NUMBER_OF_RESULTS_REQUIRED,queryResult
                                .getNumberOfDocuments()) *
120                             Config.SIZE_OF_RESULT_ENTRY);
121
122                     break;
123                 }
124             }
125         }
126
127         curr.free(memreq);
128         if (Config.DEMO_MODE)
129             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
130                     LogWriter.OTHER);
131
132         ms.querySolved(this, query, queryResult);
133         //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                getNumberOfDocuments());
134     }
135 }
```

B.1.20 simulation.processes.QueryProcessPLPM

```
1  package simulation.processes;
2  import java.util.*;
3
4  import simulation.log.LogWriter;
5  import simulation.model.Config;
6  import simulation.model.ModelS;
7  import simulation.node.Node;
8  import simulation.query.IndexHitList;
9  import simulation.query.IndexTools;
10 import simulation.query.Query;
11 import simulation.query.SubQuery;
12
13
14
15
16 import desmoj.core.simulator.*;
17
```

```
18
19  public class QueryProcessPLPM extends QueryProcess {
20
21      public QueryProcessPLPM(Model owner, String name, boolean showInTrace) {
22          super(owner, name, showInTrace);
23      }
24
25      public QueryProcessPLPM(Model owner, String name, boolean showInTrace, Query
            query, Node node) {
26          super(owner, name, showInTrace, query, node);
27      }
28
29      @Override
30      public void checkQueue(){}
31
32      @Override
33      public void lifeCycle() {
34
35          ModelS ms = (ModelS) getModel();
36          if (Config.DEMO_MODE)
37              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
38                  LogWriter.OTHER);
39
40          node.transfer(ms.getBroker(), node, this, Config.QUERY_HEADER_LENGTH +
                query.getNumberOfTerms()
41                  * Config.SIZE_OF_QUERY_ENTRY);
42
43          node.lookupTerms(this, query.getNumberOfTerms());
44          if ( query.getTerms().get(0) < 0){
45              if (Config.USE_AND_QUERIES) {
46                  node.transfer(node, ms.getBroker(), this, Config.
                        QUERY_RESULT_HEADER_LENGTH);
47                  if (Config.DEMO_MODE)
48                      ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                            ().getTimeValue(),
49                              LogWriter.FINISH, LogWriter.OTHER);
50                  ms.querySolved(this, query, queryResult);
51                  //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                        getNumberOfDocuments());
52                  return;
53              } else {
54                  query.eliminateNonExistingTerms();
55              }
56          }
57
58
59          ArrayList<Node> nodes = ms.getNodes();
60          ArrayList<SubQuery> subs = query.partitionByTerm();
61          subcnt=subs.size();
62
63          int subcntc = subcnt;
64          Node prev = null;
65          Node curr = node;
66          int memreq = Config.NUMBER_OF_DOCUMENTS*Config.SIZE_OF_ACC_ENTRY;
67
68
69          curr.malloc(this, memreq);
70          IndexHitList reslist=null;
71          if (subcnt==0){
72              queryResult.setHitList(reslist);
```

```
73              curr.transfer(curr, ms.getBroker(), this, Config.
                    QUERY_RESULT_HEADER_LENGTH);
74          } else {
75              while(true){
76                  int bestcand=0;
77                  for (int i=1; i < subcnt; i++){
78                      //  TODO: fix the heuristic part here!
79                      if (subs.get(bestcand).getLowestScore()>subs.get(i).
                            getLowestScore()){
80                          bestcand=i;
81                      }
82                  }
83
84                  prev=curr;
85                  curr=nodes.get(subs.get(bestcand).getNodeNumber());
86
87                  if (prev!=curr){
88                      prev.free(memreq);
89                      prev.transfer(curr, prev, this, Config.SUBQUERY_HEADER_LENGTH*
                            subcntc + query.getNumberOfTerms() *
90                          Config.SIZE_OF_QUERY_ENTRY+(reslist!=null?reslist.
                                getNumDocs()*Config.SIZE_OF_ACC_ENTRY:0));
91                      curr.malloc(this, memreq);
92                  }
93
94                  SubQuery subquery=subs.get(bestcand);
95                  node.lookupTerms(this, subquery.getNumberOfTerms());
96
97                  int memreq2 = subquery.getNumberOfTerms() * Config.
                        NUMBER_OF_DOCUMENTS *
98                      Config.SIZE_OF_ACC_ENTRY;
99
100                 curr.malloc(this, memreq2);
101
102                 ArrayList<IndexHitList> hitlists = new ArrayList<IndexHitList>();
103                 if (reslist!=null) hitlists.add(reslist);
104                 int totalentries = (reslist!=null)?reslist.getNumDocs():0;
105
106                 for (Integer termid : subquery.getSortedTermList()) {
107                     IndexHitList termlist=subquery.getHitList(termid);
108                     node.fetch(this, subquery.getNumberOfDocuments(termid) *
                            Config.SIZE_OF_INDEX_DOCUMENT_ENTRY);
109                     hitlists.add(termlist);
110                     totalentries += termlist.getNumDocs();
111                 }
112
113                 //simulates merging extraction from heap. fetched lists and the
                        old list
114                 reslist = IndexTools.interleaveEntries(hitlists);
115                 node.heapInterleave(this, totalentries, hitlists.size());
116                 curr.free(memreq2);
117
118                 subs.remove(bestcand);
119
120                 if (--subcnt==0){
121                     //postprocess
122                     queryResult.setHitList(reslist);
123
124                     curr.heapTopAndSort(this, queryResult.getNumberOfDocuments(),
125                         Config.NUMBER_OF_RESULTS_REQUIRED);
126                     curr.transfer(curr, ms.getBroker(), this, Config.
                            QUERY_RESULT_HEADER_LENGTH +
```

```
127                         Math.min(Config.NUMBER_OF_RESULTS_REQUIRED,queryResult.
                                getNumberOfDocuments()) *
128                         Config.SIZE_OF_RESULT_ENTRY);
129                     break;
130                 }
131             }
132         }
133         curr.free(memreq);
134         if (Config.DEMO_MODE)
135             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
136                     LogWriter.OTHER);
137         ms.querySolved(this, query, queryResult);
138         //System.out.println(query.getNumberOfTerms()+":"+queryResult.
                getNumberOfDocuments());
139     }
140 }
```

B.1.21 simulation.processes.SubQueryProcess

```
1  package simulation.processes;
2  import simulation.model.Config;
3  import simulation.model.ModelS;
4  import simulation.node.Node;
5  import simulation.query.SubQuery;
6  import desmoj.core.simulator.*;
7
8  public abstract class SubQueryProcess extends SimProcess{
9      protected SubQuery subquery;
10     protected Node node;
11     protected QueryProcess queryprocess;
12
13     public SubQueryProcess(Model owner, String name, boolean showInTrace) {
14         super(owner, name, showInTrace);
15     }
16
17     public SubQueryProcess(Model owner, String name, boolean showInTrace,
18               QueryProcess queryprocess, SubQuery subquery, Node node) {
19         super(owner, name, showInTrace);
20         this.subquery=subquery;
21         this.node=node;
22         this.queryprocess=queryprocess;
23         if (Config.DEMO_MODE)
24             ((ModelS) owner).getLogWriter().logProcess(getName(), queryprocess.
                    getName(), currentTime().getTimeValue());
25     }
26
27     @Override
28     public void lifeCycle(){
29     }
30
31     public Node getNode() {
32         return node;
33     }
34
35     public SubQuery getSubquery() {
36         return subquery;
37     }
38
39 }
```

B.1.22 simulation.processes.SubQueryProcessGI

```
1  package simulation.processes;
2
3  import simulation.log.LogWriter;
4  import simulation.model.Config;
5  import simulation.model.ModelS;
6  import simulation.node.*;
7  import simulation.query.*;
8  import desmoj.core.simulator.*;
9
10 public class SubQueryProcessGI extends SubQueryProcess{
11
12     public SubQueryProcessGI(Model owner, String name, boolean showInTrace) {
13         super(owner, name, showInTrace);
14     }
15
16     public SubQueryProcessGI(Model owner, String name, boolean showInTrace,
17               QueryProcess queryprocess, SubQuery subquery, Node node) {
18         super(owner, name, showInTrace, queryprocess, subquery, node);
19     }
20
21     @Override
22     public void lifeCycle() {
23         //example
24         Node othernode = queryprocess.getNode();
25         ModelS ms = (ModelS) getModel();
26         if (Config.DEMO_MODE)
27             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                   getTimeValue(), LogWriter.START,
28                 LogWriter.OTHER);
29
30         //transfer data
31         node.transfer(othernode, node, this, Config.SUBQUERY_HEADER_LENGTH +
               Config.SIZE_OF_QUERY_ENTRY *
32                 subquery.getNumberOfTerms());
33
34         //lookup for the terms
35         node.lookupTerms(this, subquery.getNumberOfTerms());
36
37         //allocate
38         int memreq = Config.NUMBER_OF_DOCUMENTS * (Config.
               SIZE_OF_INDEX_DOCUMENT_ENTRY + 2 * Config.SIZE_OF_ACC_ENTRY);
39         node.malloc(this, memreq);
40
41         IndexHitList reslist=null;
42         for (Integer termid : subquery.getSortedTermList()) {
43             //FIXME: just documentbased
44             IndexHitList termlist=subquery.getHitList(termid);
45             node.fetch(this, termlist.getNumDocs() * Config.
                   SIZE_OF_INDEX_DOCUMENT_ENTRY);
46
47             node.interleaveTwoLists(this, (termlist.getNumDocs()+(reslist!=null?
                   reslist.getNumDocs():0)));
48             if (reslist==null)
49                 reslist = termlist;
50             else
51                 reslist = IndexTools.interleaveEntries(reslist, termlist);
52
53         }
54         //transfer back:
```

```
55          SubQueryResult subqueryresult = new SubQueryResult(getModel(), "
                SubQueryResult", false, reslist);
56          node.transfer(node, othernode, this, Config.SUBQUERY_RESULT_HEADER_LENGTH
                +
57                  Config.SIZE_OF_ACC_ENTRY * subqueryresult.getNumberOfDocuments());
58
59          //free memory
60          node.free(memreq);
61          if (Config.DEMO_MODE)
62              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
63                  LogWriter.OTHER);
64          //ack query
65          queryprocess.ackQuery(this, subqueryresult);
66      }
67
68  }
```

B.1.23 simulation.processes.SubQueryProcessGIPM

```
1   package simulation.processes;
2   import desmoj.core.simulator.*;
3   import java.util.*;
4
5   import simulation.log.LogWriter;
6   import simulation.model.Config;
7   import simulation.model.ModelS;
8   import simulation.node.Node;
9   import simulation.query.IndexHitList;
10  import simulation.query.IndexTools;
11  import simulation.query.SubQuery;
12  import simulation.query.SubQueryResult;
13
14
15
16
17  public class SubQueryProcessGIPM extends SubQueryProcess{
18      public SubQueryProcessGIPM(Model owner, String name, boolean showInTrace) {
19          super(owner, name, showInTrace);
20      }
21
22      public SubQueryProcessGIPM(Model owner, String name, boolean showInTrace,
23                QueryProcess queryprocess, SubQuery subquery, Node node) {
24          super(owner, name, showInTrace, queryprocess, subquery, node);
25      }
26
27      @Override
28      public void lifeCycle() {
29          //example
30          Node othernode = queryprocess.getNode();
31          ModelS ms = (ModelS) getModel();
32          if (Config.DEMO_MODE)
33              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
34                  LogWriter.OTHER);
35
36          //transfer data
37          node.transfer(othernode, node, this, Config.SUBQUERY_HEADER_LENGTH +
                Config.SIZE_OF_QUERY_ENTRY *
38                  subquery.getNumberOfTerms());
39
40          //lookup for the terms
```

```
41          node.lookupTerms(this, subquery.getNumberOfTerms());
42
43          //allocate
44          int memreq = Config.NUMBER_OF_DOCUMENTS * (Config.
                SIZE_OF_INDEX_DOCUMENT_ENTRY *
45                  subquery.getNumberOfTerms() + Config.SIZE_OF_ACC_ENTRY);
46          node.malloc(this, memreq);
47
48          ArrayList<IndexHitList> hitlists = new ArrayList<IndexHitList>();
49          int totalentries = 0;
50          for (Integer termid : subquery.getSortedTermList()) {
51              //FIXME: just documentbased
52              IndexHitList termlist=subquery.getHitList(termid);
53              node.fetch(this, termlist.getNumDocs() * Config.
                    SIZE_OF_INDEX_DOCUMENT_ENTRY);
54              hitlists.add(termlist);
55              totalentries += termlist.getNumDocs();
56          }
57
58          IndexHitList reslist = IndexTools.interleaveEntries(hitlists);
59          //simulates merging extraction from heap
60          node.heapInterleave(this, totalentries, hitlists.size());
61
62          //FIXME: storage to memory
63
64          //transfer back:
65          SubQueryResult subqueryresult = new SubQueryResult(getModel(), "
                SubQueryResult", false, reslist);
66          node.transfer(node, othernode, this, Config.SUBQUERY_RESULT_HEADER_LENGTH
                +
67                  Config.SIZE_OF_ACC_ENTRY * subqueryresult.getNumberOfDocuments());
68
69          //free memory
70          node.free(memreq);
71          if (Config.DEMO_MODE)
72              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
73                      LogWriter.OTHER);
74          //ack query
75          queryprocess.ackQuery(this, subqueryresult);
76      }
77  }
```

B.1.24 simulation.processes.SubQueryProcessHD

```
1   package simulation.processes;
2   import simulation.log.LogWriter;
3   import simulation.model.Config;
4   import simulation.model.ModelS;
5   import simulation.node.Node;
6   import simulation.query.IndexHitList;
7   import simulation.query.IndexTools;
8   import simulation.query.SubQuery;
9   import simulation.query.SubQueryResult;
10  import desmoj.core.simulator.*;
11
12  public class SubQueryProcessHD extends SubQueryProcess{
13
14      public SubQueryProcessHD(Model owner, String name, boolean showInTrace) {
15          super(owner, name, showInTrace);
16      }
17
```

```
18      public SubQueryProcessHD(Model owner, String name, boolean showInTrace,
19              QueryProcess queryprocess, SubQuery subquery, Node node) {
20          super(owner, name, showInTrace, queryprocess, subquery, node);
21      }
22
23      @Override
24      public void lifeCycle() {
25          //example
26          Node othernode = queryprocess.getNode();
27          ModelS ms = (ModelS) getModel();
28          if (Config.DEMO_MODE)
29              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
30                  LogWriter.OTHER);
31
32          //transfer data
33          node.transfer(othernode, node, this, Config.SUBQUERY_HEADER_LENGTH +
                Config.SIZE_OF_QUERY_ENTRY *
34                  subquery.getQuery().getNumberOfTerms());
35
36          node.lookupTerms(this, subquery.getQuery().getNumberOfTerms());
37
38          //allocate
39          int memreq = Config.NUMBER_OF_DOCUMENTS * (Config.
                SIZE_OF_INDEX_DOCUMENT_ENTRY+2 * Config.SIZE_OF_ACC_ENTRY);
40          node.malloc(this, memreq);
41
42          IndexHitList reslist=null;
43          for (Integer termid : subquery.getSortedTermList()) {
44              //FIXME: just documentbased
45              IndexHitList termlist=subquery.getHitList(termid);
46              node.fetch(this, termlist.getNumDocs() * Config.
                    SIZE_OF_INDEX_DOCUMENT_ENTRY);
47
48              node.interleaveTwoLists(this, termlist.getNumDocs()+(reslist!=null?
                    reslist.getNumDocs():0));
49
50              if (reslist == null)
51                  reslist = termlist;
52              else
53                  reslist = IndexTools.interleaveEntries(reslist, termlist);
54          }
55
56
57          SubQueryResult subqueryresult = new SubQueryResult(getModel(), "
                SubQueryResult", false, reslist);
58          //transfer back:
59          node.transfer(node, othernode, this, Config.SUBQUERY_RESULT_HEADER_LENGTH
                + Config.SIZE_OF_ACC_ENTRY *
60                  subqueryresult.getNumberOfDocuments());
61
62          //free memory
63          node.free(memreq);
64          if (Config.DEMO_MODE)
65              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
66                  LogWriter.OTHER);
67
68          //ack query
69          queryprocess.ackQuery(this, subqueryresult);
70      }
71
```

```
72 }
```

B.1.25 simulation.processes.SubQueryProcessHDPM

```
1  package simulation.processes;
2  import java.util.ArrayList;
3
4  import simulation.log.LogWriter;
5  import simulation.model.Config;
6  import simulation.model.ModelS;
7  import simulation.node.Node;
8  import simulation.query.IndexHitList;
9  import simulation.query.IndexTools;
10 import simulation.query.SubQuery;
11 import simulation.query.SubQueryResult;
12
13
14 import desmoj.core.simulator.*;
15
16 public class SubQueryProcessHDPM extends SubQueryProcess{
17
18     public SubQueryProcessHDPM(Model owner, String name, boolean showInTrace) {
19         super(owner, name, showInTrace);
20     }
21
22     public SubQueryProcessHDPM(Model owner, String name, boolean showInTrace,
23             QueryProcess queryprocess, SubQuery subquery, Node node) {
24         super(owner, name, showInTrace, queryprocess, subquery, node);
25     }
26
27     @Override
28     public void lifeCycle() {
29         //example
30         Node othernode = queryprocess.getNode();
31         ModelS ms = (ModelS) getModel();
32         if (Config.DEMO_MODE)
33             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                   getTimeValue(), LogWriter.START,
34                 LogWriter.OTHER);
35
36         //transfer data
37         node.transfer(othernode, node, this, Config.SUBQUERY_HEADER_LENGTH +
               Config.SIZE_OF_QUERY_ENTRY *
38                 subquery.getQuery().getNumberOfTerms());
39
40         node.lookupTerms(this, subquery.getQuery().getNumberOfTerms());
41
42         //allocate
43         int memreq = Config.NUMBER_OF_DOCUMENTS * (Config.
               SIZE_OF_INDEX_DOCUMENT_ENTRY *
44                 subquery.getNumberOfTerms() + Config.SIZE_OF_ACC_ENTRY);
45         node.malloc(this, memreq);
46
47         ArrayList<IndexHitList> hitlists = new ArrayList<IndexHitList>();
48         int totalentries = 0;
49         for (Integer termid : subquery.getSortedTermList()) {
50             IndexHitList termlist=subquery.getHitList(termid);
51             node.fetch(this, termlist.getNumDocs() * Config.
                   SIZE_OF_INDEX_DOCUMENT_ENTRY);
52             hitlists.add(termlist);
53             totalentries += termlist.getNumDocs();
54         }
```

```
55          IndexHitList reslist = IndexTools.interleaveEntries(hitlists);
56          //simulates merging extraction from heap
57          node.heapInterleave(this, totalentries, hitlists.size());
58
59          SubQueryResult subqueryresult = new SubQueryResult(getModel(), "
                SubQueryResult", false, reslist);
60          //transfer back:
61          node.transfer(node, othernode, this, Config.SUBQUERY_RESULT_HEADER_LENGTH
                + Config.SIZE_OF_ACC_ENTRY *
62                  subqueryresult.getNumberOfDocuments());
63
64          //free memory
65          node.free(memreq);
66          if (Config.DEMO_MODE)
67              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
68                  LogWriter.OTHER);
69
70          //ack query
71          queryprocess.ackQuery(this, subqueryresult);
72      }
73
74  }
```

B.1.26 simulation.processes.SubQueryProcessLI

```
1   package simulation.processes;
2   import simulation.log.LogWriter;
3   import simulation.model.Config;
4   import simulation.model.ModelS;
5   import simulation.node.Node;
6   import simulation.query.IndexHitList;
7   import simulation.query.IndexTools;
8   import simulation.query.SubQuery;
9   import simulation.query.SubQueryResult;
10  import desmoj.core.simulator.*;
11
12  public class SubQueryProcessLI extends SubQueryProcess{
13
14      public SubQueryProcessLI(Model owner, String name, boolean showInTrace) {
15          super(owner, name, showInTrace);
16      }
17
18      public SubQueryProcessLI(Model owner, String name, boolean showInTrace,
19                QueryProcess queryprocess, SubQuery subquery, Node node) {
20          super(owner, name, showInTrace, queryprocess, subquery, node);
21      }
22
23      @Override
24      public void lifeCycle() {
25          //example
26          Node othernode = queryprocess.getNode();
27          ModelS ms = (ModelS) getModel();
28          if (Config.DEMO_MODE)
29              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.START,
30                  LogWriter.OTHER);
31          //transfer data
32          node.transfer(othernode, node, this, Config.SUBQUERY_HEADER_LENGTH +
                Config.SIZE_OF_QUERY_ENTRY *
33                  subquery.getQuery().getNumberOfTerms());
34
```

```
35          node.lookupTerms(this, subquery.getQuery().getNumberOfTerms());
36
37          //no results for and query
38          if ( subquery.getTerms().get(0) < 0){
39              if (Config.USE_AND_QUERIES) {
40                  node.transfer(node, othernode, this, Config.
                        SUBQUERY_RESULT_HEADER_LENGTH);
41                  queryprocess.ackQuery(this, null);
42                  if (Config.DEMO_MODE)
43                      ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                            ().getTimeValue(), LogWriter.FINISH,
44                            LogWriter.OTHER);
45                  return;
46              } else {
47                  subquery.eliminateNonExistingTerms();
48              }
49          }
50
51          //allocate
52          int memreq = Config.NUMBER_OF_DOCUMENTS / Config.NUMBER_OF_NODES * (Config
                .SIZE_OF_INDEX_DOCUMENT_ENTRY + Config.SIZE_OF_ACC_ENTRY);
53          node.malloc(this, memreq);
54
55          IndexHitList reslist=null;
56          for (Integer termid : subquery.getSortedTermList()){
57              IndexHitList termlist=subquery.getHitList(termid);
58              node.fetch(this, subquery.getNumberOfDocuments(termid) * Config.
                    SIZE_OF_INDEX_DOCUMENT_ENTRY);
59
60              node.interleaveTwoLists(this, termlist.getNumDocs()+(reslist!=null?
                    reslist.getNumDocs():0));
61              if (reslist == null)
62                  reslist = termlist;
63              else
64                  reslist = IndexTools.interleaveEntries(reslist, termlist);
65          }
66
67          //postprocess
68          SubQueryResult subqueryresult = new SubQueryResult(getModel(), "
                SubQueryResult", false, reslist);
69
70              node.heapTopAndSort(this, subqueryresult.getNumberOfDocuments(),
                    Config.NUMBER_OF_RESULTS_REQUIRED);
71
72
73          //transfer back:
74          node.transfer(node, othernode, this, Config.SUBQUERY_RESULT_HEADER_LENGTH
                + Config.SIZE_OF_RESULT_ENTRY *
75                  Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, subqueryresult.
                        getNumberOfDocuments()));
76
77          //free memory
78          node.free(memreq);
79
80          //ack query
81          queryprocess.ackQuery(this, subqueryresult);
82          if (Config.DEMO_MODE)
83              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
84                    LogWriter.OTHER);
85      }
86
```

```
87 }
```

B.1.27 simulation.processes.SubQueryProcessLIPM

```
1  package simulation.processes;
2  import java.util.ArrayList;
3
4  import simulation.log.LogWriter;
5  import simulation.model.Config;
6  import simulation.model.ModelS;
7  import simulation.node.Node;
8  import simulation.query.IndexHitList;
9  import simulation.query.IndexTools;
10 import simulation.query.SubQuery;
11 import simulation.query.SubQueryResult;
12
13
14 import desmoj.core.simulator.*;
15
16 public class SubQueryProcessLIPM extends SubQueryProcess{
17
18     public SubQueryProcessLIPM(Model owner, String name, boolean showInTrace) {
19         super(owner, name, showInTrace);
20     }
21
22     public SubQueryProcessLIPM(Model owner, String name, boolean showInTrace,
23               QueryProcess queryprocess, SubQuery subquery, Node node) {
24         super(owner, name, showInTrace, queryprocess, subquery, node);
25     }
26
27     @Override
28     public void lifeCycle() {
29         //example
30         Node othernode = queryprocess.getNode();
31         ModelS ms = (ModelS) getModel();
32         if (Config.DEMO_MODE)
33             ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                   getTimeValue(), LogWriter.START,
34                   LogWriter.OTHER);
35         //transfer data
36         node.transfer(othernode, node, this, Config.SUBQUERY_HEADER_LENGTH +
               Config.SIZE_OF_QUERY_ENTRY *
37                   subquery.getNumberOfTerms());
38
39         node.lookupTerms(this, subquery.getQuery().getNumberOfTerms());
40
41         if ( subquery.getTerms().get(0) < 0){
42             if (Config.USE_AND_QUERIES) {
43                 node.transfer(node, othernode, this, Config.
                       SUBQUERY_RESULT_HEADER_LENGTH);
44                 queryprocess.ackQuery(this, null);
45                 if (Config.DEMO_MODE)
46                     ms.getLogWriter().logProcessStatus(this.getName(), currentTime
                           ().getTimeValue(), LogWriter.FINISH,
47                           LogWriter.OTHER);
48                 return;
49             } else {
50                 subquery.eliminateNonExistingTerms();
51             }
52         }
53         //allocate
54         int memreq = Config.NUMBER_OF_DOCUMENTS / Config.NUMBER_OF_NODES *
```

```
55              (Config.SIZE_OF_INDEX_DOCUMENT_ENTRY * subquery.getQuery().
                    getNumberOfTerms() +
56              Config.SIZE_OF_ACC_ENTRY);
57
58          node.malloc(this, memreq);
59
60          ArrayList<IndexHitList> hitlists = new ArrayList<IndexHitList>();
61          int totalentries = 0;
62          for (Integer termid : subquery.getSortedTermList()){
63              IndexHitList termlist=subquery.getHitList(termid);
64              node.fetch(this, subquery.getNumberOfDocuments(termid) * Config.
                    SIZE_OF_INDEX_DOCUMENT_ENTRY);
65
66              hitlists.add(termlist);
67              totalentries += termlist.getNumDocs();
68          }
69
70          IndexHitList reslist = IndexTools.interleaveEntries(hitlists);
71
72          //simulates merging extraction from heap
73          node.heapInterleave(this, totalentries, hitlists.size());
74          node.heapTopAndSort(this, reslist.getNumDocs(), Config.
                NUMBER_OF_RESULTS_REQUIRED);
75
76          //postprocess
77          SubQueryResult subqueryresult = new SubQueryResult(getModel(), "
                SubQueryResult", false, reslist);
78
79          //transfer back:
80          node.transfer(node, othernode, this, Config.SUBQUERY_RESULT_HEADER_LENGTH
                + Config.SIZE_OF_RESULT_ENTRY *
81                  Math.min(Config.NUMBER_OF_RESULTS_REQUIRED, subqueryresult.
                        getNumberOfDocuments()));
82
83          //free memory
84          node.free(memreq);
85
86          //ack query
87          queryprocess.ackQuery(this, subqueryresult);
88          if (Config.DEMO_MODE)
89              ms.getLogWriter().logProcessStatus(this.getName(), currentTime().
                    getTimeValue(), LogWriter.FINISH,
90                  LogWriter.OTHER);
91      }
92
93  }
```

B.1.28 simulation.query.IndexHitList

```
1  package simulation.query;
2  public interface IndexHitList{
3      public int getNumDocs();
4  }
```

B.1.29 simulation.query.IndexTools

```
1  package simulation.query;
2  import java.util.*;
3  import java.io.*;
4
5  import simulation.model.Config;
```

```
6
7  public class IndexTools {
8      public static IndexHitList mergeEntries(ArrayList<IndexHitList> entries){
9          if (Config.SIMULATED_ONLY){
10             if (entries.size()==0) return new SimulatedIndexHitList(0.0);
11             SimulatedIndexHitList first = (SimulatedIndexHitList)entries.get(0);
12             SimulatedIndexHitList tmp;
13             double acc = (first!=null) ? first.getDocFrequency() : 0;
14             for(int i=1; i<entries.size(); i++){
15                 tmp=(SimulatedIndexHitList) entries.get(i);
16                 if (tmp!=null) acc += tmp.getDocFrequency();
17             }
18             return new SimulatedIndexHitList(acc);
19         } else {
20             SimpleIndexHitList res = new SimpleIndexHitList();
21             int nums = entries.size();
22             if (nums==1) return entries.get(0);
23             int scores[] = new int[Config.NUMBER_OF_DOCUMENTS];
24             for (int i=0; i<Config.NUMBER_OF_DOCUMENTS; i++){
25                 scores[i]=0;
26             }
27
28             for (IndexHitList entry : entries) {
29                 ArrayList<SimpleIndexHit> hitlist = ((SimpleIndexHitList)entry).
                       getHitList();
30                 for (SimpleIndexHit record : hitlist) {
31                     scores[record.getDocid()]=record.getHits();
32                 }
33             }
34
35             for (int i=0; i<Config.NUMBER_OF_DOCUMENTS; i++){
36                 if (scores[i]>0) res.addHits(i, scores[i]);
37             }
38             return res;
39         }
40     }
41
42     public static IndexHitList interleaveEntries(ArrayList<IndexHitList> entries){
43         if (Config.SIMULATED_ONLY){
44             if (Config.USE_AND_QUERIES){
45                 if (entries.size()==0) return new SimulatedIndexHitList(0.0);
46                 double acc = ((SimulatedIndexHitList)entries.get(0)).
                       getDocFrequency();
47                 for(int i=1; i<entries.size(); i++){
48                     acc *= ((SimulatedIndexHitList) entries.get(i)).
                           getDocFrequency();
49                 }
50                 return new SimulatedIndexHitList(acc);
51             } else {
52                 if (entries.size()==0) return new SimulatedIndexHitList(0.0);
53                 double acc = 1 - ((SimulatedIndexHitList)entries.get(0)).
                       getDocFrequency();
54                 for(int i=1; i<entries.size(); i++){
55                     acc *= 1 - ((SimulatedIndexHitList) entries.get(i)).
                           getDocFrequency();
56                 }
57                 return new SimulatedIndexHitList(1-acc);
58             }
59
60         } else {
61             SimpleIndexHitList res = new SimpleIndexHitList();
62             if (Config.USE_AND_QUERIES){
```

```
63                  int nums = entries.size();
64                  if (nums==1) return entries.get(0);
65                  int scores[] = new int[Config.NUMBER_OF_DOCUMENTS];
66                  int cnts[] = new int[Config.NUMBER_OF_DOCUMENTS];
67                  for (int i=0; i<Config.NUMBER_OF_DOCUMENTS; i++){
68                      scores[i]=0;
69                      cnts[i]=0;
70                  }
71
72                  for (IndexHitList entry : entries) {
73                      ArrayList<SimpleIndexHit> hitlist = ((SimpleIndexHitList)entry
                            ).getHitList();
74
75                      for (SimpleIndexHit record : hitlist) {
76                          if (scores[record.getDocid()] > record.getHits())
77                              scores[record.getDocid()]= record.getHits();
78                          cnts[record.getDocid()]++;
79                      }
80                  }
81
82                  for (int i=0; i<Config.NUMBER_OF_DOCUMENTS; i++){
83                      if (scores[i]>0 && cnts[i]==nums) res.addHits(i, scores[i]);
84                  }
85                  return res;
86              } else {
87                  int nums = entries.size();
88                  if (nums==1) return entries.get(0);
89                  int scores[] = new int[Config.NUMBER_OF_DOCUMENTS];
90                  for (int i=0; i<Config.NUMBER_OF_DOCUMENTS; i++){
91                      scores[i]=0;
92                  }
93
94                  for (IndexHitList entry : entries) {
95                      ArrayList<SimpleIndexHit> hitlist = ((SimpleIndexHitList)entry
                            ).getHitList();
96
97                      for (SimpleIndexHit record : hitlist) {
98                          scores[record.getDocid()]+=record.getHits();
99                      }
100                 }
101
102                 for (int i=0; i<Config.NUMBER_OF_DOCUMENTS; i++){
103                     if (scores[i]>0) res.addHits(i, scores[i]);
104                 }
105                 return res;
106             }
107         }
108     }
109
110
111     public static IndexHitList interleaveEntries(IndexHitList lista, IndexHitList
            listb){
112         if (lista==null) return listb;
113         if (listb==null) return lista;
114         if (Config.SIMULATED_ONLY){
115             if (Config.USE_AND_QUERIES)
116                 return new SimulatedIndexHitList(
117                         ((SimulatedIndexHitList) lista).getDocFrequency() *
118                         ((SimulatedIndexHitList) listb).getDocFrequency());
119             else
120               return new SimulatedIndexHitList( 1 -
121                       (1-((SimulatedIndexHitList) lista).getDocFrequency()) *
```

```
122                     (1-((SimulatedIndexHitList) listb).getDocFrequency()));
123         } else {
124                 SimpleIndexHitList res = new SimpleIndexHitList();
125                 ArrayList<SimpleIndexHit> hitsa = ((SimpleIndexHitList)lista).
                        getHitList();
126                 ArrayList<SimpleIndexHit> hitsb = ((SimpleIndexHitList)listb).
                        getHitList();
127                 int as = hitsa.size();
128                 int bs = hitsb.size();
129                 int ap=0, bp=0;
130                 while (ap<as && bp<bs ){
131                     SimpleIndexHit hita = hitsa.get(ap);
132                     SimpleIndexHit hitb = hitsb.get(bp);
133                     int hits1 = hita.getHits();
134                     int hits2 = hita.getHits();
135
136                     if ( hita.compareTo(hitb) < 0 ){
137                         if (!Config.USE_AND_QUERIES) res.addHits(hita.getDocid(),
                                hits1);
138                         ap++;
139                     } else if ( hita.compareTo(hitb) > 0){
140                         if (!Config.USE_AND_QUERIES) res.addHits(hitb.getDocid(),
                                hits2);
141                         bp++;
142                     } else {
143                         if (Config.USE_AND_QUERIES) res.addHits(hitb.getDocid(),
                                hits1<hits2?hits1:hits2);
144                         else res.addHits(hitb.getDocid(), hits1+hits2);
145                         ap++; bp++;
146                     }
147                 }
148                 if (!Config.USE_AND_QUERIES) {
149                     if (ap==as){
150                         SimpleIndexHit hitb;
151                         for (; bp < bs; bp++){
152                             hitb = hitsb.get(bp);
153                             res.addHits(hitb.getDocid(), hitb.getHits());
154                         }
155                     } else if (bp==bs){
156                         SimpleIndexHit hita;
157                         for (; ap < as; ap++){
158                             hita = hitsa.get(ap);
159                             res.addHits(hita.getDocid(), hita.getHits());
160                         }
161                     }
162                 }
163             return res;
164         }
165     }
166
167 public static IndexHitList mergeEntries(IndexHitList lista, IndexHitList listb){
168     if (lista==null) return listb;
169     if (listb==null) return lista;
170     if (Config.SIMULATED_ONLY){
171             return new SimulatedIndexHitList(
172                     ((SimulatedIndexHitList) lista).getDocFrequency() +
173                     ((SimulatedIndexHitList) listb).getDocFrequency());
174     } else {
175             SimpleIndexHitList res = new SimpleIndexHitList();
176             ArrayList<SimpleIndexHit> hitsa = ((SimpleIndexHitList)lista).
                    getHitList();
```

```
177             ArrayList<SimpleIndexHit> hitsb = ((SimpleIndexHitList)listb).
                    getHitList();
178             int as = hitsa.size();
179             int bs = hitsb.size();
180             int ap=0, bp=0;
181             while (ap<as && bp<bs ){
182                 SimpleIndexHit hita = hitsa.get(ap);
183                 SimpleIndexHit hitb = hitsb.get(bp);
184                 int hits1 = hita.getHits();
185                 int hits2 = hita.getHits();
186 
187                 if ( hita.compareTo(hitb) < 0 ){
188                     res.addHits(hita.getDocid(), hits1);
189                     ap++;
190                 } else if ( hita.compareTo(hitb) > 0){
191                     res.addHits(hitb.getDocid(), hits2);
192                     bp++;
193                 }
194             }
195             if (ap==as){
196                 SimpleIndexHit hitb;
197                 for (; bp < bs; bp++){
198                     hitb = hitsb.get(bp);
199                     res.addHits(hitb.getDocid(), hitb.getHits());
200                 }
201             } else if (bp==bs){
202                 SimpleIndexHit hita;
203                 for (; ap < as; ap++){
204                     hita = hitsa.get(ap);
205                     res.addHits(hita.getDocid(), hita.getHits());
206                 }
207             }
208         return res;
209     }
210 }
211 }
```

B.1.30 simulation.query.Query

```
1  package simulation.query;
2  import java.util.*;
3 
4  import simulation.model.Config;
5  import desmoj.core.simulator.*;
6 
7  public class Query extends Entity{
8      private HashMap<Integer, IndexHitList> termHits;
9 
10     public Query(Model owner, String name, boolean showInTrace) {
11         super(owner, name, showInTrace);
12         termHits = new HashMap<Integer, IndexHitList>();
13     }
14 
15     public void addTerm(int termid, IndexHitList hitlist){
16         termHits.put(termid, hitlist);
17     }
18 
19     public int getNumberOfTerms(){
20         return termHits.size();
21     }
22 
23     public ArrayList<Integer> getTerms(){
```

```
24          ArrayList<Integer> res = new ArrayList<Integer>(termHits.keySet());
25          Collections.sort(res);
26          return res;
27      }
28
29      public double getTotalIndexSize(){
30          double ret = 0.0;
31          for (Integer key : termHits.keySet()) {
32              ret+=termHits.get(key).getNumDocs()*Config.
                    SIZE_OF_INDEX_DOCUMENT_ENTRY;
33          }
34          return ret;
35      }
36
37      public void eliminateNonExistingTerms(){
38          ArrayList<Integer> keyset = new ArrayList<Integer>(termHits.keySet());
39          int i=0;
40          while (i<keyset.size()){
41              if (keyset.get(i) < 0){
42                  termHits.remove(keyset.get(i));
43                  keyset.remove(i);
44              } else {
45                  i++;
46              }
47          }
48      }
49
50      public ArrayList<SubQuery> partitionByDoc(){
51          ArrayList<SubQuery> res = new ArrayList<SubQuery>();
52          SubQuery subs[] = new SubQuery[Config.NUMBER_OF_NODES];
53          for (int i=0; i < Config.NUMBER_OF_NODES; i++){
54              subs[i] = new SubQuery(getModel(), "SubQuery", false, i, this);
55          }
56          if (Config.SIMULATED_ONLY){
57              for (Integer termid : termHits.keySet()) {
58                  double freqpart = ((SimulatedIndexHitList)termHits.get(termid)).
                        getDocFrequency()/Config.NUMBER_OF_NODES;
59                  for (int i=0; i < Config.NUMBER_OF_NODES; i++){
60                      subs[i].addTermFreq(termid, freqpart);
61                  }
62              }
63          } else {
64              for (Integer termid : termHits.keySet()) {
65                  ArrayList<SimpleIndexHit> hitlist = ((SimpleIndexHitList)termHits.
                        get(termid)).getHitList();
66                  for (SimpleIndexHit hit : hitlist){
67                      subs[hit.getDocid() % Config.NUMBER_OF_NODES].addTermHits(
                            termid, hit.getDocid(), hit.getHits());
68                  }
69              }
70          }
71          for (int i=0; i<Config.NUMBER_OF_NODES; i++){
72              //if ( subs[i].getNumberOfTerms()>0 ) res.add(subs[i]);
73              //just add it anyway since the query is broadcasted
74              res.add(subs[i]);
75          }
76
77          return res;
78      }
79
80      public ArrayList<SubQuery> partitionByTerm(){
81          ArrayList<SubQuery> res = new ArrayList<SubQuery>();
```

```
82          SubQuery subs[] = new SubQuery[Config.NUMBER_OF_NODES];
83          for (int i=0; i < Config.NUMBER_OF_NODES; i++){
84              subs[i] = new SubQuery(getModel(), "SubQuery", false, i, this);
85          }
86
87          //same for both simulated and real
88          for (Integer termid : termHits.keySet()) {
89              subs[termid % Config.NUMBER_OF_NODES].addTerm(termid, termHits.get(
                    termid));
90          }
91
92          for (int i=0; i<Config.NUMBER_OF_NODES; i++){
93              if ( subs[i].getNumberOfTerms()>0 ) res.add(subs[i]);
94          }
95
96          return res;
97      }
98
99
100     public ArrayList<SubQuery> partitionHybrid(){
101
102         ArrayList<SubQuery> res = new ArrayList<SubQuery>();
103         SubQuery subs[] = new SubQuery[Config.NUMBER_OF_NODES];
104         for (int i=0; i < Config.NUMBER_OF_NODES; i++){
105             subs[i] = new SubQuery(getModel(), "SubQuery", false, i, this);
106         }
107
108         if (Config.SIMULATED_ONLY){
109             double chunkimpact=((double) Config.HYBRID_CHUNK_SIZE)/Config.
                    NUMBER_OF_DOCUMENTS;
110
111             for (Integer termid : termHits.keySet()) {
112
113                 double frequency=((SimulatedIndexHitList)termHits.get(termid)).
                        getDocFrequency();
114                 int chunkcnt =(int) Math.ceil(frequency / chunkimpact);
115
116                 for (int i=0; i<chunkcnt; i++){
117                     int node = (i ^  termid) % Config.NUMBER_OF_NODES;
118                     if (i==chunkcnt-1){
119                         subs[node].addTermFreq(termid, frequency-chunkimpact*(
                                chunkcnt-1));
120                     } else {
121                         subs[node].addTermFreq(termid, chunkimpact);
122                     }
123                 }
124             }
125         } else {
126             for (Integer termid : termHits.keySet()) {
127                 ArrayList<SimpleIndexHit> hitlist = ((SimpleIndexHitList)termHits.
                        get(termid)).getHitList();
128
129                 int currChunkId = 0;
130                 int currChunkEntry = 0;
131                 for (SimpleIndexHit hit : hitlist){
132                     int docid = hit.getDocid();
133                     int numhits = hit.getHits();
134                     int node = (termid ^ currChunkId) % Config.NUMBER_OF_NODES;
135                     if (currChunkEntry + numhits < Config.HYBRID_CHUNK_SIZE){
136                         subs[node].addTermHits(termid,
137                                 docid, numhits);
138                         currChunkEntry+=numhits;
```

```
139                     } else {
140                         while (numhits > 0){
141                             int newnumhits = numhits-Config.HYBRID_CHUNK_SIZE;
142                             subs[node].addTermHits(termid,
143                                     docid, (newnumhits>0)?Config.HYBRID_CHUNK_SIZE
                                            :numhits);
144                             numhits=newnumhits;
145                             currChunkId++;
146                             currChunkEntry=0;
147                         }
148                     }
149                 }
150             }
151         }
152         for (int i=0; i<Config.NUMBER_OF_NODES; i++){
153             if ( subs[i].getNumberOfTerms()>0 ) res.add(subs[i]);
154         }
155 
156         return res;
157     }
158 }
```

B.1.31 simulation.query.QueryResult

```
1  package simulation.query;
2  import java.util.*;
3  
4  import desmoj.core.simulator.*;
5  import simulation.*;
6  import simulation.model.Config;
7  
8  public class QueryResult extends Entity{
9      private ArrayList<SubQueryResult> results;
10     private IndexHitList hitlist;
11 
12     public QueryResult(Model owner, String name, boolean showInTrace) {
13         super(owner, name, showInTrace);
14         results = new ArrayList<SubQueryResult>();
15     }
16 
17     public void addResult(SubQueryResult result){
18         results.add(result);
19     }
20 
21     public void combineResults(){
22         if (hitlist!=null) return;
23         ArrayList<IndexHitList> hitlists = new ArrayList<IndexHitList>();
24         for (SubQueryResult result : results) {
25             hitlists.add(result.getHitList());
26         }
27         if (Config.INDEXING_MODE == Config.LOCAL_INDEXING ||
28             Config.INDEXING_MODE == Config.LOCAL_INDEXING_PARALLEL_MERGE
29         ) hitlist = IndexTools.mergeEntries(hitlists);
30         else hitlist=IndexTools.interleaveEntries(hitlists);
31     }
32 
33     public void setHitList(IndexHitList hitlist){
34         this.hitlist=hitlist;
35     }
36     public IndexHitList getHitList(){
37         if (hitlist==null)
38             if (results.size()>0) combineResults();
```

```
39              else return null;
40          return hitlist;
41      }
42
43      public int getNumberOfDocuments(){
44          if (hitlist==null) combineResults();
45          return hitlist.getNumDocs();
46      }
47  }
```

B.1.32 simulation.query.SimpleIndexHit

```
1   package simulation.query;
2   public class SimpleIndexHit implements Comparable<SimpleIndexHit>{
3           private int docid, hits;
4
5           public SimpleIndexHit(int docid, int hits){
6               this.docid=docid;
7               this.hits=hits;
8           }
9
10          public int getDocid(){
11              return docid;
12          }
13
14          public int getHits(){
15              return hits;
16          }
17
18          public int compareTo(SimpleIndexHit hit) {
19              if ( this.docid > hit.docid) return 1;
20              else if ( this.docid < hit.docid) return -1;
21              else return 0;
22          }
23  }
```

B.1.33 simulation.query.SimulatedIndexHitList

```
1   package simulation.query;
2
3   import simulation.model.Config;
4
5   public class SimulatedIndexHitList implements IndexHitList{
6       private double frequency;
7
8       public SimulatedIndexHitList(){
9           frequency=0.0;
10      }
11
12      public SimulatedIndexHitList(double frequency) {
13          this.frequency = frequency;
14      }
15
16      public void addHits(double freqfrac){
17          frequency +=freqfrac;
18      }
19
20      public int getNumDocs(){
21          return (int) (Config.NUMBER_OF_DOCUMENTS * frequency);
22      }
23
```

```
24      public double getDocFrequency(){
25          return frequency;
26      }
27  }
```

B.1.34 simulation.query.SubQuery

```
1   package simulation.query;
2   import java.util.*;
3
4   import desmoj.core.simulator.*;
5
6   public class SubQuery extends Entity{
7       private HashMap<Integer, IndexHitList> termHits;
8       private int nodeNumber;
9       private Query query;
10
11      public SubQuery(Model owner, String name, boolean showInTrace){
12          super(owner, name, showInTrace);
13          termHits = new HashMap<Integer, IndexHitList>();
14      }
15
16      public SubQuery(Model owner, String name, boolean showInTrace, int nodeNumber,
             Query query){
17          super(owner, name, showInTrace);
18          termHits = new HashMap<Integer, IndexHitList>();
19          this.nodeNumber = nodeNumber;
20          this.query = query;
21      }
22
23      public int getNumberOfTerms(){
24          return termHits.size();
25      }
26
27      public ArrayList<Integer> getTermList(){
28          return new ArrayList<Integer>(termHits.keySet());
29      }
30
31      public IndexHitList getHitList(int termid){
32          return termHits.get(termid);
33      }
34
35      public int getNumberOfDocuments(int termid){
36          return termHits.get(termid).getNumDocs();
37      }
38
39      public int getNodeNumber(){
40          return nodeNumber;
41      }
42
43      public void addTerm(int termid, IndexHitList hitlist){
44          termHits.put(termid, hitlist);
45      }
46
47      /**
48       * used only by SimulatedIndexHitList
49       */
50      public void addTermFreq(int termid, double freq) {
51          IndexHitList hitlist = termHits.get(termid);
52          if (hitlist == null){
53              hitlist = new SimulatedIndexHitList();
54              termHits.put(termid, hitlist);
```

```
55          }
56          ((SimulatedIndexHitList) hitlist).addHits(freq);
57      }
58
59      /**
60       * used only by SimpleIndexHitList
61       */
62      public void addTermHits(int termid, int docid, int hits) {
63          IndexHitList hitlist = termHits.get(termid);
64          if (hitlist == null){
65              hitlist = new SimpleIndexHitList();
66              termHits.put(termid, hitlist);
67          }
68          ((SimpleIndexHitList) hitlist).addHits(docid, hits);
69      }
70
71      public SubQueryResult getSubQueryResult(){
72          return new SubQueryResult(getModel(), "SubQueryResult", false,
73                  new ArrayList<IndexHitList>(termHits.values()));
74      }
75
76      public int getLowestScore() {
77          int lowest = Integer.MAX_VALUE;
78          for (IndexHitList hitlist : termHits.values()) {
79              if (hitlist.getNumDocs() < lowest ) lowest = hitlist.getNumDocs();
80          }
81          return lowest;
82      }
83
84      public Query getQuery(){
85          return query;
86      }
87
88      public ArrayList<Integer> getTerms(){
89          ArrayList<Integer> res = new ArrayList<Integer>(termHits.keySet());
90          Collections.sort(res);
91          return res;
92      }
93
94      public void eliminateNonExistingTerms(){
95          ArrayList<Integer> keyset = new ArrayList<Integer>(termHits.keySet());
96          int i=0;
97          while (i<keyset.size()){
98              if (keyset.get(i) < 0){
99                  termHits.remove(keyset.get(i));
100                 keyset.remove(i);
101             } else {
102                 i++;
103             }
104         }
105     }
106
107     public ArrayList<Integer> getSortedTermList() {
108         // TODO Auto-generated method stub
109         int cnt = termHits.size();
110         if (cnt==1){
111             return new ArrayList<Integer>(termHits.keySet());
112         }
113         ArrayList<Integer> res = new ArrayList<Integer>(cnt);
114
115         ArrayList<Pair> vals = new ArrayList<Pair>(termHits.size());
116         for (Integer key : termHits.keySet()){
```

```
117             vals.add(new Pair(key, getNumberOfDocuments(key)));
118         }
119         Collections.sort(vals);
120 
121         for (int i=0; i<cnt; i++){
122             res.add(vals.get(i).getId());
123         }
124 
125         return res;
126     }
127 
128     private class Pair implements Comparable<Pair>{
129         private int id, value;
130 
131         public Pair(int id, int value){
132             this.id = id;
133             this.value = value;
134         }
135 
136         public int compareTo(Pair p){
137             if (value > p.value) return 1;
138             else if (value < p.value) return -1;
139             else return 0;
140         }
141 
142         public int getId(){
143             return id;
144         }
145     }
146 }
```

B.1.35 simulation.query.SubQueryResult

```
1  package simulation.query;
2  import java.util.*;
3 
4  import desmoj.core.simulator.*;
5 
6  public class SubQueryResult extends Entity {
7      private IndexHitList hitlist;
8 
9      public SubQueryResult(Model owner, String name, boolean showInTrace) {
10         super(owner, name, showInTrace);
11     }
12 
13     public SubQueryResult(Model owner, String name, boolean showInTrace,
           IndexHitList hitlist){
14         super(owner, name, showInTrace);
15         this.hitlist = hitlist;
16     }
17 
18     public SubQueryResult(Model owner, String name, boolean showInTrace, ArrayList
           <IndexHitList> hitlists){
19         super(owner, name, showInTrace);
20         this.hitlist = IndexTools.interleaveEntries(hitlists);
21     }
22 
23     public IndexHitList getHitList(){
24         return hitlist;
25     }
26 
27     public int getNumberOfDocuments(){
```

```
28          return  (hitlist != null) ? hitlist.getNumDocs() : 0;
29      }
30  }
```