Truls A. Bjørklund

Column Stores versus Search Engines and Applications to Search in Social Networks

Thesis for the degree of philosophiae doctor

Trondheim, June 2011

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU
Innovation and Creativity

NTNU
Norwegian University of Science and Technology

Doctoral thesis
for the degree of philosophiae doctor

Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Computer and Information Science

©Truls A. Bjørklund

ISBN 978-82-471-2838-1 (printed version)
ISBN 978-82-471-2840-4 (electronic version)
ISSN 1503-8181

Doctoral theses at NTNU, 2011:148

Printed by NTNU-trykk

Typeset with LaTeX 2_ε in Computer Modern 10pt

Abstract

Search engines and database systems both play important roles as we store and organize ever increasing amounts of information and still require the information to be easily accessible. Research on these two types of systems has traditionally been partitioned into two fields, information retrieval and databases, and the integration of these two fields has been a popular research topic. Rather than attempting to integrate the two fields, this thesis begins with a comparison of the technical similarities between search engines and a specific type of database system often used in decision support systems: column stores. Based on an initial assessment of the technical similarities, which includes an evaluation of the feasibility of creating a hybrid system that supports both workloads, the papers in this thesis investigate how the identified similarities can be used as a basis for improving the efficiency of the different systems.

To improve the efficiency of processing decision support workloads, the use of inverted indexes as an alternative to bitmap indexes is evaluated. We develop a query processing framework for compressed inverted indexes in decision support workloads and find that it outperforms state-of-the-art compressed bitmap indexes by being significantly more compact, and also improves the query processing efficiency for most queries.

Keyword search in social networks with access control is also addressed in this thesis, and a space of solutions is developed along two axes. One of the axes defines the set of inverted indexes that are used in the solution, and the other defines the meta-data used to filter out inaccessible results. With a flexible and efficient search system based on a column-oriented storage system, we conduct a thorough set of experiments that illuminate the trade-offs between different extremes in the solution space. We also develop a hybrid scheme in between two of the best extremes. The hybrid approach uses cost models to find the most efficient solution for a particular workload. Together with an efficient query processing framework based on our novel HeapUnion operator, this results in a system that is efficient for a wide range of workloads that consist of updates and searches with access control in a social network.

Preface

This doctoral thesis was submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree philosophiae doctor (PhD). The work herein was performed at the Department of Computer and Information Science, NTNU, and at the Computer Science Department at Cornell University. Professor Bjørn Olstad has been the main supervisor, while Øystein Torbjørnsen and Magnus Lie Hetland have co-supervised the work.

Acknowledgments

First and foremost, I would like to thank Johannes Gehrke. Despite his extremely busy schedule, he has still managed to find time to help me, and has in practice been my main supervisor throughout this work. His feedback and encouragement has improved the quality of my work dramatically, and I am very grateful. I would also like to thank my co-authors, including Nils Grimsmo, Michaela Götz, Magnus Lie Hetland and Øystein Torbjørnsen. They have all contributed with fruitful discussions and helpful feedback, and Øystein Torbjørnsen has also been my day-to-day supervisor in the early stages of the work with this thesis. Furthermore, I would like to thank my family and friends for their encouragement and support.

Contents

List of Figures

Chapter 1

Introduction

Search engines and database systems both play important roles as we store and organize ever increasing amounts of information. They both store and process large data volumes, and provide ways of querying the data. However, research on search engines and database systems has traditionally been partitioned into two separate fields; information retrieval (IR) and databases (DB), and the integration of the two fields has been a long-withstanding research challenge [8, 25, 62, 101]. Rather than attempting to integrate the two fields, this thesis uses an investigation of the similarities and differences between search engines and a specific type of database systems as a starting point for further research. The type of database systems that we focus on is called column stores, and is mainly used in decision support systems. We give a brief introduction to both decision support systems and search engines in this chapter, as a basis for a description of the research questions we address in this thesis.

1.1 Decision Support Systems

Decision Support Systems (DSSs) usually store historical structured data about an enterprise, and make the information available for analyses to guide and support high-level decision making [83]. The data is organized into tables, and so-called *star schemas* represent a best practice for the table organization. Star schemas are characterized by a large central fact table that references several smaller dimension tables [75, 78, 83]. As an example, consider the star schema presented in Figure 1.1. The fact table in the example is called Sales, and each sale is associated with a product, a supplier and a customer, all of which are represented in dimension tables.

To use the data in DSSs to guide decision-making, the information is usually processed with so-called On-Line Analytical Processing (OLAP), which is dominated by ad hoc, complex queries [83]. Typical queries on the table structure in Figure 1.1 could for

Customer
customer ID
name
address

(a) Customer table

Sales
sale ID
time
product ID
customer ID
supplier ID
quantity

(b) Sales table

Supplier
supplier ID
name
address

(c) Supplier table

Product
product ID
name
brand
price

(d) Product table

Figure 1.1: Example star schema

example find the total price of all sales to a particular customer, the most popular supplier for a particular product, or the month of the year with highest total sales. To support such queries, the tables are usually stored in a database system, and column stores are considered an attractive solution for decision support workloads [32, 33, 93].

1.1.1 Column Stores

The primary difference between column stores and more traditional database systems, referred to as row stores, is the storage layout used when the data in a table is represented on disk or in main memory. In a column store, each attribute/column in the table is stored separately, and the attribute value for all tuples are stored together in a (more or less) contiguous storage segment. In a row store, however, the values of all attributes for a single tuple (row) are stored together. The primary advantage of column stores, which makes them well suited for decision support workloads, is their ability to avoid reading columns that are not required when processing a query, while the cost of updates and of reconstructing tuples during query processing are potential disadvantages [53]. The basic idea of a column store has been known for many years [45], but they have gained significant attention recently through systems like MonetDB [32], MonetDB/X100 [33] and C-store [93].

1.2 Search Engines

Search engines are best known to users through web search engines like Google and Bing, which help users find information on the web. Other types of search engines include

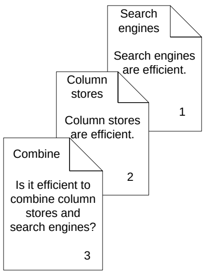

Search engines
Search engines are efficient.
1
Column stores
Column stores are efficient.
2
Combine
Is it efficient to combine column stores and search engines?
3

Figure 1.2: Example documents

Terms		**Posting lists**
and	$\longrightarrow$	$(3,1)$
are	$\longrightarrow$	$(1,1)$ $(2,1)$
column	$\longrightarrow$	$(2,2)$ $(3,1)$
combine	$\longrightarrow$	$(3,2)$
efficient	$\longrightarrow$	$(1,1)$ $(2,1)$ $(3,1)$
engines	$\longrightarrow$	$(1,2)$ $(3,1)$
is	$\longrightarrow$	$(3,1)$
it	$\longrightarrow$	$(3,1)$
search	$\longrightarrow$	$(1,2)$ $(3,1)$
stores	$\longrightarrow$	$(2,2)$ $(3,1)$
to	$\longrightarrow$	$(3,1)$

Figure 1.3: Logical view of inverted index for example documents

desktop search engines that allow users to search in the data on their own computer, and enterprise search engines that allow users to search in the data within an enterprise.

Common to all types of search engines is that users typically specify their information need through keywords. In their most basic forms, search engines take a set of keywords as input and return a set of relevant documents. Which documents that are relevant is determined based on a ranking heuristic; documents are generally considered more relevant than others when they have more occurrences of the keywords relative to the length of the document. To support such ranked queries efficiently, search engines usually construct an inverted index over the document collection [18, 102, 112].

1.2.1 Inverted Indexes

An inverted index for a document collection consists of two parts. First, all unique searchable words in the collection, denoted terms, are stored in a structure called the dictionary (also referred to as the vocabulary structure or the lexicon). A look-up in the dictionary based on a term is used to find a pointer to the term's posting list, which we also refer to as an inverted list. The second part consists of the posting lists for all terms. Each entry in the posting lists is referred to as a posting, and it typically contains a reference to a document that contains the term, as well as data used to rank the documents [18, 102, 112]. As an example, consider the documents in Figure 1.2 taken from Paper I. A logical view of an inverted index for the documents is found in Figure 1.3. In the example, each posting contains two values: (1) The document identifier (ID) of the document that contains the term (the document identifiers are found in the bottom right corner of the documents in Figure 1.2). (2) The number of occurrences of the term in the document; a value that is used in several ranking schemes [18, 85, 102, 112].

An inverted index can be interpreted as a representation of a large table, where there is a column for each term and a row for each document. The inverted index is a column-oriented representation of this table, and it is this basic observation that motivates the starting point of this thesis, namely an investigation of the technical similarities and differences between column stores and search engines based on inverted indexes.

1.3 Research Questions

The main research question for this thesis is:

> *What are the technical similarities and differences between column stores for decision support workloads and search engines, and how can systems for either workload benefit from the similarities?*

In pursuit of answers to the overall question, several smaller and more specific research questions have been addressed:

1. Which technical aspects of column stores and search engines are similar, and which are different?
2. How can identified similarities be used as a basis to lower the processing time for search workloads?
3. How can identified similarities be used as a basis to lower the processing time for decision support workloads?

1.4 Thesis Outline

This thesis is a collection of papers, and the main part of the thesis including all research results is therefore found in the papers. In Chapter 2, we give an introduction to background material, while Chapter 3 presents an overview of the research process and papers. Chapter 4 concludes the thesis and points to interesting directions for future work. The actual papers are found in Appendix A. Furthermore, during the work with this thesis I have also been involved in other research projects, and Appendix B provides an overview of some of the other papers where I have contributed, as well as an overall introduction to the topic of these papers. Appendix C contains an errata for the included papers.

Chapter 2

Background

In this chapter, we present background information to ease the understanding of the papers included in this thesis. We discuss Decision Support Systems (DSSs) in Section 2.1, and search engines in Section 2.2.

2.1 Decision Support Systems

We introduced DSSs briefly in Chapter 1.1, and have explained that column stores are considered as being well suited for decision support workloads and that star schemas are a best practice for the organization of tables in DSSs. This section discusses implications of a columnar storage layout, and introduces query processing techniques that are important for decision support workloads.

2.1.1 Columnar Storage

As explained in Chapter 1.1.1, the main discriminative factor between column stores and more traditional row-oriented database systems is that column stores store the attributes/columns of a table separately instead of storing the value of all attributes for a particular tuple together, as is done in row stores. A columnar storage layout has several implications: First, when processing a query on a particular table, only the columns that are actually part of the query need to be accessed, which often results in less data read from disk/memory compared to in a row store [53, 93]. However, the results of most queries are returned as tuples, so actual tuples need to be stitched together from the separate columns somewhere in the query plan. This can result in an overhead compared to when using row stores [4]. Furthermore, because all attributes of a tuple are stored in separate locations, adding a new tuple is slightly more involved in a column store compared to in a row store [60, 93]. Last, a columnar storage layout facilitates compression schemes that combine multiple entries in a column into one representation, like Run-Length Encoding (RLE) [2, 113].

The additional challenges associated with adding new tuples and updates in column stores are usually addressed by having two different structures which in combination represent the state of the database. The main part of the database is stored in a large and relatively static read-optimized structure, while new updates are accumulated in a smaller dynamic in-memory structure [60, 93]. In the C-store system, the small dynamic part is called Write Store (WS), and the larger read-optimized part is called Read Store (RS) [93]. At given intervals, the updates from WS are moved into RS in a batch process, a strategy that avoids the overhead associated with updating all columns for each new tuple. The RS is based on a Log-Structured Merge (LSM) tree [77], where tables are organized in a hierarchy with partitions of different sizes, and tuples are moved towards the larger partitions over time. As in more traditional database systems, support for consistency of operations is also important for column stores. Snapshot isolation, which guarantees that all reads see a consistent version of the database, is commonly used [27, 93].

The second mentioned implication of a columnar storage layout, namely the problem of when and how to stitch together the entries from different columns to form a tuple, referred to as materialization, has also been addressed in the literature [3, 4]. However, the introduced solutions are not important for the understanding of this thesis, and we therefore do not discuss them any further. We do, however, discuss the last mentioned implication of a columnar storage layout in some detail in Section 2.1.1.1 because the compression schemes used in column stores have relations to the compression schemes used in inverted indexes. Furthermore, in Section 2.1.1.2 we discuss some variants of the straight-forward column-oriented storage model we have considered up until now.

2.1.1.1 Compression

As mentioned above, storing each column of a table separately facilitates compression schemes that combine the representation of multiple entries in one column. We explain two such compression schemes in this section, RLE and PForDelta [2, 113]. The explanation is based on an example with some tuples for a subset of the columns in the Product table in Figure 1.1; the tuples are shown in Figure 2.1.

RLE: Run-Length Encoding represents runs with the same value together, typically in the form of $\langle value, runlength \rangle$ pairs. RLE therefore leads to compact representations of columns where there are reasonably sized runs of the same value. It is more likely for consecutive tuples to share a value for one attribute than for the complete tuples to be identical, which makes RLE more applicable in column stores compared to row stores [2].

In the example tuples in Figure 2.1, the Price column has runs with reasonable sizes and is therefore a candidate for RLE compression. The whole column in the example can be represented with 4 pairs: $\langle 10, 4 \rangle, \langle 9, 1 \rangle, \langle 20, 2 \rangle, \langle 10, 1 \rangle$.

PForDelta: The values in the Product ID column in Figure 2.1 are sorted, and the column is therefore a promising candidate for PForDelta compression. PForDelta was

Product ID	Price
1	10
2	10
5	10
7	10
15	9
18	20
20	20
21	10

Figure 2.1: Example tuples

originally introduced as a compression scheme in a column store [113], but has later become popular in search engines as well [109, 111]. It is also used in implementations associated with several papers in this thesis.

PForDelta is based on delta compression, and each delta is represented with PFor [113]. With delta compression, each entry is represented as the delta from the previous entry. The Product ID column in the example from Figure 2.1 would thus be represented as $1, 1, 3, 2, 8, 3, 2, 1$. The main idea of delta compression is that the deltas are generally smaller than the actual entries, and compression is achieved by using a representation of the deltas where small values require less storage space than larger ones; PFor is used as such a method in PForDelta.

PFor is basically an extension of bit-packing that also allows exceptions. A given number of bits is reserved for each entry, just like when using bit-packing. The entries that are small enough to be represented with the chosen number of bits are stored straightforwardly, while the rest are stored as ordinary integers, and are referred to as exceptions. Compared to bit-packing, this approach has the advantage that it is not necessary to choose the number of bits large enough to represent all values in a column, but it can rather be chosen to fit most values, leaving only a few to be represented as exceptions [113]. In the Product ID column from Figure 2.1 all deltas except 8 can be represented with 2 bits, and 2 is therefore a reasonable number of bits to use per entry to achieve a compact representation.

With PFor, batches of b entries are compressed and decompressed together, where b is set to a multiple of 32 [111, 113]. This approach leads to a solution with fast compression and decompression because b values can be compressed or decompressed virtually without branch mispredictions, and branch mispredictions usually have a negative impact on processing efficiency [113].

2.1.1.2 Alternative Storage Layouts

Column stores and row stores represent two alternative storage layouts, but other alternatives also exist; it is for example possible to store different subsets of the columns of a table together. A related approach is used in C-store where a table can be represented as a set of projections, where each projection contains a subset of the columns of a table (and potentially also columns from other tables) [93]. All columns in each projection are stored separately, but a common sort order is chosen for each projection. It is thus possible to adapt the sort orders to the query workload.

Another storage layout that can be interpreted as an intermediate strategy in between column stores and row stores is called PAX (Partition Attributes Across) [5]. In PAX, a tuple is always stored within a single page just like in a traditional row store, but the tuples within each page are stored with a columnar layout. This scheme has several attractive properties: First, when a tuple is stored within one page, it is straight-forward to combine the attributes into tuples during query processing without excessive disk or memory accesses. Updates are also supported efficiently because a tuple is confined to one page. Furthermore, the columnar storage layout within each page results in better cache utilization when only a subset of the columns are accessed during queries [5].

To clarify the differences between row stores, column stores and PAX, we present an example based on the tuples in Figure 2.1. We assume that each page can fit 4 integers and do not consider storage of meta-data. With a traditional row store, the tuples in Figure 2.1 can be stored as shown in Figure 2.2, while Figure 2.3 shows a straight-forward columnar layout and Figure 2.4 shows a layout based on PAX. In all figures, we use I and P to denote that the stored value is a Product ID or Price, respectively.

I	P	I	P
1	10	2	10

(a) Page 1

I	P	I	P
5	10	7	10

(b) Page 2

I	P	I	P
15	9	18	20

(c) Page 3

I	P	I	P
20	20	21	10

(d) Page 4

Figure 2.2: Row-oriented layout for example tuples

I	I	I	I
1	2	5	7

(a) Page 1

I	I	I	I
15	18	20	21

(b) Page 2

P	P	P	P
10	10	10	10

(c) Page 3

P	P	P	P
9	20	20	10

(d) Page 4

Figure 2.3: Column-oriented layout for example tuples

I	I	P	P
1	2	10	10

(a) Page 1

I	I	P	P
5	7	10	10

(b) Page 2

I	I	P	P
15	18	9	20

(c) Page 3

I	I	P	P
20	21	20	10

(d) Page 4

Figure 2.4: PAX layout for example tuples

2.1.2 Query Processing

As described in Chapter 1.1, decision support workloads are dominated by complex ad hoc queries, often over star schemas. One common type of query in such schemas is called star joins [78, 83]. A star join is a join of the fact table with several dimension tables in a star schema, and the queries typically perform aggregations over the tuples in the fact table that reference tuples in dimension tables that satisfy certain predicates [78]. An example query over the star schema in Figure 1.1 that would lead to a star join is to ask for all sales where both the customer and the supplier come from the United States. Certain customers and suppliers satisfy these predicates, and the query asks for all sales that reference both a relevant customer and supplier, resulting in a star join.

Join indexes are frequently used to improve the efficiency of star joins. They are basically a pre-computed mapping between tuples that join in the different tables. One straightforward type of join index for star schemas is based on bitmap indexes [78]. For each tuple in a dimension table, the index has a bitmap that identifies all fact table entries that reference this tuple. Bitmap indexes can also be constructed over attributes in the fact table to improve the efficiency of queries that involve predicates on the fact table itself. We take a closer look at bitmap indexes next.

It should also be noted that several papers have discussed and compared various column store specific processing strategies for queries that are relevant in DSSs [3, 4, 33, 53], but these techniques are not important for the understanding of the papers in this thesis and we therefore refer the interested reader to the actual papers. To choose between different query processing plans, Manegold et al. have introduced a cost model for the memory access costs in MonetDB [70]. Although this overall approach is relevant for the hybrid strategy based on cost models introduced in Paper IV in this thesis, we do not discuss the actual model further because it focuses mainly on memory access costs while our models focus on processing time.

2.1.2.1 Bitmap Indexes

A standard bitmap index has one bitmap per unique value of the indexed attribute. The bitmap for each value has as many bits as there are tuples in the relation, and a given bit is set to 1 if the tuple has the value the bitmap represents, and 0 otherwise. To answer predicates involving several attributes and several values, different bitmaps can be combined using bitwise operations like bitwise AND and bitwise OR.

A general problem with standard bitmap indexes is that their size grows with the cardinality of the indexed attribute, resulting in very large indexes for high cardinality attributes. Several strategies have been suggested to address this problem, including binning and bit-sliced indexes. Binning involves letting one bitmap represent several values [64, 92], and the technique efficiently reduces the number of bitmaps in a bitmap index. However, candidate checking is required when using binning to avoid false hits for some queries [86]. Bit-slicing is a technique used for numerical attributes, where one instead of constructing one bitmap per value, has one bitmap per bit in the binary representations of the values

Literal Bitmap	010	100	111	111	011	000	000	000
WAH compression	0010	0100	1110		0011	1011		

Figure 2.5: Example WAH compression

of the indexed attribute [79], and the technique has also been generalized [42, 43]. These techniques may reduce the size of the bitmap index, but they can also have a negative effect on query processing efficiency, especially for simple queries.

The most popular approach to deal with the large size of bitmap indexes for high cardinality attributes is compression, and many compression schemes have been suggested [15, 16, 61, 76, 91, 106]. BBC [15] and WAH [103, 105, 106] are two particularly interesting compression schemes because they are developed to ensure that performing bitwise boolean operations on the bitmaps is still efficient. BBC was introduced first of these two techniques, and WAH is a word-aligned version of BBC. WAH-compressed indexes are slightly larger than indexes compressed with BBC, but query processing is more efficient with WAH [104, 105]. Even though new compression schemes have been introduced later, the query processing efficiency with WAH remains attractive in most cases [76, 91]. WAH-compressed bitmap indexes were therefore chosen as a basis for comparison in Paper II in this thesis.

WAH is essentially a form of run-length encoding, and the compressed bitmaps are partitioned into sequences with $w - 1$ bits in each, where w is the computer word size. If all the bits in a sequence are identical, this and neighboring sequences with the same property are represented in one fill word representing a run of equal bits. If there are both 0's and 1's in the sequence, it is stored as a literal bitmap. The first bit in the word is used to discriminate between fill words and literal words; the bit is set to 1 if it is a fill word and 0 if it is a literal word [103, 105, 106]. The second bit in a fill word is used to define whether it is a fill of 0's or 1's, and the remaining bits describe the fill length as the number of following sequences that have the same bit value. As an example, Figure 2.5 shows how the bit string 010 100 111 111 011 000 000 000 is compressed with WAH when $w = 4$ bits. As shown in the figure, 5 sequences of bits are represented in fill words, while the other 3 are represented as literal bitmaps.

High cardinality attributes are far more compactly represented with WAH-compressed bitmaps than with uncompressed bitmaps, because most values have bitmaps consisting mostly of 0's, resulting in few literal words and many sequences being combined into fill words. The compression is even more efficient if the table is sorted based on the indexed attribute, which typically results in runs of both 1's and 0's, and only a few words being literal bitmaps [66, 81]. Because words containing both 0's and 1's are represented as literal bitmaps, they can easily be combined with other bitmaps in bitwise operations, a fact that contributes to the query processing efficiency observed when using WAH compression [103, 106].

2.2 Search Engines

In Chapter 1.2, the use of inverted indexes in search engines was briefly described. In this section, we provide a more thorough introduction with focus on the structure of inverted indexes and how new data can be added to them. We also discuss query processing, including both basic ranked queries and more advanced query types. Papers III and IV in this thesis focus on access control for search workloads in social networks. Our papers are the first to address this topic, but both access control in search engines and search in social networks are topics that have been addressed previously. We provide a brief introduction to relevant previous work on these topics in Sections 2.2.3 and 2.2.4, respectively.

2.2.1 Inverted Indexes - Structure and Loading

We described inverted indexes in Chapter 1.2.1, where Figure 1.3 showed a logical view of an inverted index for the example documents from Figure 1.2. The example index is referred to as having document-level granularity because each entry in the index describes the occurrence of a term in one document [102]. It is also possible to create indexes with finer granularities, for example by including the actual term positions for all occurrences of the term, resulting in a word-level inverted index. To include the term positions can be useful if phrase queries are supported [18, 102, 112]. Phrase queries require that a set of terms should occur immediately after each other in a specified order.

The exact contents of the postings in an inverted index is typically chosen based on the types of queries and ranking functions that should be supported, and therefore varies between different systems [34]. It is for example also possible to record the context of each occurrence of a term in the index, e.g., whether the term occurs in the title or the body of an HTML document. This information can be used during ranking to favor documents where the search term occurs in the title of the document.

Regardless of the exact contents of the postings, the overall structure of an inverted index remains the same, and consists of a vocabulary structure and the posting lists, as mentioned previously. Many different data structures can be used to store the vocabulary, and B-trees, sorted arrays, hash-based structures and tries are commonly used alternatives [18, 102, 112]. Each posting list is usually stored sequentially, either in memory or on disk, to facilitate efficient scans during query processing [102, 112]. The combined size of the posting lists can be significant, especially when the inverted index has fine granularity, and there is an extensive literature on compression of posting lists [10, 11, 89, 102, 109, 111–113]. We give an introduction to a few commonly used compression methods in Section 2.2.1.1.

When a new document is added to an inverted index, the posting list for every unique term in the document needs to be updated, a process which is potentially inefficient. We give an overview of inverted index construction methods and approaches that support incrementally adding new documents efficiently in Section 2.2.1.2.

2.2.1.1 Compression

How inverted indexes are compressed depends on the sort order in the posting lists, and different sort orders are used depending on both the ranking scheme and the query processing strategy. In the papers in this thesis, the posting lists are usually sorted on document IDs, which is also common in current large-scale search engines [35, 40, 109]. Our introduction in this section therefore focuses on this case, but slight variations of the presented schemes can be used with other sort orders.

The posting lists are usually accessed in a sequential manner, and the document IDs can therefore be represented with delta compression as explained in Section 2.1.1.1. All common compression schemes represent the document IDs as deltas, but the representations of the deltas differ. One simple scheme called VByte represents each delta with an integral number of bytes. The first bit in each byte describes whether or not this is the last byte in the representation of the current delta, while the remaining 7 bits are part of the binary representation of the delta [89]. VByte does not lead to very compact indexes, but can result in faster decompression than schemes with very compact representations, such as Golumbs code [102, 111].

Even though there are many alternative compression methods [10, 102, 112], recent comparison studies have shown that versions of PForDelta (explained in Section 2.1.1.1) combines quite attractive compression ratios with efficient decompression [109, 111, 113]. PForDelta is therefore used to compress document IDs in several implementations used for experiments in the papers in this thesis.

Delta compression can also be useful when compressing the term positions in a word-level inverted index, but it is not very useful when compressing term frequencies. Most frequencies are relatively small, and can therefore be represented just like the deltas between the document IDs, e.g., with PFor. Some approaches specifically adapted to compress the frequencies have also been developed [109], and likewise for term positions [108], but we do not discuss these further here.

2.2.1.2 Loading data

Several methods for constructing inverted indexes have been introduced, but the most efficient approaches usually follow the same overall strategy [37, 55, 102, 112]. They typically partition the document collection into batches of documents, and an inverted index for one batch of documents is accumulated in an updatable structure. When all documents in the batch have been added, the index is stored in compressed form, often on disk. When all batches are processed, all partial indexes are merged to form a single large index [102].

Recent approaches designed to support efficient incremental updates in inverted indexes are usually based on a hierarchy of indexes [36, 39, 52, 69, 71]. In all approaches based on a hierarchy, batches of documents are accumulated in an updatable structure, just like when constructing indexes. However, to ensure low latency indexing it is necessary that the in-memory structure can support queries while loading data. At given intervals, the updatable structure is merged into the hierarchy of indexes. The indexes in the hierarchy are only updated through merges, and they thus support read-only workloads after they have been constructed. Notice the parallels between the updatable structure and the Write Store in C-Store, and between the hierarchy and the Read Store.

The structure of the hierarchy varies between different approaches. The following explanation focuses on a straight-forward strategy called geometric partitioning which was introduced by Lester et al. [68, 69]. With geometric partitioning, all indexes contain a subset of the documents, and the larger indexes contain the least recent documents. The maximum sizes of the different indexes are defined through a configurable parameter, r. Index i has a maximum capacity that is r times the capacity of index $i-1$. Consequently, if r is large, the total number of indexes for a document collection is kept relatively low, while low values of r typically lead to many indexes. This has two implications: First, choosing a high value for r leads to lower query processing costs, because only a limited number of indexes must be accessed to process a query. Second, choosing a low value for r leads to lower update costs, because each batch of documents is involved in fewer merges overall. The choice of r thus represents a trade-off between update and query processing costs [69].

One of the extensions of geometric partitioning was introduced by Büttcher et al. [39]. The main novel feature of their methods is based on the observation that the cost of merges in the hierarchy is dominated by long posting lists. Updates to these long lists can therefore be supported more efficiently with an in-place update strategy, and the hierarchy is only used for short posting lists [39]. Another interesting aspect of their work is that a detailed cost model is developed to analyze the technique. Cost models have also been used in related work to analyze the relative efficiency of inverted index construction methods [102, 112].

2.2.2 Query Processing

The most common type of query in most search engines is ranked queries where users submit a set of keywords and the top-k ranked documents are returned. The documents are selected based on a ranking function that tries to estimate how relevant each document is to the query keywords. Although this process is conceptually simple, it has many aspects. Examples of such aspects include how documents and queries are transformed into sets of terms, what ranking functions that are supported, and how the queries are processed in the inverted index [18, 102, 112].

When documents and queries are parsed to find terms, it is common to ensure that two words are considered similar regardless of whether they are written in upper- or lowercase, and regardless of whether they are written in plural form or not. Although such techniques are important to find the most relevant documents for a particular query, their implications for the work presented in this thesis are limited, and we refer the reader to overviews of such techniques in the literature [18, 102, 112].

A vast number of ranking functions can be used, and they take different meta-data into account when estimating the relevance of a document for a particular query [18, 34, 112]. A traditional set of relatively straight-forward techniques are based on so-called TF-IDF measures, which consist of two parts:

- **TF (Term Frequency)**. Ensures that a document where a query term occurs often relative to the length of the document is ranked higher (given a higher similarity score) than one with few occurrences of the query term relative to its length.
- **IDF (Inverse Document Frequency)** In multi-term queries, query terms that occur rarely in the document collection are given more weight than common terms when computing the similarity score between a document and a query.

The calculation of a similarity score based on TF-IDF is often based on considering both the query and the document as vectors and use the cosine product to find the angle between the two vectors; the cosine of the angle is commonly used as the similarity score. One possible way to define the value for term t in the query vector (given that t occurs in the query) is [112]:

$$w_{q,t} = \ln\left(1 + \frac{N}{f_t}\right)$$

N in the above formula represents the total number of documents in the document collection, while f_t is the number of documents in the collection that contains t. The entry for term t in the vector for document d, given that t occurs in d, can be defined as [112]:

$$w_{d,t} = 1 + \ln f_{d,t}$$

$f_{d,t}$ is defined as the frequency of term t in document d. The similarity score for a document given a query can now be calculated based on the cosine product between the two vectors:

$$S_{q,d} = \frac{\sum_t w_{q,t} w_{d,t}}{\sqrt{\sum_t w_{q,t}^2} \cdot \sqrt{\sum_t w_{d,t}^2}}$$

Notice that the TF part in this formula is $w_{d,t}$ which grows with the frequency of the term in the document, while $w_{q,t}$ is the IDF part, and decreases as the frequency of the term in the collection increases. One of the most popular ranking models that incorporates ideas from TF-IDF is called Okapi BM25 [85]. It is originally a probabilistic model, and the cosine product is therefore not used in the calculation of its similarity scores.

Given a ranking function, the process of answering a keyword query reduces to finding the k documents with the highest similarity score with the query, and many approaches to this problem have been suggested [13, 35, 40, 49, 80, 94, 95, 109]. Current large-scale search engines often employ AND or WAND style queries [35, 40, 109]. With AND style queries, the similarity score is only computed for documents that contain all query terms. WAND, however, is an operator that can be configured to work as different hybrids in between AND and OR, and an example query processing strategy based on WAND first computes the similarity score for the documents that contain all query terms, and falls back to an OR strategy if not enough relevant documents are found [35, 109]. With both AND and WAND style queries, a processing strategy where the posting lists for all query terms are processed in parallel, called document-at-a-time processing, is typically used [35, 95, 97, 109]. In order to find the documents that contain all query terms one needs to find the intersection of the posting lists. We introduce a set of solutions to this and related problems in Section 2.2.2.1. Other processing strategies for finding the top-k ranked results of keyword queries often assume that the posting lists are sorted in an order defined by the ranking function. A set of such strategies have been introduced by Fagin et al. [49], and we give a brief introduction to these methods in Section 2.2.2.2. The introduction serves as background for Section 2.2.4 where we discuss methods for search in social networks that build upon the work by Fagin et al..

Search engines are not restricted to processing simple ranked queries. It may also be possible to specify a phrase as one of the query terms, resulting in a ranked phrase query. Other more complex query types may also be supported. We do not go into detail about complex query types, but we introduce a strategy referred to as faceted search in Section 2.2.2.3. For an overview of phrase query processing, see the survey by Zobel and Moffat [112] and the references therein.

2.2.2.1 Intersection, Union and Skipping

The problems of computing unions and particularly intersections of lists have received a lot of attention in previous work [17, 19, 23, 24, 28, 44, 48, 59, 84, 96], most of which has a theoretical focus. In this section, we describe two approaches that compute the intersection of two lists, one introduced by Demaine et al. [48] and the other by Baeza-Yates [17]. Both of these algorithms compute the intersection of sorted uncompressed lists. We also explain how we can extend the inverted index to support such strategies efficiently on compressed lists. While computing the intersection of lists is the most important set operation in most search engines, several papers in this thesis use a query processing framework where combinations of unions and intersections can be computed. Although the problem of computing unions of lists has also been addressed, combinations of unions and intersections have received limited attention [44, 84]. However, Raman et al. have introduced a framework targeted at providing an efficient practical solution to this problem [84], and we therefore explain their approach, which is also relevant for query processing in DSSs.

Intersection: The method introduced by Demaine et al. to compute the intersection of lists [48] is based on performing galloping searches in the lists [26]. A galloping search is used to find a particular value, v, in a sorted list, and we assume in the following that it returns the smallest list element larger than v if v does not exist in the list. When a galloping search starts from the beginning of the list, it checks the list elements at indexes 1,2,4 $\ldots$ 2^i, until a value larger than v is found (or the end of the list is reached). Assuming that a larger value is found at index 2^i, a binary search is performed between indexes 2^{i-1} and 2^i to find the exact value to return. Figure 2.6 illustrates the approach.

The method from Demaine et al. finds the intersection of two lists by starting with the first value from one of the lists as a candidate. A galloping search is then performed in

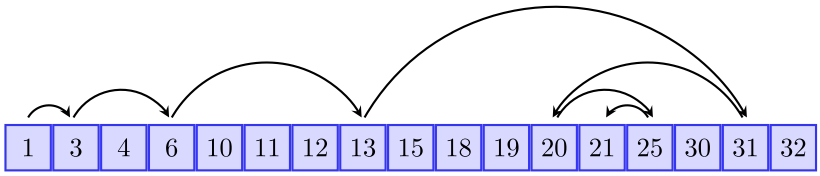


Figure 2.6: A galloping search for 21

the other list to check whether the same value exists there. If the candidate is found, it is returned as part of the intersection, and we can proceed to find the next value in the intersection by restarting the process from the positions where the last value was found. If the candidate is not found, the smallest element larger than the candidate is returned from the galloping search in the second list, and this value becomes the new candidate and is searched for in the first list. This process continues by alternating between the lists until a common value is found. The steps in the process of finding the first element in the intersection of two example lists are shown in Figure 2.7.

The approach introduced by Demaine et al. has the nice property that if there is significant skew among the lists, for example if all values in one list are smaller than a value w, and all values in the other list are larger than w, the method efficiently determines that the intersection of the two lists is empty [48].

The method introduced by Baeza-Yates is also able to process intersections between skewed lists efficiently, but uses a different approach based on binary searches [17]. In its simplest form, the method starts with the median in the shortest list, and performs a binary search for the value in the largest list. If the value is found, it is part of the intersection. Regardless of whether it is found or not, the overall problem can now be partitioned into two smaller sub-problems that are solved recursively. The sub-problems consist of the elements in the short list that are smaller than (larger than, respectively) the median, and these should be intersected with the elements in the long list that are smaller than (larger than) the result of the binary search [17]. Figure 2.8 illustrates the first partitioning into sub-problems for the same lists as in the example in Figure 2.7. An optimization of this basic strategy has also been introduced. The idea of the optimization is to ensure that only overlapping segments of the two lists are actually processed in the recursive calls [17].

Skipping: Both methods for computing the intersection of two lists explained above process uncompressed lists. The methods rely on galloping and binary searches, which essentially lead to random look-ups in the lists. As explained in Section 2.2.1.1, posting lists in search engines are usually delta compressed. A disadvantage of delta compression is that in order to find the value of a specific entry, all entries up to and including this entry must be decompressed, and this makes random look-ups inefficient.

One approach to address this problem is to always decompress the complete list, but that can lead to unnecessary processing if significant parts of the list are not accessed.

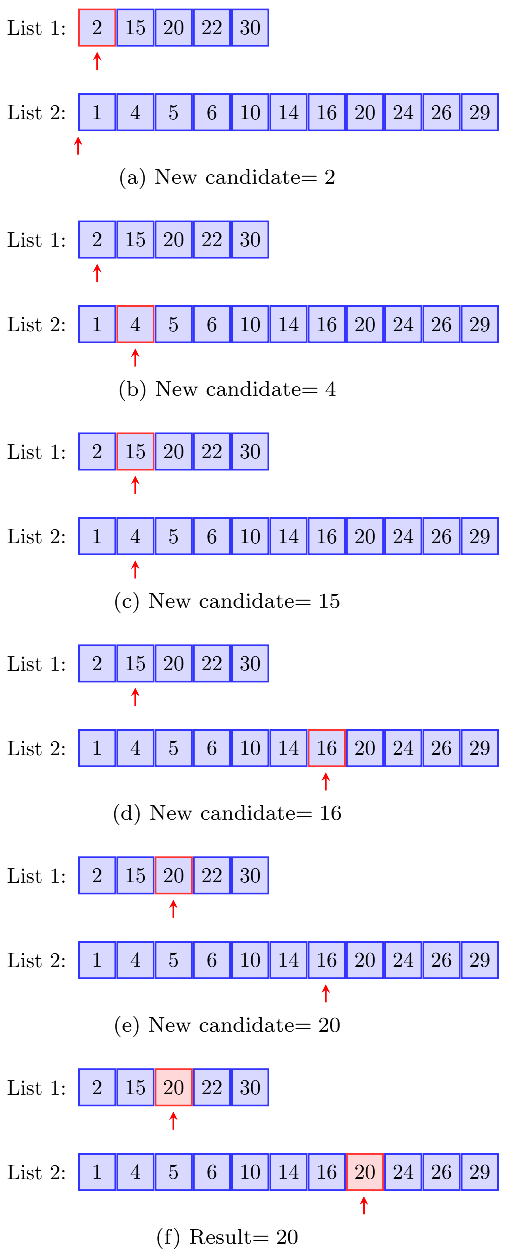


Figure 2.7: Example intersection calculation with method from Demaine et al.

Another frequently used approach is to introduce synchronization points in the lists. At these synchronization points, one can find the actual uncompressed value in the list

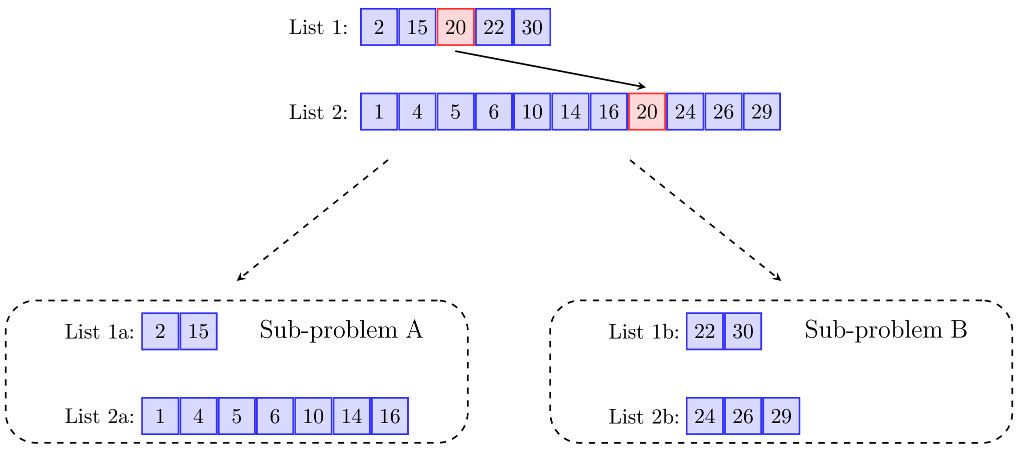


Figure 2.8: Intersection calculation with method from Baeza-Yates

and it is therefore possible to start decompressing from these points. Different methods based on synchronization points vary in sophistication. Several methods construct a search structure over the synchronization points, often inspired by the skip list data structure [82], and the approach of searching forward in the list to find a particular value is therefore referred to as skipping [30, 74]. Other methods simply store the values in the synchronization points in a simple auxiliary array that is usually kept in memory during query processing [46]. Both galloping and binary searches can be performed in the auxiliary array, but a segment of document IDs must be decompressed to find specific values in the list (apart from the synchronization points).

Union: When all the results of a union between a set of sorted lists are to be outputted, a multi-way merge strategy that traverses all the lists in parallel is commonly used. However, when the union of a set of lists is part of a larger query, it may be possible to avoid returning all results and there is potential for optimized processing strategies. Raman et al. have developed a framework for processing queries which follow the template outlined in Figure 2.9. The queries are intersections of unions of lists and have applications both in search engines and DSSs [84]. Notice that what we refer to as lists can also be implemented as B-trees or similar structures.

The algorithm used by Raman et al. to compute the intersection is an extension of the method by Demaine et al. that was introduced above. It searches for candidate results in the unions of lists to find values that occur in all unions, and it is thus important to support skipping over a union of lists efficiently. Several different implementations are possible, including the following straight-forward approaches [84]:

1. **Eager Merge**. The lists that are inputs to the union are pre-merged into a single list of intermediate results, and skipping is implemented with standard techniques.
2. **No Merge**. Each skip for the union is processed as a skip in each input list, and

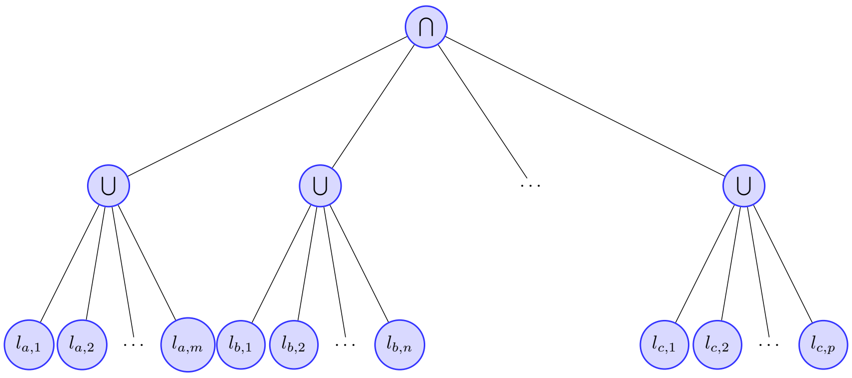


Figure 2.9: Query template

the minimum returned value from all of the lists is returned.

Both of these approaches have advantages and disadvantages. Eager Merge is efficient if there are many skips for the union compared to the total length of the inputs, because each operation only leads to a skip in the single list of intermediate results. However, if only a few skips are performed, the overhead of pre-merging all the inputs outweighs the benefit of not having to search in each input, and No Merge is then a better strategy [84].

Raman et al. have introduced an intermediate strategy which assumes that to process a given number of skips for a union as efficiently as possible, a subset of the inputs to the union should be merged before processing any skips, and the rest of the inputs should be handled as with No Merge. Because the number of skips for each union is unknown when processing begins, an adaptive method called Lazy Merge is introduced [84]. Lazy Merge is configured with a parameter α and begins processing without merging any inputs, but continuously records the number of skips processed for the union. When this number reaches α times the length of one of the input lists, this input is merged into a list of intermediate results. Overall, this method adaptively finds the optimal subset of input-lists to merge into the list of intermediate results. It has been shown that with a properly chosen α, the processing cost of this approach is never more than twice the processing cost of a solution that pre-merges the optimal subset of the input lists [84].

2.2.2.2 Threshold Algorithms

As mentioned above, Fagin et al. have introduced a set of strategies that can be used to retrieve the top-k ranked results from an inverted index. All the strategies assume that the posting lists are sorted on ranking scores. A ranking score in this setting represents the score added to the similarity score of a document based on the occurrences of the term in the document. The methods introduced by Fagin et al., which we refer to as threshold algorithms, were originally introduced in a more general setting, but we explain the methods in the context of inverted indexes [49]. Several other information retrieval papers present similar ideas and extensions [12, 13, 80, 94], but because the methods introduced by Fagin et al. are the starting points in several of the papers on search in social networks that we discuss in Section 2.2.4, we focus on them here.

The threshold algorithms discriminate between sorted and random accesses to data. Sorted accesses can be used to scan through posting lists, while random accesses can be used to retrieve the ranking score associated with a document for a particular term. Notice that the cost of a random access is implementation-dependent. In a straight-forward inverted index where lists are sorted on ranking scores, the worst-case cost of a random access is equal to the cost of scanning the complete posting list.

The Threshold Algorithm (TA) uses both sorted and random accesses. The posting lists for all query terms are scanned in parallel, and each time a document is found in any list, the algorithm performs random accesses to find its ranking score for the rest of the terms. All these ranking scores are then combined into the total similarity score between the document and the query. If this value is among the k highest seen up until now, the document and its similarity score is stored in an intermediate structure that continuously contains the top-ranked results. At the same time, the last ranking score seen in every posting list is recorded, denoted x_i for input i. The algorithm takes advantage of the fact that a bound on the maximum possible similarity score for any unprocessed document is found by combining the x_i's for all posting list. When this value drops below the kth highest seen similarity score, the top-k results have been found, and no further processing is required. Fagin et al. show that if sorted and random accesses have the same cost, and if the function that combines the individual ranking scores into a similarity score is monotone and fixed (and thereby also that the number of processed lists is constant), there exists no algorithm that: (1) always finds the correct answer, (2) does not make wild guesses, and (3) still performs asymptotically better than TA in terms of number of accesses on any possible input [49].

In light of the discussion above, it is often not realistic to assume that random and sorted accesses have the same cost. Fagin et al. have introduced another algorithm, denoted No Random Access (NRA), for settings where random accesses are either impossible or very costly. The main idea of this algorithm is again to scan all posting lists in parallel. At any point in time, two values are monitored for all documents that potentially are part of the top-k results: (1) the lowest possible similarity score for the document, and (2) the highest possible similarity score for the document. These values are easily found by storing in which posting lists we already have found references to the document, the document's partial similarity score, as well as the last ranking score seen in the different posting lists, x_i. The highest possible score for a document is found by combining its partial similarity score with the x_i's for all lists where the document has not yet been found. Likewise, the lowest possible score is found by combining its partial score with 0 for each of the remaining lists. The main idea of NRA is to continue processing until the kth highest among the lowest possible scores is larger than the highest possible score for all documents not among the top-k. Fagin et al. also show that there exists no algorithm that (1) always finds the top-k ranked documents according to a monotone and fixed similarity score, (2) only performs sorted accesses, and (3) still performs asymptotically better than NRA in terms of number of sorted accesses on any possible input [49].

In several possible implementations, random accesses are possible, but they have significantly higher cost than sorted accesses. Fagin et al. also introduce a method for such settings, denoted Combined Algorithm (CA), which combines the techniques of TA and NRA [49].

2.2.2.3 Faceted Search

Faceted search is a technology that falls in between searching and browsing, and lets users choose between these two strategies in every step of the information gathering process. The main idea is to present categories based on facets associated with the results together with the actual search results [54]. The number of search results within each category is usually also listed. The facets are based on meta-data associated with the documents, and they can for example describe the author of the document, the time of publication and so on. The meta-data is often hierarchical with categories that can be partitioned further into sub-categories. When a set of search results is presented together with a set of categories based on facets, the user can choose whether she wants to access one of the listed results, specify the query further with more keywords, or restrict the search results to those within a certain category [47, 54].

If we assume that the document collection consists of research papers, possible facet hierarchies include:

- The science discipline the paper belongs to, with possible facet categories such as physics, computer science and biology. The science disciplines could further be partitioned into sub-fields and potentially also into specific topics.
- The time of publication, where the top-level of the facet hierarchy for example describes the decade of publication, while lower levels could have finer granularity.
- The publication channel, resulting in a facet hierarchy with publisher at the top level, the exact journal or conference next and so on.

The possibility of restricting the search results to within specific categories can help users find the information they are looking for more efficiently.

It is possible to choose a set of facets that is presented for all queries, but Dash et al. [47] explore how one can find the most interesting set of facets dynamically for a given query result. They define how interesting/surprising a facet is by calculating the expected number of documents in the result set that belongs to a particular category based on statistical models, and compare it to the actual number. If there is a significant difference, the facet is considered interesting.

To identify the most interesting facets for a given result set, the number of documents in the result set that belongs to the different categories must be determined. One possible approach to find this information is based on having a look-up structure where the facets associated with a certain document can be found, and then aggregate the results of look-ups for all documents in the result set. However, Dash et al. follow another approach which is based on having representations of the sets of documents that belong to the different categories, and compute the intersection between these sets and the result set to find the number of results that belong to the different categories. This approach is based on the solution used in Apache Solr.[1] To support facet hierarchies efficiently, a tree of sets is constructed for each hierarchy, and intersections are calculated in a search starting from the root in the tree. The search for interesting facets in a tree is pruned when a set that has no documents in common with the set of results is found. The sets in the tree are represented as WAH-compressed bitmaps to limit the space usage, and Dash et al. report that this scheme leads to a significantly more efficient solution than the above mentioned solution based on looking up the facets for each document in the result set [47].

2.2.3 Access Control

Papers III and IV in this thesis address the problem of supporting access control for search workloads in social networks. Approaches that support access control for search over multi-user (enterprise) file systems have been discussed in the literature, and we provide an overview of suggested solutions in this section.

When access control is taken into account in search workloads, different subsets of the overall document collection are *searchable* by different users. When a user submits a search query, the results should only depend on the documents searchable by the user, and it should not be possible to infer information about the rest of the collection [38, 90].

A straight-forward approach to support access control in search workloads is to construct a single index for each user, where only documents accessible to the user are indexed. This approach is secure because it is impossible for a user to infer information about other parts of the document collection than the one she has access to. However, when many files in the system are accessible to many users, the approach leads to a large space overhead because these files are indexed multiple times [38].

To avoid the large space overhead of separate indexes, a seemingly natural approach is to construct a single system-wide index that contains all documents. Each query is processed in this single index, and before the results are returned to the user, the documents the user does not have access to are filtered out. However, filtering out the inaccessible documents in the end does actually not enforce access control, as noted by Büttcher and Clarke [38], at least not for queries ranked using TF-IDF or similar strategies. This is due to the use of statistics about the whole document collection in the ranking schemes, which can help a user infer information about the occurrences of terms in inaccessible documents. For a detailed description of how this can be done when the Okapi BM25 model is used for ranking, see the paper by Büttcher and Clarke [38]. Here, we only provide some intuition on how a user with limited access privileges can obtain information about the number of occurrences of a term in the total collection when the user has access to add documents to

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

the index. We assume that the user is interested in determining the frequency of the term ”bankruptcy” in the collection. The user can obtain information by adding one document with the term ”bankruptcy”, and a set of documents that all contain a new fake word that does not occur elsewhere in the collection. If the user searches for ”bankruptcy” and the new term together in one query, the relative ordering of the document that contains ”bankruptcy” compared to the documents with the new term reveals information about the total number of occurrences of ”bankruptcy”, because the IDF part of the similarity score makes less frequent terms have a larger impact on the similarity score.

To address this issue, Büttcher and Clarke suggest an approach where they filter out inaccessible results from the posting lists before any ranking calculations are performed. This approach leads to slightly slower query processing than the post-filtering approach, but the overhead can be limited with various optimizations [38]. Notice that the post-filtering approach can still be used for queries with ranking functions that do not use statistics about the complete collection.

Singh et al. suggest a different approach for supporting access control in search workloads by partitioning the documents into so-called Access Control Barrels (ACBs). All the documents in one ACB have the same access permissions, and a query submitted by a user has to be processed over all ACBs this user has access to. This approach is secure, and optimizations can be used to limit the number of ACBs [90]. Work by Bailey et al. is also relevant, and it considers how filtering of inaccessible data can be integrated into an overall enterprise architecture [20], while Zerr et al. consider security attacks on an enterprise search architecture, and try to limit the information leakage when a certain fraction of the servers are compromised [110].

2.2.4 Search in Social Networks

As mentioned above, the main focus of Papers III and IV in this thesis is on search with *access control* in a social network, a previously unaddressed topic. However, several papers have discussed other aspects of the interplay between search engines and social networks, and in this section we give a brief introduction to the parts of this work that is most relevant for the papers in this thesis.

Most previous work that explore search and social networks focus on how aspects from the social network can be taken into account in the ranking functions to improve the relevancy of the search results (see for example [6, 7, 21, 58, 87, 88]). The main focus has been on social tagging sites where users can tag items (which we refer to as documents) with keywords; examples of such sites include del.icio.us and Flickr. The different ranking schemes based on social tagging sites typically rank documents that are tagged with a keyword that is part of a user’s query higher than documents that are not tagged with this keyword. Furthermore, if the users who tagged the document with the keyword are friends of the user who submitted the query, the document is ranked even higher. Two of the papers on ranking in social tagging sites focus specifically on efficient processing of top-k queries [7, 88], and because efficient query processing is also discussed in Papers III and IV, we give a more detailed presentation of these two papers. Both of them address ranking in social tagging sites, and present query processing strategies based on the threshold algorithms introduced by Fagin et al. that we described in Section 2.2.2.2.

Schenkel et al. have introduced a ranking scheme for social tagging sites that consists of two parts, one global part and one user-specific part [88]. In the global part, documents are ranked based on how often they have been tagged with the query keywords regardless of which users who added the tags. The user-specific part, however, only considers the tags added by friends of the user, u, who submits the query, and the documents are ranked based on how often and by which friends of u they have been tagged with the query keywords. The impact of a tag added by a particular friend, v, depends on the strength of the friendship between v and u. The ranking scheme also supports query expansion to semantically related keywords.

To support efficient query processing with the mentioned ranking scheme, Schenkel et al. store three different sets of lists:

1. A posting list per keyword, kw, which lists the documents tagged with kw and the associated (global) frequency, sorted on descending frequency.
2. A list for each user that stores her friends, sorted on descending friendship strength.
3. A list per $\langle user, keyword\rangle$-pair that lists all documents tagged with the keyword by the particular user.

The first set of lists is used for the global part of ranking, while the last two are used to incorporate the user-specific part. The lists that are relevant for a particular query are processed with an algorithm based on the threshold algorithms introduced by Fagin et al.. All lists are scanned in parallel, and the next list to process is chosen based on its expected contribution to the similarity scores of the documents. The upper and lower bounds on the score for each document are continuously monitored, and when the kth largest lower bound is higher than all other upper bounds, the top-k results have been found [88].

Amer-Yahia et al. address the same problem as Schenkel et al., but they only consider the user-specific part of ranking. Their ranking function is based solely on the number of times a document has been tagged with the query keywords by friends of u [7], and they explore different index organizations to support top-k queries with this ranking scheme efficiently. In this setting, the score assigned to a document for a query clearly depends on the user who submits the query, so in order to apply the algorithms introduced by Fagin et al. (TA and NRA) directly, one would need to construct one inverted index per user. The index should contain one list per keyword, kw, and the list entries should reference the tagged document and store the frequency with which the document has been tagged with kw by friends of u. If the lists are sorted on frequency, top-k algorithms such as TA and NRA could be applied directly. However, having a separate index per user would clearly result in significant space overhead, and Amer-Yahia et al. consider such a solution unrealistic. A solution based on a single global index is therefore introduced as a starting point for further refinement. The postings in this index contain lists of the users who tagged the document with the represented keyword, kw. This makes it possible to perform local aggregations to identify the exact number of friends of u who have tagged the document with kw. However, recall that top-k algorithms such as TA and NRA require the lists to be sorted, and it is obviously not possible to sort the lists according to the tagging frequencies within the sub-networks of all users. Amer-Yahia et al. sort the lists on upper bounds, namely the highest frequency within any users sub-network. In this way, it is possible to use the threshold algorithms by performing local aggregations to find the exact frequencies. However, the performance of the threshold algorithms may be suboptimal for many users, especially for those were the global order differs significantly from their local order [7].

Because one index per user leads to a large space overhead and the global index may lead to slow query processing, Amer-Yahia et al. introduce a set of intermediate strategies where an index with several lists per keyword is constructed. The posting lists are similar in structure to the posting lists in the global index, but each list only contains data for a subset of the users, and the subsets are chosen with clustering algorithms. Two clustering methods are explored; the first method clusters seekers and the second method clusters taggers. When clustering seekers, each list contains the data from all friends of the users in a cluster. When a user submits a query to such an index, only one posting list must be accessed per query keyword, but the index may have a significant space overhead. With clustering on taggers, however, the tagging behavior of all users in a cluster is represented in a list. This method leads to a smaller space overhead, but several posting lists might have to be accessed per query keyword. Amer-Yahia et al. recommend using a hybrid solution in between these two [7].

Chapter 3

Research Summary

This chapter gives a summary of the research underlying this thesis. The chapter begins with a description of the process that lead to this thesis in Section 3.1, where both successful and unsuccessful research efforts are described. In Section 3.2, an overview of the papers is presented, including the roles of the different authors and retrospective views.

A schematic overview of the papers is presented in Figure 3.1, where the papers are assigned to fields according to which application they address. It should be noted, however, that all papers use techniques from both decision support systems and search engines. The dashed box in the figure indicates that Papers III and IV address the same overall problem. I am the first author of all papers presented here, and the actual papers are included in Appendix A. Papers where I am a co-author are described and included in Appendix B. The topics of those papers differ from the overall topic of this thesis, but some of the underlying concepts and processing strategies are similar.

3.1 Research Process

The work with this thesis has been part of the Integrated PhD program at the Faculty of Information technology, Mathematics and Electrical engineering (IME) at NTNU. In the Integrated PhD program, the last year of the master's program is integrated with the first year of the PhD program. This section presents an overview of the process that lead from the master's thesis to the current research results, and both fruitful and not so fruitful research efforts are mentioned.

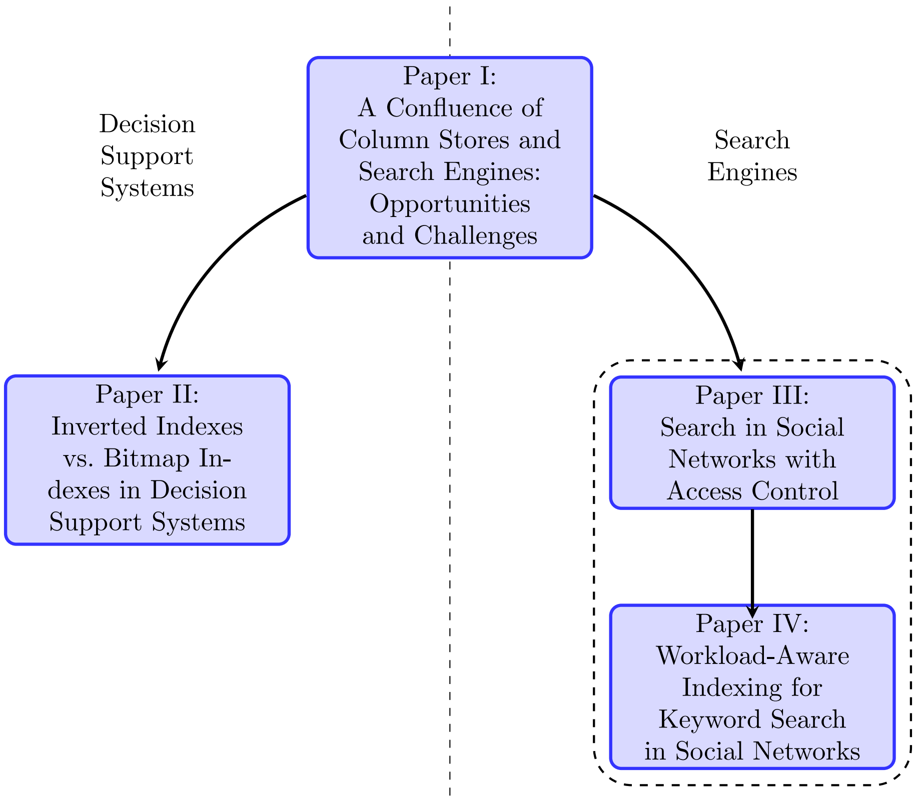

Decision Support Systems
Paper I: A Confluence of Column Stores and Search Engines: Opportunities and Challenges
Search Engines
Paper II: Inverted Indexes vs. Bitmap Indexes in Decision Support Systems
Paper III: Search in Social Networks with Access Control
Paper IV: Workload-Aware Indexing for Keyword Search in Social Networks

Figure 3.1: Overview of papers

3.1.1 First Steps

The topic of my master's thesis was updatable inverted indexes [29], and the initial plan was to continue to work on this topic in the PhD program. We therefore started to extend a system used for experiments in the master's thesis with the goal of developing a system that could support the following operations efficiently: (1) Add a new document to the index, (2) delete a document from the index and (3) update the index to reflect updates in a document. One of the specific ideas we investigated was to take the expected lifetime of a document into account, and create an index structure specifically adapted to documents with different expected lifetimes.

After having spent quite some time on updatable inverted indexes, I had gained significant insight into how to make various aspects of the implementation fast in practice, but the research contributions were unfortunately limited. I was then introduced to Johannes Gehrke. He immediately took interest in my work, and suggested performing a literature study on column stores. After some investigation, it became clear that column stores had several aspects in common with search engines, and this lead to a change of topic to the overall topic of this thesis.

3.1.2 Column Stores vs. Search Engines

The literature study on column stores lead to the idea that one could use a columnar storage layout within each posting list in the inverted index. The main motivation was that search engines process different types of queries, and that these queries require different data in the index. A simple example is that phrase queries often use information about the actual occurrences of a term inside a document, whereas this information is not used in simple ranked queries. As explained in Chapter 2.1.1, the ability to avoid reading data that is not required when processing a query is one of the main basic advantages of column stores, and we believed that a columnar layout would enable processing different kinds of queries efficiently with a single index. We were specifically considering a layout based on PAX [5] to avoid excessive disk seeks/memory accesses.

However, after significant efforts were put into the development of a system based on these ideas, we discovered a paper by Anh and Moffat that actually explored most of the same ideas [14]. Although Anh and Moffat had not described the parallels between their technique and column stores, the evaluation seemed thorough and many different aspects were tested. We consequently changed focus, and tried to find other ideas from one type of system that could be used in the other.

As we looked into more aspects of both types of systems, we realized that there were clearly parallel developments in the two fields. The idea of writing a qualitative paper about these observed parallels then emerged, which eventually resulted in Paper I. Paper I also evaluates the possibility of creating a hybrid system to support both workloads.

3.1.3 Bitmap Indexes vs. Inverted Indexes

While investigating the parallels between decision support systems based on column stores and search engines, we looked into the use of bitmap indexes in star joins and as general indexes in DSSs. Inspired by the fact that most search engines do not represent their indexes as bitmaps anymore, the idea of comparing inverted indexes to bitmap indexes for decision support workloads emerged; this was identified as an opportunity in Paper I. We developed a query processing framework for DSSs based on inverted indexes, and compared it to using WAH-compressed bitmaps in a thorough set of experiments, which resulted in Paper II. Paper II was submitted as a full paper, but was accepted as a short paper. We have not written an extended version of the short paper for reasons mentioned in the retrospective view on Paper II (see Section 3.2). The submitted version is therefore included together with the published version in this thesis, and it includes additional contributions on adaptations to sorted tables, as well as more experiments.

3.1.4 Loading in Column Stores

Another opportunity identified in Paper I was technology transfer of incremental loading strategies between column stores and search engines. Although C-store was introduced with a strategy for handling updates, further development of these techniques was considered important future work [1, 93].

Our initial idea on this topic was to see whether some of the incremental loading strategies for inverted indexes described in Chapter 2.2.1.2 and their extensions could be useful in column stores. To evaluate these ideas, we initiated the development of a simple system capable of storing and loading tables with snapshot isolation. Apart from the immediate results we hoped to achieve for loading in column stores, we also wanted to see whether the system could be extended into a true hybrid system over time.

The first operation we wanted the system to support was to load tables with different sort orders. We soon realized that some new trade-offs were introduced by the more versatile data found in column stores compared to in inverted indexes. As an example, when accumulating index data for an inverted index there is one list per term, and each list has entries that are sorted on document ID. The documents are often inserted in order according to the document ID, so it is straight-forward to achieve the actual sorting. However, records in a table in a column store can be inserted in any order, and have any specified set of attributes as the sort key. We tried to develop techniques that were efficient for a wide range of configurations, but the progress was limited.

3.1.5 Search in Social Networks

While investigating loading in column stores, the problem of search with access control in social networks came up. We saw this as a timely problem where we already had some relevant background knowledge, and we also believed that access control had received limited attention in the search engine literature. Johannes Gehrke, Michaela Götz and I started discussing possible solutions to this problem. It soon became clear that there were some interesting topics we could look into, and the focus gradually shifted from loading in column stores to search in social networks with access control.

The different solutions we considered for search with access control in social networks involved various organizations of the index and meta-data, and we realized that using the system I had already developed for loading in column stores as a back-end for a search system could be a good idea. By doing this, we could easily test different approaches through simple changes in configurations. This is related to one of the advantages of a hybrid system identified in Paper I, namely flexible definitions of meta-data. However, it should be noted that our solution is not a real hybrid system; it is a system that is able to store columns, but it lacks many of the features traditionally associated with a database system. With proper configurations, this system works just like a search engine, and it has the advantage that it is straight-forward to configure different solutions. We describe the system in more detail in Section 3.1.6.

After some investigations on the problem of supporting search with access control in a social network, we realized that the most efficient solution to the problem probably depended on the actual workload. We initially focused on what Paper III refers to as different index designs, and the goal was to develop a method based on cost models that could find the most efficient index design for a particular workload. However, after some experiments, we realized that the organization of meta-data and how they are used to filter out inaccessible results is essential for the overall performance of several index designs. We therefore defined a solution space based on both an index design and an access design where the access design describes the storage of meta-data. Faced with a large solution space and the difficulties with developing accurate cost models (see Section 3.1.7), we decided to write a paper that outlined the solution space and compared a set of basic strategies, and this resulted in Paper III.

3.1.6 Search System with Columns

As explained above, the search system used for experiments in Papers III and IV consists of a system capable of storing and loading columns of sorted data, referred to as JCS, and a thin search layer which essentially configures the system to work like a search engine, referred to as RTS. In JCS, new data is accumulated in an updatable structure, and the updates are combined with the main table structure when the updatable structure reaches a given size. The part of a table that is stored in the main table structure can be organized in a hierarchy, which is common in solutions supporting efficient updates in both search engines and database systems, or it can be stored as a single large table. The main part of the tables can in principle be stored either on disk or in-memory, but all experiments in our papers use the in-memory version. The storage representation of a column can be chosen such that one type of compression is used in the updatable accumulation structure, while another compression scheme is used in the main part of the table.

RTS, the search layer on top of JCS, defines how the indexes are stored through definitions of tables, and has support for some query processing operators. To support ranking based on recency, the index table is defined to use a form of VByte encoding that can be read in reverse order in the updatable in-memory structure, while in the main part of the table, we use compression based on PForDelta where document IDs are stored in descending order. When we use a ranking function based on recency, as is done in Papers III and IV, these compression schemes enable early termination of query processing after the top-k results have been returned.

In Paper I, a set of challenges associated with a hybrid system is identified, and some of these challenges apply to our simple search system based on columns as well. First, the problem of many short columns that arises from mapping each posting list to an individual (set of) columns must be addressed. We do so simply by storing all posting lists in one table, a strategy that is similar to the one used when MonetDB/X100 was tested as a back-end in an IR system [56, 57]. To achieve attractive compression rates even when lists with widely different deltas are stored in one column, we calculate the optimal number of bits to use per segment in PForDelta. This strategy leads to slightly slower compression speed, but it results in compact indexes. The challenge of latency versus through-put discussed in Paper I is not that relevant in this setting because we experiment with an in-memory system.

In conclusion, the search system based on columns works like a search engine because the technical solutions are essentially the same. However, it is configurable through the definitions of tables, and this has been a key factor that made us able to experiment with many different designs in Papers III and IV.

3.1.7 Cost Models

The experimental results in Paper III gave us an overview of the solution space we had outlined for search in social networks with access control, but it did not give a clear conclusion on what method to use because most methods had a particular subset of the workloads where they performed best. However, a limited number of methods performed best for workloads we considered reasonable, e.g., workloads with more than 1,000 users. Two of these methods both used a single index, but due to differences in the amount of meta-data stored, one of them was best when updates dominated the workload and the other when search queries dominated. It was straight-forward to define a set of hybrid solutions in between these two extremes, and the idea for the next paper emerged. The idea was to develop cost models which could be used to find the most efficient hybrid strategy for a given workload. Together with a treatment of the query processing strategies used in our system, this became the basis for Paper IV.

The process of developing the cost models used in Paper IV turned out to be a challenge, and we now give an introduction to some of the lessons learned in this process. We wanted to test cost models that varied in complexity in order to find a model that was as simple as possible while still being accurate. We first explored simple models, and for several workloads, we managed to fit the constants so that the models were reasonably accurate.

However, different workloads resulted in widely different constants, so the models clearly did not generalize well.

We consequently increased the complexity of the models we tested, and over time they became rather advanced. However, in retrospect we can see that parts of the models were to some extent based on guessing. An estimate of the skip lengths that occur when computing the intersection of two lists with average deltas m and n can be mentioned as an example of an estimate that persisted for a long time although it was not really sound. The intersection algorithm maintains a current candidate, and tries to skip forward to find this candidate in both input lists, just like in the method introduced by Demaine et al. described in Chapter 2.2.2.1. We tried to determine how far the candidate would be forwarded after one skip in each of the inputs, and our estimate was for quite some time that the candidate would be forwarded $\max(n, m)$ document IDs. However, the accuracy of the models increased dramatically when we later tried to use simple statistical models as a basis for our models. We ended up considering all lists and sets of results in the system as sets of Bernoulli trials. If a list was estimated to have n results and there were N documents overall in the system, the list was considered to consist of N Bernoulli trials with a probability of success (that a given element occurs in the list) as $\frac{n}{N}$. Although both this and other mathematical assumptions we made are simplifications compared to reality, basing our models on statistics proved very useful. In the example mentioned above, these assumptions resulted in an adjustment of the estimate to $m + n$, which improved the overall accuracy dramatically.

Another difficult challenge was to determine reasonable constants for our cost models with micro benchmarks, mainly because our system is implemented in Java. Due to techniques such as just-in-time compilation and adaptive optimization, warm-up is required to ensure consistent performance of a code segment in Java, where warm-up implies to run the actual code segment a significant number of times before measuring the amount of time used. However, it is also important to avoid unintended cache effects, so the warm-up must use different data from the timed runs, or the data sets must be large enough to ensure that the cache effects in the micro benchmarks do not invalidate the results. Although these concepts are simple, it is easy to make mistakes, especially because all parts of the code should be warmed up sufficiently. The issues involved in conducting proper micro benchmarks complicated the development of the cost models because we often wondered: Is the low accuracy of a tested model due to the actual model, or is it caused by wrong constants as a result of errors in the micro benchmarks? As we tested more complex models, it also became a challenge to design micro benchmarks that isolated the specific effects we modeled.

We thus learned a few lessons during the work with cost models. First, we realized that it is not easy to develop them. We also experienced that it is a good idea to base the models on statistics even if the assumptions made are relatively rough. Last, we experienced the difficulties involved with conducting micro benchmarks in Java. After learning these lessons, we managed to develop cost models that were reasonably accurate and used them with success in Paper IV.

3.2 Research Results

This section presents an overview of the papers in this thesis, including their abstract, a description of the roles of the authors, and some comments on the papers in retrospect.

3.2.1 Paper I

A Confluence of Column Stores and Search Engines: Opportunities and Challenges

Abstract

IR and DB integration has been a long-withstanding research challenge. Most of the work trying to integrate the two fields is motivated by specific application scenarios. In this paper we approach this problem from another perspective. Instead of focusing on IR and DB as whole fields, we restrict the focus to search engines and column stores. We present observations of similarities in the two technologies, and aggregate information on parallel developments in the two fields. We argue that these developments point towards a confluence of column stores and search engines, and one may in fact argue that this confluence has already started. We evaluate the potential in developing an engine capable of handling the workloads traditionally supported by the different systems, namely decision support and search workloads, by identifying potential opportunities and challenges. The opportunities include potential areas for technology transfer and more efficient support for features. The identified challenges outline areas for future work whose successfulness will help decide whether a confluence of column stores and search engines is feasible.

Roles of the authors

The idea of this paper came as a result of advice given by Gehrke. I did most of the work on the paper, while Gehrke and Torbjørnsen had roles as supervisors and provided constructive and helpful feedback throughout the process.

Retrospective view

This paper provided an overview of observed similarities and differences between column stores and search engines. The possibility of having one hybrid system replacing the two different systems was discussed, and opportunities and challenges associated with a confluence of the two types of systems were considered. The paper used a qualitative approach, and the opportunities and challenges identified in the paper needs to be demonstrated experimentally. In an attempt to evaluate the relevance of the ideas in retrospect, we give a brief overview of the aspects discussed in the paper where there has been new research results since the publication of our paper. A more thorough discussion is found in Chapter 4.

Neither we nor others we are aware of have developed a true hybrid system along the lines suggested in Paper I, but several opportunities and challenges have been addressed. First, the opportunity for technology transfer of indexing methods for decision support workloads is the topic of Paper II in this thesis. Furthermore, the system underlying Papers III and IV can be seen as taking advantage of a columnar storage layout for search workloads because it enables flexible index and meta-data organizations. The system also addresses the challenge of many short columns in a search workload to some extent. We did not make much progress on technology transfer of loading techniques as mentioned in Section 3.1.4, but Héman et al. have made contributions on this topic recently with a different strategy [60]. Other opportunities and challenges have not been addressed, and their relevance is therefore uncertain. However, the fact that some opportunities have lead to new research results indicates that the paper presented some reasonable ideas even though they were not evaluated quantitatively.

3.2.2 Paper II

Inverted Indexes vs. Bitmap Indexes in Decision Support Systems

Abstract

Bitmap indexes are widely used in Decision Support Systems (DSSs) to improve query performance. In this paper, we evaluate the use of compressed inverted indexes with adapted query processing strategies from Information Retrieval as an alternative. In a thorough experimental evaluation on both synthetic data and data from the Star Schema Benchmark, we show that inverted indexes are more compact than bitmap indexes in almost all cases. This compactness combined with efficient query processing strategies results in inverted indexes outperforming bitmap indexes for most queries, often significantly.

Roles of the authors

Gehrke and Torbjørnsen had roles as supervisors on this paper as well, while I did most of the work. Grimsmo also provided helpful feedback on writing, participated in some technical discussions, and helped configure FastBit properly.

Retrospective view

This paper considered using inverted indexes as an alternative to bitmap indexes in DSSs and compared the two techniques quantitatively. Both bitmap indexes, inverted indexes and traditional B-tree indexes contain the same information given that they index the same data collection; they only represent the information differently. In their basic uncompressed forms, it is straight-forward to discriminate between these three methods.[1]

[1]It is possible to configure a B-tree index and an inverted index that are quite similar. However, the inverted index we use in the paper uses a dictionary based on hashing, so it is not a B-tree index.

However, when compression is used, one can in principle use the same representation for all these types of indexes, and the differences between them are therefore not so clear anymore. One might argue that we could have referred to the method we suggest as a form of bitmap index with a specific compressed representation and an adapted query processing framework. After all, it is to which extent the compressed representation and the query processing framework together form an efficient solution that is essential. However, I believe there are two good reasons to refer to our solution as an example of an inverted index: (1) The solution uses several methods that originate from the information retrieval literature, and indexes such as the one we use in our paper are referred to as inverted indexes there. (2) When we decompress during query processing, we decompress into lists of references to tuples, and this format differs from the bitmaps traditionally associated with bitmap indexes. With bitmap indexes compressed with WAH, on the other hand, query processing is based on combining bit sequences using bitwise operations.

As mentioned above, it is essential to find an index representation and a query processing framework that form an efficient entity, and the experimental results in our paper show that our inverted indexes are both more compact and usually more efficient than WAH-compressed bitmap indexes. However, I believe that our paper could have been stronger with more focus on the query processing framework, especially because a more thorough review of related work later revealed that there is novelty in our solution, more specifically in the union/OR operator. Although our paper gives a presentation of the query processing framework that makes it possible to implement our techniques, identification of the novel aspects would obviously have made the paper stronger, especially because the novel union operator is correctly identified as relevant for the query processing performance seen in the experiments in the paper. We have considered to write an extended version of this paper that describes the novelty more thoroughly, in addition to including the extra contributions from the full version of the paper. However, the topic addressed in Papers III and IV has been interesting, and it has therefore been given a higher priority than an extended version of Paper II. Furthermore, the novel features of the union operator are actually even more useful in methods discussed in Papers III and IV. We therefore decided to introduce an improved version of this operator as novel there, and our additional contributions in an extended version of this paper would therefore be limited.

3.2.3 Paper III

Search in Social Networks with Access Control

Abstract

More and more important data is accumulated inside social networks. Limiting the flow of private information across a social network is very important, and most social networks provide sophisticated privacy settings to control this flow. Creating such extensive access control knobs makes the search for content a hard problem since each user sees a unique subset of all the data.

In this work, we take a first step at integrating access control based on a social network in a search system. We describe a set of solutions to the problem, including what indexes to construct and how to filter out inaccessible results. An experimental analysis illustrates the tradeoffs of the various strategies, and we point out a set of interesting future research directions in this area.

Roles of the authors

All authors were part of the initial discussions leading to this paper. I outlined the technical solutions and implemented the complete system used in the experiments. Götz made significant contributions on generating and gathering data, while I conducted the experiments. The writing of this paper was a collaboration process where all authors were involved, but I wrote the initial version of most sections.

Retrospective view

Because this paper is fairly recent, a mature retrospective view has not yet evolved, but there are some aspects of the paper that could be discussed. Two examples of aspects that can be discussed are whether the solution space we consider is reasonable, and whether the data used in the experiments is relevant.

The solution space presented in the paper is restricted to inverted indexes, but other solutions are clearly also possible, e.g., solutions that do not construct indexes but rather search in the actual documents. Such a strategy might be useful when there are virtually no searches compared to updates. However, I still believe that focusing on inverted indexes was reasonable when taking into consideration that they have proven to be a very efficient strategy for text search [112].

The data used in the experiments in this paper is either crawled from Twitter or generated synthetically. The generated networks were based on Barabasi's preferential attachment model [22], and it is clearly worth questioning whether this model is relevant. Leskovec et al. address the evolution of social networks, and they find that the preferential attachment model models the friendships of several social networks with reasonable accuracy, and that preferential attachment is more accurate than other more complex models which have been suggested [67]. Leskovec et al. also suggest an extended model for how a social network evolves over time which extends preferential attachment. We could have used this or other extended models, but because we focused on a static network, using preferential attachment seemed reasonable. Furthermore, we see the same tradeoffs with the network crawled from Twitter as with the synthetically generated networks, which indicates that the model is reasonable for our purposes. However, one can also question whether the crawled data is reasonable to use as a basis for experiments. When crawling a subset of a network, the characteristics do not necessarily generalize to the complete network. The crawled network used in our experiments has a higher average degree (number of relations per user) than other crawls [41, 65], probably because a larger crawl finds more inactive users. But, if these inactive users do not submit search queries, they are not really relevant for our experiments anyway, so not including them might not be a big problem. In conclusion, to ensure validity of experiments, real data and benchmarks are required, but privacy concerns limit the availability of such data. Given these restrictions, I believe that the data used in our experiments is reasonable.

3.2.4 Paper IV

Workload-Aware Indexing for Keyword Search in Social Networks

Abstract

More and more data is accumulated inside social networks. Keyword search provides a simple interface for exploring this content. However, a lot of the content is private, and a search system must enforce the privacy settings of the social network.

In this paper, we present a workload-aware keyword search system with access control based on a social network. We make two technical contributions: (1) HeapUnion, a novel union operator that improves processing of search queries with access control by up to a factor of two compared to the best previous solution; and (2) highly accurate cost models that vary in sophistication and accuracy. These cost models provide input to an optimization algorithm that selects the most efficient organization of access control meta-data for a given workload. Our experimental results with real and synthetic data show that our approach outperforms previous work by up to a factor of three.

Roles of the authors

Götz and Gehrke contributed in technical discussions leading to this paper, but I did most of the work. I implemented the necessary extensions to the main system that was also used in Paper III. Grimsmo was involved in some discussions on cost models, but I otherwise developed them on my own. I also wrote most of the paper, but got constructive feedback from all co-authors, and Gehrke made specific contributions to the writing of the first sections.

Retrospective view

The discussion for Paper III applies here as well. Otherwise, the paper is too recent for insightful retrospective comments.

Chapter 4

Concluding Remarks

This chapter concludes the thesis. It begins with overall conclusions in Section 4.1 before Section 4.2 presents a summary of the contributions. Interesting directions for future work are discussed in Section 4.3.

4.1 Conclusions

The work in this thesis has explored parallels between decision support systems based on column stores and search engines based on inverted indexes. With an investigation of the common technical aspects of the two types of systems as a starting point, the work proceeded in several directions where aspects from one type of system were found to be useful in the other. All of the work has addressed the following overall research question:

> *What are the technical similarities and differences between column stores for decision support workloads and search engines, and how can systems for either workload benefit from the similarities?*

This thesis does not provide a complete answer to this question, but contains several contributions that illuminate different aspects of the question. The technical similarities and differences between the two types of systems are discussed, and the possibility of constructing one hybrid system to support both workloads is considered. Furthermore, we have shown that the use of inverted indexes with a query processing framework for decision support workloads represents an alternative to bitmap indexes in DSSs, and that the resulting strategies both lead to more compact indexes and more efficient processing of most queries. The investigation of a solution space for search in social networks with access control has been made possible through a system based on storing columns. The investigation lead to the development of a system that is efficient for a wide range of workloads by using cost models to find an efficient organization of the index meta-data for each particular workload.

Topic	**Aspects**	**Addressed in**
Confluences	Columnar inverted indexes	[14]
	Column stores as search engine back-ends	[56, 57]
	Other hybrid systems	[25]
	Facets vs OLAP dimensions	[47, 54, 107]
Opportunities	Taking advantage of columns	Papers III and IV
	Loading	[60]
	Consistency and database offloading	
	Technology transfer of indexing methods	Paper II
Challenges	Numbers of columns and sizes	Papers III and IV
	Throughput vs. latency	
	Different query languages	

Figure 4.1: Table from Paper I

4.2 Summary of Contributions

In order to summarize the contributions in this thesis, we revisit a table from Paper I which is shown in Figure 4.1.

Paper I contains a qualitative comparison of the technical similarities and differences between column stores and search engines, and the table in Figure 4.1 provides an overview of the findings. We have included a new column in the version of the table shown in Figure 4.1 that shows were the mentioned topics have been addressed. The confluences mentioned in the table refer to parallel developments in the two fields that had started before Paper I was published. The opportunities were considered potential areas for further investigations, and the papers that address the opportunities are published after Paper I. The challenges in the table refer to challenges associated with creating a hybrid system to support both decision support and search workloads. Because such a hybrid system has not yet been developed, most of the challenges are not addressed. However, as the overview in the table shows, there have been developments on several of the identified opportunities.

Paper II in this thesis explores the opportunity of technology transfer of indexing methods by investigating the use of inverted indexes as an alternative to bitmap indexes in decision support systems. Papers III and IV are both based on a system capable of storing and loading columns of data. The flexibility that comes from being able to configure the meta-data as sets of columns has made it possible to investigate a wide range of solutions. Based on this underlying system, we have developed a solution to support search in social networks with access control efficiently for a wide range of workloads. Papers III and IV have thus used techniques from column stores to help develop an efficient solution for search workloads. The underlying system has also to some extent addressed the challenge with many short columns in search workloads identified in Paper I. This is achieved by defining that all documents IDs in all posting lists are stored in a single column, and by supporting a compression scheme that represents this column efficiently.

While the work in this thesis has not lead to results that make use of the other opportunities mentioned in the table in Figure 4.1, Héman et al. have described a new technique for loading data incrementally in column stores while supporting concurrent queries efficiently [60]. They suggest that their technique may also be useful for loading in search engines, indicating that there is potential for technology transfer in the opposite direction of what we suggested in Paper I. It will be interesting to see whether future work finds any sort of technology transfer on this topic useful.

In Chapter 1.3, the overall research question of this thesis was decomposed into three more specific questions, and we now discuss how the papers in this thesis address the different questions.

1. Which technical aspects of column stores and search engines are similar, and which are different?

This question is addressed in Paper I where we explore the possibility of creating one system to support the workloads traditionally supported by these two different systems. The table in Figure 4.1 summarizes the findings in the paper.

2. How can identified similarities be used as a basis to lower the processing time for search workloads?

Papers III and IV address this question through the development of efficient solutions for search in social networks with access control using an underlying system based on columns. Paper I contains a more general discussion of the topic.

3. How can identified similarities be used as a basis to lower the processing time for decision support workloads?

Paper II addresses this question by showing how technology transfer of indexing methods can help to improve the efficiency of query processing in decision support workloads. Paper I contains a general discussion of this topic as well.

Apart from the papers discussed above, the work with this thesis has also involved co-authoring papers on other topics. Further details about this work is presented in Appendix B.

4.3 Future Work

Many interesting directions for future work can be identified based on this thesis. First, Paper I discusses the possibility of developing a hybrid system that supports both search and decision support workloads efficiently. This thesis has not provided a clear answer as to whether it is a good idea to develop such a system, and it thus remains an interesting open question for future work.

Addressing some of the limitations in the work presented in this thesis can also lead to interesting research with impact in several application areas. One limitation in our papers on search in social networks with access control is that changes in the network structure are not taken into account, and the access privileges for each user are therefore static. It is possible to adopt the outlined solutions to a dynamic network structure, but an investigation of whether this introduces new trade-offs between the methods would be interesting. This question also has implications for enterprise search, because updates to the access privileges of documents in an enterprise should also be reflected in the search results. New results on these topics will thus have a wider application area than search in social networks.

Recency is used as the basis for ranking results in both Paper III and IV, and extending this ranking function is obviously reasonable, e.g., with techniques based on TF-IDF. As described in Chapter 2.2.3, it is important that the statistics used in the ranking calculations are not global, because this can reveal inaccessible information to users. Because several of the most efficient strategies discussed in our papers use a global index, specific solutions are required to avoid the use of global statistics in ranking calculations, and several possible approaches exist. One previously suggested solution is to filter out all inaccessible results before any ranking calculations are performed, but it is also possible to maintain statistics about the occurrences of terms for each user. Another potential approach is to filter out the inaccessible results for a subset of each of the posting lists involved in the query, and estimate the overall statistics based on the filtered subsets. An investigation of these and other alternatives could be interesting, and even more so if it is combined with dynamic access privileges.

Opportunities for future work also exist on the index structures discussed and compared in Paper II. More specifically, it seems reasonable that one needs to combine aspects from different methods to find an index solution that performs well for all kinds of inputs. One example of how solutions can be combined is by using different representations for different parts of the same index, as has also been the topic of related work [73, 76, 109]. An important area for future work is the development of more efficient query processing strategies for indexes that use a combination of multiple representations [9].

Bibliography

[1] D. J. Abadi. Query execution in column-oriented database systems. MIT PhD Dissertation, 2008.

[2] D. J. Abadi, S. R. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proc. SIGMOD*, 2006.

[3] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proc. SIGMOD*, 2008.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *Proc. ICDE*, 2007.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. VLDB*, 2001.

[6] S. Amer-Yahia, M. Benedikt, and P. Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2), 2007.

[7] S. Amer-Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *Proc. VLDB Endow.*, 1(1), 2008.

[8] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at SIGMOD 2005. *SIGMOD Record*, 34(4), 2005.

[9] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *Proc. VLDB*, 2000.

[10] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. ADC*, 2004.

[11] V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Inf. Retr.*, 8(1), 2005.

[12] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proc. SIGIR*, 2005.

[13] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. SIGIR*, 2006.

[14] V. N. Anh and A. Moffat. Structured index organizations for high-throughput text querying. In *Proc. SPIRE*, 2006.

[15] G. Antoshenkov. Byte-aligned bitmap compression. In *Proc. DCC*, 1995.

[16] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *Proc. VLDB*, 2006.

[17] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. CPM*, 2004.

[18] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[19] R. Baeza-Yates and A. Salinger. Fast intersection algorithms for sorted sequences. *Algorithms and Applications*, 2010.

[20] P. Bailey, D. Hawking, and B. Matson. Secure search in enterprise webs: tradeoffs in efficient implementation for document level security. In *Proc. CIKM*, 2006.

[21] S. Bao, G. Xue, X. Wu, Y. Yu, B. Fei, and Z. Su. Optimizing web search using social annotations. In *Proc. WWW*, 2007.

[22] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439), 1999.

[23] J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Algorithms*, 4(1), 2008.

[24] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14, 2009.

[25] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR&DB integration. In *Proc. CIDR*, 2007.

[26] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3), 1976.

[27] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *Proc. SIGMOD*, 1995.

[28] P. Bille, A. Pagh, and R. Pagh. Fast evaluation of union-intersection expressions. In *Proc. ISAAC*, 2007.

[29] T. A. Bjørklund. Experimentation with inverted indexes for dynamic document collections. Master's thesis, Norwegian University of Science and Technology, 2007.

[30] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *Proc. SPIRE*, 2005.

[31] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proc SIGMOD*, 2006.

[32] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal*, 8(2), 1999.

[33] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proc. CIDR*, 2005.

[34] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7), 1998.

[35] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *Proc. CIKM*, 2003.

[36] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In *Proc. CIKM*, 2005.

[37] S. Büttcher and C. L. A. Clarke. Memory management strategies for single-pass index construction in text retrieval systems. Technical report, University of Waterloo, Waterloo, Canada, 2005.

[38] S. Büttcher and C. L. A. Clarke. A security model for full-text file system search in multi-user environments. In *Proc. FAST*, 2005.

[39] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *Proc. SIGIR*, 2006.

[40] B B. Cambazoglu, V. Plachouras, and R. Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *Proc. SIGIR*, 2009.

[41] M. Cha, H. Haddadi, F. Benevenuto, and K. P. Gummadi. Measuring user influence in Twitter: The million follower fallacy. In *Proc. ICWSM*, 2010.

[42] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 27(2), 1998.

[43] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proc. SIGMOD*, 1999.

[44] E. Chiniforooshan, A. Farzan, and M. Mirzazadeh. Worst case optimal union-intersection expression evaluation. In *Proc. ICALP*, 2005.

[45] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Proc. SIGMOD*, 1985.

[46] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. SPIRE*, 2007.

[47] D. Dash, J. Rao, N. Megiddo, A. Ailamaki, and G. Lohman. Dynamic faceted search for discovery-driven analysis. In *Proc. CIKM*, 2008.

[48] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proc. SODA*, 2000.

[49] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66, June 2003.

[50] G. Gou and R. Chirkova. Efficiently querying large XML data repositories: A survey. *Knowl. and Data Eng.*, 2007.

[51] N. Grimsmo. Bottom up and top down — twig pattern matching on indexed trees. NTNU PhD Dissertation, 2011.

[52] S. Gurajada and S. Kumar P. On-line index maintenance using horizontal partitioning. In *Proc. CIKM*, 2009.

[53] S. Harizopoulos, V. Liang, D. J. Abadi, and S. R. Madden. Performance tradeoffs in read-optimized databases. In *Proc. VLDB*, 2006.

[54] M. Hearst, A. Elliott, J. English, R. Sinha, K. Swearingen, and K.-P. Yee. Finding the flow in web site search. *Commun. ACM*, 45(9), 2002.

[55] S. Heinz and J. Zobel. Efficient single-pass index construction for text databases. *J. Am. Soc. Inf. Sci. Technol.*, 54(8), 2003.

[56] S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. MonetDB/X100 at the 2006 TREC terabyte track. In *Proc. TREC*, 2006.

[57] S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Efficient and flexible information retrieval using MonetDB/X100. In *Proc. CIDR*, 2007.

[58] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Information retrieval in folksonomies: Search and ranking. In *Proc. ESWC*, 2006.

[59] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Inf.*, 1(2), 1971.

[60] S. Héman, M. Zukowski, N. J. Nes, E. Sidirourgos, and P. A. Boncz. Positional update handling in column stores. In *Proc. SIGMOD*, 2010.

[61] T. Johnson. Performance measurements of compressed bitmap indices. In *Proc. VLDB*, 1999.

[62] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR-DB system. In *Proc. ICDE*, 2003.

[63] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *Proc. SIGMOD*, 2002.

[64] N. Koudas. Space efficient bitmap indexing. In *Proc. CIKM*, 2000.

[65] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *Proc. WWW*, 2010.

[66] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *CoRR*, abs/0901.3751, 2009.

[67] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins. Microscopic evolution of social networks. In *Proc. SIGKDD*, 2008.

[68] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *Proc. CIKM*, 2005.

[69] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM Trans. Database Syst.*, 33(3), 2008.

[70] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *Proc. VLDB*, 2002.

[71] G. Margaritis and S. V. Anastasiadis. Low-cost management of inverted files for online full-text search. In *Proc. CIKM*, 2009.

[72] T. Milo and D. Suciu. Index structures for path expressions. *Proc. ICDT*, 1999.

[73] A. Moffat and J. S. Culpepper. Hybrid bitvector index compression. In *Proc. ADC*, 2007.

[74] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4), 1996.

[75] E. O'Neil, P. O'Neil, and X. Chen. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

[76] E. O'Neil, P. O'Neil, and K. Wu. Bitmap index design choices and their performance implications. In *Proc. IDEAS*, 2007.

[77] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Inf.*, 33(4), 1996.

[78] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3), 1995.

[79] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. SIGMOD*, 1997.

[80] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10), 1996.

[81] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. In *Proc. ICDE*, 2005.

[82] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990.

[83] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* McGraw-Hill, Inc., 2003.

[84] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proc. SIGMOD*, 2007.

[85] S. E. Robertson, S. Walker, S. Jones, M. Hancock-Beaulieu, and M. Gatford. Okapi at trec-3. In *Proc. TREC*, 1994.

[86] D. Rotem, K. Stockinger, and K. Wu. Optimizing candidate check costs for bitmap indices. In *Proc. CIKM*, 2005.

[87] N. Sarkas, G. Das, and N. Koudas. Improved search for socially annotated data. *Proc. VLDB Endow.*, 2, August 2009.

[88] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *Proc. SIGIR*, 2008.

[89] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.

[90] A. Singh, M. Srivatsa, and L. Liu. Efficient and secure search of enterprise file systems. In *Proc. ICWS*, 2007.

[91] M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on run-length and huffman encoding. *Information Systems*, 2008.

[92] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In *Proc. DEXA*, 2004.

[93] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *Proc. VLDB*, 2005.

[94] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proc. SIGIR*, 2007.

[95] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proc. SIGIR*, 2005.

[96] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proc. VLDB Endow.*, 2(1), 2009.

[97] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6), 1995.

[98] W3C. XPath 1.0, 1999. `http://w3.org/TR/xpath`.

[99] W3C. Extensible markup language (XML) 1.0 (fourth edition), 2006. `http://www.w3.org/TR/2006/REC-xml-20060816/`.

[100] W3C. XQuery 1.0, 2007. `http://w3.org/TR/xquery`.

[101] G. Weikum. DB&IR: both sides now. In *Proc. SIGMOD*, 2007.

[102] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes.* Academic Press, 1999.

[103] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, 2004.

[104] K. Wu, E. J. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *Proc. CIKM*, 2001.

[105] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Proc. SSDBM*, 2002.

[106] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1), 2006.

[107] P. Wu, Y. Sismanis, and B. Reinwald. Towards keyword-driven analytical processing. In *Proc. SIGMOD*, 2007.

[108] H. Yan, S. Ding, and T. Suel. Compressing term positions in web indexes. In *Proc. SIGIR*, 2009.

[109] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, 2009.

[110] S. Zerr, E. Demidova, D. Olmedilla, W. Nejdl, M. Winslett, and S. Mitra. Zerber: r-confidential indexing for distributed documents. In *Proc. EDBT*, 2008.

[111] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, 2008.

[112] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[113] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, 2006.

Appendix A

Papers

Appendix A.1

Paper I

A Confluence of Column Stores and Search Engines: Opportunities and Challenges

Truls A. Bjørklund, Johannes Gehrke and Øystein Torbjørnsen

Using Search Engine Technology for Information Management (USETIM),
workshop at VLDB
2009

Abstract

IR and DB integration has been a long-withstanding research challenge. Most of the work trying to integrate the two fields is motivated by specific application scenarios. In this paper we approach this problem from another perspective. Instead of focusing on IR and DB as whole fields, we restrict the focus to search engines and column stores. We present observations of similarities in the two technologies, and aggregate information on parallel developments in the two fields. We argue that these developments point towards a confluence of column stores and search engines, and one may in fact argue that this confluence has already started. We evaluate the potential in developing an engine capable of handling the workloads traditionally supported by the different systems, namely decision support and search workloads, by identifying potential opportunities and challenges. The opportunities include potential areas for technology transfer and more efficient support for features. The identified challenges outline areas for future work whose successfulness will help decide whether a confluence of column stores and search engines is feasible.

1 Introduction

IR and DB integration has been a long-withstanding research challenge. Most of the work on this problem has focused on specific application scenarios requiring systems that are a mixture of traditional database systems and search engines [6, 26, 20]. This paper focuses on the technical similarities between the two fields. Instead of considering database systems as a single field, we focus on a particular class of database systems. Decision Support Systems (DSS) supporting On-Line Analytical Processing (OLAP) is one example of a specialized solution for a particular workload, and we believe that this field has clearer similarities to search engines than other database workloads.

Column-oriented databases (called column stores) have received a lot of attention in recent years through systems like MonetDB [10] and C-store [25]. The primary difference between such systems and more traditional row-oriented systems is that they store a table one column at a time, instead of one tuple (row) at a time. The primary advantage of column stores is their ability to avoid reading data which is not required to process a query. Column stores also facilitate efficient compression schemes because they compress one column at a time instead of complete tuples. The representation of an attribute value can therefore easily be based on previous values for the same attribute, improving the effectiveness of schemes such as run-length encoding. The cost of updates and of reconstructing tuples during queries are potential disadvantages of column stores [16], but they are considered a good fit for decision support workloads.

Search engines are the primary systems used in information retrieval, known to users through web search engines like Google and Yahoo!. These systems are typically based on an inverted index. An inverted index has for each term, defined as a searchable word in the document collection, an inverted list of the documents containing that term, typically including extra information used to rank the results [31]. If we consider the data stored in a search engine as a large table with terms as attributes and documents as tuples, an inverted index can be interpreted as a column-oriented representation of this table.

The basic observation that search engines and column stores both store tables in a column-oriented fashion motivates the topic of this paper, namely an evaluation of possible synergies obtainable by implementing one system to support both decision support and search workloads. We begin with observations supporting that this confluence has already started in Section 2. Opportunities and challenges associated with an integration are identified in Sections 3 and 4 respectively, before we conclude in Section 5. A summary of the topics discussed in this paper is shown in Table 1.

2 Confluences

Decision support systems and search engines serve the same purpose in general, namely to provide read-mostly querying capabilities over a knowledge base. Decision support systems have traditionally only supported querying structured data while search engines have focused on collections of unstructured documents. The querying capabilities of the two systems also differ to some extent; decision support systems summarize the answer

Topic	Aspects	Section
Confluences	Columnar inverted indexes [7]	2.1
	Column stores as search engine back-ends [18, 19]	2.2
	Other hybrid systems [9]	2.3
	Facets vs OLAP dimensions [17, 15, 29]	2.4
Opportunities	Taking advantage of columns	3.1
	Loading	3.2
	Consistency and database offloading	3.3
	Technology transfer of indexing methods	3.4
Challenges	Numbers of columns and sizes	4.1
	Throughput vs. latency	4.2
	Different query languages	4.3

Table 1: Summary of the Paper

to queries through aggregations, while search engines answer queries with the top-k documents ranked according to some heuristic. Because the typical query results are so different, it is not obvious whether it is possible to design one system supporting both workloads efficiently. Returning the top-k results has been investigated for structured data as well, but the semantics of ranking documents deviate from ranking tuples [4, 14]. Despite these differences, an analysis of recent developments within both fields suggests that similar ideas are explored. We will take a closer look at some such similarities in this section. An overview of the issues that we discuss is shown in Table 1.

2.1 Columnar Inverted Indexes

As mentioned in the introduction, an inverted index can be interpreted as a column-ordered representation of a table with terms as attributes and documents as tuples, where one inverted list represents one column. The inverted list entries are typically more complex than simple attributes. Exactly what they contain depends on the granularity of the index. In a document-level inverted index, each entry typically consists of the document identifier and the frequency of the term in the document and/or other pre-calculated ranking values. In a word-level inverted index, each entry also contains a list of the actual occurrences of the term in the document [27, 31]. Different types of queries require different subsets of the inverted list entries. Boolean queries only ask for all documents containing some terms combined with boolean operators. Such queries only require reading the document identifiers. Basic traditional ranked queries require reading information used to rank the results, like the frequency of terms in documents. Phrase queries also require reading and processing the occurrences.

A clear advantage of column stores compared to row stores is that, in a column store, data for attributes not required to process a query does not have to be read. As search engine workloads include queries which require reading different subsets of the entries stored in inverted lists, a columnar storage of the inverted lists themselves might also be

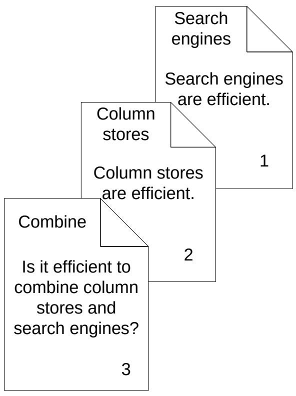

Search engines
Search engines are efficient.
1
Column stores
Column stores are efficient.
2
Combine
Is it efficient to combine column stores and search engines?
3

Figure 1: Example documents

a good idea. Anh and Moffat experiment with several such schemes in recent work [7], even though the similarities to column stores are not identified. They present three different column-oriented storage schemes for inverted indexes. The first method uses three overall columns, one for document identifiers, one for frequencies and one for the occurrences, and partitions each inverted list into these three columns. The second method stores all the data for each inverted list sequentially, but stores all document identifiers in the list first, before all frequencies and then the occurrence lists. The last approach partitions each inverted list into blocks and uses column-oriented storage only within each block, a scheme that is similar in spirit to PAX [5]. To see how these index schemes differ from a traditional inverted index, consider the documents in Figure 1. A logical view of a document-level inverted index for these documents is given in Figure 2. We only consider a document-level inverted index here; generalization to a word-level inverted index is straight-forward.

Figure 3 shows how the inverted index for the example documents in Figure 1 would be stored with the different storage schemes. Figure 3a shows a traditional inverted index where all entries in the inverted lists are stored sequentially. In the remaining figures, we assume that each column is stored sequentially, and for simplicity that columns are stored immediately following each other. Under such assumptions, the disk layout for the version with two separate columns is shown in Figure 3b. Figure 3c shows the scheme where there are two columns per term, and Figure 3d shows the scheme where each inverted list is partitioned into blocks. We assume in the figure that each block is capable of storing two inverted list entries, so the only difference from Figure 3c to Figure 3d is in the list for the term "efficient". Apart from the traditional inverted index, all of these schemes are in a sense column-oriented; a fact showing that developments towards a confluence of column stores and search engines have already started in the search engine literature.

Terms		Inverted lists
and	$\longrightarrow$	$(3,1)$
are	$\longrightarrow$	$(1,1)$ $(2,1)$
column	$\longrightarrow$	$(2,2)$ $(3,1)$
combine	$\longrightarrow$	$(3,2)$
efficient	$\longrightarrow$	$(1,1)$ $(2,1)$ $(3,1)$
engines	$\longrightarrow$	$(1,2)$ $(3,1)$
is	$\longrightarrow$	$(3,1)$
it	$\longrightarrow$	$(3,1)$
search	$\longrightarrow$	$(1,2)$ $(3,1)$
stores	$\longrightarrow$	$(2,2)$ $(3,1)$
to	$\longrightarrow$	$(3,1)$

Figure 2: Logical view of inverted index for example documents

2.2 Column Stores as Search Engine Back-Ends

Developments towards a confluence of column stores and search engines have also started in the literature on column stores, where MonetDB/X100 has been suggested as a back-end for a search engine [18, 19]. The data stored in the search engine back-end is organized in standard tables in MonetDB/X100 with the table structure shown in Figure 4.

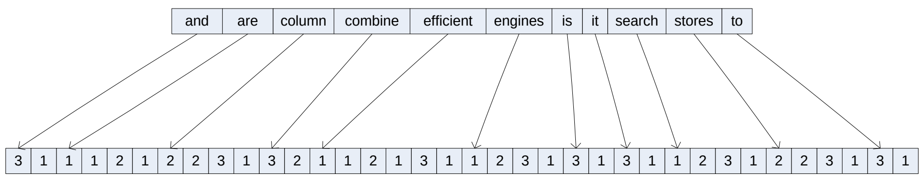
The **Document** table contains meta-data about the documents that is typically stored in all search engines. A proper storage structure for the **Term** table is essentially similar to the dictionary in an inverted index, which is a searchable structure containing all terms in the document collection. The two last attributes in the **TermInDocument** table contain the data stored in inverted lists with document-level granularity. As MonetDB/X100 is a column-oriented database system, the storage structure resulting from the chosen table layout is actually equivalent to the one tested by Anh and Moffat where all document IDs are stored in one column and all frequencies in another column. The support for compressing one column at a time in column stores enables using similar compression schemes to those used for inverted lists in IR systems, and the resulting index is thus both compact and efficient to decompress [18].

2.3 Other Hybrid Systems

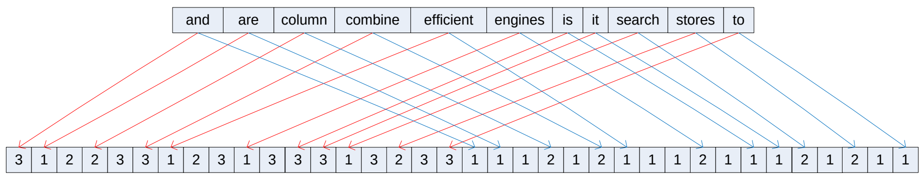
Other examples of systems that start to bridge the gap between column stores and search engines also exist. One example is CompleteSearch, which has support for some database workloads based on the HYB data structure, which is a slightly modified inverted index [9].

2.4 Facets vs. OLAP dimensions

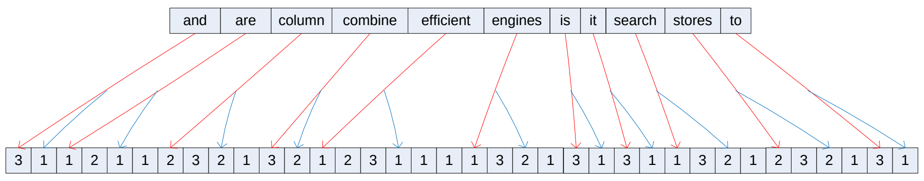
Recall that one difference between search engines and decision support systems is that search engines answer most queries with the top-k documents, while decision support



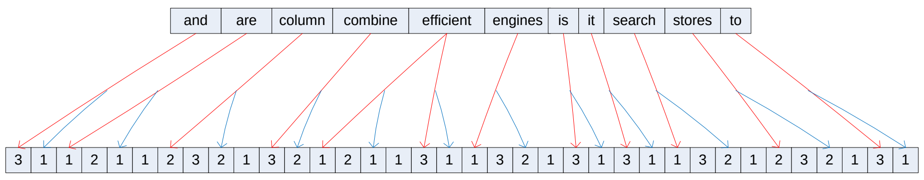
(a) Traditional inverted index for example documents



(b) Index with separate columns



(c) Index with two columns per term



(d) PAX-like index

Figure 3: Inverted indexes with different storage schemes

Term
term ID
term

(a) Term table

Document
document ID
name
length

(b) Document table

TermInDocument
term ID
document ID
within document frequency

(c) TermInDocument table

Figure 4: Tables used in Search Engine back-end in MonetDB/X100

systems typically answer structured queries with aggregations. Through the introduction of faceted search, typical query results in search engines are no longer restricted to ranked lists of documents [17]. Faceted search is an approach that falls between keyword search and browsing, in the sense that it supports both and lets the user choose a strategy on the fly. The result of a query is typically presented as a ranked list of the top-k results along with some facets. These displayed facets represent categories associated with the search, and the results typically include the number of hits within each category from the entire set of results, not only for the top-k. A search involving facets can thus be characterized as processing many group-by's at once in database terminology, one for each displayed facet.

A search on a web page for an electronic store for "phone" could return facets enabling choices between cell phones and regular phones, different brands, and different price ranges. A user could then choose to select one of the top-k results, or to specify the search by adding additional query terms, or to further specify the query through facets.[1] The ability to drill-down into the search results through facets makes them similar to dimensions in OLAP systems [15, 29].

3 Opportunities

Replacing two systems with one which has equal performance and functionality is obviously attractive from a management and cost perspective. Even so, it is probably not enough motivation to go ahead and try to implement such a system. This section outlines other potential advantages that we believe are obtainable in a hybrid column store and search engine system. An overview of the issues that we discuss is shown in Table 1.

[1]See http://www.bestbuy.com for an example

3.1 Taking Advantage of Columns

Column stores have the advantage that they avoid reading data that is not required to process a query. They are also able to store sparse data efficiently [1]. In this section we discuss the advantages associated with columnar storage in a search engine, both for the inverted index itself, and when storing meta-data.

3.1.1 Extending the Inverted Index

By vertically partitioning the entries in inverted lists as explained in the previous section, potential disadvantages of adding additional attributes to each entry are limited. If we were to add additional attributes in a traditional inverted index, every query would have to read these attributes, which would obviously slow them down. In the example in the previous section, we constructed a document-level inverted index for the example documents of Figure 1. The advantages of columnar storage are clearer for a word-level index, because it contains the occurrences as well, and the occurrences are not required for simple queries. Examples of even more attributes that can be useful in some queries include attributes to enable personalized search results, and the contexts in which the occurrences are found. The context could for example indicate whether an occurrence is contained in the title or body of the document. A word-level inverted index including such a context for occurrences for Document 1 from Figure 1 is shown in Figure 5.

Terms		**Inverted lists**
are	$\longrightarrow$	$(1, 1, [4], [body])$
efficient	$\longrightarrow$	$(1, 1, [5], [body])$
engines	$\longrightarrow$	$(1, 2, [1, 3], [title, body])$
search	$\longrightarrow$	$(1, 2, [0, 2], [title, body])$

Figure 5: Logical view of word-level inverted index with context information

The example index in Figure 5 can clearly be extended with more attributes, for instance chapters, sections and pages of occurrences. As we add more attributes to an inverted index, it becomes less and less likely that all queries require all of them. The benefits of column-oriented storage thus increase with more attributes.

3.1.2 Additional Meta-data

As noted in the previous section, all search engines store meta-data about the documents, typically including their length and name. In faceted search, there are facets associated with each document, and the facets are thus part of the meta-data. To see examples of possible facets, let us consider the documents in Figure 1. We could have one facet describing that a document is about technology. The facet would only be associated with the documents in the collection that have techonological topics, which includes all in our example. We could also include facets that describe the topics of the documents more specifically. A facet describing that a document is about search engines could be associated with documents 1 and 3 in our example, while documents 2 and 3 could be associated with the facet "column stores". Facets with values are also possible, and a simple example of a facet with numerical values is year of publication. This facet would have a value for all documents.

In a large system, the number of different facets is typically large, and most documents have null-values for most of the facets. To store this information, we could leverage that column stores are space efficient for sparse data [1].

3.2 Loading

Loading data has been identified as a key problem in column store databases [2]. C-store was introduced as using LSM-trees, but the solution has never been outlined in detail [25]. Updatable indexes, especially indexes supporting adding more data, have been investigated in detail for search engines on the other hand [11, 12, 13, 23, 21, 22].

While many solutions have been investigated for updatable inverted indexes, solutions based on a hierarchy of indexes tend to perform best overall in experimental comparisons. In such solutions, new documents are added by accumulating an index for them in memory. When the memory set aside for accumulation is filled, the new partial index is merged into the hierarchy. How this merge proceeds depends on the exact method, but in general it is rare that all indexes are merged, a fact that makes the average merge relatively inexpensive. The disadvantage of such approaches is of course that a search for a single term must be performed in several indexes. Retrieving the inverted list for one term with entries in all indexes will thus not take just 1 disk access, as is assumed for a straight-forward inverted index.

The research on loading in search engines seems applicable to column stores, and a hybrid system is a good starting point for exploring this potential technology transfer.

3.3 Consistency and Database Offloading

Although there has been much research on loading data into search engines, there is one fundamental aspect of it that is important for decision support systems which has received limited attention for search engines, namely consistency. In some sense, this difference is a challenge. However, we choose to consider this a potential area for technology transfer because improved support for consistency of operations may enable more extensive use of search engines.

Outlining a particular solution for consistency in a hybrid system is beyond the scope of this paper, but it seems likely that recently developed consistency models with slightly weaker guarantees than the traditional ACID properties can be supported without a major impact on performance. An example of such a model is snapshot isolation, which has been suggested used in C-store [25].

Although not much previous work has been concerned with consistency when adding new documents to a search engine index, we argue that there are advantages of supporting it, especially if the performance impact is limited. Improved support for consistency in search engines is especially important for database offloading. In database offloading, several related records in a database will typically be transformed to several documents in a search engine, and support for adding these documents as an atomic operation might be important for correctness.

3.4 Technology Transfer of Indexing Methods

Bitmap indexes are commonly used in decision support systems. Traditional bitmap indexes are best fitted for attributes with low cardinality, but bitmap compression techniques have been developed to handle high-cardinality attributes as well [28]. Bitmap indexes have been used previously in information retrieval, but are now mostly replaced by inverted indexes. The primary advantage of inverted indexes which has made them the de facto standard index method in search engines is that they are very compressible [27, 31]. A comparison of the two approaches for decision support workloads might therefore be interesting.

4 Challenges

Even though one can argue that a confluence of column stores and search engines has already started, there are still interesting challenges associated with designing a hybrid system. In particular there are some differences between the workloads traditionally supported by the two systems, and these differences result in different trade-offs. We will outline three of the challenges that have to be solved in order to construct an efficient hybrid system in this section. An overview of the issues that we discuss is shown in Table 1.

4.1 Number of Columns and Column Sizes

Studies have shown that the occurrences of terms in natural language usually follow a Zipfian distribution [30, 8]. In practice, this means that there are a few very common terms that appear in most documents, and that most terms only have one or two occurrences overall. By considering each inverted list to be a representation of a column in the overall term-document table, there are many columns in a search engine index, most of which are very short.

Column stores are mostly used for workloads with limited numbers of columns, but each column is often relatively large. Column stores have been suggested for other application areas than decision support workloads, like storing semantic web data [3]. The solution suggested for semantic web data involves a larger number of columns than what is typically used in a column store, and this is found to cause problems for the directory management in existing column store solutions [24].

Search engine workloads will typically involve more extreme numbers of columns than RDF data, and a hybrid system would thus probably require more sophisticated directory management than column stores have today. The fact that this is a problem for RDF data as well also indicates opportunities. A solution for handling many short columns efficiently in a hybrid system may also help supporting storage of RDF data in column stores more efficiently.

4.2 Throughput Versus Latency

Users expect answers from search engines to be produced in milliseconds even on web-scale document collections. In addition to scaling to many queries per seconds, low latency is thus an important performance requirement for search engines. Query processing with latency-constraints has not been a focus for column stores, as they have rather been optimized for throughput. This difference may lead to quite different trade-offs for different aspects of the system. One example of such an aspect is the unit of transfer from disk. In a system focusing on throughput a large unit of transfer is preferable, because it leads to fewer read operations from secondary storage. Large units of transfer are thus preferable in column stores for decision support workloads. For search engines on the other hand, using large units of transfer may force a query to read much more data from disk than actually required. This problem also becomes more severe when the typical columns are very short, as explained in the last subsection. Even though the data which is unnecessary for one query might be used by another, it will give a higher latency to the query performing the read. A hybrid system must thus be able to balance these two requirements.

The quest for low latency in search engines can even sacrifice correctness in some cases. A simple example of a technique doing just that is removal of stop words [8, 27]. Stop words are the most frequently occurring words in the document collection, and whether such words occur in the query and/or document is not considered vital for the relative ranking of documents. They are thus sometimes removed from the index altogether saving significant amounts of space. Sacrificing correctness is typically not an option in database systems, and a hybrid system must overcome such differences.

Part of the challenge with latency has already gained significance in search engine workloads as well, through the introduction of faceted search. While standard search engine queries only return the top-k results, faceted search queries also present statistics on the complete result set for the shown facets. To do so, a straight-forward solution would have to process the complete result set, posing challenges with respect to latency. To enable efficient query responses in OLAP workloads involving dimensions, it is common practice to materialize certain frequently used data cubes based on selections along the dimensions. Employing such techniques in faceted search, and thus also in a hybrid system, is probably not as beneficial, because the initial selection in search engines is based on keyword queries. Keyword queries create irregular selections compared to selections based on dimensions in a data warehouse.

4.3 Different Query Languages

A potential system based upon the observations in this paper should be able to support both decision support and search workloads. If the challenges outlined in this section are solved, one might be able to support even more workloads efficiently. In particular, such a system would be a good fit for other workloads requiring low latency on read-mostly workloads based on databases and/or search engines, potentially with many columns. One example of such a workload is RDF data, as discussed in Section 4.1, and XML search is another possibility. It is of course attractive in general if the same system can be reused for several workloads, but it also poses problems, like finding a reasonable physical algebra as a basis for the system.

The above mentioned workloads generally support different query languages today. While SQL is standard in structured data, XPath and XQuery are the most commonly used languages for semi-structured XML data. Search engines on the other hand generally only support queries as lists of words, potentially with a few additional operators like phrases, proximity and some boolean operators. Constructing a hybrid system thus requires finding a reasonable algebra which can form a basis for supporting all the mentioned query languages efficiently, which is clearly a challenge.

It should be noted that through the support of top-k queries for relational data, parts of this problem have already been addressed. But there are semantic differences between ranking documents and tuples [4, 14], making the challenge posed by a hybrid system more involved.

5 Conclusion

This paper has discussed parallel developments within search engines and column stores for decision support workloads. These developments point towards a confluence between the two fields. Opportunities and challenges associated with such a confluence have been identified, and Table 1 provides a summary. We believe that addressing the challenges in this paper can have significant practical impact for both search and decision support workloads, in addition to new types of workloads with similar characteristics.

Acknowledgments: This material is based upon work supported by New York State Science Technology and Academic Research under agreement number C050061. This material is also based upon work supported by the National Science Foundation under Grants 0534404 and 0627680. This research was also supported by the iAd Project funded by the Research Council of Norway. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NYSTAR, the National Science Foundation or the Research Council of Norway.

References

[1] D. J. Abadi. Column stores for wide and sparse data. In *Proc. CIDR*, 2007.

[2] D. J. Abadi. Query execution in column-oriented database systems. MIT PhD Dissertation, 2008.

[3] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proc. VLDB*, 2007.

[4] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. In *Proc. ICDE*, 2002.

[5] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proc. VLDB*, 2001.

[6] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at SIGMOD 2005. *SIGMOD Record*, 34(4), 2005.

[7] V. N. Anh and A. Moffat. Structured index organizations for high-throughput text querying. In *Proc. SPIRE*, 2006.

[8] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[9] H. Bast and I. Weber. The CompleteSearch engine: Interactive, efficient, and towards IR&DB integration. In *Proc. CIDR*, 2007.

[10] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proc. CIDR*, 2005.

[11] S. Büttcher and C. L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. In *Proc. CIKM*, 2005.

[12] S. Büttcher and C. L. A. Clarke. A hybrid approach to index maintenance in dynamic text retrieval systems. In *Proc. ECIR*, 2006.

[13] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *Proc. SIGIR*, 2006.

[14] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *Proc. VLDB*, 2004.

[15] D. Dash, J. Rao, N. Megiddo, A. Ailamaki, and G. Lohman. Dynamic faceted search for discovery-driven analysis. In *Proc. CIKM*, 2008.

[16] S. Harizopoulos, V. Liang, D. J. Abadi, and S. R. Madden. Performance tradeoffs in read-optimized databases. In *Proc. VLDB*, 2006.

[17] M. Hearst, A. Elliott, J. English, R. Sinha, K. Swearingen, and K.-P. Yee. Finding the flow in web site search. *Commun. ACM*, 45(9), 2002.

[18] S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. MonetDB/X100 at the 2006 TREC terabyte track. In *Proc. TREC*, 2006.

[19] S. Heman, M. Zukowski, A. P. de Vries, and P. A. Boncz. Efficient and flexible information retrieval using MonetDB/X100. In *Proc. CIDR*, 2007.

[20] N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ engine: A hybrid IR-DB system. In *Proc. ICDE*, 2003.

[21] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *Proc. CIKM*, 2005.

[22] N. Lester, J. Zobel, and H. Williams. Efficient online index maintenance for contiguous inverted lists. *Inf. Process. Manage.*, 42(4), 2006.

[23] N. Lester, J. Zobel, and H. E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *Proc. CRPIT*, 2004.

[24] L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for RDF data management: Not all swans are white. *Proc. VLDB Endow.*, 1(2), 2008.

[25] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: a column-oriented dbms. In *Proc. VLDB*, 2005.

[26] G. Weikum. DB&IR: both sides now. In *Proc. SIGMOD*, 2007.

[27] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes.* Academic Press, 1999.

[28] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, 2004.

[29] P. Wu, Y. Sismanis, and B. Reinwald. Towards keyword-driven analytical processing. In *Proc. SIGMOD*, 2007.

[30] G. K. Zipf. *Human Behavior and the Principle of Least-Effort.* Addison-Wesley, 1949.

[31] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

Appendix A.2

Paper II

Inverted Indexes vs. Bitmap Indexes in Decision Support Systems

Truls A. Bjørklund, Nils Grimsmo, Johannes Gehrke and Øystein Torbjørnsen

Conference on Information and Knowledge Management (CIKM)
2009

Published version

Abstract

Bitmap indexes are widely used in Decision Support Systems (DSSs) to improve query performance. In this paper, we evaluate the use of compressed inverted indexes with adapted query processing strategies from Information Retrieval as an alternative. In a thorough experimental evaluation on both synthetic data and data from the Star Schema Benchmark, we show that inverted indexes are more compact than bitmap indexes in almost all cases. This compactness combined with efficient query processing strategies results in inverted indexes outperforming bitmap indexes for most queries, often significantly.

1 Introduction

Decision Support Systems (DSSs) support queries over large amounts of structured data, and bitmap indexes are often used to improve the efficiency of important query classes involving selection predicates and joins [16, 17].

Bitmap indexes were formerly also used in Information Retrieval (IR), but are today mainly replaced by *inverted indexes*. Part of the reason why inverted indexes gained popularity in IR was that they easily support integrating new fields required to support ranked queries. The switch from bitmap indexes to inverted indexes lead to a flood of research on efficient inverted indexes [25, 30, 6, 13, 24, 3, 31, 29], and inverted indexes are now the preferred indexing method in search engines [30].

In this paper, we are asking (and answering) the question: What are the trade-offs of using inverted indexes in DSSs, and should they be considered a serious alternative to bitmap indexes? The main contributions of this paper are (1) the study of how to use and implement inverted indexes in DSSs, and (2) a thorough performance evaluation that compares inverted indexes and bitmap indexes in DSSs. In particular, we compare inverted indexes with FastBit,[1] a state-of-the-art bitmap query processing and indexing system based on WAH-compressed bitmap indexes [27].

2 Background

A standard bitmap index has one bitmap per distinct value for the indexed attribute, with 1's at positions for tuples with the represented value, and 0's elsewhere. Bitmaps can be combined using bitwise operators to answer complex boolean queries. For attributes with few distinct values, bitmap indexes are relatively compact, but their space usage increases linearly with the cardinality. One approach to limit the space usage of bitmap indexes for high cardinality attributes is compression. WAH [27] is one of several introduced compression schemes. Although there are schemes with more compact indexes, WAH supports efficient query processing. This combined with the fact that FastBit is openly available motivates the use of WAH-compressed bitmap indexes in the experiments in this paper.

WAH-compression is a form of word-aligned run-length encoding for bitmap indexes, where consecutive words containing only 0's or 1's are stored as fill words, and other words are stored literally [26, 27]. WAH-compressed bitmaps for high cardinality attributes are relatively compact because most words contain only 0's.

In IR, inverted indexes consist of a search structure for all searchable words called a dictionary, and lists of references to documents containing each searchable word, called inverted lists. An inverted index for an attribute in a DSS consists of a dictionary of the distinct values in the attribute, with pointers to inverted lists that reference tuples with the given value through tuple identifiers (TIDs). To reduce both space usage and the I/O requirements in query processing, the inverted lists are often compressed by storing the deltas between the sorted references [30]. This approach makes small values more

[1] `http://sdm.lbl.gov/fastbit/`

likely, and several compression schemes that represent small values compactly have been suggested. According to a recent study, PForDelta [31] is currently the most efficient method [29], and is therefore used in this paper. PForDelta stores deltas in a word-aligned version of bit packing, which also includes exceptions to enable storing larger values than the chosen number of bits allows [31].

Two overall query processing approaches exist in search engines. Document-at-a-time strategies avoid materializing intermediate results by processing all inverted lists in a query in parallel [6, 24], and are well suited for boolean query processing. They can be combined with skipping, which is used in search engines to avoid reading and decompressing parts of inverted lists that are not required to process a query [13]. We give a brief description of how we use these ideas in the query processing in this paper in the next section.

3 Query Processing

Recall that we use document-at-a-time strategies that avoid materializing intermediate results to process inverted index queries in this paper. We support three operators which can be combined to answer complex queries. They all support skipping to the next result with a given minimum TID value, in addition to standard Volcano-style iteration [9].

The `SCAN` operator can iterate through an inverted list. To support skipping, each kth TID in each inverted list is stored in an external list. The external list is kept in memory during scans, and supports binary searches to find the correct part of the inverted list to process when skipping.

The `OR` operator provides an iterator interface over the sorted merge of its multiple input iterators. The iterators are organized in a priority-queue based on a heap, which is maintained to make sure that the input with the smallest next TID is at the top. Skipping in the `OR` operator is based on a breadth-first search in the heap. A skip may not result in an actual skip for a given input iterator. If so, we know that neither of its children in the heap can do any skipping either, and we therefore avoid testing. After the search, we make sure that only the part of the heap involving iterators that actually skipped is maintained. This approach is reasonably efficient when actually performing skips in both large and small fractions of the set of input iterators.

The `AND` operator expects that the input iterators are sorted in ascending order according to the expected number of returned results. To find the next result, we start with a candidate from the iterator with the fewest number of expected results. We then try to skip to the candidate in the other input iterators, re-starting with a new candidate if the current candidate is absent in one iterator. A candidate found in all inputs is returned as a result. To support skipping, we start with the value to skip to as the candidate and proceed as in normal iteration.

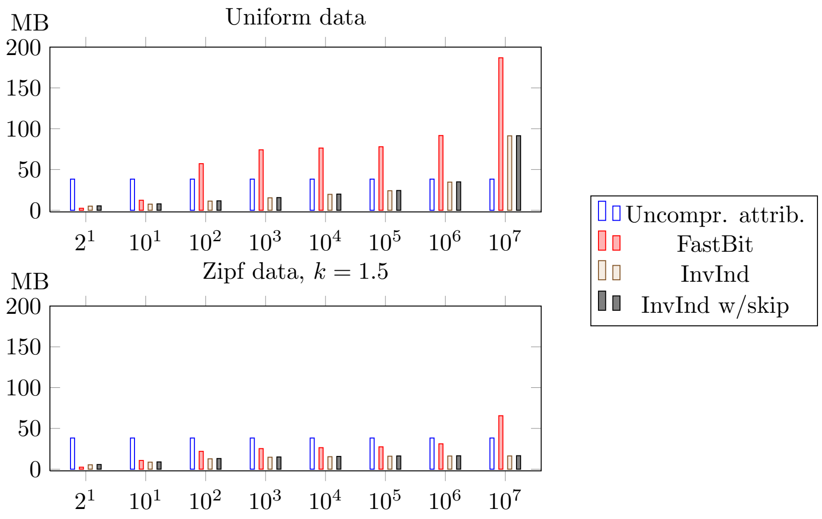


Figure 1: Index sizes. X-axis gives attribute cardinality.

4 Experiments

To investigate the trade-offs between inverted indexes and bitmaps, we experiment with FastBit and our inverted index solutions with and without support for skipping. We present results from experiments with synthetic data and data from the Star Schema Benchmark (SSB) [14].

All experiments are run on a quad-core Intel Xeon CPU at 3.2 GHz with 16 GB main memory. All indexes are stored on disk, but queries are run warm by performing 10 runs during one invocation of the system, and reporting the average from the last 8, thus measuring the in-memory query processing performance. We run FastBit version 1.0.5 (implemented in C++), with extra stack space to enable processing queries with many operands. Our approaches are implemented in Java (version 1.6). We use additional warm-up for our system to enable run-time optimizations in the Java virtual machine that reduce variance between runs. Additional warm-up did not change the performance for FastBit.

4.1 Synthetic Data

To experiment with synthetic data we generate two tables. Both tables have 10 million tuples and 8 indexed attributes with maximum cardinalities ranging from 2 via all powers of 10 up to 10 million. The attributes in the first table follow a uniform distribution, while a Zipf distribution (with $k = 1.5$) is used in the other.

4.1.1 Index Size

The sizes of the uncompressed attributes in the synthetic tables and their indexes are shown in Figure 1. When using standard PForDelta compression on the attribute with cardinality 2 in the table with uniform data, each value is represented with 4 bits in the most compact index. The reason why a lower number of bits results in a larger index is that the implementation of PForDelta might result in artificial extra exceptions when using a small number of bits per value [31]. Bitmap indexes are known to be compact when the cardinality is 2, and FastBit outperforms our approaches in this case. PForDelta results in compact indexes for higher cardinality attributes, and most of the space usage for the highest cardinality attribute comes from the dictionary (62 out of 91MB). The WAH-compressed bitmaps for the same attribute can in worst case contain nearly 3 computer words per tuple, resulting in a space usage of over 228MB on a 64-bit architecture. The actual results are significantly better, but compressed inverted indexes are clearly more compact.

Indexes for Zipf distributed attributes are more compact than for uniformly distributed attributes with the same maximum cardinality, because skewed distributions make it less likely for the actual cardinality to be equal to the maximum.

4.1.2 Query processing

To experiment with query processing performance, we test four different query types which all vary the attribute on which there is a single value predicate:

1. **Query type `SCAN`:** Finds all tuples with value 0 for a varied attribute.
2. **Query type skewed `AND`:** Finds all tuples having value 0 for the attribute with cardinality 10, in addition to 0 in one other varied attribute.
3. **Query type `OR`:** Finds all tuples having values in the lower half range for a varied attribute.
4. **Query type `AND-OR`:** Finds all tuples with value in the lower half range for the attribute with cardinality 100000, and value 0 for another varied attribute.

All queries compute the sum of the primary keys of the matching tuples, to ensure that the output from the index is used to perform table look-ups. In the table with uniform distributions, there were no tuples with value 0 for the highest cardinality attribute, so all single valued predicates on this attribute was changed to require the value to be 2. The results are shown in Figure 2.

Compared to bitmaps, decompressed inverted lists are well suited for looking up other attributes for the qualifying tuples, a factor contributing to faster scans for uniform data. The difference in index size also seems to have an impact. All scans are relatively slow for Zipfian data because we always search for the most common attribute value in a skewed distribution, except for in the highest cardinality attribute as noted above.

Skewed `AND` favors methods capable of taking advantage of the different density in the operands. Inverted indexes with skipping are therefore efficient for uniform data, but introduce overhead for Zipfian data because both operands are dense when the most common values in skewed distributions are accessed. FastBit performs well on dense

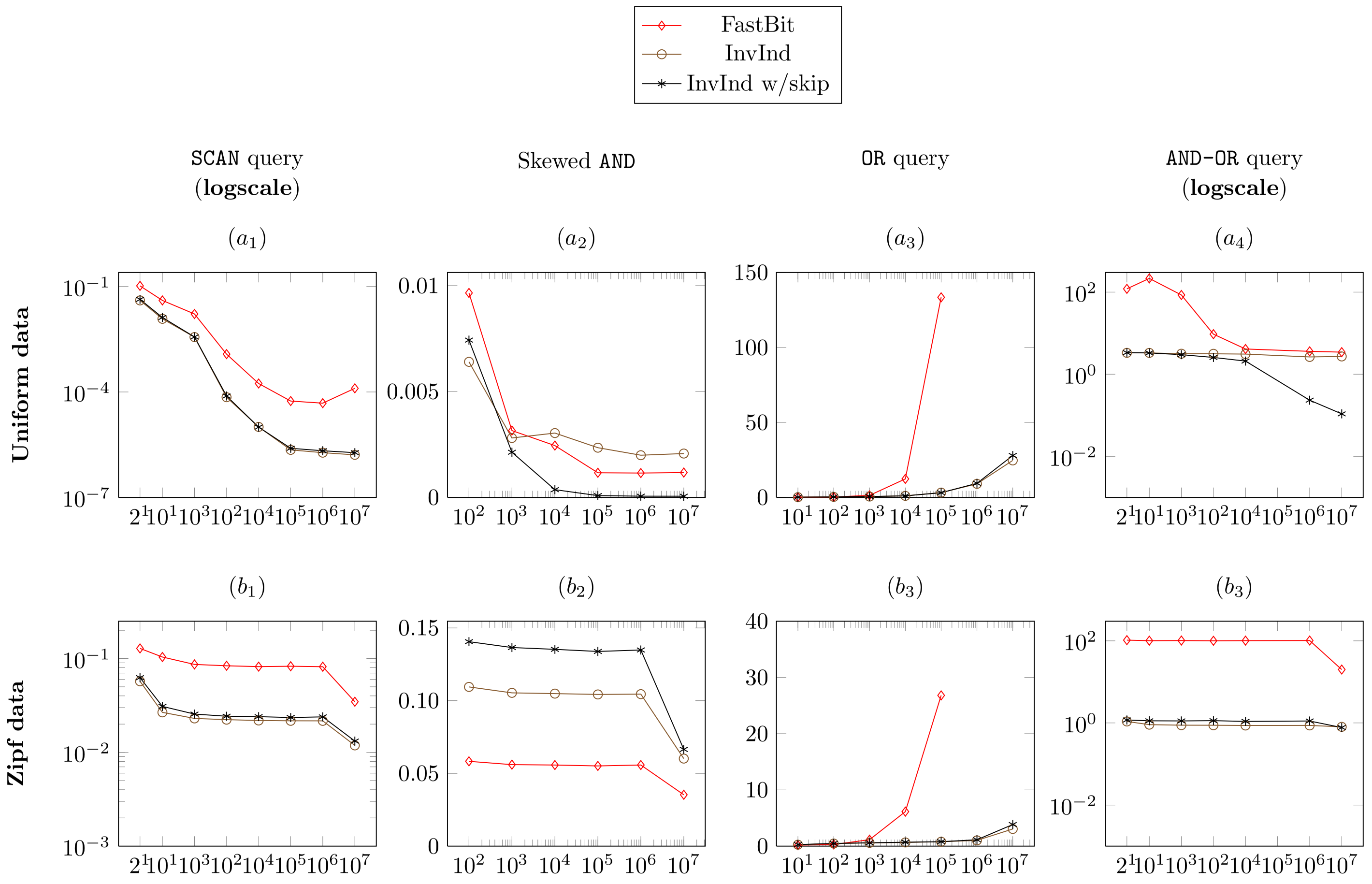


Figure 2: Results from running queries on generated tables, showing query time in seconds for varying cardinalities.

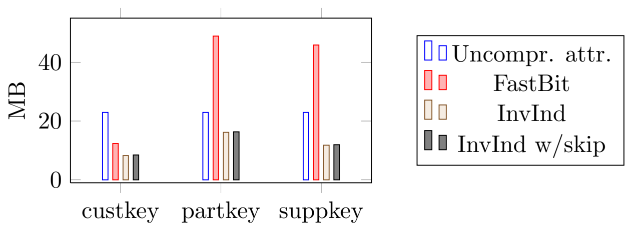


Figure 3: Size of indexes for foreign key columns in SSB

operands, both because it can combine multiple logical TIDs using one CPU instruction, and because it performs the operator before extracting the tuple references. Because neither input is smaller than the output for AND operators, FastBit decodes fewer references compared to the inverted indexes.

The multi-way OR operators in our solution demonstrate better scalability than FastBit with respect to the number of inputs for both tables.

The idea of skipping in OR operators is ideally suited for query type AND-OR, but it is only useful when the other operands to the AND return data that enables reasonable skip lengths, which occurs for high cardinality attributes with uniform distributions.

4.2 Star Schema Benchmark

Star schemas represent a best practice for how to organize data in decision support systems, and are characterized by a central fact table that references several smaller dimension tables. Typical queries on such schemas involve joins of the fact table with relevant dimension tables called star joins. Bitmap indexes can be constructed over the foreign keys in the fact table to speed up such joins, and are then called join indexes [16, 17]. We experiment with using inverted indexes as an alternative to bitmap indexes for this purpose in the Star Schema Benchmark (SSB) [14]. We use Scale Factor 1, and precalculate the foreign keys that match the queries in SSB. They are submitted as part of the query to the tested systems. We also avoid calculating the exact answer, but rather let all queries return the sum of an attribute of the fact table. This isolates the effects of the indexes while making sure the returned results are suitable for further look-ups in the fact table. There are four dimension tables in SSB, but we avoid constructing join indexes for the `Date` table because FastBit is unable to process the queries involving all tables without a very significant stack size.

4.2.1 Index Size

Figure 3 shows the join index sizes in both systems. FastBit has significantly larger indexes because the foreign keys have relatively high cardinalities. The attribute `custkey` is partly sorted, resulting in longer runs in WAH-compressed indexes, and the relative difference between FastBit and inverted indexes is therefore smaller in that case.

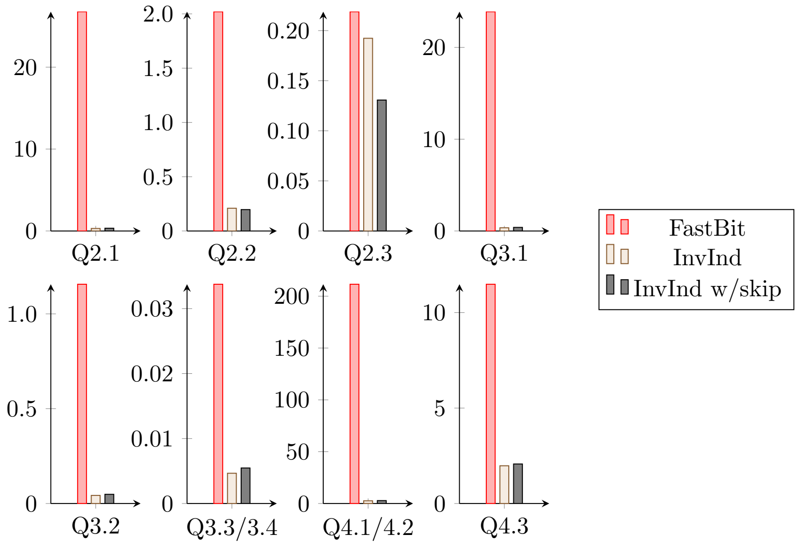


Figure 4: Query processing time in seconds for SSB queries.

4.2.2 Join Processing

The query processing results for SSB are given in Figure 4. Within each set of queries, the predicates on the dimension tables become increasingly selective, making FastBit perform better relative to inverted indexes because the `OR` operators that combine the tuples representing each qualifying foreign key have fewer inputs. Query 2.3 has skewed input to an `AND` operator making skipping important for performance. The `OR` operator providing dense input to the `AND` is over the attribute with the lowest cardinality, contributing to the smaller performance difference between FastBit and inverted indexes for this query.

5 Related Work

Several alternatives to the compression schemes discussed in this paper have been suggested both for bitmaps [4, 10, 5, 15, 21] and inverted indexes [25, 20, 1]. Experiments have shown that the query processing efficiency of WAH remains attractive, even though there are approaches resulting in smaller indexes. WAH is known to result in smaller indexes when the table is sorted on the indexed attribute [19, 12]. Due to space restrictions, we do not experiment with sorted tables in this paper. Experiments with compression in inverted indexes in IR have shown that PForDelta currently is the most efficient technique [29], and further improvements to the technique have also been suggested recently [28].

As an alternative to compression, there are several approaches that reduce the number of bitmaps in the index [17, 7, 8, 11, 22]. Strategies for operating on many bitmaps by processing two at a time have been explored for WAH-compressed bitmap indexes [26], and a recent paper suggests using multi-way operators for bitmaps, but the idea is not tested [12]. Query processing approaches in inverted indexes in IR have focused on term-at-a-time strategies in addition to the document-at-a-time approach used in this paper [6, 24, 18, 2, 3, 23].

6 Conclusions

In this paper, we have evaluated the applicability of compressed inverted indexes as an alternative to bitmap indexes in DSSs. Inverted indexes are generally significantly more space efficient. The only case where WAH-compressed bitmaps are clearly more compact is when the cardinality of the indexed attribute is very low. FastBit performs well on simple queries with dense operands, but inverted indexes are better in other cases, often significantly.

Acknowledgments: This material is based upon work supported by New York State Science Technology and Academic Research under agreement number C050061, by Grant NFR 162349, by the National Science Foundation under Grants 0534404 and 0627680, and by the iAd Project funded by the Research Council of Norway. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NYSTAR, the National Science Foundation or the Research Council of Norway.

References

[1] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. ADC*, 2004.

[2] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proc. SIGIR*, 2005.

[3] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. SIGIR*, 2006.

[4] G. Antoshenkov. Byte-aligned bitmap compression. In *Proc. DCC*, 1995.

[5] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *Proc. VLDB*, 2006.

[6] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *Proc. SIGIR*, 1995.

[7] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 27(2), 1998.

[8] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proc. SIGMOD*, 1999.

[9] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1), 1994.

[10] T. Johnson. Performance measurements of compressed bitmap indices. In *Proc. VLDB*, 1999.

[11] N. Koudas. Space efficient bitmap indexing. In *Proc. CIKM*, 2000.

[12] D. Lemire, O. Kaser, and K. Aouiche. Sorting improves word-aligned bitmap indexes. *CoRR*, abs/0901.3751, 2009.

[13] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4), 1996.

[14] E. O'Neil, P. O'Neil, and X. Chen. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

[15] E. O'Neil, P. O'Neil, and K. Wu. Bitmap index design choices and their performance implications. In *Proc. IDEAS*, 2007.

[16] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3), 1995.

[17] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. SIGMOD*, 1997.

[18] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10), 1996.

[19] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. In *Proc. ICDE*, 2005.

[20] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.

[21] M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on run-length and huffman encoding. *Information Systems*, 2008.

[22] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In *Proc. DEXA*, 2004.

[23] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proc. SIGIR*, 2007.

[24] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proc. SIGIR*, 2005.

[25] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes.* Academic Press, 1999.

[26] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, 2004.

[27] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1), 2006.

[28] H. Yan, S. Ding, and T. Suel. Inverted index compression and query processing with optimized document ordering. In *Proc. WWW*, 2009.

[29] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, 2008.

[30] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[31] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, 2006.

Submitted version

Abstract

Bitmap indexes are widely used in Decision Support Systems (DSSs) to improve query performance. In this paper, we evaluate the use of inverted indexes combined with adapted query processing strategies from Information Retrieval as an alternative. In a thorough experimental evaluation on both synthetic data and data from the Star Schema Benchmark, we show that inverted indexes are more compact than bitmap indexes in almost all cases. This compactness combined with efficient query processing strategies results in inverted indexes outperforming bitmap indexes for most queries, often significantly.

1 Introduction

Decision Support Systems (DSSs) need to support queries over large amounts of structured data, and bitmap indexes are often used to improve the efficiency of important query classes [19, 20]. Bitmap indexes can speed up query processing for both selection predicates and joins.

A standard bitmap index has one bitmap per distinct value for the indexed attribute, with 1's at positions for tuples with the represented value, and 0's elsewhere. Bitmaps can also be combined using bitwise operators to answer complex boolean predicates. For attributes with low cardinality, a bitmap index is relatively compact, but its space usage increases linearly with the number of attribute values, and for attributes with high cardinality space usage can be prohibitive. The main technique to reduce the space usage for high cardinality attributes is compression [5, 14, 35, 27], and there exist several compression schemes specifically developed for bitmaps such as BBC [5] and WAH [34, 32, 35]. Both schemes are based on run-length encoding, which leads to more compact representations of sparse bitmaps.

Bitmap indexes were a long time ago also used in Information Retrieval (IR), but they have today been mainly replaced by *inverted indexes*. An inverted index has, for each word appearing in the indexed documents, a reference to the documents containing the word [31, 7, 37]. Part of the reason why inverted indexes gained popularity in IR, was that most queries required ranking, and additional data related to the ranking function had to be integrated into the index. Adding additional fields is hard to do in a bitmap index, but relatively easy to do in an inverted index. This switch from bitmap to inverted indexes lead to a flood of research on efficient inverted indexes [31, 37, 10, 16, 30, 4, 38, 36], and inverted indexes are now the preferred indexing method in search engines [37].

In this paper, we are asking (and answering) the question: What are the trade-offs of using inverted indexes also in DSSs, and should they be considered a serious alternative to bitmap indexes? The main contributions of this paper are (1) the study of how to use and implement inverted indexes in DSSs, and (2) a thorough performance evaluation that compares inverted indexes and bitmap indexes in DSSs. In particular, we compare inverted indexes with FastBit[1], a state-of-the-art bitmap query processing and indexing system based on WAH-compressed bitmap indexes.

The rest of the paper is organized as follows. We first give some background on bitmap and inverted indexes (Section 2). We then introduce a query processing system based on multi-way operators that avoid materializing intermediate results. The operators support a Volcano-style iterator interface [13] with an extra operation, skipping, that helps processing skewed inputs more efficiently (Section 3). Compression schemes for bitmap indexes are known to be more efficient for sorted than unsorted data [22], and we show how inverted indexes can be adapted to perform well for sorted data (Section 4). We compare FastBit and our system experimentally with both synthetic data and data from the Star Schema Benchmark [17] (Section 5).

[1]http://sdm.lbl.gov/fastbit/

2 Background

2.1 Bitmap Indexes

The support for bitwise operations on CPUs enables efficient query processing of complex predicates with bitmap indexes, especially for relatively dense bitmaps. But when basic bitmap indexes are created for high cardinality attributes, the resulting indexes become large due to the large number of bitmaps required. There are several techniques that limit the number of bitmaps required. All of these techniques are in some sense orthogonal to bitmap compression because compression is applicable regardless of whether they are used or not, although the efficiency of the compression can be affected. We focus on compression in this paper, and leave a further investigation of using the orthogonal techniques in inverted indexes to future work.

Several compression schemes have been considered for bitmap indexes, both general-purpose and bitmap-specific approaches [5, 14, 34, 35, 6, 18, 27]. The first bitmap-specific techniques were introduced to ensure that performing boolean operators on the bitmaps was still efficient, even when the bitmaps were compressed. BBC [5] was an early example of such a technique, and WAH has been introduced later as a simpler variant with slightly larger indexes but improved query processing efficiency [33, 34]. Other recent approaches also have better compression rates than WAH, but the query processing efficiency in WAH remains attractive in most cases [18, 27]. FastBit is an openly available system using WAH. Its availability and attractive query processing efficiency motivated our choice of using it as a representative for bitmap indexes in our experiments.

Bitmaps represented using WAH are partitioned into sequences with $w-1$ bits in each, where w is the computer word size. If all the bits in a sequence are identical, this and neighboring sequences with the same property are represented in one fill word representing a run of equal values. If there are both 0's and 1's, the sequence is stored as a literal bitmap, and the remaining bit in the word is used to discriminate between fill words and literal words [34, 32, 35]. High cardinality attributes will be far more compactly represented with WAH than in uncompressed bitmaps, because most values will have bitmaps consisting mostly of 0's, resulting in few literal words and many sequences being combined into fill words. The compression is even more efficient if the table is sorted on the indexed attribute, which will typically result in runs of both 1's and 0's, and only a few words being literal bitmaps [22].

When bitmaps are used to index high cardinality attributes potential queries may involve many bitmaps. Wu et al. have considered approaches to process many bitmaps efficiently with focus on `OR` operators [32]. Five strategies are considered, and they differ in the order in which the bitmaps are combined, and how the intermediate results are stored. All of the strategies combine two and two bitmaps, making them fundamentally different from the multi-way operators we use for inverted indexes.

2.2 Inverted Indexes

Inverted indexes is the indexing method of choice in search engines. In IR terms, an inverted index has, for each word appearing in the indexed documents, a reference to

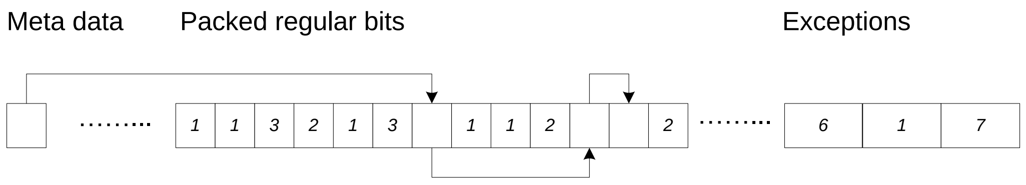


Figure 1: Compressed seq. $1, 1, 3, 2, 1, 3, 7, 1, 1, 2, 1, 6, 2$

the documents containing the word [31, 7, 37]. Transferring this idea directly to DSSs leads to a solution where, for each distinct value of an indexed attribute, there is a list of tuples with the represented value, referenced through the tuple identifier (TID). This basic scheme is similar to Value-List indexes [20].

In search engines, inverted indexes are usually compressed to reduce space usage and I/O [37]. Instead of storing each individual identifier in the inverted lists, they are stored as deltas from the previous entry. An advantage with storing deltas is that small values become more likely than large, and schemes capable of representing small values more compactly can be used to compress the lists. When compression was introduced in search engines, schemes with near-optimal compression rates, like Golombs code, were in focus [31]. Such a scheme has also been suggested used for compression of bitmap indexes [14].

In recent years, the main focus when evaluating compression schemes has been decompression speed. If decompression is efficient, compression can be a tool to improve query processing speeds when I/O bottlenecks limit the performance of the system. An example of a scheme developed with this in mind is VByte [25], and the most efficient scheme at the moment is PForDelta [38] according to a recent comparison study [36]. PForDelta provides good compression rates and very efficient decompression, making it the natural choice of compression method for the inverted indexes in this paper.

PForDelta is delta-coding using PFor to represent the deltas. The basic idea of PFor is to represent values using bit packing, but allowing exceptions to enable reducing the default number of bits per value. The method compresses and decompresses X values at a time, where X is a multiple of 32. X values compressed together using PFor is denoted a sequence. The sequence is stored within one disk block, possibly with data in three different parts of the block. One part has room to store the sequence with the chosen number of bits. Metadata for each sequence is stored in the beginning of the block, containing a pointer to the first entry (if any) that can not fit in the chosen number of bits. The value causing the exception will be stored in a list at the end of the disk block. If there are more than one such value within the sequence, the next is pointed to from the position left open by the last exception. Notice that the number of bits available there is limited, so it might be impossible to point all the way to the next exception. This is solved by introducing an artificial exception. An example of using PForDelta with 2 bits for each delta is shown in Figure 1.

There are two overall approaches to processing queries in search engines, term-at-a-time and document-at-a-time [10, 21, 30, 4, 29]. Term-at-a-time involves processing one inverted list at a time, and is intended for ranked queries where such strategies can avoid reading parts of the index. As we expect boolean predicates in DSSs and no ranking, document-at-a-time is the strategy we will focus on. The main idea of the strategy is that all relevant inverted lists are processed in parallel.

The idea of skipping has been introduced in search engines to avoid reading and decompressing parts of inverted lists that are not required to process a query. The idea stems from skip lists [23], but was introduced in a simple version in inverted indexes to enable forwarding to relevant matches. More advanced variants have been introduced later [16, 9].

3 Query Processing with Inverted Indexes

Bitmap indexes in DSSs support answering complex boolean predicates. Inverted indexes must support the same queries to represent a real alternative. Our solution consists of three basic operators. They all provide the same interface based on an extension to Volcano-style iterators [13]. In addition to the ability to iterate through TIDs in sorted order, our operators support skipping which is inspired by the idea of skipping in inverted lists in search engines. The skip function takes one argument describing the lowest relevant TID, and forwards the iterator accordingly. Our solution supports `SCAN` and multi-way `OR` and `AND` operators.

3.1 The `SCAN` Operator

The `SCAN` operator accesses the index structure. It finds the correct inverted list by looking up in the dictionary, which is a search structure over all distinct values for the indexed attribute. The basic inverted lists are compressed using PForDelta, and iterating through the list thus involves decompressing, and returning one value at a time.

A simple approach to implement a skipping function is to continuously request the next value until the returned value is relevant. This approach is in effect similar to not supporting skipping at all, and is used in our system as the baseline for all operators not supporting skipping. To support skipping for the `SCAN` operator, each kth TID is stored in a list external from the inverted list. Performing skipping in this case involves checking whether the TID to skip to is larger than or equal to the next TID in the external list. If so, we perform a binary search in the list to find the part of the inverted list to skip to. When the correct part is found, the baseline approach is used to find the exact result. This approach is similar to approaches used in inverted indexes in IR [16].

3.2 Multi-way `OR`

The `OR` operator takes several iterators as input and provides an iterator interface over the sorted merge of all the inputs. We follow an approach different from those described for bitmap indexes [32], and ours is inspired by both query processing and index construction in inverted indexes [31, 10, 30]. The main idea of the scheme is that we iterate through all input iterators at the same time, and organize them in a priority-queue based on a heap to find the next iterator to return a result from. When the next result is requested, the operator requests the next result from the iterator at the top of the priority-queue, and then maintains the priority queue. The ability to avoid materializing intermediate results is an advantage of this approach.

A simple approach for supporting skipping in the `OR` operator is to start with skipping in the iterator at the top of the priority-queue. If the iterator advances, the priority queue is maintained, before we repeat the procedure. We continue doing so until the iterator at the top of the priority queue has a next value larger than or equal to the value we skip to. In the worst-case, all iterators may require skipping, leading to a linear number of calls to maintain the priority queue. Alternatively, we could go through all input iterators and skip before we rebuild the complete priority queue. While the asymptotic complexity of this approach is better than for the first one, it might not work that well in practice, as there can be many iterators which do not need to skip at all.

Our approach is a compromise between these two. We essentially perform a breath-first search in the heap starting from the top. If the iterator being processed has a smaller next value than we skip to, we perform the skip in the iterator, add the children in the heap to the BFS queue, and add the entry itself to a stack of entries which require maintenance in the priority queue if the entry is not a leaf. Maintaining the priority queue will thus never be more expensive than in the second approach above, and the method might still perform well in practice when there are small skips.

3.3 Multi-Way `AND`

Many techniques can be used to implement a multi-way `AND` operator that does not materialize intermediate results. We follow a simple approach we expect to perform well if the input iterators support skipping. We order the input iterators according to the expected number of returned results, from lowest to highest. When the next result is requested from the iterator, we start by asking for the next result from the input iterator with the lowest number of expected results, and get a candidate returned. We then try to skip to the candidate in the following input iterators. If all have the value, it is returned. If one iterator does not have it, its next value becomes the next candidate and we go back to the first iterator.

Supporting skipping in the `AND` operator is straight forward. We just start with the value to skip to as the candidate, and try skipping to it for the input iterators.

4 Exploiting Sort Orders

If an attribute is part of the sort order for a table, it will typically have the same value for consecutive tuples. When such attributes are indexed with delta-compressed inverted indexes, many deltas will be one. Each of them will be stored explicitly when using standard compression schemes, even though simple alternative schemes might lead to better compression rates for such workloads. In search engines, where inverted indexes are most commonly used, each entry in the inverted lists will have associated extra data for ranking. This, along with the fact that sorting the input data is significantly more manageable for structured data, might be the reason why there is limited special support for workloads involving many deltas of one in inverted indexes.

WAH-compressed bitmap indexes are space efficient when representing sorted data, as explained in Section 2. In an attempt at matching WAHs compression rates for sorted data, we introduce a simple scheme based on run-length encoding. For a run of deltas of one, the scheme stores two values. It stores the delta from the end of the previous run to the beginning of this run, and the number of entries in the run. In order to keep things simple, we store these lists using PFor as well. For some types of data, it might be possible to achieve better compression rates when storing the deltas between runs using one number of bits and the length of the runs with another number of bits. In the interest of simplicity our system does not support such a solution, and we use one specific number of bits to represent both numbers.

A scheme based on run-lengths of deltas of one will obviously not be efficient in all cases. In order to enable choosing when to use this scheme and not, we support the same interface as the `SCAN` operator for PForDelta-compressed inverted indexes, and the two methods can be used interchangeably. We do not implement special support for skipping in the `SCAN` operator for this compression scheme, but only iterate through the runs until we find one containing the skip value or with a higher TID as the start of the run.

By interpreting this compression scheme in terms of bitmap indexes, we notice that the two numbers used have meaning there as well. The first number for each run represents the number of 0's before a 1 is found, and the second represents the number of 1's (after the first one) before the next 0 is found. The same scheme can thus be used to compress bitmaps, which is not a surprise as the two structures store the same data.

5 Experiments

To investigate the advantages and disadvantages of inverted indexes compared to bitmaps, we experiment with FastBit and our solutions based on inverted indexes. We present results from experiments with synthetic data and data from the Star Schema Benchmark (SSB) [17].

All experiments are run on a computer with a quad-core Intel Xeon CPU at 3.2 GHz, with 6 MB cache and 16 GB main memory. All implementations only run one thread processing the queries, and store their indexes on disk. We make FastBit construct indexes so that it has the same number of bitmaps as there are inverted lists in our implementations. This ensures that we compare both systems' ability to process queries with the same number of operands.

The index sizes referred to in the results are the sizes of the files used to store the indexes in the two systems. Queries are run hot by running them 10 times, and taking the average from the last 8 runs. To give the two systems equal opportunities to cache data, both perform all runs for a query in one invocation. We use FastBit version 1.0.5 (implemented in C++), and all queries are run without estimation, as that was most efficient. We give FastBit extra stack space (by calling `ulimit`), which is required to process queries involving many operators. Our system is implemented in Java (version

1.6). We use additional warm-up for our system, to enable run-time optimizations in the Java virtual machine that reduce variance between runs. Additional warm-up did not change the performance for FastBit.

5.1 Synthetic Data

In the first experiment we generate two tables of data, both with 9 attributes and 10 million tuples. The first attribute is a primary key which we do not index, while the other 8 are indexed by both systems. In the first table, all attributes have uniform distributions, while all attributes follow a Zipf distribution (with $k = 1.5$) in the second table. The attributes have different maximum cardinalities, ranging from 2 via 10, 100, 1 000, 10 000, 100 000, 1 000 000 to 10 000 000. We explore how the systems perform with respect to query processing performance and index sizes on these tables. We also explore the effects sorting has on these aspects.

6 different approaches are included in the experiments:

1. FastBit - a representative of WAH-compressed bitmap indexes.
2. Normal - the basic version of our system, which uses PForDelta to compress inverted lists and has no support for skipping.
3. Normal with skip - similar compression as normal, but supports skipping in all operators.
4. Run-length - uses the run-length based compression scheme introduced in Section 4, and does not support skipping.
5. Optimal space - chooses for each attribute the compression scheme that results in the smallest index. No skipping is used.
6. Optimal space w/ operator skip - This scheme uses the same indexes as Optimal space. Because the Run-length compressed indexes do not support skipping, we use the normal approach without skipping in `SCAN`s. Skipping in `AND` and `OR` operators is supported.

5.1.1 Index Size

The sizes of the indexes constructed for the tables with uniform and Zipf distributed data are shown in Figure 2, along with results for sorted versions of the tables, which will be discussed later. The size of the uncompressed attributes are included for reference.

When using standard PForDelta compression on the attribute with cardinality 2 in the table with uniform data, the most compact index uses 4 bits to represent each delta. This value seems unintuitively large for indexing an attribute with cardinality 2, but choosing lower numbers leads to many artificial exceptions, as mentioned in Section 2.2. Modifications to PForDelta that avoid artificial exceptions can thus lead to more compact indexes. Run-length compression does not result in very compact indexes either, as the runs in the data are not long enough. Bitmap indexes are known to be compact when the cardinality is 2, and FastBit outperforms our approaches in this case.

PForDelta compressed inverted indexes are compact for high cardinality attributes on the other hand. Taking the attribute with maximum cardinality 10 million from the

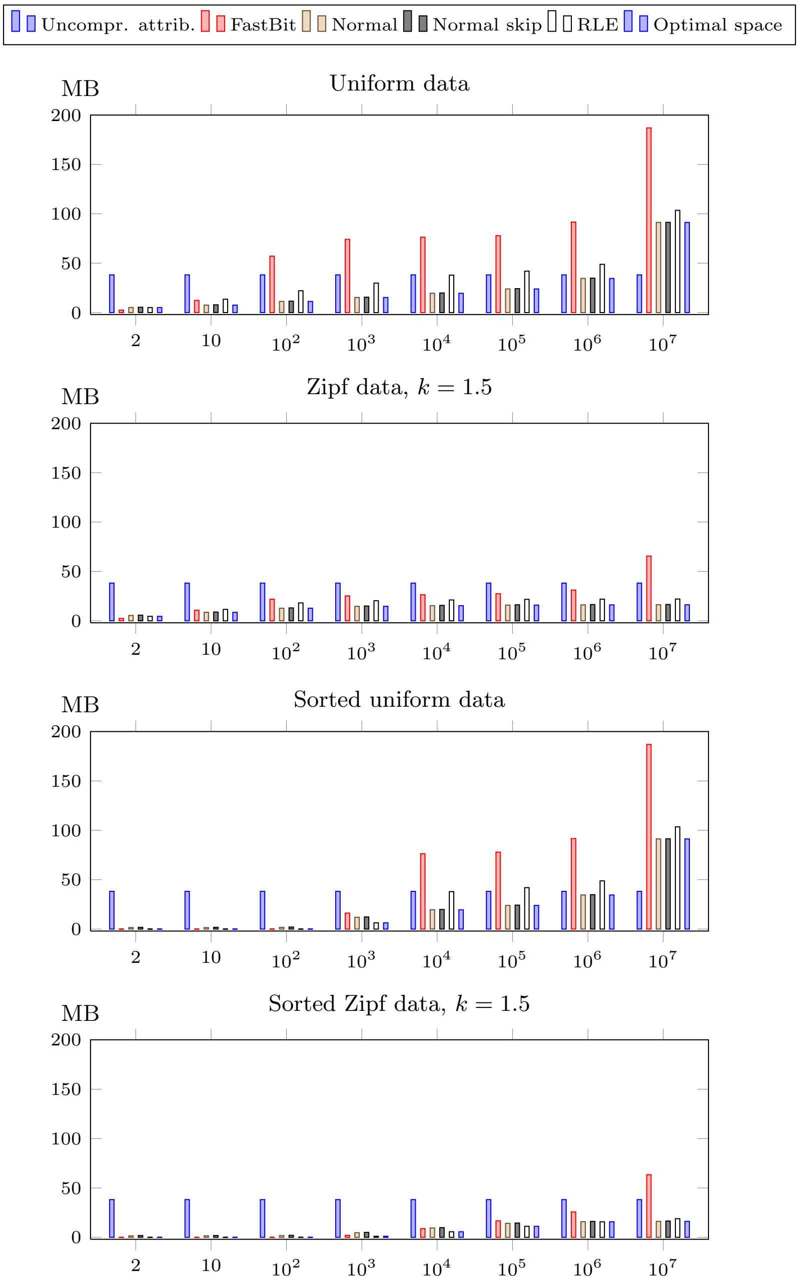


Figure 2: Index sizes. The x-axis represents the cardinality of the attribute.

table with uniform data as an example, our normal scheme uses 24 bits to represent each delta. This choice results in zero exceptions as the largest possible delta is 10 million. The inverted lists should thus consume approximately 29 MB of space in addition to some metadata. The reason why the reported index size is significantly larger is that the dictionary occupies approximately 62 MB. For the same attribute, bitmap indexes in FastBit will contain 10 million bits that are set to 1. In the worst case, all of these are in different bitmaps. Except for the bits in the first and last word, all of them require three words to store their bitmaps. There is a fill with 0's first, followed by the bitmap where the set bit is found, and another fill word with 0's, making the overall worst case space consumption over 228MB on a 64-bit architecture. The actual number from the experiments is significantly smaller, especially when considering that the reported size includes space used to store the dictionary. This reflects that the estimate is a worst-case. Still, the compression rates using PForDelta are clearly more attractive for high cardinality attributes.

Indexes for Zipf distributed attributes are more compact than indexes for uniformly distributed attributes with the same maximum cardinality, because a skewed distribution makes it less likely that the actual cardinality is equal to the maximum. The dictionary therefore becomes smaller, and the number of bitmaps or inverted lists is reduced.

WAH-compressed bitmap indexes are known to be more compact when tables are sorted. We therefore examine the effect of sorting on all tested indexes. Testing the effects of sorting involves choosing sort orders to test. Testing all sort orders for the tables introduced above is not feasible. We therefore start out with sorting the full tables according to a sort order based on the cardinalities, from lowest to highest. We will later extract some of the columns from the tables, on which we experiment with all sort orders. The sizes of all indexes with the chosen sort order for the full tables are included in Figure 2.

For attributes that are (partially) sorted, the length of runs of similar values increases, and Run-length therefore performs better relative to basic PForDelta compression. Run-length is preferable for more attributes when all attributes follow a Zipfian distribution compared to a uniform distribution, because the skew typically makes the actual cardinality lower than the maximum. One attribute with high cardinality makes the following attributes in the sort order less ordered than one with low cardinality.

To be able to test varying the sort order, we extract three attributes from the table with uniform data, and test the index sizes with all sort orders on the resulting tables. The attributes with maximum cardinalities 10, 1 000 and 100 000 are extracted, and the results for all sort orders are shown in Figure 3.

Attributes that occur early in the sort order are generally more compactly compressed because the runs of similar bits in WAH and deltas of one in Run-length are longer. The PForDelta compressed indexes are also smaller when the data is sorted, because the deltas are generally smaller, making it possible to use fewer bits per delta.

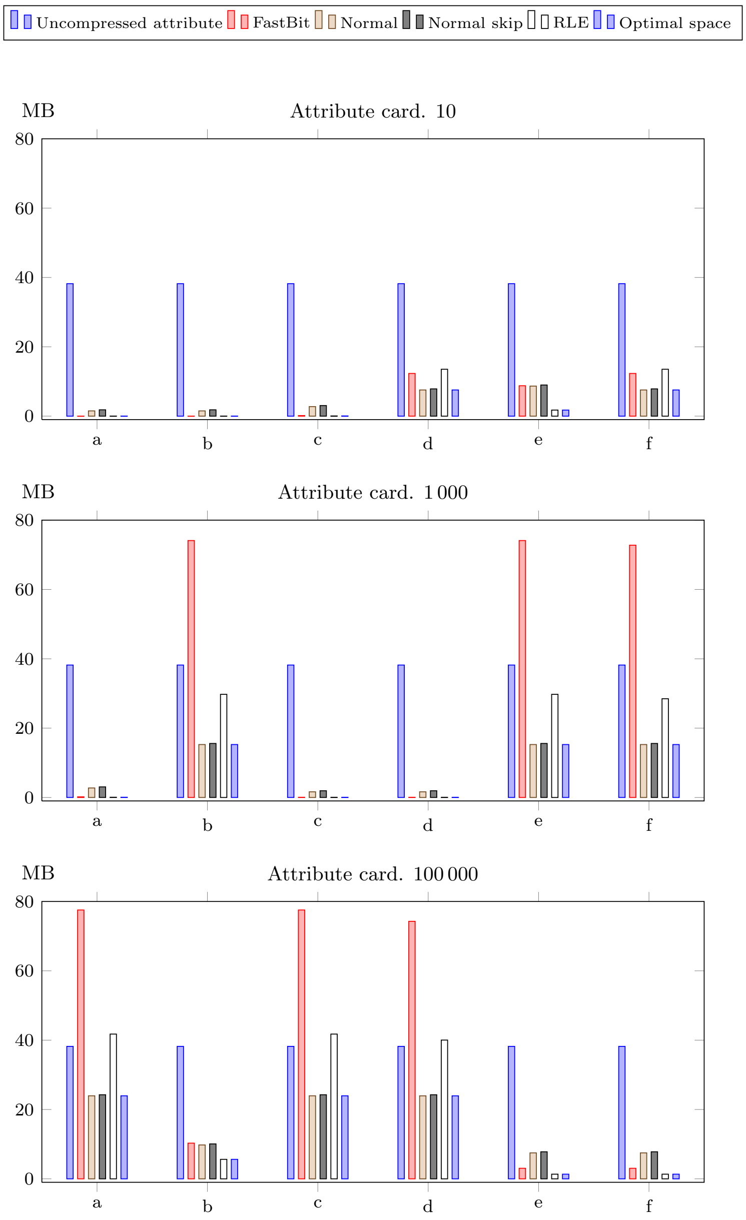


Figure 3: Sizes of indexes for sorted data with all sort orders: a=(10, 1 000, 100 000), b=(10, 100 000, 1 000), c=(1 000, 10, 100 000), d=(1 000, 100 000, 10), e=(100 000, 10, 1 000), f=(100 000, 1 000, 10).

5.1.2 Query processing

To experiment with query processing performance, we test four different query types which all vary the attribute on which there is a predicate. All queries ask for the sum of the primary keys of the tuples matching the predicates. This ensures that the results of the index accesses are suitable for further operations.

1. **Query type `SCAN`:** Finds all tuples with value 0 for a varied attribute.
2. **Query type skewed `AND`:** Finds all tuples having value 0 for the attribute with cardinality 10, in addition to 0 in one other varied attribute.
3. **Query type `OR`:** Finds all tuples having values in the lower half range for a varied attribute.
4. **Query type `AND-OR`:** Finds all tuples with value in the lower half range for the attribute with cardinality 100 000, and value 0 for another varied attribute.

In the generated table with uniform distributions, there were no tuples with value 0 for the highest cardinality attribute. When running queries with this attribute and using a predicate with a single value, we therefore substitute 2 for 0 to make sure that the scan returns a result.

The results from running all query types on both unsorted and sorted versions of the tables generated from uniform and Zipfian distributions are found in Figures 4, 5, 6 and 7.

When delta-compressed inverted indexes are decompressed, the result is a list of TIDs, which are well suited for looking up other attributes for the qualifying tuples. Performing lookups based on bitmaps representing qualifying tuples are not necessarily as straightforward, which can explain that FastBit is slower at virtually all scans compared to the other methods. The difference in index size also seems to have an impact as FastBit is relatively slower when its index size increases relative to the other approaches.

The scan queries on Zipfian data have different performance characteristics compared to on the uniform table. All attributes generated with Zipfian distributions are skewed so that low values are most common. The number of tuples having value 0 is thus far higher than for uniformly distributed data. All scans therefore return many tuples, and are relatively slow. However, `SCAN`s on the highest cardinality attribute are more efficient than on the others, because the value is required to be 2 as noted above. The query therefore returns fewer results, and is faster than the other scans.

Both FastBit and our schemes using Run-length compression take advantage of sorted data, and their decompression efficiency clearly benefits from a compact index, as can be seen for the lower cardinality attributes on both uniform and Zipfian data. The small difference in terms of number of results returned for different attributes with Zipfian data makes the curves for the sorted tables quite different.

Query type skewed `AND` tests whether the different methods can take advantage of the fact that one of the inputs is sparser than the other. Skipping is implemented to support such operations efficiently, but Normal with skip performs poorly for Zipfian data. As mentioned above, attributes with Zipfian data have value 0 for many tuples, and both inputs to the `AND` operator in the query are thus dense. The skip length will therefore never be particularly long, and it turns out to be overhead to try skipping for each step

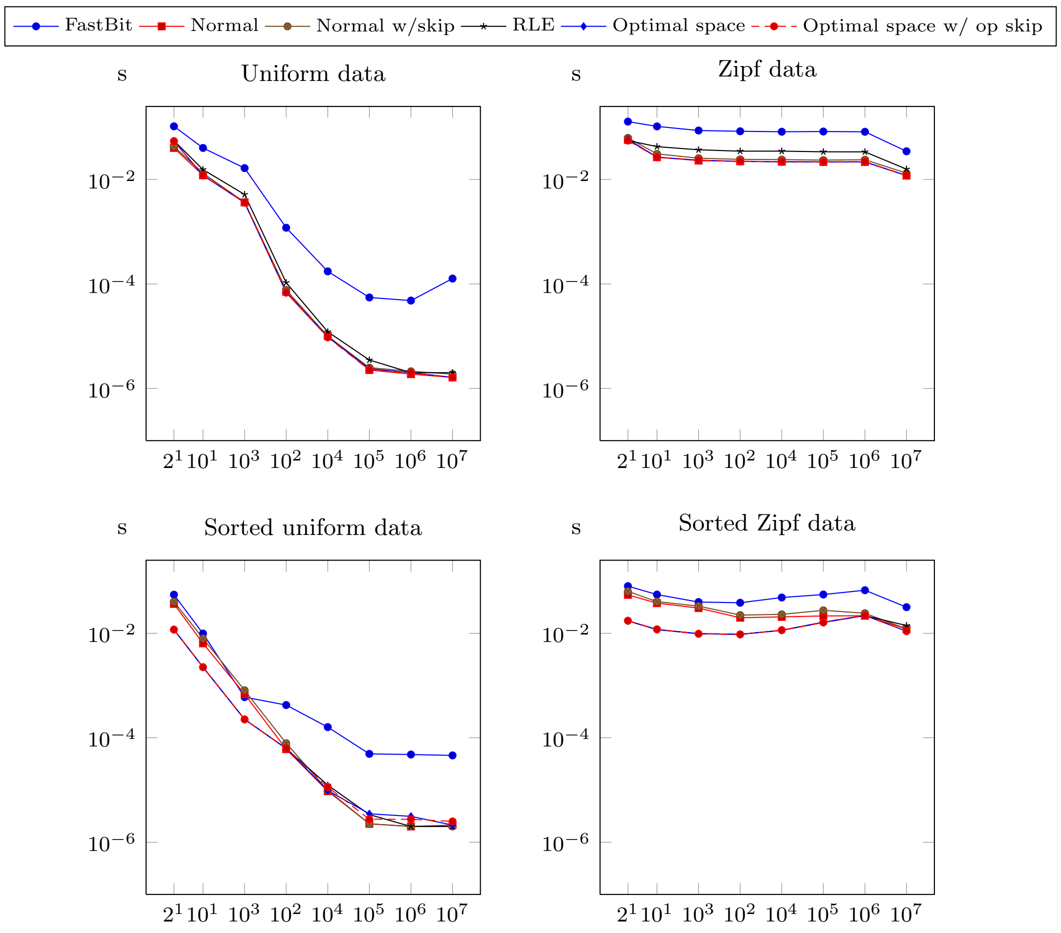


Figure 4: Query processing times for `SCAN` Query (**logscale**)

when using unsorted Zipfian data. For the sorted version of the Zipfian data however, supporting skipping is better than using Normal for the lower cardinality attributes, because the sorting makes certain large skips possible.

FastBit on the other hand performs well on all dense operands, partly because it can combine multiple logical TIDs using one CPU instruction. An additional advantage FastBit has when processing `AND` operators is that while inverted indexes decompress TIDs before performing operands, FastBit performs the operands first, before extracting references into the requested attribute. In `AND` operators, neither input is smaller than the output, making sure that FastBit decodes fewer references.

Even though a simple `AND` operator on dense operands with sorted data is an operation where we expect FastBit to have superior performance, our inverted indexes based on

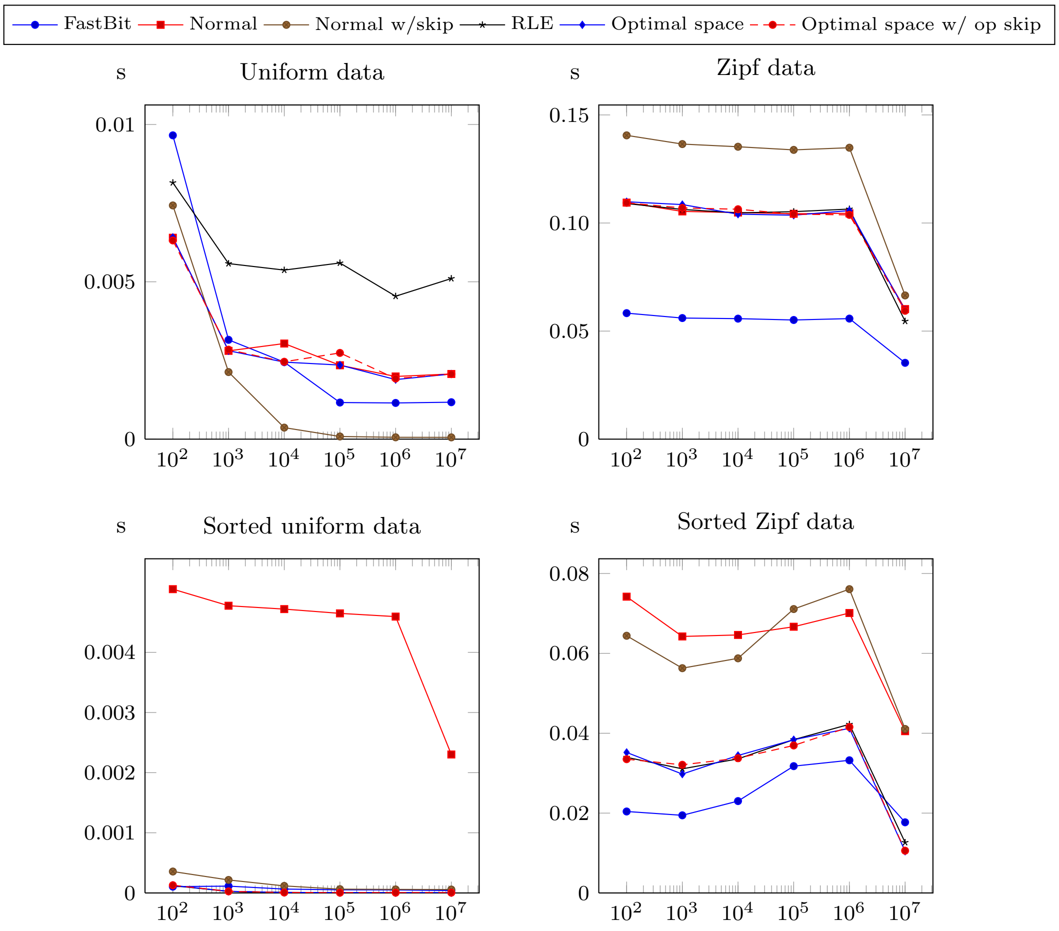


Figure 5: Query processing times for skewed `AND` query

Run-length compression also perform very well on the sorted Zipfian data. Run-length compression obviously results in compact indexes in those cases, and despite their lacking support for advanced skipping, they actually perform skipping quite efficiently on sorted data. In their basic skip operation, these operators check whether the TID to skip to is within this run or smaller. If so, the iterator is advanced to the appropriate TID. If not, the next run is processed. When the data is sorted and the runs are long, this results in a very efficient technique for skipping despite the lack of extra support for skipping through self-indexing.

The ability to skip is the reason why Run-length and the two Optimal approaches perform well for sorted uniform data. The Optimal space approaches use PForDelta without skipping for some of the other attributes, but as long as the most dense attribute is

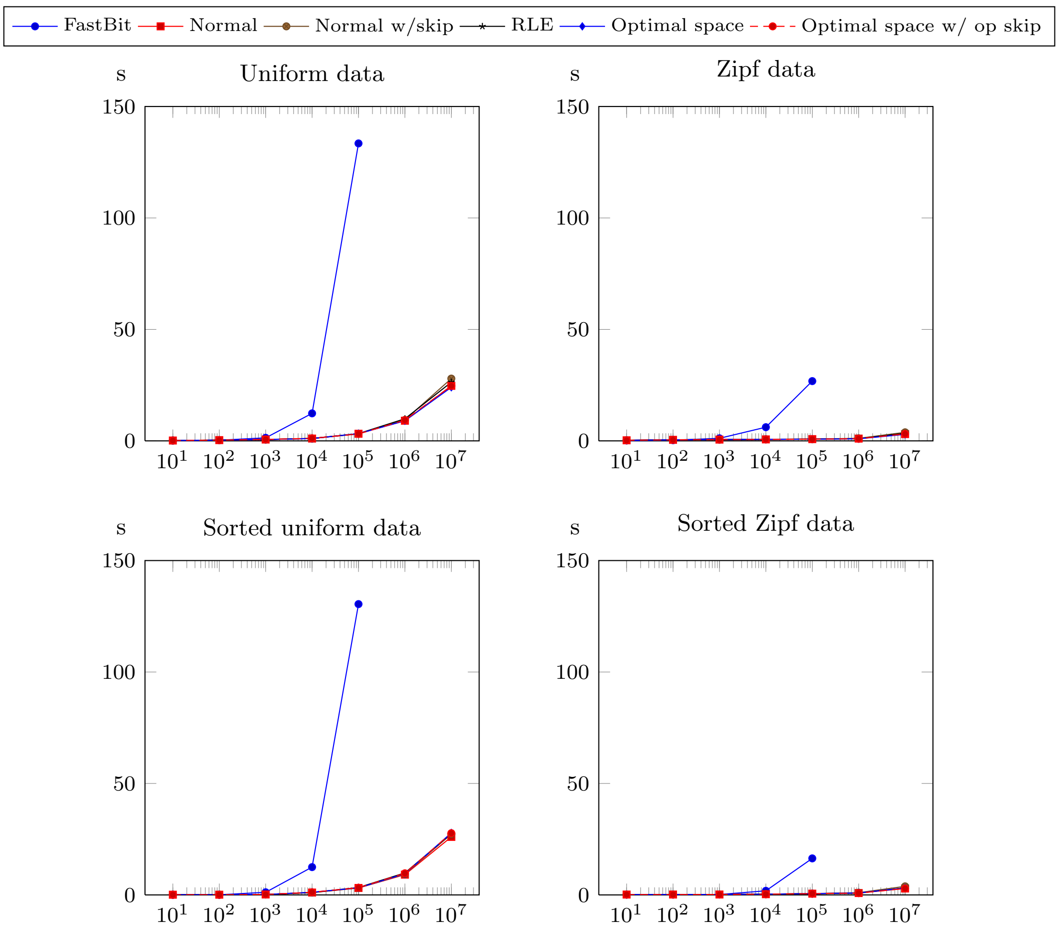


Figure 6: Query processing times for `OR` query

sorted and represented with Run-length, the skipping ensures good performance. Normal with skip achieves more or less the same performance with specific skipping support in the `SCAN` operator. Normal performs significantly worse than the others as it is the only approach without any support for skipping for sorted uniform data. The only attribute with value 2 for the last attribute in the sorted table with uniform data has a low TID when the table is sorted with the chosen sort order, and the query is thus terminated before Normal has scanned through the whole list for the attribute with cardinality 10, making the performance difference to other approaches less significant in that case.

While Run-length has good performance for sorted data, the large index sizes for unsorted uniform data has a negative effect on performance. The large difference in performance between sorted and unsorted data indicates that the idea of a scheme which

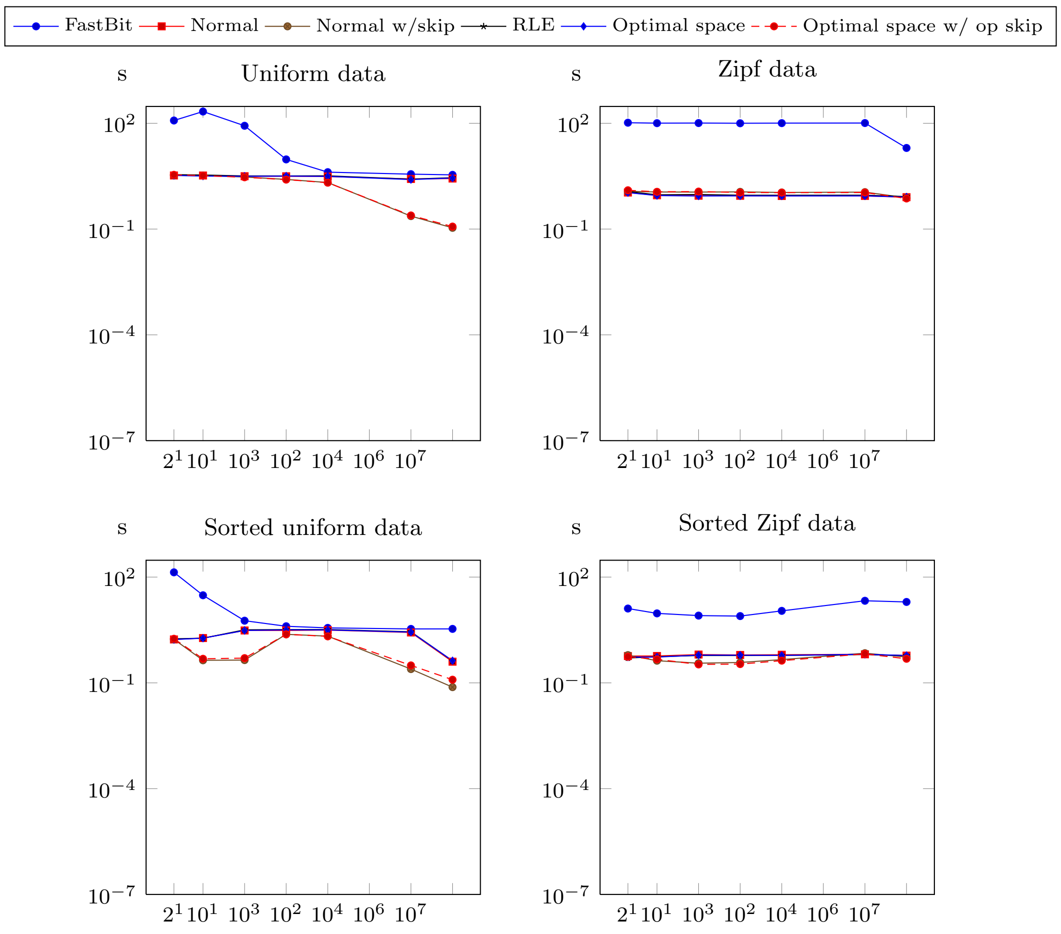


Figure 7: Query processing times for `AND-OR` query (**logscale**)

is optimal with respect to space consumption also makes sense in terms of query processing performance.

The multi-way `OR` operators in our solution demonstrate better scalability than FastBit when performing an `OR` between many inputs, as can be seen for all tables for query type `OR`.

The idea of skipping in `OR` operators is ideally suited for query type `AND-OR`, but it is only useful when the other operands to the `AND` return data that enables reasonable skip lengths. For unsorted uniform data, the skip length is long enough to have a significant impact when the attribute with the simple value predicate has high cardinality. For a sorted attribute on the other hand, the multi-way `OR` only has to be processed for the few ranges of TIDs that have the selected value, and skipping in the `OR` operator therefore has a performance impact for several cardinalities in the sorted uniform table. FastBit also seems to be able to take advantage of predicates to filter the results from the `OR`, and approaches the performance of those of our schemes without support for skipping in operators for high cardinalities.

To investigate the effect of sorting on query processing further, we use the tables generated to test the index sizes for all sort orders. All of the queries used for the full tables that access only the extracted attributes are run, and the results are shown in Figures 8 and 9. Because of space constraints, we only include results for FastBit and Optimal space with operator skip. We chose this scheme because it provides good compression in all cases, and comparable query processing efficiency to the best approach in all query types except skewed `AND` on unsorted tables in previous experiments.

Scans for tuples with a given value in an attribute are more efficient when the scanned table is sorted on the attribute, because the compression is more efficient both for FastBit and our scheme. For high cardinality attributes, the compression rates in FastBit are less attractive, and more data must be accessed relative to our approach. All scans of the attribute with cardinality 100 000 are efficient, so otherwise negligible overheads might also have a significant impact.

The choice between using standard PForDelta compression without skipping and the Run-length scheme for each attribute, along with different degrees of sorting, have significant impacts on the efficiency of the indexes for query type skewed `AND`. In sort order f in query skewed `AND` 1 000, our scheme suddenly performs poorer than FastBit. In that sort order, the optimal scheme will only choose to use the run-length scheme for the attribute with cardinality 100 000, indicating that the others are not really sorted. Because none of the relevant inverted lists represent sorted data, we can not expect one of the lists to end at an early TID. In addition, as PForDelta compression is used, the skipping abilities of Run-length explained above do not have any effect. Our approach does a simple `AND` operation on two inverted lists, an operation which FastBit performs well with reasonable cardinalities. By enabling skipping, we could have gotten better performance in this case. Part of the same intuition explains the performance for sort order e. The attribute with cardinality 10 is stored with Run-length compression there however, implying that the data is partly sorted and therefore enabling some skipping. The significant outlier for our approach for query skewed `AND` 100 000 occurs because both operands are stored using PForDelta and the same explanation as above applies. We might expect sort order f to perform equally poorly, as the attribute with cardinality 10 is represented using PForDelta there as well. That does not happen because the table is completely sorted on the attribute with cardinality 100 000. After the iterator for value 0 for that attribute has processed approximately 100 tuples, the `AND` can terminate because there are no more results from one of the operands. The results indicate that FastBit does not take full advantage of such aspects.

The other results follow the intuitions from previous results in that FastBit consistently performs better relative to our approaches when the number of operands is kept low.

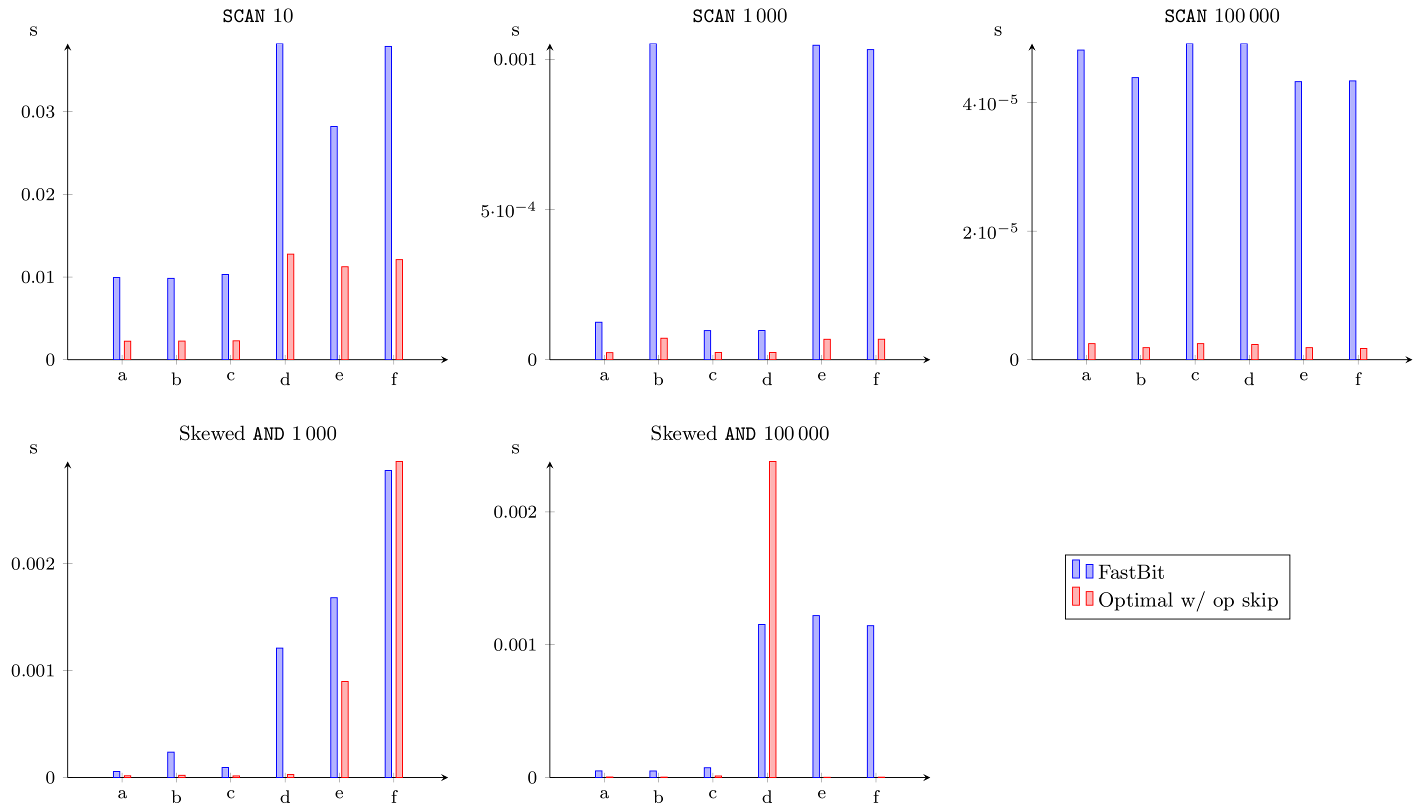


Figure 8: Query processing times on different sort orders: a=(10, 1 000, 100 000), b=(10, 100 000, 1 000), c=(1 000, 10, 100 000), d=(1 000, 100 000, 10), e=(100 000, 10, 1 000), f=(100 000, 1 000, 10).

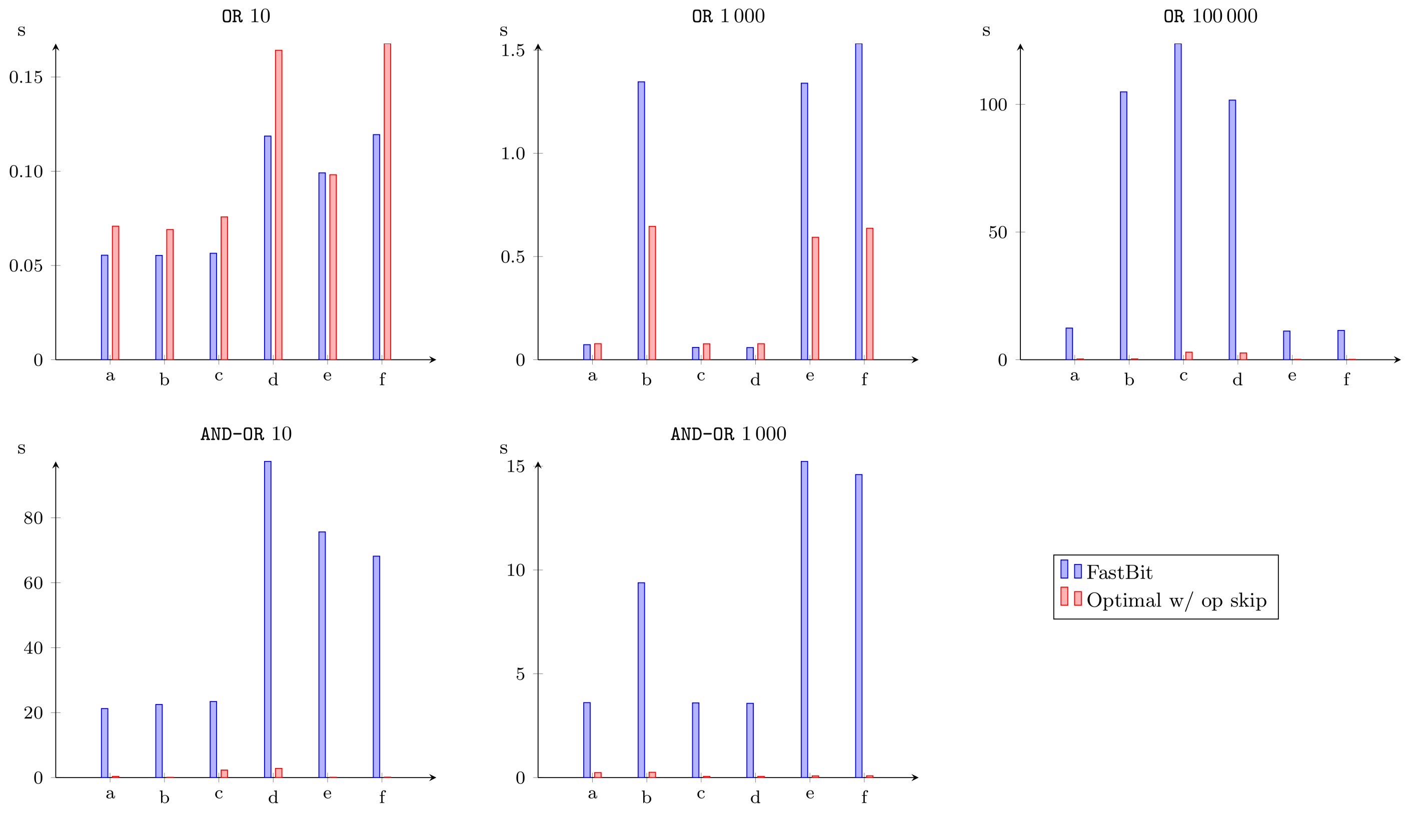


Figure 9: Query processing times on different sort orders: a=(10, 1 000, 100 000), b=(10, 100 000, 1 000), c=(1 000, 10, 100 000), d=(1 000, 100 000, 10), e=(100 000, 10, 1 000), f=(100 000, 1 000, 10).

5.2 Results for Star Schema Benchmark

Star schemas represent a best practice for how to organize tables in decision support systems, with their central fact table that references several smaller dimension tables. Aggregations over tuples in the fact table that reference tuples satisfying some predicates on dimension tables are typical queries on such schemas. Those queries involve processing a join between the fact table and all relevant dimension tables and are called star joins. Bitmap indexes have been suggested used as join indexes to speed up such joins [19, 20], and we therefore experiment with using inverted indexes as an alternative for this purpose. Our experiments are based on the Star Schema Benchmark (SSB) [17], which is a simplification of TPC-H. It consists of a central fact table and four dimension tables, `Customer`, `Supplier`, `Part` and `Date`. When bitmaps are used as join indexes, they are built over the foreign key attributes. The bitmap index is used to lookup bitmaps for the primary keys of those tuples that satisfy the selection predicates for the dimension tables. In our experiments we try to isolate the effect of the indexes, by assuming that the foreign keys representing matches are already known. We let all queries return the sum of an attribute of the fact table, as this shows that the result is returned on a format suitable for looking up data used in aggregations. The queries we run are based on the queries in SSB, but as explained above we precalculate the foreign keys we look for in each foreign key attribute and do not compute the complete answer.

As some queries involve a lot of tuples that qualify in the different dimension tables, they become quite long and FastBit would require a very significant stack size to process them. To limit this problem we chose to construct indexes over all foreign key columns except `Date`, thereby limiting the size of several queries significantly. We present the results both in terms of index size and query processing efficiency on data from SSB with scale factor 1. As in our previous experiments, we use indexes with one bitmap/inverted list per distinct value.

5.2.1 Index Size

Figure 10 shows the sizes of the join indexes constructed in both systems, with the size of the attributes stored uncompressed included for reference. The reason why FastBit in general results in significantly larger indexes is that the foreign key columns have relatively high cardinalities. The attribute `custkey` is partly sorted, and the relative differences between FastBit and our compression schemes are therefore smaller in that case.

5.2.2 Join Processing

The results from processing the joins from SSB with the different indexes are shown in Figure 11. Within each set of queries in SSB, the predicates on the dimension tables become more and more selective, and FastBit thus performs better and better as the `OR` operator combining bitmaps for all qualifying foreign keys involve less and less bitmaps. Query 2.3 has a significant fraction of qualifying tuples in one of the dimension tables and few in the other, giving the `AND` operator skewed inputs. The methods supporting

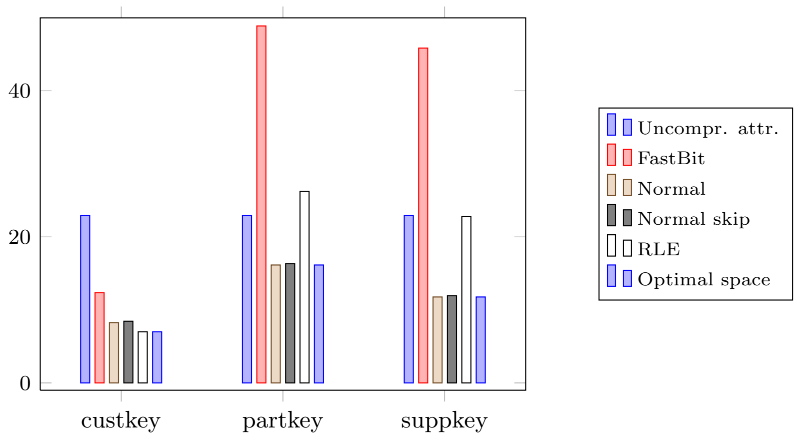


Figure 10: Size of indexes for foreign key columns in SSB

skipping in the `OR` operator thus perform better than the others. The foreign key attribute with the highest fraction of qualifying tuples in this case is also the one with the lowest cardinality, a fact that ensures that the performance difference between FastBit and our approaches is less significant in this case than for other queries.

6 Related Work

Several compression techniques for bitmaps apart from BBC and WAH have been suggested. Some approaches are based on run-length encoding of the bits set to 0 which results in schemes which are essentially equivalent to delta compression of tuple identifiers. [14, 26, 27]. These solutions all use different compression techniques than we do, for example based on Huffman coding [26, 27], and do not consider using aspects of the query evaluation strategies with inspiration from IR used in this paper. Another approach includes storing verbatim bitmaps for attributes up to a certain density, and as TID-lists for higher densities [18]. The TIDs are not compressed except from storing a within block TID using 2 bytes instead of a global TID. Other techniques take different approaches like using a compression technique based on bloom filters for bitmap indexes [6]. As mentioned in Section 2, the focus of compression schemes for inverted indexes in recent years has been fast decompression. Examples of schemes with such a focus include VByte, which represents each delta with a variable number of bytes, using fewer bytes for lower values [25]. Another approach is simple-9 [2] which is a word-aligned approach. Experimentation with several methods is found in [36].

The size of bitmap indexes can also be reduced through transformations. One example of such a technique is bit-slicing [20], where a value in an attribute is represented by a one in bitmaps representing bits set to one in a binary representation of the value. Bit-slicing has also been generalized to general decompositions [11, 12]. To speed up certain queries,

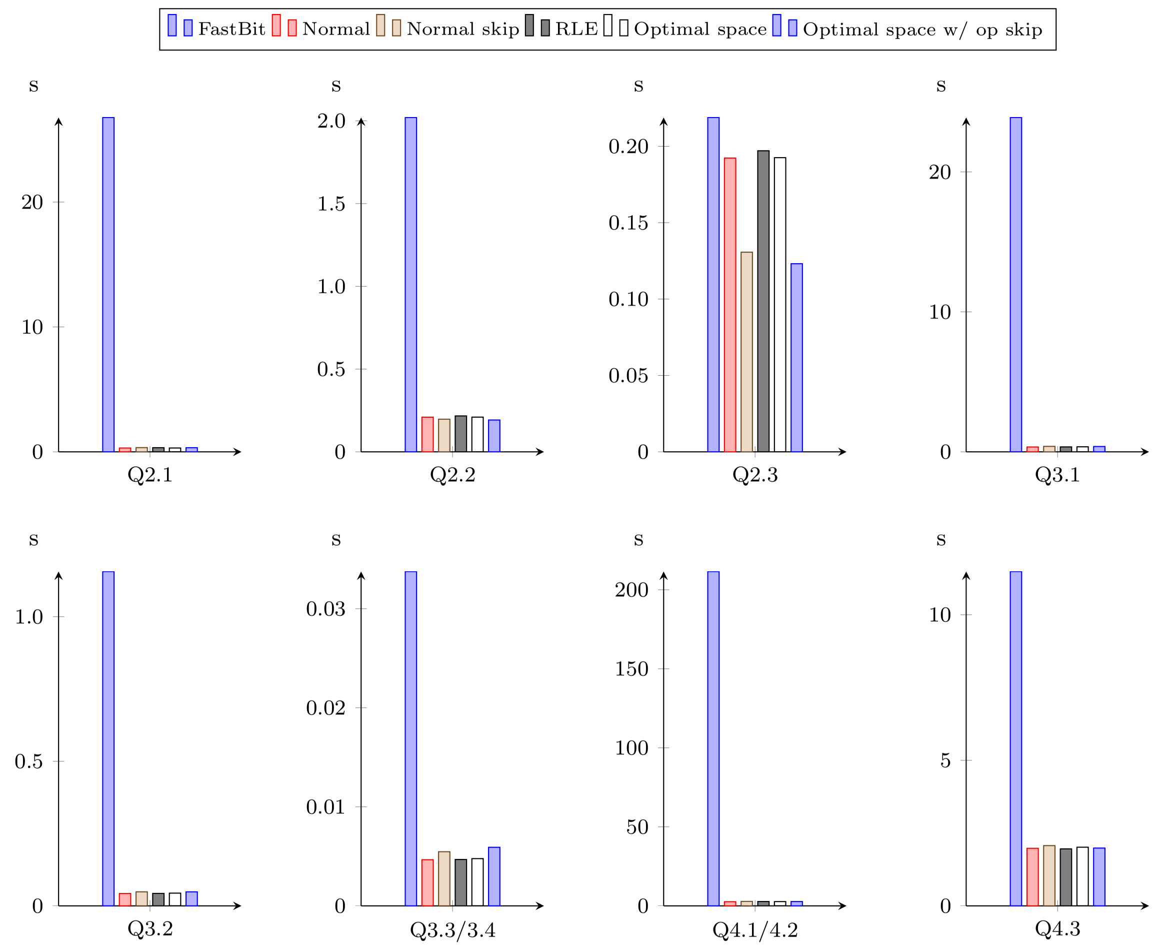


Figure 11: Query processing times for star joins from SSB queries

it has been suggested to let each bitmap represent overlapping ranges or intervals of values [12]. Another technique is to combine bitmaps for several values into one. This technique is called binning [15, 28]. Binning requires candidate checking to avoid false hits [24]. Techniques reducing the cardinality in the index have not received much attention for inverted indexes, but the idea occurs in a technique based on merging the inverted lists for lexicographically close words to support prefix search in search engines [8].

Strategies for query processing when different bitmaps are represented with different compression schemes has been addressed for bitmap indexes [14, 1], and strategies for operating on many bitmaps have been explored when WAH is used as the compression scheme [32], as mentioned in Section 2. Query processing approaches in inverted indexes in IR have focused mainly on when the different inverted lists are accessed to support efficient ranked queries [10, 30, 21, 3, 4, 29].

7 Conclusion

In this paper, we have evaluated the applicability of compressed inverted indexes as an alternative to bitmap indexes in DSSs. In terms of index sizes, our approaches are generally significantly more space efficient. The only case for which WAH-compressed bitmaps are clearly more compact than all of our approaches is when the cardinality of the indexed attribute is very low and there are only short runs of similar values. Modifications to the compression scheme used in our approach could help limit this difference.

FastBit performs well on simple queries with dense operands, whereas our approaches are better in most other cases tested, and often significantly better. Join indexes has been a traditional application scenario for bitmap indexes, and our experiments show that inverted indexes based on ideas from IR are more compact and give better performance than WAH-compressed bitmap indexes on the joins in all queries from SSB. We do not integrate transformations of the indexes to reduce the cardinality in our experiments however. A further comparison with such features enabled in both approaches is part of future work. It may also be interesting to include ideas from inverted indexes used in this paper, such as multi-way operators, into WAH-compressed bitmaps.

References

[1] S. Amer-Yahia and T. Johnson. Optimizing queries on compressed bitmaps. In *Proc. VLDB*, 2000.

[2] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. In *Proc. ADC*, 2004.

[3] V. N. Anh and A. Moffat. Simplified similarity scoring using term ranks. In *Proc. SIGIR*, 2005.

[4] V. N. Anh and A. Moffat. Pruned query evaluation using pre-computed impacts. In *Proc. SIGIR*, 2006.

[5] G. Antoshenkov. Byte-aligned bitmap compression. In *Proc. DCC*, 1995.

[6] T. Apaydin, G. Canahuate, H. Ferhatosmanoglu, and A. S. Tosun. Approximate encoding for direct access and query processing over compressed bitmaps. In *Proc. VLDB*, 2006.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[8] H. Bast and I. Weber. Type less, find more: fast autocompletion search with a succinct index. In *Proc. SIGIR*, 2006.

[9] P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In *Proc. SPIRE*, 2005.

[10] E. W. Brown. Fast evaluation of structured queries for information retrieval. In *Proc. SIGIR*, 1995.

[11] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. *SIGMOD Rec.*, 27(2), 1998.

[12] C.-Y. Chan and Y. E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proc. SIGMOD*, 1999.

[13] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1), 1994.

[14] T. Johnson. Performance measurements of compressed bitmap indices. In *Proc. VLDB*, 1999.

[15] N. Koudas. Space efficient bitmap indexing. In *Proc. CIKM*, 2000.

[16] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4), 1996.

[17] E. O'Neil, P. O'Neil, and X. Chen. http://www.cs.umb.edu/ poneil/StarSchemaB.PDF.

[18] E. O'Neil, P. O'Neil, and K. Wu. Bitmap index design choices and their performance implications. In *Proc. IDEAS*, 2007.

[19] P. O'Neil and G. Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Rec.*, 24(3), 1995.

[20] P. O'Neil and D. Quass. Improved query performance with variant indexes. In *Proc. SIGMOD*, 1997.

[21] M. Persin, J. Zobel, and R. Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *J. Am. Soc. Inf. Sci.*, 47(10), 1996.

[22] A. Pinar, T. Tao, and H. Ferhatosmanoglu. Compressing bitmap indices by data reorganization. In *Proc. ICDE*, 2005.

[23] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6), 1990.

[24] D. Rotem, K. Stockinger, and K. Wu. Optimizing candidate check costs for bitmap indices. In *Proc. CIKM*, 2005.

[25] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.

[26] M. Stabno and R. Wrembel. RLH: bitmap compression technique based on run-length and huffman encoding. In *Proc. DOLAP*, 2007.

[27] M. Stabno and R. Wrembel. RLH: Bitmap compression technique based on run-length and huffman encoding. *Information Systems*, 2008.

[28] K. Stockinger, K. Wu, and A. Shoshani. Evaluation strategies for bitmap indices with binning. In *Proc. DEXA*, 2004.

[29] T. Strohman and W. B. Croft. Efficient document retrieval in main memory. In *Proc. SIGIR*, 2007.

[30] T. Strohman, H. Turtle, and W. B. Croft. Optimization strategies for complex queries. In *Proc. SIGIR*, 2005.

[31] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes.* Academic Press, 1999.

[32] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *Proc. VLDB*, 2004.

[33] K. Wu, E. J. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In *Proc. CIKM*, 2001.

[34] K. Wu, E. J. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *Proc. SSDBM*, 2002.

[35] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1), 2006.

[36] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, 2008.

[37] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[38] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, 2006.

Appendix A.3

Paper III

Search in Social Networks with Access Control

Truls A. Bjørklund, Michaela Götz and Johannes Gehrke

Keyword Search on Structured Data (KEYS), workshop at SIGMOD 2010

Abstract

More and more important data is accumulated inside social networks. Limiting the flow of private information across a social network is very important, and most social networks provide sophisticated privacy settings to control this flow. Creating such extensive access control knobs makes the search for content a hard problem since each user sees a unique subset of all the data.

In this work, we take a first step at integrating access control based on a social network in a search system. We describe a set of solutions to the problem, including what indexes to construct and how to filter out inaccessible results. An experimental analysis illustrates the tradeoffs of the various strategies, and we point out a set of interesting future research directions in this area.

1 Introduction

We consider the problem of keyword search in a social network where the network determines what data a user has access to. Search companies have already set their eyes on data in social networks: Google and Bing support search over Twitter posts. But due to privacy concerns, users are more and more limiting visibility of their data to only their social contacts. Facebook already allows users to search over the most recent posts of their friends, but the technical details are proprietary. Google also plans to include data shared by friends on social networks in search results.[1] Although there is obvious commercial interest in the problem of search in social networks with access control, we are not aware of previous academic work addressing it.

Enforcing access control in search engines is already supported in most desktop and enterprise search engines; a straight-forward solution is to create a separate index for each user, indexing only the documents accessible to that user. However, related work has shown that this approach does not scale to a large number of users because of the redundancy resulting from documents accessible by several users. Büttcher and Clarke suggest to build a global index for all documents and filter out results in a post-processing step [4], while Singh et al. propose to build one index per data collection with the same access permissions [11].

In this paper we describe the problem of keyword search in social networks with access control, and we make a first step towards a solution to the problem. We make the following contributions:

- We lay out a design space with two axes that exhibit beautiful symmetry: The first axis describes how to organize data into indexes, and the second axis describes how to organize access control information into author lists. (Section 3)
- In a thorough experimental evaluation with a real system and real and synthetic data, we show tradeoffs between important points in the design space. (Sections 4 and 5)
- We describe an exciting agenda for future research. (Section 7)

We discuss further related work in Section 6.

2 Problem Statement

Our goal is to support keyword search over content generated in a social network; we will refer to a piece of content both as a document and a post. Each document is *authored* by one user in the network. We view the social network as a directed graph (V, E). This captures both undirected graphs such as Facebook as well as directed graphs such as Twitter. Each node $u \in V$ corresponds to a user in the social network, and there is a directed edge $(u, v) \in E$ if user v is a friend of user u in the social network; we also say that u is one of v's *followers*. We denote by $F_u = \{v|(u, v) \in E\}$ the set of friends of user u and by $O_u = \{v|(v, u) \in E\}$ the set of followers of u.

[1] `http://www.networkworldme.com/v1/news.aspx?v=1&nid=3236&sec=netmanagement`

A user has access to the documents authored by herself and to the documents authored by her friends. Thus, the result of a keyword query submitted by user u should include only relevant posts that (1) contain the search terms and that (2) are authored by a user in F_u. We call such keyword queries *queries with access control*, and since all queries considered in this paper are queries with access control, we will refer to them simply as *queries.* We rank the results of a query according to recency with more recent posts ranked higher than older posts; then we retrieve the top-k results. We can now (very informally) define the problem of keyword search in social networks with access control: Design a search system that enables fast queries with access control, efficient document updates, and that is also space-efficient.

We would like to emphasize that this problem definition makes several simplifying assumptions (for example, it assumes that the network structure is static); we will discuss extensions of this basic case when we discuss the research agenda in this space in Section 7.

3 Design Space

Keyword search is usually enabled through an *inverted index.* Given a set of documents, an inverted index contains a *posting list* for each unique keyword in the documents. A posting list consists of *postings*; each posting describes an occurrence of the keyword in one document, including metadata used for ranking. Inverted indexes are the most commonly used data structure in today's search engines, and they have highly optimized implementations [14]. All solutions in this paper use the inverted index as a basic building block. We are now ready to describe the two axes of our design space.

3.1 The Index Axis

Our first axis is motivated by the observation that we can enforce access control by building not just a single inverted index, but a set of inverted indexes where each index contains all documents from a set of users. When processing a query from a user u, we do not need to involve indexes that do not contain documents authored by users in F_u. We can describe this more formally with the following two definitions.

Definition 1 (Group-Index) *Given a set of users U, a* group-index *for U is an inverted index I that indexes all documents of users $u \in U$ and only documents of users $u \in U$; we say that $u \in U$ is a* member *of I.*

For convenience, we will often identify an index I with its members U.

Definition 2 (Index Design) *Let $G = (V, E)$ be a social network. An* index design *$\mathcal{I}$ for G is a set of group-indexes $\mathcal{I} = \{I_1, \dots, I_k\}$ such that for each user $u \in V$ there exists a $j \in \{1, \dots, k\}$ such that u is a member of I_j. We call k the* cardinality *of the index design, and we call the average number of group-indexes that a user is a member of the* redundancy *of the index design.*

We can now characterize different index designs based on their cardinality and redundancy. Intuitively, a high cardinality has the advantage that for a user u there might be a set of group-indexes that closely "match" F_u; it has the disadvantage that we may have to combine the results from many indexes to answer a query. High redundancy implies both a large space consumption and poor update performance, but can again help to design group-indexes that closely "match" F_u for a user u with a small number of indexes.

There is a very large number of possible index designs. In this first paper, we only explore the space of extreme points to understand basic tradeoffs: We only consider cardinality 1 and $|V|$, and we only consider redundancy 1 and a, which is the average number of followers of a user. This gives rise to the following four index designs written as (cardinality, redundancy):

- (1,1): A single index stores all the documents; we call this design *global index.*
- (1,a): This solution is obviously suboptimal, and we will not consider it further.
- ($|V|$,1): One index per user u with the single member u; we call this design *user indexes.*
- ($|V|$,a): One index per user u with members F_u; we call this design *friends indexes.*

3.2 The Access Axis

Our second axis is motivated by the observation that we can enforce access control by storing explicitly which user has authored a document. We do this by creating *author lists* which contain pairs of *authorings*, where an authoring is a pair of (document identifier d, user identifier u), indicating that user u has authored document d. For a query by user u, an author list may allow us to filter the results from an inverted index: For each posting, we can check whether the document it occurs in was authored by a friend of u.

The design of author lists is our second axis, and it beautifully mirrors the design of the index axis. We can again define the notion of a *group-author list* as an author list containing all the authorings of a subset of users, and we can define an *access design* analogously as a set of group-author lists. Even the notions of cardinality and redundancy of a design carry over.

Analogously to index designs, there is a very large number of possible access designs. We again only explore the space of extreme points: We only consider cardinality 1 and $|V|$, and we only consider redundancy 1 and a, which is the average number of followers of a user. This gives rise to the following four access designs written as (cardinality, redundancy):

- (1,1): A single group-author list stores all the documents; we call this design *global-list.*
- (1,a): This solution is obviously suboptimal, and we will not consider it further.
- ($|V|$,1): One group-author list per user u with the single member u; we call this design *user-lists.*
- ($|V|$,a): One group-author list per user u with members F_u; we call this design *friends-lists.*

We would like to emphasize the nice symmetry between index design and access design; this gives a very clean characterization of the design space that we explore in our experimental evaluation.

3.3 Query Processing

Given an index design and an access design, we now explain how these two can be combined to answer a query submitted by a particular user u. We first select a set of group-indexes from the index design so that all friends of u are members of at least one group-index. If the set of group-indexes only contains postings from F_u and no other users, we can answer the query directly, and do not need to worry about the access design.

If the set of selected group-indexes have members $v \notin F_u$, we choose a set of group-author lists from the access design so that all friends F_u are members of at least one group-author list. If this selected set of group-author lists contains only members $v \in F_u$, then we only need to intersect the authorings in the group-author lists with the results of the query on the group-indexes on the document identifiers. If the group-author lists contain members $w \notin F_u$, then for each authoring, we need to check whether the author is a friend of u; this check can be performed in a lookup structure that stores all friendships in the graph.

4 Implementation

We have implemented a main-memory search engine in Java that supports the index and access designs outlined in Section 3. Postings resulting from a new document are accumulated in an updatable memory structure where postings are compressed using VByte [10], and become searchable immediately. The accumulated postings are routinely combined with the rest of the data in a hierarchy based on geometric partitioning [8]. More advanced techniques exist [7, 9], but they would not change the tradeoffs that we show. We use PForDelta [15] for compression, which has shown efficient decompression performance in recent studies [13]. For each index, we maintain a dictionary that allows us to look up term identifiers and retrieve the location and length of their posting lists.

Query processing is supported through a set of basic operators in addition to a *filter* operator used to filter the results of an intersection between a set of group-author lists and posting lists based on a lookup to test friendships. Together, these operators support the general query processing strategy outlined in Section 3.3. All operators work like Volcano-style iterators [6]. Their API has two methods: The GET NEXT method returns the next result from the operator while the SKIP method forwards to the next result with a given minimum value.

5 Experiments

We have performed a thorough experimental evaluation with the system that we described in the previous section. Table 1 shows our hardware specifications.

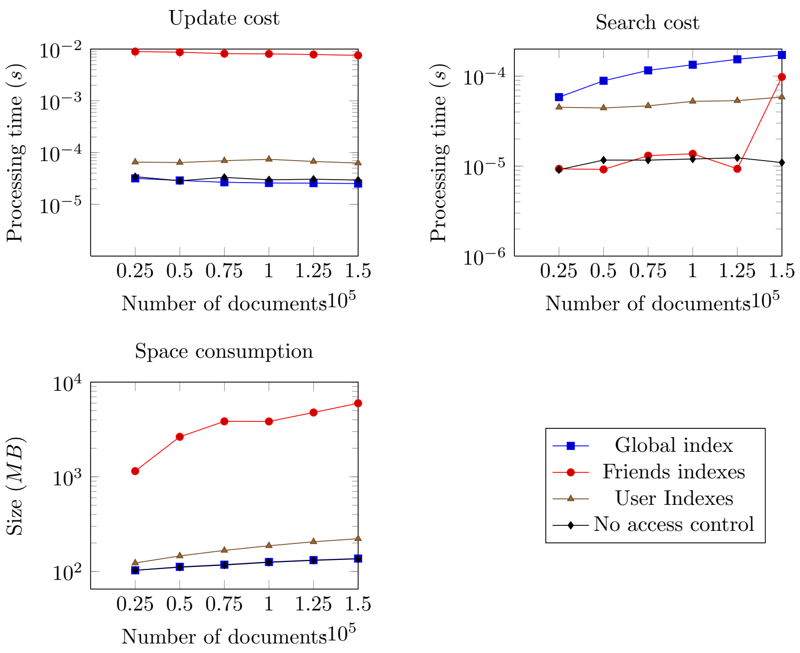


Figure 1: Time spent per update and search with different index designs (1,000 users, 20 friends per user)

CPU	Intel Xeon (3.2 GHz)
Cache size	6 MB
Memory size	16 GB
Java version	1.6
OS	Red Hat Enterprise 5.3

Table 1: Experiment environment

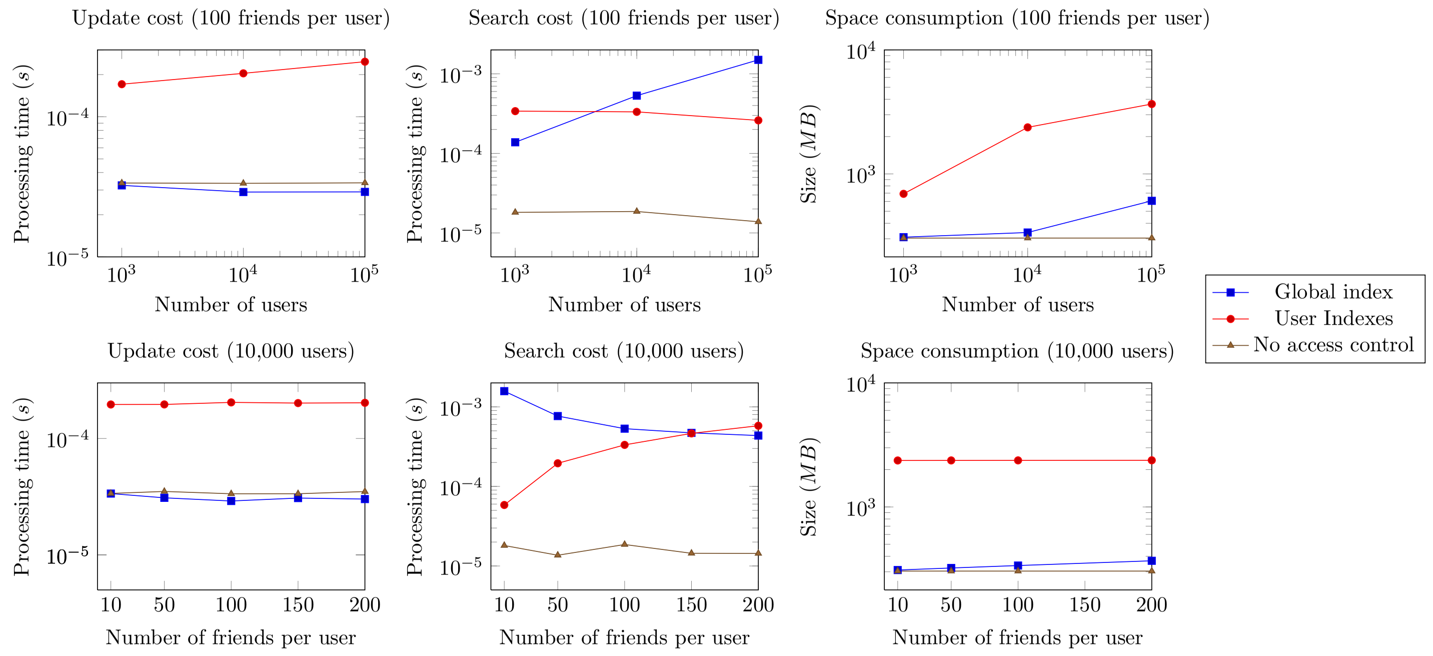


Figure 2: Results from varying network characteristics with different index designs (1,000,000 documents)

5.1 Data

We explore the design space with workloads that are described in terms of networks, documents, and queries. We chose to first load all the documents before processing any queries, and measure the time it takes to load a document or to process a query separately; thus our results are easier to interpret than results for a mixed workload of queries and updates.

Networks. Our experiments use both synthetic networks and a real network from Twitter. The Twitter network was obtained by crawling roughly 417,000 users in February 2010. The synthetic networks of varying size and connectivity were generated using Barabasi's preferential attachment model [2].

Documents. All documents in our experiments were obtained from a crawl of Twitter. For the synthetic network we assigned Twitter documents to users by giving each user a posting frequency of a random Twitter user, and we assigned documents to users accordingly. This preserves the overall distribution of posting frequencies but not any correlation of frequencies and network structure. For the real network we retrieved the 200 most recent documents of the 417,000 users, and we then selected the 5 million most recent ones.

Queries. Since we do not have access to real search logs we generated synthetic query workloads as follows. For each search, we picked a random user u who searched for a random term occurring in the documents of u's friends. We process 100,000 queries returning the 100 top ranked results in all workloads.

Measurements. In all experiments, we report the average query and update times in addition to the space consumption. The space consumption is measured by collecting a dump of the complete memory heap when the index is loaded and garbage collection has been run. The reported results will therefore also include memory used in all configurations regardless of index or access design.

To explore both axes of the design space, we will first consider different index designs (Section 5.2), and then we focus on different access designs (Section 5.3).

5.2 Index Designs

In this section, we compare the performance of user indexes, friends indexes and the global index. The global index is combined with the global-list to enforce access control, while the other two strategies do not require an access design to enforce access control because all users can find a set of group-indexes that have exactly their friends as members. We also include a configuration with a global index without any access design as a baseline. Although this approach does not enforce access control, it enables us to illustrate the overhead of enforcing access control.

5.2.1 Scalability with Number of documents

The first experiment is based on a synthetic network with 1,000 users and 20 friends per user. The size of this network is so small to ensure that all index designs can handle it.

We vary the number of documents between 25,000 and 150,000. The results are shown in Figure 1.

The search efficiency for friends indexes is clearly attractive and comparable to not enforcing access control because no unnecessary data is read or processed. User indexes, on the other hand, require combining results from 20 indexes during each search in this experiment, which makes them slower than friends indexes. The global index is even slower because many lookups to test friendship might be required before the results are found. When the number of documents increases, more and more documents will contain the search terms and more and more lookups to test whether the user who submitted the query is friends with the authors are then required.

While the friends indexes seem to be the most attractive design with respect to search performance, their huge size reduces performance with 150,000 documents due to swapping from memory to disk. The large size of friends indexes comes from the redundancy of indexing a document once for each of the author's followers. In addition to the increased number of postings resulting from the redundancy, the total size of the dictionaries also increases significantly because terms with a single occurrence will have one dictionary entry for each of the author's followers, while common terms will have one for each user. User indexes also suffer from larger dictionary sizes overall compared to a global index, because common terms will be represented in the index for many users, although the effect is small for the network size used in this experiment.

5.2.2 Varying Network Characteristics

The number of documents is not the only factor that determines performance, and we therefore conducted a set of experiments where we vary the network characteristics. After having established that friends indexes do not scale even to very small networks of 1000 nodes we focus on the remaining methods and move to larger input sizes. We vary the number of users in the network and the number of friends per user in two separate experiments. The number of documents is kept constant at 1,000,000. The results are shown in Figure 2.

The update cost for user indexes increases with increasing number of users, because the combined dictionary size is larger, which is also reflected in the index size. Although the update cost for the global index is constant regardless of network characteristics, the size of the lookup structures to test friendship increases with larger networks and with it the total space consumption.

When the fraction of documents that are accessible to the user who submitted the query increases, the search cost for the global index with global-list decreases because fewer look-ups are required before 100 results are found. The opposite effect is seen for user indexes; as the number of friends who have authored documents containing the search terms increases, their search cost also increases.

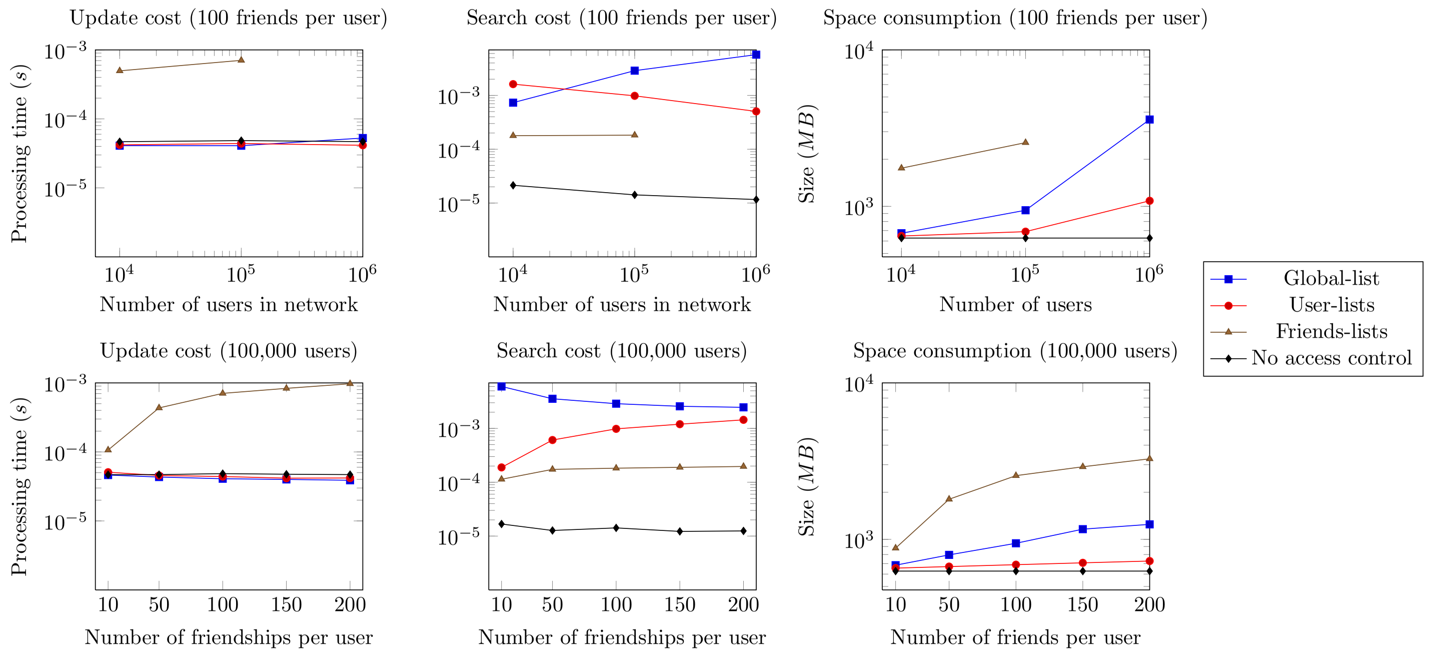


Figure 3: Results from varying network characteristics with different access designs (2,500,000 documents)

5.3 Access Designs

In this section we study the performance of the three access designs introduced previously. We test them in combination with the global index since neither user indexes nor friends indexes require an access design. Our conclusions apply more generally to any index design that relies on access designs to enforce access control. We also include a method without any access design for reference. The designs are tested with different workloads. We have excluded a workload with varying number of documents for access designs due to space restrictions, but experiment with varying network characteristics as we did for index designs, and also include a workload based on a real network.

5.3.1 Synthetic Networks

The experiments with synthetic networks are similar to the experiments with varying network characteristics for index designs except that the number of documents is kept constant at 2,500,000 and that we increase the number of users. The results are shown in Figure 3.

The update cost for all methods except friends-lists is comparable because the cost of updating the meta-data associated with access control contributes only a small fraction of the overall cost ($< 10\%$). For friends-lists, however, the index size grows linearly with the number of friendships in the network (creating an overhead for update costs of more than 2000% for 100,000 users with 100 friends per user) until we run out of space for 1,000,000 users. The space usage with the global-list is significantly higher than with user-lists for large networks because of the overhead associated with the lookup structure to test friendships. This structure is not required during query processing with the other access designs.

Friends-lists have an attractive search performance that does not depend on the size or the connectivity of the network. Filtering with friends-lists can still be 1500% slower than having no access control at all because instead of only scanning the 100 most recent postings in a posting list, we potentially have to skip forward in the lists many times. However, both user-lists and global-list are even slower and their performance depends on the network characteristics. Searches with user-lists become slower when the number of friends of the searching user increases because more group-author lists must be combined. The search cost with the global-list, on the other hand, decreases as the fraction of accessible documents increases because fewer postings must be processed to find the top 100 results.

5.3.2 Real Networks

The results from experiments with access designs on data from a real network are shown in Figure 4. The network used in the experiments contains roughly 417,000 users with 178 friends on average, and we generally observe the exact same tradeoffs as in the synthetic networks with similar sizes. Because the number of documents is scaled up in this experiment, each update is slightly slower due to more expensive merges in the hierarchy of indexes as the index size grows. In this particular network, the average

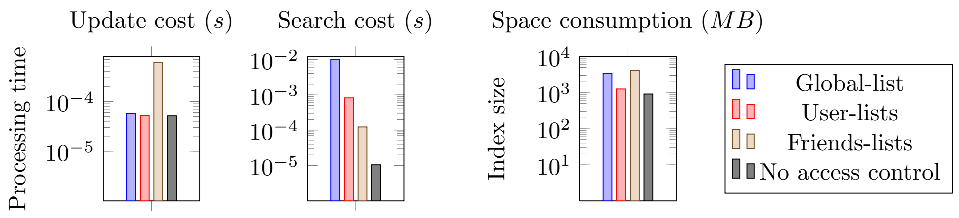


Figure 4: Results for real network

number of followers for the author of each document is slightly lower than in our synthetic networks with the same size, which makes updates for friends-lists slightly faster here relative to the other approaches. The search cost is also affected by the increased number of documents, because each query will on average have more results.

5.4 Discussion

Our experiments show that our designs represent different tradeoffs between index size, update and search performance. The designs along the two axes reveal similar tradeoffs which is due to the symmetry between the axes. However, the differences between the solutions for different index designs are generally much larger than between the solutions for different access design.

Based on our experiments, we believe that user indexes, or a global index with either friends-lists or user-lists as the access design is the best of the basic solutions in real-world scenarios. We do not recommend to use friends indexes because of their scalability problems. The choice between user indexes, or a global index with either user-lists or friends-lists as the access design should be made dependent on the expected workload of updates and searches, the network structure and possible space constraints.

6 Related work

For related work on access control models for structured data we refer the reader to the excellent survey by Bertino et al. [3].

The problem of enforcing access control occurs in both desktop and enterprise search systems, and a straight-forward solution is to create a separate index for each user, indexing only the documents accessible to that user. In a social network we introduced this design as friends indexes. Related work has shown that this approach does not scale to a large number of users because of the redundancy resulting from shared documents, a fact that is confirmed by our experiments. Several alternatives have been suggested [4, 11]. Singh et al. propose to group files based on their access permissions, so that all files in a group are accessible to the same users [11]. Each group is indexed and each user forwards a search to a specified set of indexes. When a social network determines the access permissions, the approach would typically yield an index design close to user indexes, because most users have a unique set of followers. Singh et al. also describe how the set of indexes can be reduced when introducing redundancy, and they also mention that it is possible to filter out results, just like we do with different access designs. However, they do not consider these solutions relevant in their usage scenarios and do not explore them. Büttcher and Clarke suggest to enforce access control by filtering results from a global index while making sure that ranking statistics are based solely on information accessible to the user conducting the search [4]. The exact filtering strategy is not described in detail. Bailey et al. describe how filtering can be integrated into an overall enterprise architecture [1]. Our work can be seen as an extension that explores different access designs which can be interpreted as different filtering strategies, and our experiments indicate that the choice of access design has a significant impact on performance.

Zerr et al. consider security attacks on enterprise search architectures where a certain fraction of the servers is compromised, and propose a system that limits the amount of information leakage in such scenarios [12], an orthogonal problem to what we consider in this paper.

7 Conclusions and Future Work

In this paper, we have taken the first steps towards addressing the problem of search when a social network determines the access permissions. To do so, we have considered keyword search over documents in the social network, and we outlined a symmetric design space consisting of index designs and access designs. Our experiments with several basic solutions indicate that user indexes or global indexes with either user-lists or friends-lists as access designs are the most promising solutions in real-world scenarios.

We believe that the problem we have addressed in this paper only scratches the surface of an important research area for future work. The design space of both axes is wide open and requires further exploration beyond the basic strategies we have evaluated in our experiments. Identifying the best strategy for a particular workload is an important direction of future work.

Challenges on the systems-side include the scalability to large networks and adaptability to their dynamic structural changes. A search system for a large network will probably have to be distributed in order to scale, and partitioning users and their data across machines is an open problem.

Extensions to support more advanced ranking functions are important, possibly including functions that take the actual social network structure into account (see [5] for an example). With more sophisticated ranking functions, it is important to avoid that the ranking reveals protected information to users [4, 11].

The overall problem of search in social network goes well beyond keyword search. Documents are not the only entities that could be searchable in a social network, and extensions to allow queries over structured data in a social network is another avenue of future research.

Acknowledgments. This material is based upon work supported by the New York State Foundation for Science, Technology, and Innovation under Agreement C050061, by the National Science Foundation under Grants 0121175 and 0534404, and by the iAd Project funded by the Research Council of Norway. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] P. Bailey, D. Hawking, and B. Matson. Secure search in enterprise webs: tradeoffs in efficient implementation for document level security. In *Proc. CIKM*, 2006.

[2] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439), 1999.

[3] E. Bertino, S. Jajodia, and P. Samarati. Database security: research and practice. *Inf. Syst.*, 20(7), 1995.

[4] S. Büttcher and C. L. A. Clarke. A security model for full-text file system search in multi-user environments. In *Proc. FAST*, 2005.

[5] D. Carmel, N. Zwerdling, I. Guy, S. Ofek-Koifman, N. Har'el, I. Ronen, E. Uziel, S. Yogev, and S. Chernov. Personalized social search based on the user's social network. In *Proc. CIKM*, 2009.

[6] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1), 1994.

[7] S. Gurajada and S. Kumar P. On-line index maintenance using horizontal partitioning. In *Proc. CIKM*, 2009.

[8] N. Lester, A. Moffat, and J. Zobel. Fast on-line index construction by geometric partitioning. In *Proc. CIKM*, 2005.

[9] G. Margaritis and S. V. Anastasiadis. Low-cost management of inverted files for online full-text search. In *Proc. CIKM*, 2009.

[10] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.

[11] A. Singh, M. Srivatsa, and L. Liu. Efficient and secure search of enterprise file systems. In *Proc. ICWS*, 2007.

[12] S. Zerr, E. Demidova, D. Olmedilla, W. Nejdl, M. Winslett, and S. Mitra. Zerber: r-confidential indexing for distributed documents. In *Proc. EDBT*, 2008.

[13] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, 2008.

[14] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[15] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, 2006.

Appendix A.4

Paper IV

Workload-Aware Indexing for Keyword Search in Social Networks

Truls A. Bjørklund, Michaela Götz, Johannes Gehrke and Nils Grimsmo

Version without proofs submitted to Special Interest Group on Management Of Data Conference (SIGMOD)
2011

Abstract

More and more data is accumulated inside social networks. Keyword search provides a simple interface for exploring this content. However, a lot of the content is private, and a search system must enforce the privacy settings of the social network.

In this paper, we present a workload-aware keyword search system with access control based on a social network. We make two technical contributions: (1) HeapUnion, a novel union operator that improves processing of search queries with access control by up to a factor of two compared to the best previous solution; and (2) highly accurate cost models that vary in sophistication and accuracy. These cost models provide input to an optimization algorithm that selects the most efficient organization of access control meta-data for a given workload. Our experimental results with real and synthetic data show that our approach outperforms previous work by up to a factor of three.

1 Introduction

More and more data is accumulated inside social networks where users tweet, update their status, chat, post photos, comment on each other's lives, get updates through news feeds, and search for information. Examples of such social interaction platforms include Facebook, Twitter, LinkedIn, YouTube and Flickr. From a user's perspective, some of her content may be private and should only be accessible to a selected set of users in the network. To limit arbitrary information flow, social networks enable users to adjust their privacy settings at a fine granularity; e.g., to ensure that only friends can see the content they have posted. Thus any component that enables access to data in the social network *must* adhere to the privacy settings in place.

Search over collections of documents is a well-studied problem [36]. However, when supporting search over content in a social network, new opportunities and challenges arise. A document in a social network may be considered as consisting of two types of properties: *document-centric properties* and *network-centric properties*. The document-centric properties consist of the document and its metadata, for example the time when it was posted, the terms in the document, or properties of the user who posted the document. The network-centric properties consist of additional information added by other users, such as comments, tags, or ratings. The ranking of a document for a given search query could thus be based on both the document-centric and the network-centric part, where properties such as the relationship between the user who tagged a document and the user who submitted the query can be taken into account [1, 27].

A challenge associated with search in social networks is to enforce the access restrictions that are determined by the network structure. In the remainder of this paper, we will assume that a user can access all of her documents and the documents of her friends,[1] and the challenge of supporting efficient keyword search while enforcing these restrictions is the topic of this paper. This is a hard problem since nearly every user has access to a unique subset of the documents, and since the resulting solution needs to scale both with the number of documents and the number of users. We address this problem by materializing special-purpose metadata called *author-lists*, which are lists of identifiers for all documents authored by a set of users. However, with this solution arise two major challenges: First, queries may now need to access a large number of author-lists in order to determine the set of documents that a user has access to. Efficiently processing search queries while scaling with the number of author-lists is essential for an efficient solution. Second, we need to select which author-lists to materialize to process a workload efficiently. We explore this space of possible solutions with cost models. Our experiments show that highly accurate cost models are required to find the best solutions. However, accurate cost models for query costs have received very limited attention in the literature on search as compared to the literature in database systems, probably because the space of possible query plans for a search query is usually small compared to the space for a query in a database system. Thus a second challenge is the development of highly accurate cost models for a search workload.

When enforcing access control, it is important to avoid using global statistics about the

[1]Our techniques are easily extended to other cases such as Facebook Groups.

document collection in the ranking functions, because this will reveal information about the content of inaccessible documents to users who submit queries [12]. In this paper we therefore focus on ranking functions based on simple document-centric properties such as the time when the document was posted or the user who posted the document. This provides an important step towards full support for search in social networks with access control for two reasons: First, there are situations where it is natural to rank documents only on simple properties and thus the techniques proposed in this paper solve the whole problem. An example is the search for tweets on Twitter where recency is a well-understood and reasonable ranking function. Second, the technical contributions of this paper can be used as building blocks for a solution that uses more general ranking functions. All solutions need to enforce access control, and query processing operators that efficiently process access control structures are therefore important. Furthermore, accurate cost models are important components for finding an efficient organization of the access control meta-data.

The main technical contributions of our work can be summarized as follows:

- We introduce a new operator called HeapUnion which efficiently allows us to process a large number of author-lists while skipping over irrelevant documents in each of them. Compared to previous approaches, it improves the query processing time by up to a factor of two. (Section 4)
- To the best of our knowledge, this is the first paper to provide highly accurate cost models of the query operators in a keyword search system. We also describe how our cost models interact with the solution space of the problem of selecting the most efficient author-list design for a particular workload. (Section 5.)
- We show the results of a thorough experimental evaluation of all of our techniques with real and synthetic data. (Sections 4, 5 and 6)

We discuss related work in Section 7 and conclude in Section 8. We now continue with a more formal description of the problem we address in Section 2, and Section 3 describes our experimental setup.

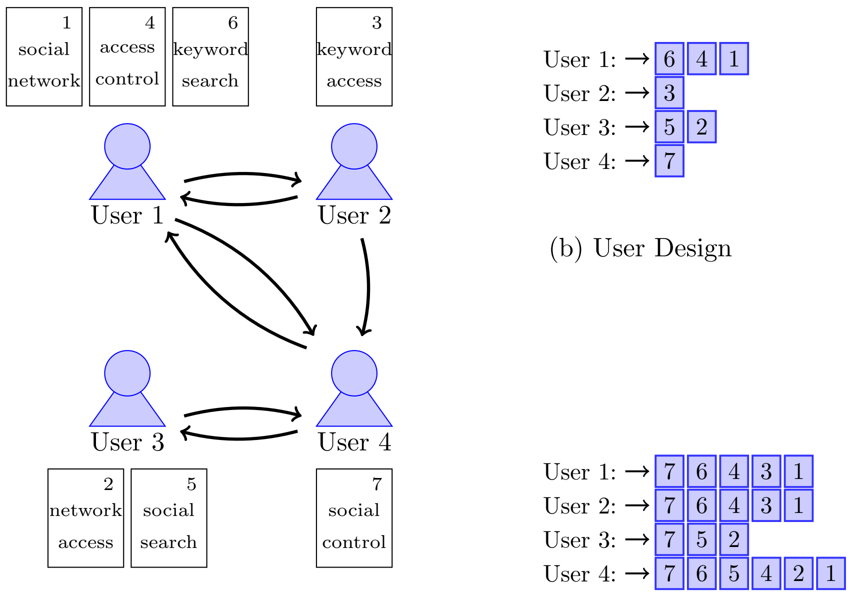
2 Problem Definition

In this section, we will introduce some notation which is used to define the problem addressed in this paper.

2.1 Data and Query Model

We view a social network as a directed graph $\langle V, E\rangle$, where each node $u \in V$ represents a user. There is an edge $\langle u, v\rangle \in E$ if user v is a friend of user u, denoted $v \in F_u$, or alternatively that u is one of v's followers, denoted $u \in O_v$. We always have $u \in F_u$ and $u \in O_u$.

We consider workloads that consist of two different operations: posting new documents and issuing queries. A new document, which we also refer to as an *update*, consists of a set



(a) Example social network with posted documents

(b) User Design

(c) Friends Design

Figure 1: Social Network and Basic Designs

of terms. We will call the user who posted document d the *author* of d, and we will also say that the user *authored* d. The new document gets assigned a unique document ID; more recently posted documents have higher IDs. Let n_u denote the number of documents authored by user u, and let $N = \sum_u n_u$ denote the total number of documents in the system.

A query submitted by a user u consists of a set of keywords. As mentioned in the previous section, we assume that only documents authored by users in F_u are accessible to u. For the remainder of this paper, we will also assume that the ranking is based on recency, with newer documents ranked higher than older documents. Thus the results of a keyword query are the k documents that (1) contain the query keywords, (2) are authored by a user in F_u, and (3) have the k highest document IDs among the documents that satisfy (1) and (2). Facebook currently supports these queries (with an artificial limit to posts of the last 30 days).[2] However, the technical details are proprietary.

Figure 1(a) shows an example of a social network with four users, where User 4 is friends with Users 1 and 3, and User 2 is friends with Users 1 and 4. All the users have posted documents, and each document has an ID which is shown in its top right corner. User 3 has posted Documents 2 and 5, and in our model she can search across Documents

[2] `http://blog.facebook.com/blog.php?post=115469877130`

2, 5, and 7.

2.2 User and Friends Designs

Bjørklund et al. have developed a conceptual framework for solutions to this problem that we build upon [11]. They have characterized solutions along two axes; the index axis and the access axis. The *index axis* captures the idea that instead of creating one single inverted index over all the content in the social network, it is possible to create several inverted indexes, each containing a subset of the content. A set of inverted indexes and their content is called an *index design*. The *access axis* mirrors the index axis and describes the meta-data used to filter out inaccessible results; the meta-data is organized into *author-lists*. As described in the introduction, an author-list contains the IDs of all documents authored by a set of users. An *access design* describes a set of author-lists.

Previous work experimented with a few extreme points in this solution space; it showed that two of the most promising solutions both use an index design with a single index containing all users' documents, while the access design in the two approaches differ. The first approach is called *user design* and has one author-list per user that contains the document IDs posted by that particular user. The second approach is called *friends design*; it also has one author-list per user, but this author-list contains the documents posted by the user and all of her friends. The author-lists for the user and friends designs for our example from Figure 1(a) are shown in Figures 1(b) and 1(c), respectively. In both of these designs, a keyword query from a user is processed in the single inverted index. To enforce access control, the results from the index are intersected with a set of author-lists containing all friends of the user. In the friends design, all friends of the user are represented in the author-list for the user, whereas in the user design, we need to calculate the union of the author-lists for all friends.

2.3 Beyond User and Friends Designs

The relative merits of the user and friends designs motivate the work in this paper. Note that in the user design, updates are efficient because only u's author-list is updated when u posts a document; queries, however, need to access the author-lists of all users in F_u. In the friends design, queries are more efficient because only the author-list of u is accessed. Updates, however, need to change the author-lists of all users in O_u.

In our new approach, we start out with the user design. In addition, we add one additional author-list l_u for each user u; l_u contains the IDs of all documents authored by a selected set of users $L_u \subseteq F_u$. When user u submits a query, there is no need to access specific author-lists for users in L_u, and queries therefore become more efficient as more users are represented in L_u. On the other hand, representing more users in L_u also leads to higher update costs. We therefore determine the contents of L_u (and thus l_u) based on the workload characteristics.

3 Experimental Setup

Before we dive into the technical contributions of this paper, we will describe the setup for our experiments. This setup is used to evaluate both our novel query processing operator HeapUnion and our cost models and the resulting optimized access designs in the following sections.

Indexing System. An updatable keyword search system is usually implemented with a hierarchy of indexes [22, 13, 19, 24]. New data is accumulated in a small updatable structure that also supports concurrent queries, while the main part of the hierarchy consists of a set of read-only indexes. The read-only indexes are formed by merging a set of smaller read-only indexes, and will subsequently be used to answer queries. Documents will be merged into larger and larger indexes over time, and the largest read-only indexes will thus contain the least recent documents. Such an index hierarchy is well suited for search in social networks, especially when used in combination with an access design that adapts to the workload. The time at which indexes are merged represents an opportunity to modify the access design and adapt it to the current workload, so that different indexes in the hierarchy potentially have different access designs. When ranking documents based on recency as we do in this paper, the largest indexes in the hierarchy (which contain the oldest documents) will probably be accessed less frequently than the smaller indexes, and using different access designs among the indexes in a hierarchy might be very beneficial in such settings.

All individual indexes in the hierarchy except the small updatable part process *stratified* workloads because the index is constructed before it is used to answer queries. Because the stratified workloads therefore dominate in the index system, we focus on stratified workloads in this paper. We experiment with a system that constructs an index for a set of documents, and then processes search queries with the index. Our system is main-memory based and accumulates an index for a batch of documents at a time in a structure where the lists are compressed using VByte [28]. The batches are combined in the end to form the complete index, where the lists are compressed using PForDelta [37, 35]. How queries are processed in the resulting index is described in Section 4.

Workloads. In our experiments we are using a set of workloads that are based on a crawl of a subset of Twitter from February 2010. The first workload is based on the actual crawled Twitter network and is denoted Workload Real. The workload consists of 417,156 users with 74,326,889 unique friendships, and their 2,500,000 most recently posted documents. Due to a restriction in the Twitter API, none of the users have more than 200 posted documents in this workload. We have also generated two workloads with synthetic networks, Workload 1 and Workload 2, to enable to test our solutions with varying social network characteristics. Workload 1 has 10,000 users and Workload 2 has 100,000. In both networks, users have 100 friends each and the friendships are generated with the widely used preferential attachment model [6]. Documents in both workloads were obtained from the Twitter crawl. In both workloads, each user is assigned a posting frequency from a Twitter user, and the documents are assigned to users according to the resulting distribution. In Workload 1, we assign 1,500,000 documents to the users and there is a strong correlation between the number of posted documents and the number of followers, while the 2,500,000 documents in Workload 2 are assigned to users without such a correlation.

The workloads also involve search queries. As we do not have access to actual search logs for the crawled data, we generated the search queries based on the actual document collection. The query terms were selected by removing all the stop words in the collection, and then choose a random remaining term. We thus select query terms based on term frequency except for stop words. We use two different strategies for selecting the user who submits the search query. We either use a uniform distribution such that each user is selected with equal probability, or a Zipfian distribution with exponent 1.5 such that a few users are selected much more frequently than others. In our plots, the workloads using the Zipfian distribution have ”Z” as a suffix. Different complete workloads are generated by combining the 3 basic workloads above with different numbers of queries. We return the top-100 results for all queries unless explicitly stated otherwise.

Hardware. All experiments were run on a computer with an Intel Xeon 3.2 GHz CPU with 6 MB cache and 16 GB main memory running RedHat Enterprise 5.3; our system is implemented in Java, and we ran Java 1.6.

4 HeapUnion

In this section, we describe how query processing with a large number of author-lists works in our system. The explanations assume single-term queries, but this is only to simplify our presentation and does not reflect a limitation of our system.

4.1 Query Processing

Our search system answers queries by computing the intersection of a posting list p_t for a term t with a union of author lists $a_1 \cup \cdots \cup a_m$.[3] A template for the resulting query plan is shown in Figure 2. It uses three operators (intersection, union, and list iterator) that all support the following interface:

- *Init*(): Initialize the operator;
- *Current*(): Retrieve the current result;
- *Next*(): Forward to the next result and retrieve it;
- *SkipTo*(*val*): Forward to the next result $\leq$ *val*.

All results are returned in sorted order based on descending document IDs to facilitate efficient ranking on recency, and the query plan follows a document-at-a-time processing strategy [33]. The top-k ranked results are therefore found by calling *Next*() on the intersection operator k times. We use a standard intersection operator that alternates between the inputs and skips forward to find a value that occurs in both [18]. With this solution, the *SkipTo*(*val*) operation in the union operator will be called repeatedly, and thus its implementation is essential to the overall processing efficiency. We will therefore

[3]Similar types of queries occur in several scenarios, e.g., in star joins in data warehouses [26].

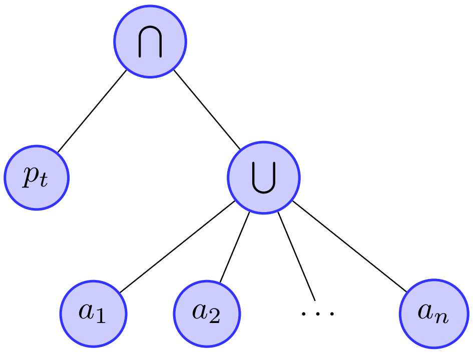


Figure 2: Query Template

focus on the union operator in the remainder of this section. For completeness, the implementation of the other operators in our system is described in Appendix A.

There exist two straight-forward implementations of the union operator, called Eager Merge and No Merge, respectively [26]. Eager Merge merges all inputs to the union first, and then supports skipping in the intermediate merged list. The initial merge is the dominating cost when using Eager Merge. Assuming that a standard multi-way merge strategy is used and that there are R entries in total in all input lists, the worst-case total merge cost is $\Theta(R \log m)$. No Merge, on the other hand, processes a skip operation on the union by performing a skip in each input and returning the maximum result. The cost of one skip operation with this implementation is equal to the skip cost for each input plus $\Theta(m)$ to find the correct return value. Eager Merge and No Merge are thus preferable in different situations: If there are many skip operations compared to the total number of entries in the inputs, Eager Merge is preferable. On the other hand, No Merge is preferable when the number of skip operations is small compared to the input sizes.

Raman et al. have introduced a union operator called Lazy Merge, which is based on the idea that if the number of skip operations in the union is large compared to the length of an input, it would have been ideal to pre-merge this input into a set of intermediate results. Lazy Merge adaptively merges an input into the intermediate results when the number of skip operations exceeds the length of the input times a constant α. Raman et al. show that Lazy Merge never uses more than twice the processing time of a solution that pre-merges the optimal set of inputs [26]. However, the approach has some drawbacks for our workloads. First, the analysis by Raman et al. does not take the actual cost of merges into account. The actual running time can therefore be quite unattractive when the union has many inputs, as we will see in Section 4.3. Second, we usually process top-k queries and therefore only need the first k results from the intersection. It is inefficient to merge a set of complete inputs when only a small fraction of the results are used during query processing. It can also be considered a disadvantage that the constant α needs to be determined in order to use Lazy Merge. To address these issues, we develop a union operator called HeapUnion which is described next.

4.2 The HeapUnion Operator

HeapUnion, our novel union operator, is designed to be efficient regardless of whether all or only a fraction of the results are actually needed, and to scale gracefully to a very large number of inputs regardless of the characteristics of the skip operations. To achieve these goals, HeapUnion is based on a binary heap. The heap contains one entry for each input operator, and it is ordered based on the value obtained from calling $Current()$ on each input operator (referred to as the current value for the input operator), just as in a standard multi-way merge strategy.

We support the standard operator interface by always having the input with the highest current value at the top of the heap, so that this value is also the current value for HeapUnion. The heap is initialized the first time *Next*() or *SkipTo*(*val*) is called. When the first call is a *Next*() (*SkipTo*(*val*) resp.), HeapUnion calls *Next*() (*SkipTo*(*val*)) on all inputs, and the heap is constructed using Floyd's Algorithm [16]. Floyd's algorithm calls a sub-procedure called *heapify*() which can be used to construct a legal heap from an entry with two legal sub-heaps as children. The *heapify*() operation has logarithmic worst-case complexity in the size of the heap, and Floyd's Algorithm runs in linear time [16]. We will also use these operations during heap maintenance.

After initialization, HeapUnion works as shown in the pseudo code in Algorithm 1. The *Current*() operation either returns the current value from the input operator at the top of the heap, or it indicates that there are no more results if the heap is empty. The *Next*() operation forwards the input with the current value, and calls *heapify*() to ensure that the input with the new highest current value is at the top of the heap. The worst-case complexity of this operation is thus logarithmic in the number of input operators.

The *SkipTo*(*val*) operation is based on a breadth-first search (BFS) in the heap. When forwarding to a value *val*, only the inputs with a current value $>$ *val* actually need to be forwarded. Because the heap is organized according to the current value for all inputs, we know that if a given input has a current value $\leq$ *val*, the same is true for all of its descendants. If we determine that no skip is necessary in a given input, we thus also know that no skip is required in any of its children in the heap, and there is no need to process the children in the BFS. Furthermore, if an input is not forwarded, we know that its position in the heap relative to its children will not change. We also take advantage of this observation by calling *heapify*() only for the inputs where an actual skip occurred, and use a complete run of Floyd's algorithm only in the worst-case.

We will now compare the worst-case performance of HeapUnion with the worst-case of the basic strategies presented above. First, we provide a bound on the combined complexity of all skip operations in one HeapUnion:

Lemma 1. *Assuming that the cost of skipping forward s entries in an input has cost in $O(s)$ and that each skip operation on HeapUnion forwards at least one input, the total cost of all skip operations for a HeapUnion is in $O(R \log m)$.*

Proof. Forwarding an input results in heap maintenance costs that are always $O(\log m)$, except for in the single initialization step where they are $O(m)$. Because it is possible to forward an input $O(R)$ times in total, the combined heap maintenance costs are $O(m + R \log m) = O(R \log m)$.

Algorithm 1 HeapUnion Operator

```
function Init():
    Allocate heap

function Next():
    heap[0].Next()
    heapify(0)
    return Current()

function SkipTo(int Val):
    Perform a breadth-first search in the heap from the root
    while BFS queue is not empty:
        if Current iterator is forwarded in SkipTo(val):
            Add to LIFO-list of entries for heap reorganization
            Add children to BFS queue
    Call heapify() for forwarded inputs in LIFO-list
    return Current()

function Current():
    if size(heap) == 0:
        return EOF
    else:
        return heap[0].Current()
```

In all but the first call to skip in HeapUnion, the BFS will ensure that when m_s inputs are actually forwarded, there will be at most $m_s + 1 = O(m_s)$ checks of whether additional inputs should be forwarded. Because there may be $O(R)$ input forwards in total, the total cost of testing whether inputs should be forwarded is $O(R)$. Furthermore, by assumption, the total cost of actually forwarding the inputs is $O(R)$. This makes the combined cost of all skips in HeapUnion $O(R \log m)$. □

As a consequence of Lemma 1, the worst-case cost for all skip operations in HeapUnion is no worse than the worst-case cost of all skips in Eager Merge because the initial merge in Eager Merge has worst-case complexity $\Theta(R \log m)$. Furthermore, the following lemma follows immediately from the fact that the heap maintenance reduces to Floyd's algorithm in the worst-case and that Floyd's algorithm is $O(m)$ for m heap entries.

Lemma 2. *The heap maintenance cost in one skip operation in HeapUnion is $O(m)$.*

As a consequence of Lemma 2, the cost of a specific skip operation with HeapUnion is comparable to the cost of the same skip operation with No Merge: In both methods, the same skips will occur on the inputs, and No Merge may potentially try to skip on more inputs that are not forwarded compared to HeapUnion (due to the BFS). Furthermore, the cost of finding the largest return value with No Merge is $\Theta(m)$, and the cost of heap maintenance in HeapUnion is $O(m)$. The worst-case cost of one skip operation in HeapUnion is therefore not worse than with No Merge. HeapUnion will thus achieve the best of both worlds: When there are only a few skip operations compared to the lengths of the inputs, it is an advantage that its worst-case performance for each operation is as good as with No Merge. And, when there are many skip operations, it is an advantage that its worst-case performance is as good as with Eager Merge. In addition, HeapUnion does not pre-merge any inputs, and retrieving only a fraction of the results is therefore supported efficiently. Compared to Lazy Merge it also has the advantage that no configuration parameters are required. We will now present a set of experiments to compare the efficiency of HeapUnion and Lazy Merge.

4.3 Experiments

We compare the relative efficiency of HeapUnion and Lazy Merge by exchanging the HeapUnion operator in our query template with Lazy Merge. Our implementation of Lazy Merge stores the intermediate results in an uncompressed list where skips are implemented with a galloping search [9]. As explained above, the parameter α describes how eager Lazy Merge is at merging inputs into the intermediate results. When setting α to 0, Lazy Merge behaves as Eager Merge, and with $\alpha = \infty$, Lazy Merge behaves as No Merge.

Recall our experimental setup from Section 3. We limit the access design to the user design because it provides a real test of any solutions' ability to process queries with many author-lists efficiently. We report the time spent processing 100,000 queries for each of the workloads, and vary α between 0 and ∞. We have argued that one of the reasons why Lazy Merge is not ideal for our workloads is that we typically process top-k queries. To isolate this effect, we experiment both with only returning the top-100 results and with returning all results.

The results from the experiments are shown in Figure 3. Notice that HeapUnion does not depend on α, and its cost is therefore constant. When using Lazy Merge, the difference between the cost of top-100 queries and retrieving all results increases with the size of α. This reflects the inadequacy of approaches that pre-merge when processing top-k queries. The processing time of HeapUnion is clearly dependent on the number of retrieved results, and HeapUnion is therefore an attractive solution for top-k queries as expected.

Lazy Merge performs best with α set to extreme values. Poor performance in other cases is often caused by the large number of merges resulting from many inputs with different lengths. In what end of the scale the best α value is found for a particular workload depends on the average length of the author-lists compared to the posting list. HeapUnion outperforms the best configurations of Lazy Merge in all workloads with a speed-up between 1.12 and 2.36, reflecting that HeapUnion is efficient regardless of workload characteristics.

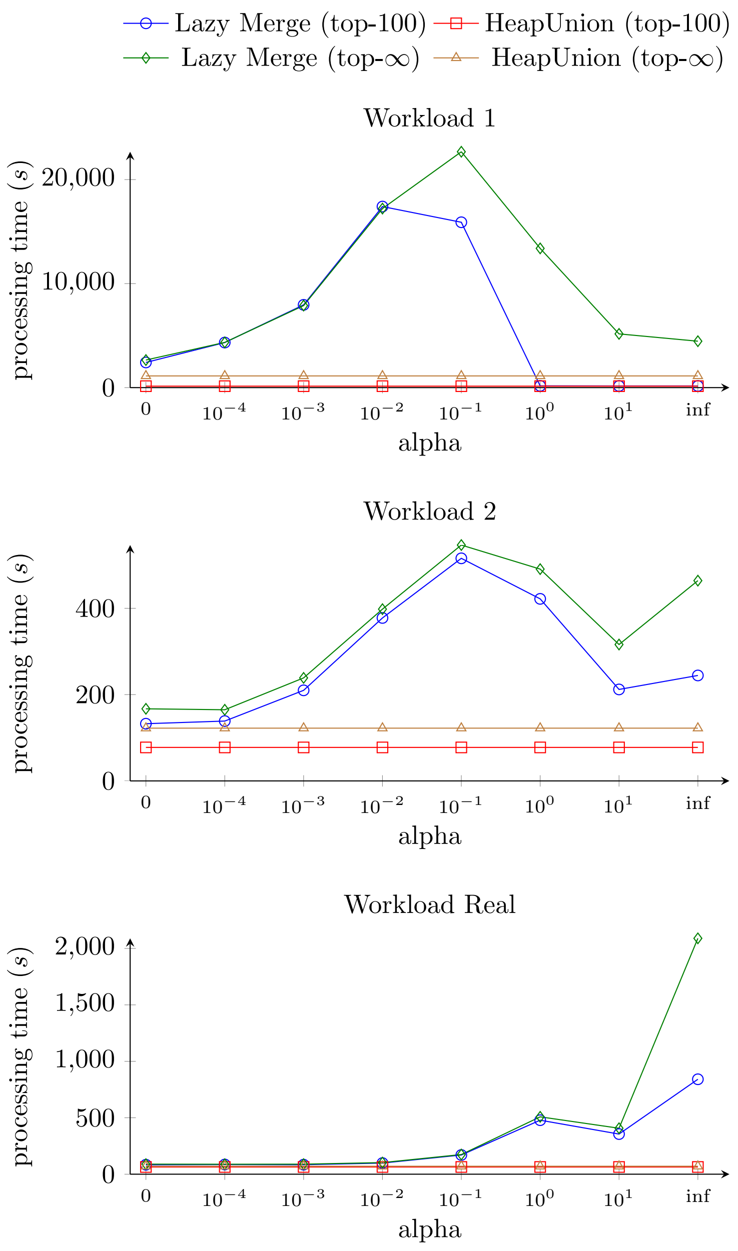


Figure 3: HeapUnion vs. Lazy Merge varying α.

5 Cost Models and Optimization

The efficiency of our system for a particular workload depends on the contents of the additional author-lists, and selecting a good set of lists is therefore essential. For each user u, any subset of F_u can be included in L_u, which leads to $2^{\sum_{u \in V} |F_u|} = 2^{|E|}$ possible designs. We use cost models to explore this large space. In information retrieval, cost models have traditionally been used to explain and compare the relative merits of different algorithms [34, 13, 14]. In this paper, however, the cost models are used in optimization algorithms to select between different access designs for a stratified workload of updates and queries. Are there any models from the literature that we can use?

5.1 Simple

A simple cost model has recently been developed by Silberstein et al. for the problem of efficiently finding the most recent events in users' event feeds in social networks [29], and it is straight-forward to adapt their model to our problem. The model, which we refer to as *Simple*, assumes that the cost of processing a query is linear in the number of accessed author-lists, and that the cost of constructing a list is linear in the number of document IDs in the list. (A formal description of Simple is found in Figure 4.) Simple thus captures the intuition that including more users in the additional author-lists leads to lower query processing costs and higher update costs. Silberstein et al. have shown that when using Simple as a cost model in their optimization problem, a globally optimal solution can be found by making only local decisions for each friend pair, and we could extend their result in a straight-forward way to our problem. This implies that only $\sum_{u \in V} |F_u| = |E|$ different designs have to be explored in order to find the optimum [29].

To test the accuracy of Simple, we want to compare its predictions to the actual running times in our system. We use Workloads 1 and 2 described in Section 3 together with a set of 100,000 queries uniformly distributed among the users as a basis for these experiments. We also need to define a set of access designs to test so that the results will be indicative of to which extent using Simple in an optimization algorithm will lead to efficient access designs. Here we make the observation that the optimization algorithms associated with all our cost models (detailed in the upcoming subsections) will select a limit for each user, u, such that all friends of u who post fewer documents than this limit will be included in L_u. Thus to test Simple, we choose a single *limit* per access design, such that all users will include a friend v in their L_u if the number of documents n_v posted by user v satisfies $n_v < \textit{limit}$. By choosing a set of different values for *limit*, we obtain a range of designs ranging from empty additional author-lists to additional author-lists that include all friends.

Figure 5 compares the actual running times of our system to predictions from Simple. The query cost estimates are clearly far from accurate, and as we will see in Section 6, this inaccuracy can lead to access designs that are up to 67% slower than designs resulting from more accurate models. In the following subsections, we will introduce two more accurate cost models. The first model, *Monotonic*, is introduced in Section 5.2, and even though it is much more accurate than Simple, it still has the nice property that an optimization algorithm must only check a limited number of access designs to find the most efficient solution. *Non-monotonic*, described in Section 5.3, increases accuracy at the expense of an increase in the size of the search space.

5.2 Monotonic

Monotonic is designed to be an accurate yet tractable cost model, were only a small number of access designs must be checked in the optimization algorithm to find a globally optimal solution. The model for updates in Monotonic is the same as in Simple, but we use a different approach to model query costs. Monotonic has one cost model for each operator and the total query costs are estimated by combining the models for all operators in the query. The cost model for an operator describes the cost of each method supported by the operator. For operators that have other operators as inputs, like HeapUnion and Intersection, the cost is calculated by combining the cost of operations within the operator with the cost of method calls on the inputs. To find the cost of the queries we use in this paper, we combine the operators according to the template in Figure 2, and calculate the cost of k *Next*() calls for the Intersection to retrieve the top-k results (assuming there are at least k).

Monotonic is described in Figure 4. *Skip*(s) is a model for *SkipTo*(val). If *SkipTo*(val) forwards the current value of an operator with Δv document IDs, we model the cost of the operation by *Skip*($\frac{r\Delta v}{N}$), where r is the number of results of the operator. The number of results for an operator is the number of times we can call *Next*() and retrieve a new result. Monotonic and Non-monotonic use the same models for List Iterator and Intersection, but they have different models for HeapUnion. We will now explain Monotonic's model for HeapUnion; models for the other operators are explained in Appendix B.[4]

HeapUnion. Let us assume that HeapUnion has m inputs $k_1, \ldots, k_m$. We use r_i to denote the number of results from input i, and define $R = \sum_{i=1}^{m} r_i$. We assume that the cost of initialization within the HeapUnion operator itself is negligible, and therefore model the cost of initialization as the sum of the initialization costs for all inputs.

Recall from Section 4.2 that the first call to either *Next*() or *SkipTo*(val) will involve construction of the heap; we therefore have two different cases in the model for *Next*() and *SkipTo*(val), depending on whether it is the first or a subsequent call. The cost of the first call to *Next*() includes the cost of calling *Next*() on all m inputs, and the cost of the heap construction using Floyd's Algorithm. For heap construction, we model the cost of each call to *heapify*() as being constant, c_h. With Floyd's algorithm, *heapify*() is called for half the heap entries, and then recursively every time there is a reorganization. The average case complexity of Floyd's algorithm is well known, and the number of relocations in the heap is approximately $\gamma m = 0.744m$ [31]. Thus we model the cost of heap construction as $(\gamma + \frac{1}{2}) \cdot m \cdot c_h$.

A first call to *SkipTo*(val) involves skipping in all inputs, in addition to heap construction. Given that s results are skipped in this operation, we simply assume that the

[4]We determined all the constants in the cost models with microbenchmarks as described in Appendix D.

	Update Cost ($\lvert l_u \rvert = n$)	Query Cost (user u)
Simple	$c_{update}n$	$c_{list}\,(\lvert F_u\rvert - \lvert L_u\rvert + 1)$
Monotonic	$c_{update}n$	see below
Non-monotonic	$\begin{cases} c_{1b} * n & \text{if } \frac{N}{n} < b_1 \\ c_{2b} * n & \text{if } b_1 \le \frac{N}{n} < b_2 \\ c_{3b} * n & \text{otherwise} \end{cases}$	see below

	Init()	*Next*()	*Skip*(s)
List Iterator	$\begin{cases} c_{einit} & \text{empty list} \\ c_{init} & \text{otherwise} \end{cases}$	c_{next}	$\begin{cases} c_c + \frac{s}{b}c_d + scan(s)c_{sc} & \text{if } s \le b \\ Skip(b) + c_g \log\left(\frac{s}{b}\right) & \text{otherwise} \end{cases}$
Intersection	$k_1.Init() + k_2.Init()$	$k_1.Next() + (t-1)k_1.Skip(1 + \frac{r_1}{r_2})$ $+t \cdot k_2.Skip(\frac{t-1}{t}(1 + \frac{r_2}{r_1}) + \frac{r_2}{r_1 t})$	not used
Monotonic HeapUnion	$\sum_{i=1}^{m} k_i.Init()$	$\begin{cases} \sum_{i=1}^{m} k_i.Next() + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \frac{r_i}{R} k_i.Next() + (\gamma + 1) \cdot c_h & \text{otherwise} \end{cases}$	$\begin{cases} \sum_{i=1}^{m} k_i.Skip(\frac{r_i s}{R}) + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \min(1, \frac{r_i s}{R}) k_i.Skip(\max(1, \frac{r_i s}{R})) & \\ \quad +(\gamma + 1) \cdot m_s \cdot c_h & \text{otherwise} \end{cases}$
Non-monotonic HeapUnion	$\sum_{i=1}^{m} k_i.Init()$	$\begin{cases} \sum_{i=1}^{m} k_i.Next() + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \frac{r_i}{R}(k_i.Next() + \log(1 + p_i) \cdot c_h) & \text{otherwise} \end{cases}$	$\begin{cases} \sum_{i=1}^{m} k_i.Skip(\frac{r_i s}{R}) + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \min(1, \frac{r_i s}{R}) k_i.Skip(\max(1, \frac{r_i s}{R})) & \\ \quad + \max(m_s\gamma + \min(m_s, \frac{m}{2}), & \\ \quad m_s(\sum_{j=1}^{m} \frac{r_j}{R} \log(1 + p_j)) - h(\lfloor m_s \rfloor)) \cdot c_h & \text{otherwise} \end{cases}$

Figure 4: Overview of Cost Estimates

number of entries skipped in input i is $\frac{r_i s}{R}$, resulting in a cost of $\sum_{i=1}^{m} k_i.\mathit{Skip}(\frac{r_i s}{R})$. The cost of heap construction is modeled as explained for *Next*() above.

Subsequent calls to *Next*() involve a call to *Next*() for the input at the top of the heap and heap reorganization. We estimate the cost of the call to *Next*() for the input at the top as a weighted average over the inputs. The model for the cost of heap maintenance is simple. We assume that there will be γ relocations when a single operator is forwarded, leading to $\gamma + 1$ calls to *heapify*().

Subsequent calls to *SkipTo*(*val*) will potentially forward all inputs, and then reorganize the heap according to the new current values of the inputs. On average, each input will be forwarded past $\frac{r_i s}{R}$ entries. However, HeapUnion will ensure not to call *SkipTo*(*val*) for inputs that will not be forwarded. Therefore, when the average skip length for an input is less than 1, we model the cost as $\frac{r_i s}{R}$ calls that skip 1 entry. To find the cost of heap maintenance we estimate the number of forwarded inputs as: $m_s = \sum_{i=1}^{m} \min(\frac{s r_i}{R}, 1)$. Assuming that there will be as many relocations in the heap as when constructing a heap with m_s entries, the cost of heap maintenance is modeled as $(\gamma + 1) \cdot m_s \cdot c_h$.

Optimization Algorithm. Although Monotonic is much more complex than Simple, it still has the nice property that testing only $|E|$ access designs is sufficient to find the optimal solution. We show this in two steps. First, we show in the following lemma that we can find a globally optimal solution by choosing the contents of the additional author-list for each user individually; it follows directly from the definitions in Figure 4.

Lemma 3. *When using Monotonic as a cost model, the cost of including a user v_1 in L_{u_1} is independent of the cost of including a user v_2 in L_{u_2} for $u_1 \neq u_2$.*

Lemma 3 reduces the number of access designs to test in the optimization algorithm from $2^{\sum_{u \in V} |F_u|}$ to $\sum_{u \in V} 2^{|F_u|}$. The following theorem shows that we can reduce the size of the search space further under certain conditions.

Theorem 4. *If Monotonic estimates that for a user u and a given workload, the performance is improved if $v \in F_u$ is included in L_u, then Monotonic will predict that it leads to a performance improvement to also include user w in L_u if $w \in F_u$, $0 < n_w < n_v$, and $c_d - \frac{c_{sc}}{2b} \geq \frac{c_g}{\ln 2}$.*

The proof of Theorem 4 is found in Appendix C. We notice that in our case, $c_d - \frac{c_{sc}}{2b} \geq \frac{c_g}{\ln 2}$ translates to $0.331 \geq 0.114$, which holds with a significant margin. Theorem 4 implies that if we sort all friends of u based on the number of documents they post, the optimal contents of L_u is a prefix of this sorted list. We thus need to check only $|E|$ designs in total in order to find the optimal solution. Furthermore, notice that there is no cost associated with including users who do not post documents in the additional author-lists, and it is therefore always beneficial to do so.

Validation. We tested Monotonic in the same way we tested Simple; the results are shown in Figure 5. The query cost estimates are much more accurate than the estimates of Simple, but there is still room for improvement when *limit* is close to 0 in Workload 2. Inaccurate modeling of the cost of heap maintenance is one factor that contributes to this error, and we will therefore improve this aspect in our second cost model, Non-monotonic.

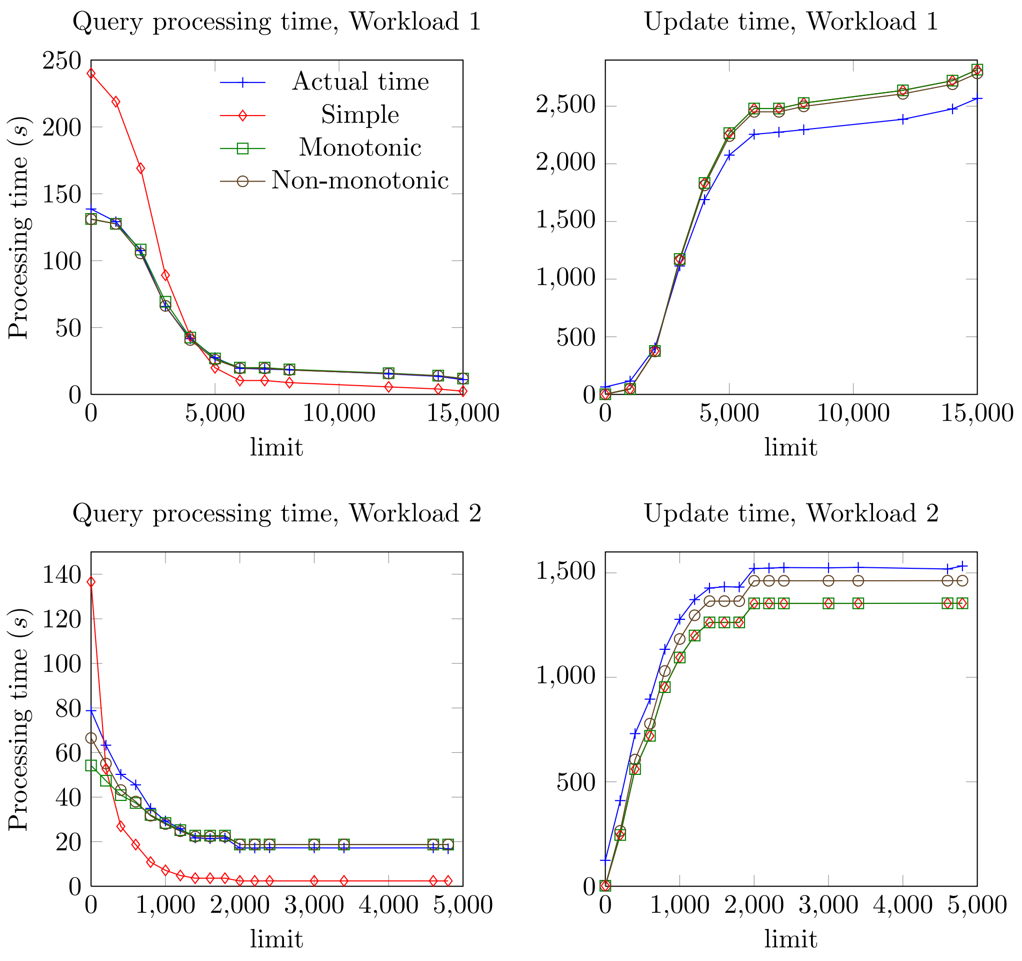


Figure 5: Accuracy of cost models for queries and updates in Workload 1 and Workload 2

5.3 Non-monotonic

Non-monotonic is designed to be more accurate than Monotonic; however, it sacrifices the nice property that only a small number of designs must be checked to find the globally optimal access design. To achieve better accuracy, we improve the model for heap maintenance costs, and we use a slightly more advanced update model. The formal description of Non-monotonic is found in Figure 4.

The update model from Monotonic is extended by taking list compression into account. During accumulation, the lists are compressed using VByte, which implies that lists with few entries result in long deltas that use more space. The model assumes that the cost of updating a list depends on the number of bytes used by VByte to represent the average delta length.

The models for heap maintenance in subsequent calls to *Next*() and *SkipTo*(*val*) reflect that the cost of heap maintenance often depends on the total number of inputs as well as on the number of forwarded inputs. Given that input i is at the top of the heap when *Next*() is called, let p_i denote the number of inputs that will have $Current()$ values larger than input i after the call to *Next*(). We estimate p_i as $\sum_{k=1}^{m} \min(\frac{r_k}{r_i}, 1)$, and assume that the heap maintenance cost when input i is at the top of the heap is $\log(1 + p_i)c_h$. By calculating a weighted average over all inputs, we end up with an average cost of heap maintenance for a *Next*() as shown in Figure 4.

The model for heap maintenance in *SkipTo*(*val*) is slightly more complex, and the maximum of two different estimates is used: (1) The first alternative is similar to the estimate in Monotonic, but incorporates that Floyd's algorithm will never call *heapify*() for more than half the inputs, which yields the following estimate: $(\gamma m_s + \min(m_s, \frac{m}{2}))c_h$. (2) The other alternative reflects that the cost can be logarithmic in the number of inputs when the number of forwarded inputs is low. We have already estimated the average number of calls to *heapify*() when only one input is forwarded in the model for *Next*(), denoted h_{next} in the following. We now assume that all forwarded inputs will lead to h_{next} *heapify*() operations, but compensate for the fact that many of the inputs are not at the top of the heap when *heapify*() is called. The compensation is achieved with the function $h(m_s)$ in Figure 4, which returns the minimum possible total distance from the root to m_s entries in the heap.

Optimization Algorithm. Lemma 3 also holds for Non-monotonic. However, the proof of Theorem 4 does not hold due to the extensions in Non-monotonic. As a result, an optimization algorithm that uses Non-monotonic must test $\sum_{u \in V} 2^{|F_u|}$ access designs to find the optimal approach. In social networks, users have hundreds of friends, and it is therefore not feasible to test such a large space of access designs. We thus choose to limit the space to the same space that the optimization algorithm explores for Monotonic. If Non-monotonic is actually more accurate than Monotonic, the resulting design should be at least as efficient.

Validation. We test Non-monotonic with the same methodology as the other two cost models; the results are shown in Figure 5. The extended update model represents a slight improvement. For query processing costs, Non-monotonic is clearly more accurate than Monotonic for low limits in Workload 2, indicating that an accurate model for heap maintenance actually plays a significant role.

In the next section, we can now finally move to the ultimate reason why we developed the two new cost models: to find good access designs for search with access control in a social network.

6 Workload-Aware Designs

We have tested the accuracy of the different cost models in Section 5, but the key success factor for a cost model is whether using it in an optimization algorithm leads to efficient designs. We therefore conducted a set of experiments to compare the access designs suggested by the different cost models and associated optimization algorithms. To do so, we use the basic workloads in Section 3, and combine them with different numbers of queries. We compare the designs based on Simple, Monotonic and Non-Monotonic to the user design and the friends design.

The results of our experiments are shown in Figure 6, where the first column in the first three lines show the results for workloads with uniformly distributed queries. To get a better view of the relative differences between the methods, the second column shows the performance of the approaches relative to the best of the user design and the friends design.

For the workloads with uniformly distributed queries, Simple often leads to designs that are slower than choosing the best of the user design and the friends design due to the prediction inaccuracies. Compared to Monotonic and Non-monotonic, the designs from Simple are up to 67% slower, a difference that occurs in Workload Real. Monotonic and Non-monotonic generally lead to reasonable designs that are comparable to or faster than the basic approaches. However, for Workload 2, both approaches lead to sub-optimal designs when queries are frequent relative to updates. This reflects the inaccuracies in the estimates for low limits for Workload 2 in Figure 5. However, Non-monotonic clearly results in better designs than Monotonic in this case; the designs from Monotonic are up to 12% slower, so the additional complexity pays off.

The results from Workload 2 with Zipfian queries are shown in the fourth line in Figure 6, denoted Workload 2Z. We have omitted the results for other workloads with Zipfian queries because the same patterns are observed. The results show that our overall solution performs much better compared to the basic designs when there is skew, with a speed-up of up to 3.4. With skew, the optimization problem is simple because a few users submit almost all queries, and all cost models are able to reflect that these users should have additional author-lists with nearly all their friends represented. Non-monotonic is still slightly better than the others, but the difference is small.

In summary, these experiments confirm the benefits of using our more accurate cost models and associated optimization algorithms; we are able to find access control designs that result in significantly better performance than the basic approaches from previous work.

7 Related Work

Due to its clear commercial value there has recently been significant interest in search in social networks. Search engines like Google and Bing already support search over public Twitter posts, and they plan to include private posts as well.[5] Facebook allows users to search their friends' posts. However, the details of these commercial solutions have not been published. Several recent papers also address search in social networks [27, 2, 11]. Some of the papers introduced ranking functions that incorporate different network-centric properties of social networks (see for example [20, 5, 1]). Furthermore, efficient processing techniques for top-k queries with ranking functions based on network-centric

[5]`http://www.networkworldme.com/v1/news.aspx?v=1&nid=3236&sec=netmanagement`

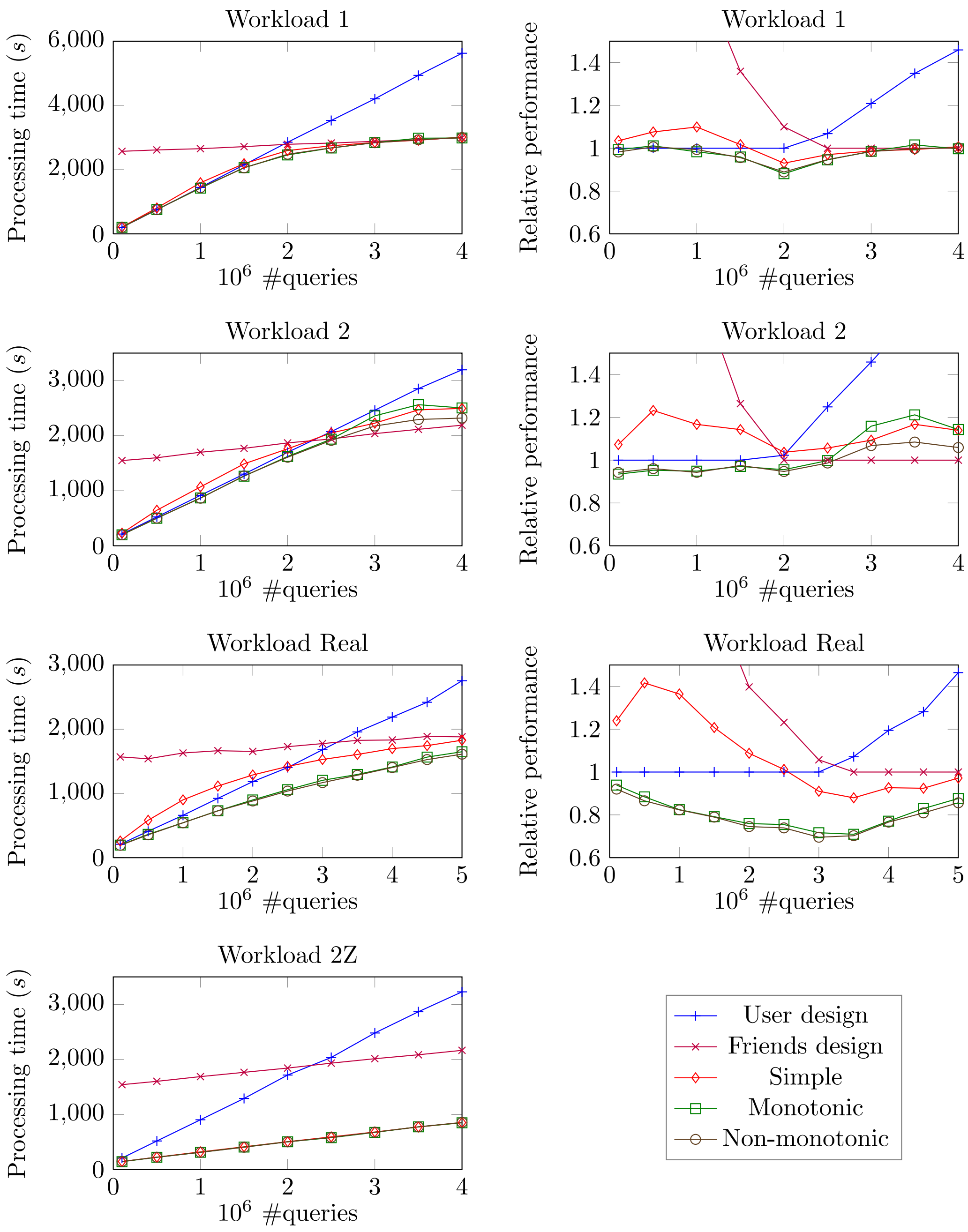


Figure 6: Performance for workloads with different fractions of documents vs. queries

properties have been introduced [2, 27]. However, none of these methods enforce access control because they may return any document tagged by users connected to the user who submitted the query, and this may include inaccessible documents in our model.

The problem we address is related to access control in both information retrieval [12, 30] and for structured data [10]. Our approach is similar to work by Silberstein et al. on retrieving the most recent events in users' event feeds in a social network [29], but our problem is more complex because we support keyword search.

Cost models have been used to estimate the efficiency of processing strategies in both information retrieval and databases [34, 13, 14, 23]. There exist advanced cost models to evaluate different index construction and update strategies [13, 34]. For search queries, however, simple models are most commonly used, sometimes without verification on an actual system [14]. We use simple cost models for updates in this paper, but relatively advanced models are required to predict query processing costs accurately. Manegold et al. outline a generic strategy to estimate the memory access costs in databases [23]. The memory access costs in our advanced models can be considered part of the model for the list iterator. However, our focus is on estimating processing time, whereas Manegold et al. focus on memory access costs.

The problems of calculating unions and particularly intersections of lists have attracted a lot of attention, both through the introduction of new algorithms with theoretical analyses [21, 18, 3, 7], and experimental investigations [8, 4]. The algorithms for single operations are also combined into full query plans [26, 15]. Most algorithms assume that the input sets are uncompressed. Motivated by the ability to store more data and in some cases improve computational efficiency, work from the IR community relaxes this assumption by introducing synchronization points in the compressed lists to speed up random look-ups [17, 25]. In addition, there exists work on algorithms adapted to new hardware trends [32]. Our intersection operator is based on the ideas from Demaine et al. [18], and we use a strategy similar to the one introduced by Culpepper and Moffat for synchronization points in the lists, but we use a novel union operator. Another discriminating factor is that we attempt to estimate the exact running time of the queries through our cost models.

8 Conclusions

In this paper we have presented an efficient system for keyword search in social networks with access control. Through the introduction of accurate cost models and associated workload-aware optimization algorithms, we are able to find designs of access control meta-data that speed up performance by a factor of up to 3.4 over previous work. We also introduced HeapUnion, a novel query processing operator that efficiently supports skipping over unions of sorted inputs. HeapUnion improves query processing efficiency with a factor between 1.12 and 2.36 in our system.

With this foundation in place, we have the basis for extensions to more advanced ranking functions such as ranking based on network-centric properties in social networks [27, 2] while enforcing access control. We may also be able to apply the techniques in this paper in other areas, such as in star joins in data warehouses [26], where HeapUnion might be an interesting query processing operator.

References

[1] S. Amer-Yahia, M. Benedikt, and P. Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2), 2007.

[2] S. Amer-Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *Proc. VLDB Endow.*, 1(1), 2008.

[3] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. CPM*, 2004.

[4] R. Baeza-Yates and A. Salinger. Fast intersection algorithms for sorted sequences. *Algorithms and Applications*, 2010.

[5] S. Bao, G. Xue, X. Wu, Y. Yu, B. Fei, and Z. Su. Optimizing web search using social annotations. In *Proc. WWW*, 2007.

[6] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439), 1999.

[7] J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Algorithms*, 4(1), 2008.

[8] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14, 2009.

[9] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3), 1976.

[10] E. Bertino, S. Jajodia, and P. Samarati. Database security: research and practice. *Inf. Syst.*, 20(7), 1995.

[11] T. A. Bjørklund, M. Götz, and J. Gehrke. Search in social networks with access control. In *Proc. KEYS*, 2010.

[12] S. Büttcher and C. L. A. Clarke. A security model for full-text file system search in multi-user environments. In *Proc. FAST*, 2005.

[13] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *Proc. SIGIR*, 2006.

[14] B B. Cambazoglu, V. Plachouras, and R. Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *Proc. SIGIR*, 2009.

[15] E. Chiniforooshan, A. Farzan, and M. Mirzazadeh. Worst case optimal union-intersection expression evaluation. In *Proc. ICALP*, 2005.

[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001.

[17] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. SPIRE*, 2007.

[18] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proc. SODA*, 2000.

[19] S. Gurajada and S. Kumar P. On-line index maintenance using horizontal partitioning. In *Proc. CIKM*, 2009.

[20] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Information retrieval in folksonomies: Search and ranking. In *Proc. ESWC*, 2006.

[21] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Inf.*, 1(2), 1971.

[22] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM Trans. Database Syst.*, 33(3), 2008.

[23] S. Manegold, P. A. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *Proc. VLDB*, 2002.

[24] G. Margaritis and S. V. Anastasiadis. Low-cost management of inverted files for online full-text search. In *Proc. CIKM*, 2009.

[25] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4), 1996.

[26] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proc. SIGMOD*, 2007.

[27] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *Proc. SIGIR*, 2008.

[28] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.

[29] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: Selectively materializing users. event feeds. In *Proc. SIGMOD*, 2010.

[30] A. Singh, M. Srivatsa, and L. Liu. Efficient and secure search of enterprise file systems. In *Proc. ICWS*, 2007.

[31] R. Sprugnoli. Recurrence relations on heaps. *Algorithmica*, 15(5), 1996.

[32] D. Tsirogiannis, S. Guha, and N. Koudas. Improving the performance of list intersection. *Proc. VLDB Endow.*, 2(1), 2009.

[33] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6), 1995.

[34] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes.* Academic Press, 1999.

[35] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, 2008.

[36] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.

[37] M. Zukowski, S. Heman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *Proc. ICDE*, 2006.

A Query Processing Operators

A.1 List Iterator

The List Iterator is used to iterate through posting lists and author lists in our system. Recall from Section 3 that the lists are compressed using PForDelta. PForDelta is a form of delta compression which is commonly used to compress inverted indexes in search engines [37, 35]. With delta compression, an entry in a sorted list is stored as the delta from the previous entry. Small values are therefore likely, and compression is achieved by using a representation where small values require little space.

To achieve efficient decompression rates, we decompress batches of $b = 128$ values at a time [37]. With this in mind, the implementation of *Init*(), *Current*() and *Next*() is straight-forward. During initialization, we allocate an array to store the batch being processed in an uncompressed format. A standard call to *Next*() will forward the current result to the next in the batch, and every bth call will result in decompression of a new batch.

The use of delta compression complicates the implementation of *SkipTo*(*val*), because previous entries in the list must be decompressed to find the value of an entry; straight-forward random look-ups are therefore impossible. We can implement *SkipTo*(*val*) by calling *Next*() until the returned value is $\leq$ *val*, but that is potentially inefficient when many entries are skipped. We enable more efficient random look-ups by storing the first value of each batch uncompressed in an auxiliary array [17]. A *SkipTo*(*val*) is processed by first checking whether the entry to skip to is located in the current batch. If not, we do a *galloping search* in the auxiliary array to find the correct batch and decompress it [9]. When the correct batch is stored in the intermediate array, we forward within that batch until the correct answer is found.

A.2 Intersection

The intersection operator sorts its inputs according to their expected number of results. The number of results for an operator is the number of times we can call *Next*() and retrieve a new result. Both *Next*() and *SkipTo*(*val*), will begin with the input with fewest results, and call *Next*() or *SkipTo*(*val*), respectively. From then on, these methods are similar. They both alternate between the inputs and try to skip forward to the last value returned from the other input. When the same value is found in both inputs, it is returned as part of the intersection. The outlined processing strategy is similar to the one introduced by Demaine et al. [18].

B Cost Models for Operators

Monotonic and Non-monotonic use the same models for the List Iterator and Intersection operators, and the underlying intuition for these models are presented here. A more formal description is given in Figure 4.

B.1 List Iterator

The list iterator operator is initialized by using a look-up structure to find the list to process. Recall that we decompress lists in batches of b values at a time. An array for the b intermediate results is therefore allocated during initialization, and the first batch of values is decompressed. However, if the look-up indicates that the list is empty, no intermediate array allocation or decompression is required. In conclusion, we model the cost of *Init*() for list iterators with two constants, c_{init} and c_{einit}, depending on whether the list is empty or not.

Every bth call to *Next*() will trigger decompression of a new batch. However, we estimate the average cost of a call to *Next*() to be constant, c_{next}.

The model for the *SkipTo*(*val*)-operation is slightly more complex and depends on the number of entries skipped, denoted s. As described in Appendix A.1, a galloping search is used to find the correct batch to decompress if many entries are skipped, and we will therefore discriminate between cases for when $s > b$ or not. If $s \leq b$, we assume that there is some constant cost associated with a skip operation, c_c. It might be necessary to decompress a new batch of values to find the correct return value. We model the cost of decompressing a segment as a constant, c_d, and the probability that it happens as $\frac{s}{b}$. In addition, we have to forward within the correct batch of values until we find the value we seek. The number of scanned values is lower than s when we need to decompress a new batch, and the average number of entries scanned in our implementation is $scan(s) = 1 + \frac{(2b-1)s}{2b} - \frac{s^2}{2b}$. We assume that scanning an entry also has a constant cost, c_{sc}. When $s > b$, the logarithmic cost of the galloping search is incorporated into the model as $c_g \log\left(\frac{s}{b}\right)$, and the rest of the cost is equal to *Skip*(b).

B.2 Intersection

The initialization within the Intersection operator is assumed to have negligible cost overall, and the cost of *Init*() is the sum of the initialization costs in the two inputs, k_1 and k_2. The inputs are assumed to have r_1 and r_2 results, respectively, with $r_1 < r_2$.

As explained in Appendix A.2, the Intersection operator works similarly for *Next*() and *SkipTo*(*val*)-operations, but we focus on *Next*() in this paper because that is the method we will use. The processing of a *Next*() call will begin with a call to *Next*() on input k_1. Then, there will be a set of skips in each input until we find a value that occurs in both. We assume that deltas between entries in the inputs are $\frac{N}{r_1}$ and $\frac{N}{r_2}$ document IDs, respectively. Furthermore, we assume that the inputs are independent, so that the deltas between results of the Intersection can be estimated as $\frac{N^2}{r_1 r_2}$. The amount skipped in one round of skips (one skip in each input) is estimated to $\frac{N}{r_1} + \frac{N}{r_2}$. We now calculate t, the expected number of rounds with skips, as:

$$t = 1 + \frac{\frac{N^2}{r_1 r_2} - \frac{N}{r_1}}{\frac{N}{r_1} + \frac{N}{r_2}} = 1 + \frac{N - r_2}{r_1 + r_2}$$

Because input 1 will be forwarded with a *Next*() call in the first round, there will be $t-1$ skips in it with average length $\frac{\frac{N}{r_1}+\frac{N}{r_2}}{\frac{N}{r_1}} = 1 + \frac{r_1}{r_2}$. Similarly, there will be t skips in input 2, but one of the skips will, according to our assumptions, arrive at the last value from input 1 which results in a shorter skip in the last round. The average skip length is therefore: $\frac{t-1}{t}(1 + \frac{r_2}{r_1}) + \frac{r_2}{r_1 t}$.

C Proof of Theorem 4

Proof. To see this, we will show that the additional costs from also including user w are never larger per document posted by w than the costs of including v per document posted by v, and that the savings during query processing from also including w are at least as large per posted document as for v.

The cost of updates to l_u is linear in the number of document IDs. The costs of including the documents from v and w in l_u are therefore $c_{update} n_v$ and $c_{update} n_w$, respectively. Per document for each user, the update cost is thus the same, namely c_{update}.

We will now focus on queries. Recall that the queries are processed with the query template in Figure 2. The only things that will change in the query plans for search queries from user u when L_u changes are the inputs to HeapUnion. Notice that what HeapUnion returns for a particular call to *SkipTo*(*val*) or *Next*() will not change, and the processing cost for the intersection operator and the iterator for the posting list is thus unaffected. We therefore only need to show that for the methods for Monotonic HeapUnion in Figure 4, the savings from including w in addition to v in L_u are at least as large per posted document as when only including v, assuming that all inputs to the union are List Iterators. We will go through each of the methods in order.

Init(): When v is included in L_u, the cost will decrease with c_{init} if $|L_u| > 0$ before v was included, because the included list does not require initialization. Otherwise, the cost will remain constant because we have to initialize l_u instead. When w is also included in L_u, the cost will definitely decrease with c_{init} because we know that $|L_u| > 0$ before w was included. Because $\frac{c_{init}}{n_v} < \frac{c_{init}}{n_w}$ when $n_v > n_w$, the savings during *Init*() are at least as large per entry for w as for v.

Next(), first call: From the formula in Figure 4, it is clear that when assuming that $|l_u| > 0$ before v is included, the savings from including v are $c_{next} + (\gamma + \frac{1}{2})c_h$ (it is less if $|l_u| = 0$). The savings from including w in L_u are at least the same (or $c_{next} + 2(\gamma + \frac{1}{2})c_h$ if adding w to L_u makes $L_u = F_u$, in which case no heap is required). By a similar argument as for *Init*(), the savings per n_w for w are at least as large as per n_v for v.

Next(), subsequent calls: Because all inputs to the union are List Iterators, the cost of the call to *Next*() for the input at the top of the heap remains the same when inputs are removed. There will be no savings from the heap maintenance costs either unless including w makes $L_u = F_u$ (when no heap is required). It is therefore straight-forward to conclude that the savings from including w are at least as large per n_w as the savings from including v per n_v.

Skip(s), first call: We will first consider the cost of the calls to skip in the inputs that are processed in this operation. Notice that when including a new user in L_u, the resulting changes in skip costs are: 1) Each skip in l_u will be longer, and the added length depends on the number of documents posted by the new user. And, 2) the skips in the list that is included will no longer be necessary. In the following, we will refer to the cost of performing a skip with length s in a List Iterator as $LI.Skip(s)$. Given that there are $|l_u|$ entries in the additional author-list before v or w is included, the savings from including v are:

$$\begin{aligned} S_v = LI.Skip\left(\frac{sn_v}{R}\right) + LI.Skip\left(\frac{s|l_u|}{R}\right) \\ - LI.Skip\left(\frac{s(|l_u| + n_v)}{R}\right) \end{aligned}$$

The savings from including w as well are:

$$\begin{aligned} S_w = LI.Skip\left(\frac{sn_w}{R}\right) + LI.Skip\left(\frac{s(|l_u| + n_v)}{R}\right) \\ - LI.Skip\left(\frac{s(|l_u| + n_v + n_w)}{R}\right) \end{aligned}$$

We need to show that $\frac{S_v}{n_v} \leq \frac{S_w}{n_w}$. Or, in particular that:

$$\begin{aligned}
&\frac{LI.Skip\left(\frac{sn_w}{R}\right)}{n_w} \\
&- \frac{(LI.Skip\left(\frac{s(|l_u|+n_v+n_w)}{R}\right) - LI.Skip\left(\frac{s(|l_u|+n_v)}{R}\right)}{n_w} \\
&- \frac{LI.Skip\left(\frac{sn_v}{R}\right)}{n_v} \\
&+ \frac{LI.Skip\left(\frac{s(|l_u|+n_v)}{R}\right) - LI.Skip\left(\frac{s|l_u|}{R}\right)}{n_v} \geq 0
\end{aligned}$$

We will now show that $LI.Skip(s)$ is concave, a fact we will use to show the above. First, the second derivative of $LI.Skip(s)$ when $s < b$ is $-\frac{c_{sc}}{b}$, which is negative because c_{sc} and b are both positive constants. When $s > b$, the second derivative is $-\frac{c_g}{s^2 \ln 2}$, which is also negative. Furthermore, $LI.Skip(s)$ is continuous at $s = b$. To show that it is concave, it thus remains to show that the derivative as s approaches b from below is not smaller than when s approaches b from above. By calculating the derivatives and finding the limits, we see that the following must hold:

$$c_d - \frac{c_{sc}}{2b} \geq \frac{c_g}{\ln 2}$$

And this holds by assumption in the theorem. We thus know that $LI.Skip(s)$ is concave.

For a concave function $f(x)$ and two points x_1 and x_2 where $x_1 < x_2$, it follows that the slope between the two points ($\frac{f(x_2)-f(x_1)}{x_2-x_1}$) will never increase when either x_1, x_2 or both increase. We can therefore conclude that

$$\begin{aligned}
&\frac{LI.Skip(\frac{s(|l_u|+n_v)}{R}) - LI.Skip(\frac{s|l_u|}{R})}{n_v} \\
&\geq \frac{(LI.Skip(\frac{s(|l_u|+n_v+n_w)}{R}) - LI.Skip(\frac{s(|l_u|+n_v)}{R})}{n_w}
\end{aligned}$$

It remains to show that $\frac{LI.Skip(\frac{sn_w}{R})}{n_w} \geq \frac{LI.Skip(\frac{sn_v}{R})}{n_v}$. To do so, we return to $f(x)$ as discussed above, and substitute $x_1 = 0$. As long as $f(0) \geq 0$, the average slope between $(0,0)$ and $(x_2, f(x_2)$ is smaller than between $(0,0)$ and $(x_3, f(x_3))$ when $x_3 < x_2$ because $f(x)$ is concave. Because $LI.Skip(0) = c_c + c_{sc}$ is positive as no costs are negative, we can conclude that $\frac{LI.Skip(\frac{sn_w}{R})}{n_w} \geq \frac{LI.Skip(\frac{sn_v}{R})}{n_v}$.

We know that the theorem holds for the heap construction from the description of *Next*(), first call, above.

Skip(s), subsequent calls: The proof for the savings in calls to *SkipTo*(*val*) for the inputs is mostly as for the first call to *Skip*(s) above. The only difference occurs when the skip length for an input is less than 1. We thus need to prove that this resulting function is still concave, and that it is not negative for $s = 0$. Because it is linear when $s < 1$, the second derivative is 0. Furthermore, for the derivative not to increase at $s = 1$, we require that:

$$\begin{aligned} c_c + \frac{c_d}{b} + (1 + \frac{2b-1}{2b} - \frac{1}{2b})c_{sc} \\ \geq \frac{c_d}{b} + (\frac{2b-1}{2b} - \frac{2}{2b})c_{sc} \end{aligned}$$

This holds when:

$$c_c + (1 + \frac{1}{2b})c_{sc} \geq 0$$

This holds because all costs are positive. Furthermore, when the skip length is 0, the cost is 0, so we reach the same conclusion as for *Skip*(s), first call, above.

The model for the cost of heap maintenance in subsequent calls to *SkipTo*(val) is linear in the number of forwarded inputs. Hence, we proceed to show that the number of forwarded inputs decrease at least as much per n_w when w is included in the additional author-list as per n_v when only v is included. The reduction when including v in L_u is:

$$\begin{aligned} R_v = \min\left(\frac{s|l_u|}{R}, 1\right) + \min\left(\frac{sn_v}{R}, 1\right) \\ - \min\left(\frac{s(|l_u| + n_v)}{R}, 1\right) \end{aligned}$$

And, the reduction when also including w is:

$$\begin{aligned} R_w = \min\left(\frac{s(|l_u| + n_v)}{R}, 1\right) + \min\left(\frac{sn_w}{R}, 1\right) \\ - \min\left(\frac{s(|l_u| + n_v + n_w)}{R}, 1\right) \end{aligned}$$

We need to show that $\frac{R_w}{n_w} \geq \frac{R_v}{n_v}$. It is clear that $\frac{\min\left(\frac{sn_v}{R},1\right)}{n_v} \leq \frac{\min\left(\frac{sn_w}{R},1\right)}{n_w}$ when $n_w < n_v$, and it remains to show that:

$$\begin{aligned} \frac{\min\left(\frac{s(|l_u|+n_v)}{R}, 1\right) - \min\left(\frac{s(|l_u|+n_v+n_w)}{R}, 1\right)}{n_w} \\ - \frac{\min\left(\frac{s|l_u|}{R}, 1\right) - \min\left(\frac{s(|l_u|+n_v)}{R}, 1\right)}{n_v} \geq 0 \end{aligned}$$

Assume first that $\frac{s(|l_u|+n_v+n_w)}{R} \leq 1$. In that case, we see that the statement sums to 0 because $|l_u|, n_v, n_w \geq 0$. There are three more possible cases:
a) $s\frac{|l_u|+n_v+n_w}{R} > 1$ and $s\frac{|l_u|+n_v}{R} < 1$,
b) $s\frac{|l_u|+n_v}{R} \geq 1$ and $s\frac{|l_u|}{R} < 1$, and

c) $s\frac{|l_u|}{R} \geq 1$.

In case a), $-\min\left(\frac{s(|l_u|+n_v+n_w)}{R}, 1\right)$, which is negative, is limited so we subtract less relative to when the statement summed to 0, so the statement is positive. In case b), $\min\left(\frac{s(|l_u|+n_v)}{R}, 1\right)$ and $-\min\left(\frac{s(|l_u|+n_v+n_w)}{R}, 1\right)$ cancel. Furthermore, $\min\left(\frac{s|l_u|}{R}, 1\right) < \min\left(\frac{s(|l_u|+n_v)}{R}, 1\right)$, making the statement positive. In case c), the result is 0.

We have now showed that the costs of including w in addition to v in the additional author-list for u are exactly the same per document posted by w, n_w, as the cost of including only v per n_v. Furthermore, we have showed that the savings per posted document are at least as large for all relevant functions in HeapUnion when also including w as when including v in L_u. Because it by assumption leads to a performance improvement to include v, it will thus lead to a further performance improvement to also include w. $\square$

D Microbenchmarks

This section contains an overview of the microbenchmarks used to determine the constants in the cost models in Section 5. All constants are summarized in Table 1.

The costs of initialization and *Next*() operations in list iterators are estimated in an experiment using author-lists with deltas between entries ranging from 1 to 100,000. The results are shown in Figure 7. In a similar experiment using posting lists instead of author-lists, the estimate for c_{init} was higher. This is probably due to that we use a different look-up structure in the inverted index, and we therefore have two values for c_{init}.

The cost of accessing an empty list is estimated with an experiment where we have a user who is friends with x users that do not post documents. We perform 100,000 queries on behalf of this user and record the time per search. The results are shown in Figure 8 along with the estimates.

The cost of skipping in the list iterator involves several constants. We first estimate the scan cost with an experiment that tests all scan lengths up to $b = 128$. The results and the resulting estimates are shown in Figure 8. The rest of the constants for skipping in list iterators are estimated from a set of experiments where skips with various lengths (1 to 10,000) are performed in 1000 lists. All results from these experiments are shown in Figure 9. We use these results to first estimate c_c and c_d by considering all skip lengths below b and subtracting the average scan cost (from the above experiment). The constant for long skips is estimated by subtracting the estimated cost of skipping b values from the longer skips. The results and resulting estimates from these experiments are shown in Figure 10.

The cost of a *heapify*() operation is estimated by running union queries over a set of iterators that simply return numbers with given intervals. We are thus able to predetermine the number of *heapify*() operations.

The constants used in Simple are estimated from a complete workload. We process

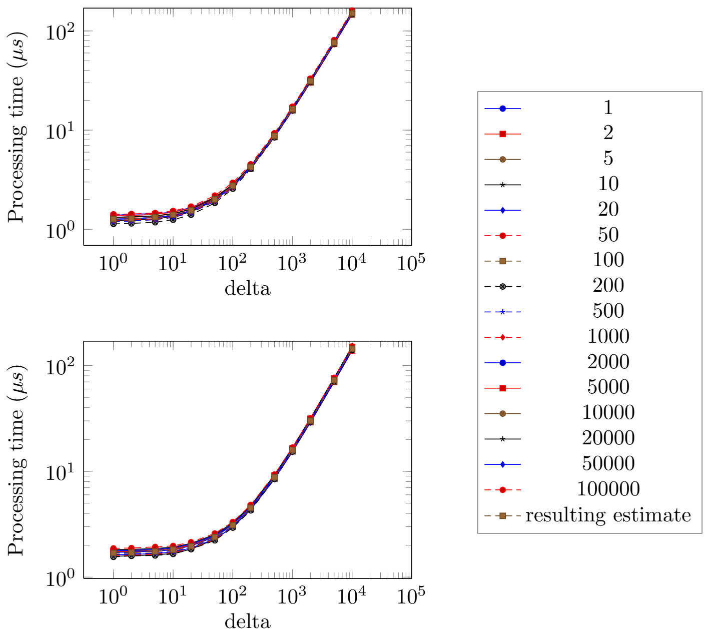
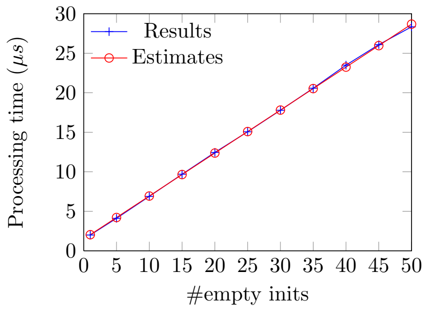


Figure 7: Time spent on x next operations with different deltas. For author-lists (above) estimates are $c_{init} = 1.262$ and $c_{next} = 0.015$. For posting-lists (below) estimate is $c_{initposting} = 1.676$.

Workload 2 from Section 3 and measure update cost as a function of author-list length, and search cost as a function of the number of merged lists in the queries. Results and estimates are shown in Figure 11. To estimate the constants in construction for Non-monotonic, we test a workload with 100,000 lists and accumulate 1,000 entries in each list. The deltas between the entries vary from 1 to 100,000, which results in different numbers of bytes used during accumulation. Results and estimates are shown in Figure 12.

Constant	**Estimate**	**Constant**	**Estimate**
c_{init}	1.262	c_g	0.079
$c_{init-posting}$	1.676	c_h	0.010
c_{einit}	0.544	c_{list}	24.111
c_{next}	0.015	c_{update}	1.002
c_{sc}	0.001	c_{1b}	0.990
c_c	0.025	c_{2b}	1.083
c_d	0.331	c_{3b}	1.136

Table 1: Estimates from microbenchmarks (in μs)



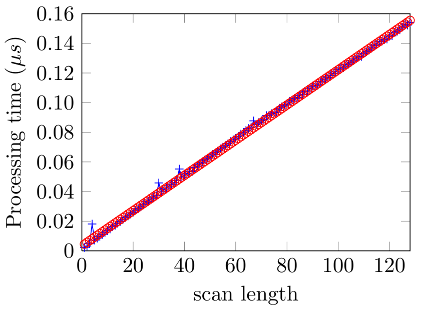


Figure 8: Left: Time spent on x initializations of empty lists $c_{einit} = 0.544$. Right: Cost of scanning x entries in array ($c_{sc} = 0.001$).

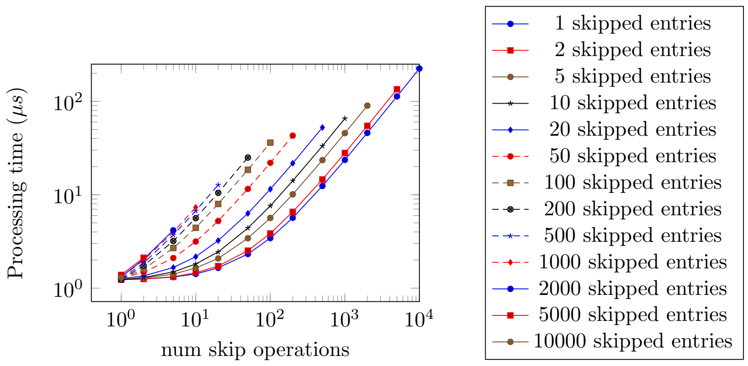
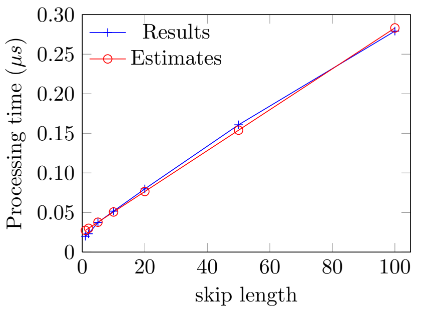


Figure 9: Cost of $SkipTo(val)$ in list iterator



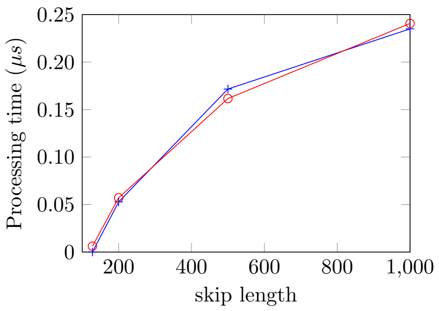
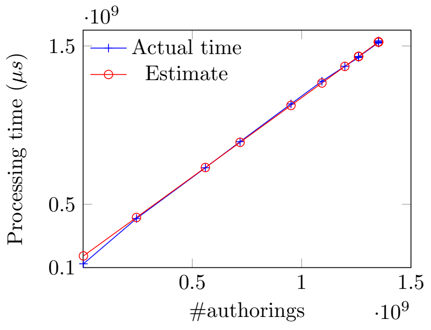


Figure 10: Determination of skip constants for short skips (left: $c_c = 0.025$ and $c_d = 0.331$) and long skips (right: $c_g = 0.079$).



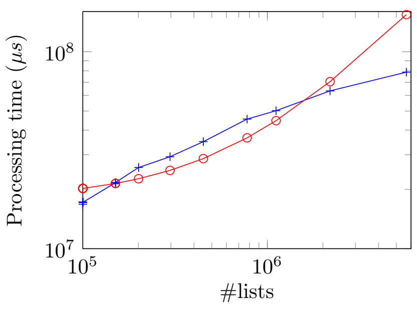
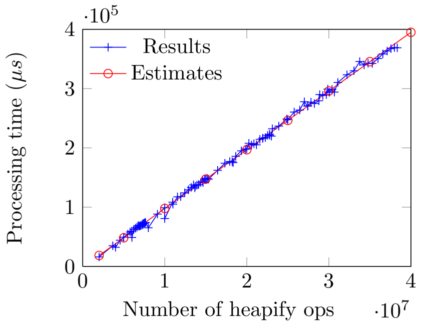


Figure 11: (Left) update workload with x list entries ($c_{update} = 1.002$), (right) search workload with x accessed author-lists ($c_{list} = 24.111$).



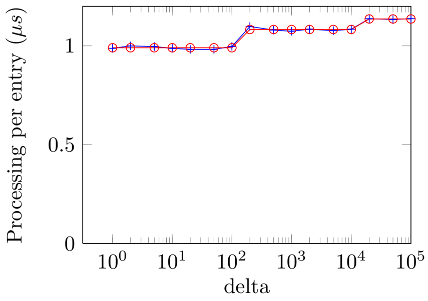


Figure 12: Left: Time per heapify operations ($c_h = 0.010$). Right: Microbenchmark for construction ($c_{1b} = 0.990$, $c_{2b} = 1.083$ and $c_{3b} = 1.136$).

Appendix B

Other Papers

This appendix presents some of the papers I have co-authored during the work with this thesis. The papers address Twig Pattern Matching (TPM), which is a pattern matching problem where both the data and the query are node-labeled trees. A brief introduction to TPM is presented before the actual papers are included.

B.1 Twig Pattern Matching

Twig Pattern Matching (TPM) is a tree matching problem where both the data and query are node-labeled trees. An example data tree based on an example from Paper V is shown in Figure B.1, and Figure B.2 shows an example query tree with three nodes.

A match for a query tree consists of a set of nodes in the data tree where each data node has the same label as the matched query node, and there are structural relationships between the nodes in the data tree that follow the specifications in the query tree. There are two different types of edges in query trees in TPM: a double edge represents an ancestor-descendant (A-D) relationship that in the example in Figure B.2 requires a matching node with label ”a” to have a descendant with label ”b”, while a single edge represents a parent-child (P-C) relationship that in the example requires the matching node with label ”a” to also have a direct child with label ”c”. A match of the query tree in the data tree is shown in Figure B.3.

In summary, the result of a TPM query is the set of mappings of query nodes to data nodes where the mapped nodes have the same labels, and the structure between the mapped data nodes also satisfy the A-D and P-C edges in the query tree. Twig pattern matching is an abstract problem, but it has applications in XML search, as will be explained further in the following subsection. The TPM solutions discussed in Papers V and VI are based on indexing the data nodes and we will give an introduction to a set of different indexing strategies in Section B.2. The indexing strategies are related to techniques from the main

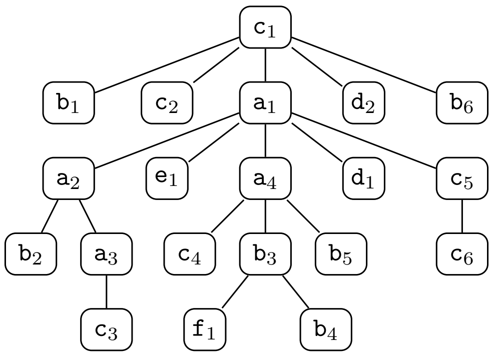


Figure B.1: Example labeled data tree

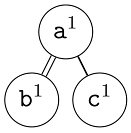


Figure B.2: Example query tree

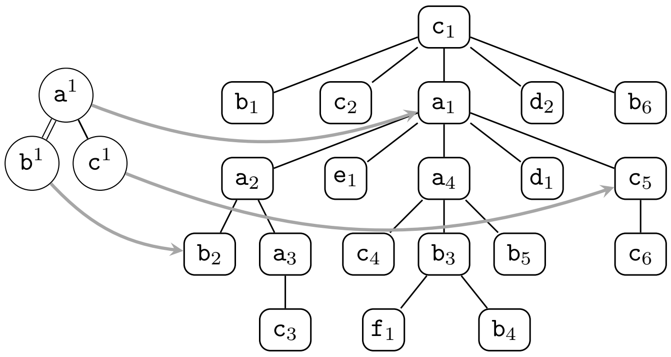


Figure B.3: Match of example query tree in example data tree

```
<library>
   <book ISBN="13">
      <title>Managing Gigabytes</title>
      <author>Witten</author>
      <author>Moffat</author>
      <author>Bell</author>
   </book>
   <book ISBN="14">
      <title>Database Management Systems</title>
      <author>Ramakrishnan</author>
      <author>Gehrke</author>
   </book>
   ...
</library>
```

Figure B.4: Example XML document

part of this thesis, and we will discuss the relationship between these papers and the rest of the thesis in Section B.3. We present an overview of the papers in Section B.4, and the actual papers are included in Section B.5. A more thorough introduction to the topic can be found in Nils Grimsmo's PhD thesis [51].

B.1.1 XML Search

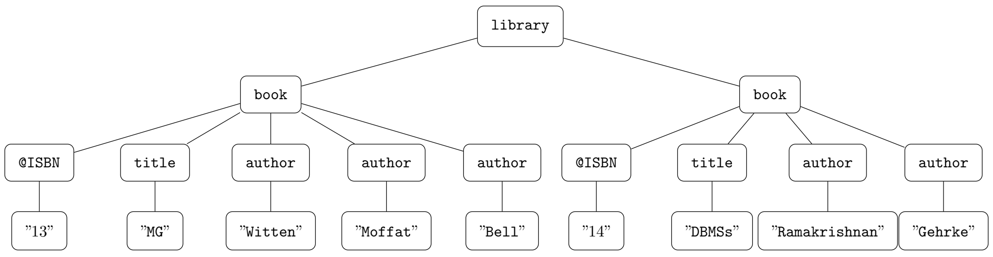
XML [99] has become a de facto standard for storing semi-structured data, and XML documents usually have a tree-structure. An example of an XML document is shown in Figure B.4.

XPath [98] and the slightly more involved XQuery [100] are standard query languages for XML. XPath is a relatively simple declarative language, and XQuery uses XPath as a building block. An XPath query that asks for the title of all books authored by Moffat is shown in Figure B.5. A single forward slash in XPath specifies that the node to the right of the forward slash should be a direct child of the node to the left of the forward slash, while a double forward slash specifies an ancestor-descendant relationship. The square brackets in XPath specifies a predicate, and the right-most node which is not part of a predicate is the output node, i.e., it specifies what the result of the query should be. In the example in Figure B.5, the requirement that Moffat should be an author is a predicate and is therefore specified in brackets, while the title is the output node, and titles are therefore returned as results for this query.

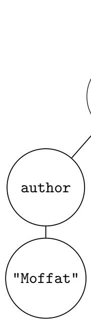
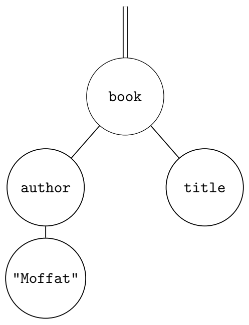
TPM is relevant for XML search because it covers a subset of XPath that represents the majority of the workload in many XML processing systems [50]. One possible translation

```
//book[author/text()="Moffat"]/title
```

Figure B.5: Example XPath query



(a) Data

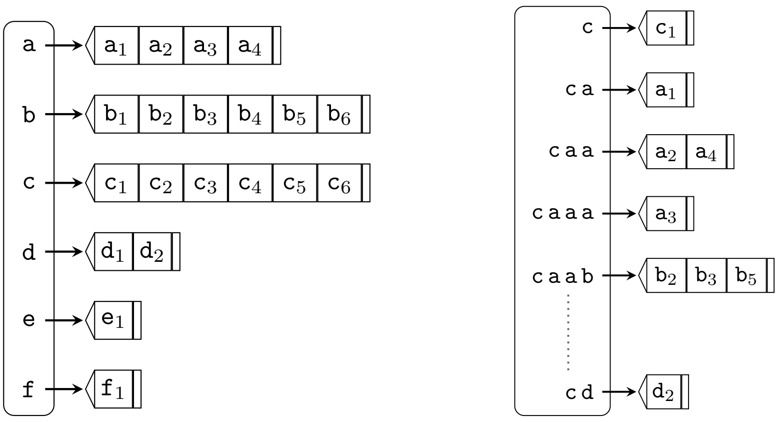
(b) Query

Figure B.6: TPM translations of XML data and query in Figures B.4 and B.5

of the XML data and query in Figures B.4 and B.5 into trees that can be processed with TPM is shown in Figure B.6. Note that a few aspects are not identical in the two versions, e.g., the output node in the query is not specified in the TPM version. However, the overall structure is the same in the two versions, and the simplicity of TPM compared to XPath makes it easier to reason about complexity and to implement prototypes; TPM is therefore often used in academic work.

B.2 Indexing Strategies

Papers V and VI in this thesis address TPM on indexed data. Different types of indexes exist, but they generally support look-ups to find a list of data nodes with some specified features, and the resulting index is similar to an inverted index used in search engines. A simple example of a feature the index can be based on is the label of the node, and a



(a) Indexing on label (b) Indexing on path

Figure B.7: Indexes for the data tree from Figure B.1.

node-label index for the example data tree in Figure B.1 is shown in Figure B.7a.

With an index based on node labels, like the one in Figure B.7a, queries are answered by looking up the labels of all query nodes in the index. The lists of data nodes for all query nodes are then joined into full query matches, an operation that is referred to as a *twig join*. Such joins are the topic of Paper V in this thesis.

With an index based on node labels, all data nodes with a label that is equal to the label of a query node are processed in a straight-forward solution, but many of them are often irrelevant for the query results. An index based on node labels can be interpreted as a grouping of the nodes according to label, but it is possible to extend the number of groups, e.g., by defining that the nodes in each group share the same label and also that the parents of all nodes in the group belong to a specific group [72]. With this recursive definition, we end up with an index where the nodes are partitioned based on the labels on their paths, and an example of such an index for the data tree in Figure B.1 is shown in Figure B.7b.

An advantage of an index based on path labels is that many structural queries can be answered without processing a join at all. Furthermore, many queries will not require reading as much data as with an index based on node labels. However, the look-up structure in the index becomes larger, and several queries will also require merging different lists for a single query node.

The idea of a path index can further be extended so that which group a node belongs to both depends on the group of its parent and of its children [63]. The resulting index is called an F&B Index, and Paper VI in this thesis addresses the efficient construction of such indexes.

B.3 Relationship to Column Stores and Search Engines

Although TPM does not belong to the same topic as the papers in the main part of this thesis, the indexing and processing strategies used in the different fields have clear similarities. Regardless of which of the indexing strategies mentioned in the previous section that are used, the overall index structure is quite similar to an inverted index, as mentioned above. The look-up structure in this case contains the different node groups, while the posting lists contain references to data nodes in the particular group. We have noted the similarities between column stores and search engines in the main part of this thesis, and several XML query processing systems are actually based on column stores, for example MonetDB/XQuery [31]. Twig joins are a type of merge of posting lists, and although the criteria for a match in a twig join is much more complex than in a simple merge of posting lists, it clearly has relations to operations on posting lists in search engines.

Because I had some experience with inverted indexes and column stores, I was involved in the work Nils Grimsmo did on TPM. In the beginning, I contributed in discussions on the implementation of various aspects with clear relationships to inverted indexes and column stores, and I also became involved in other aspects of the work over time.

B.4 Paper Overviews

An overview of the papers I have co-authored on TPM is presented in this section. An abstract of each paper is presented together with a description of the roles of the authors.

B.4.1 Paper V

Fast Optimal Twig Joins

Abstract

In XML search systems twig queries specify predicates on node values and on the structural relationships between nodes, and a key operation is to join individual query node matches into full twig matches. Linear time twig join algorithms exist, but many non-optimal algorithms with better average-case performance have been introduced recently. These use somewhat simpler data structures that are faster in practice, but have exponential worst-case time complexity. In this paper we explore and extend the solution space spanned by previous approaches. We introduce new data structures and improved strategies for filtering out useless data nodes, yielding combinations that are both worst-case optimal and faster in practice. An experimental study shows that our best algorithm outperforms previous approaches by an average factor of three on common benchmarks.

On queries with at least one unselective leaf node, our algorithm can be an order of magnitude faster, and it is never more than 20% slower on any tested benchmark query.

Roles of the authors

Grimsmo did most of the work on this paper. I participated in initial discussions when we generated ideas towards the paper, and in discussions on some of the technical solutions. I also made contributions in the writing phase. Hetland made most contributions on the formal parts of the paper, both on structure and in the writing phase. He also contributed with the idea of how the post-processing for the pre-order algorithm TJStrictPre should be implemented.

B.4.2 Paper VI

Linear Computation of the Simultaneous Forward and Backward Bisimulation for Node-Labeled Trees

Abstract

The F&B-index is used to speed up pattern matching in tree and graph data, and is based on the maximum F&B-bisimulation, which can be computed in loglinear time for graphs. It has been shown that the maximum F-bisimulation can be computed in linear time for DAGs. We build on this result, and introduce a linear algorithm for computing the maximum F&B-bisimulation for tree data. It first computes the maximum F-bisimulation, and then refines this to a maximal B-bisimulation. We prove that the result equals the maximum F&B-bisimulation.

Roles of the authors

As for Paper V, Grimsmo did most of the work on this paper as well. Hetland and I participated in discussions on some of the technical aspects, and in the writing of the paper.

B.5 Papers

Appendix B.5.1

Paper V

Fast Optimal Twig Joins

Nils Grimsmo, Truls A. Bjørklund and Magnus Lie Hetland

Conference on Very Large Data Bases (VLDB)

2010

Abstract

In XML search systems twig queries specify predicates on node values and on the structural relationships between nodes, and a key operation is to join individual query node matches into full twig matches. Linear time twig join algorithms exist, but many non-optimal algorithms with better average-case performance have been introduced recently. These use somewhat simpler data structures that are faster in practice, but have exponential worst-case time complexity. In this paper we explore and extend the solution space spanned by previous approaches. We introduce new data structures and improved strategies for filtering out useless data nodes, yielding combinations that are both worst-case optimal and faster in practice. An experimental study shows that our best algorithm outperforms previous approaches by an average factor of three on common benchmarks. On queries with at least one unselective leaf node, our algorithm can be an order of magnitude faster, and it is never more than 20% slower on any tested benchmark query.

1 Introduction

XML has become the de facto standard for storing and transferring semistructured data due to its simplicity and flexibility [6], with XPath and XQuery as the standard query languages. XML documents have tree structure, where elements (tags) are internal tree nodes, and attributes and text values are leaf nodes. Information may be encoded both in structure and content, and query languages need the expressional power to specify both.

Twig pattern matching (TPM) is an abstract matching problem on trees, which covers a subset of XPath, which again is a subset of XQuery. TPM is important because it represents the majority of the workload in XML search systems [6]. Both data and queries (twigs) in TPM are node-labeled trees, with no distinction between node types. Figure 1 shows a twig query and data with a match. A match is a mapping of query nodes to data nodes that respects labels and the ancestor-descendant (A–D) and parent-child (P–C) relationships specified by the query edges, respectively represented by double and single lines in figures here.

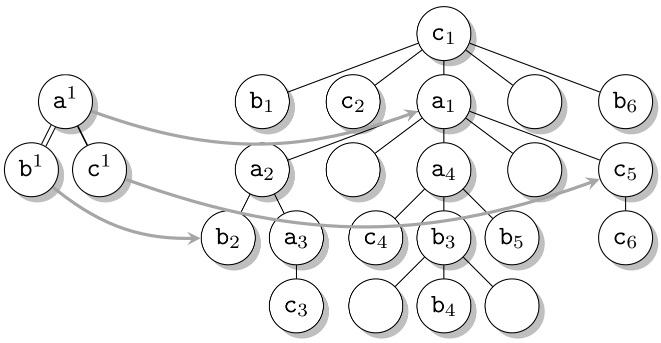


Figure 1: Twig query and data with matches.

Twig joins are algorithms for evaluating TPM queries on indexed data, where the index typically has one list of data nodes for each label. A query is evaluated by reading the label-matching data nodes for each query node, and combining these into full query matches. There exist algorithms that perform twig joins in worst-case optimal time [3], but current non-optimal algorithms that use simpler data structures are faster in practice [10, 11].

In this paper we present twig join algorithms that achieve worst-case optimality without sacrificing practical performance. Our main contributions are (*i*) a classification of filtering methods as *weak* or *strict*, and a discussion of how filtering influences practical and worst-case performance; (*ii*) level split vectors, a data structure yielding linear-time result enumeration with almost no practical overhead; (*iii*) getPart, a method for merging input streams that gives additional inexpensive filtering and practical speedup; (*iv*) TJStrictPost and TJStrictPre, worst-case optimal algorithms that unify and extend previous filtering strategies; and (*v*) a thorough experimental comparison of the effects of combining different techniques. Compared to the fastest previous solution, our best algorithm is on average three times as fast, and never more than 20% slower.

The scope of this paper is twig joins reading simple streams from label-partitioned data. See Section 6 for orthogonal related work that introduces other assumptions on how to partition and access the underlying data.

2 Background

A schema-agnostic system for indexing labeled trees usually maintains one list of data nodes per label. Each node is stored with position information that enables checking A–D and P–C relationships in constant time. A common approach is to assign intervals to nodes, such that containment reflects ancestry. Tree depth can then be used determine parenthood [15].

An early approach to twig joins was to evaluate query tree edges separately using binary joins, but when A–D edges are involved, this can give huge intermediate results even when the final set of matches is small [2]. This deficiency led to the introduction of multi-way twig join algorithms. TwigStack [2] can evaluate twig queries without P–C edges in linear time. It only uses memory linear in the maximum depth of the data tree. However, when P–C and A–D edges are mixed, more memory is needed to evaluate queries in linear time [13]. The example in Figure 2 hints at why.

More recent algorithms, which are used as a starting point for our methods, relax the memory requirement to be linear in the size of the input to the join. They follow a general scheme illustrated in Figure 3. The scheme has two phases, where the first phase has two components. The first component merges the stream of data node matches for each query node into a single stream of query and data node pairs. The second component organizes these matches into an intermediate data structure where matched A–D and P–C relationships are registered. This structure is used to enumerate results in the second phase.

The algorithms broadly fall into two categories. So-called top-down and bottom-up algorithms process and store the data nodes in preorder and postorder, respectively, and filter data nodes on matched *prefix paths* and *subtrees* before they are added to

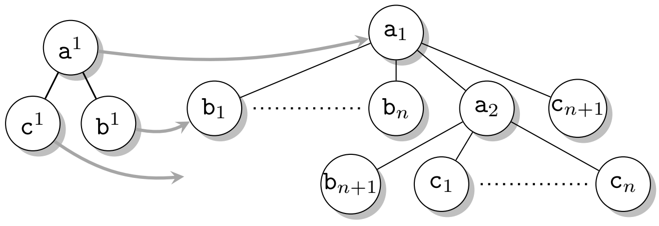


Figure 2: Hard case with restricted memory. It cannot be known whether $\mathtt{b}_1, \ldots, \mathtt{b}_n$ are useful before $\mathtt{c}_{n+1}$ is seen, or whether $\mathtt{c}_1, \ldots, \mathtt{c}_n$ are useful before $\mathtt{b}_{n+1}$ is seen.

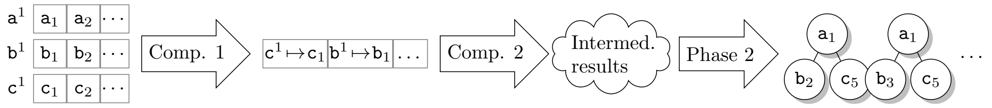
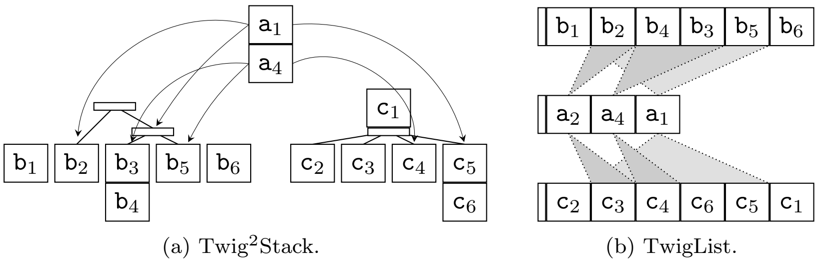


Figure 3: Work-flow of twig join algorithms, with input stream merge component, intermediate result construction component, and result enumeration phase.

the intermediate results. Many algorithms use both types of filtering, which means the processing is a hybrid of top-down and bottom-up.

Twig^{2}Stack [3] was the first linear twig join algorithm. It reorders the input into a single postorder stream to build intermediate results bottom-up. The data nodes matching a query node are stored in a composite data structure (postorder-sorted lists of trees of stacks), as shown in Figure 4a. Matches are added to the intermediate results only if relations to child query nodes are satisfied, and each match has a list of pointers to usable child query node matches.



(a) Twig^{2}Stack. (b) TwigList.

Figure 4: Intermediate result data structures when evaluating the query in Figure 1.

HolisticTwigStack [8] uses similar data structures, but builds intermediate results top-down in preorder, and filters matches on whether there is a usable match for the parent query node. It uses the *getNext* function from the TwigStack algorithm [2] as input stream merge component, which implements an inexpensive weaker form of bottom-up filtering. The combined filtering does not give worst-case optimality, but results in faster average-case evaluation of queries than for Twig^{2}Stack [8].

One approach for improving practical performance is using simpler data structures for intermediate results. TwigList [11] evaluates data nodes in postorder like Twig^{2}Stack, but stores intermediate nodes in simple vectors, and does not differentiate between A–D and P–C relationships in the construction phase. Given a parent and child query node, the descendants of a match for the parent are found in an interval in the child's vector, as shown in Figure 4b. Interval start indexes are set as nodes are pushed onto a global stack in preorder, and end indexes are set as nodes are popped off the global stack in postorder. A node is added to the intermediate results if all descendant intervals are non-empty. Compared to Twig^{2}Stack, this gives weaker filtering and non-linear worst-case performance, but is more efficient in practice [11], according to the authors because of less computational overhead and better spatial locality.

TwigFast [10] is an algorithm that uses data structures similar to those of TwigList, but stores data nodes in preorder. It uses the same preorder filtering as HolisticTwigStack, and inherits postorder filtering from the getNext input stream merge component. There are several algorithms that utilize both types of filtering, but among these TwigFast has the best practical performance [1,3,10]. Figure 5 gives an overview of twig join algorithms, and various properties that are introduced in Section 3.

		Filtering	**Checking of**		**Interm.**	
Algorithm	**Ref.**	**order**	**path**	**subtree**	**results**	**Optimal**
getNext	[2]	GN	none	weak	N/A	N/A
TwigStack	[2]	GN+pre	strict	weak	complex	no
Twig^{2}Stack	[3]	post	none	strict	complex	yes
$\frac{1}{2}$PathStack + T^{2}S	[3]	pre+post	weak	strict	complex	yes
$\frac{1}{2}$TwigStack + T^{2}S	[1]	GN+pre+post	weak	strict	complex	yes
HolisticTwigStack	[8]	GN+pre	weak	weak	complex	no
TwigList	[11]	post	none	weak	vectors	no
TwigMix	[10]	GN+pre+post	weak	weak	vectors	*incorrect*
TwigFast	[10]	GN+pre	weak	weak	vectors	no
TJStrictPost	Sect. 4	pre+post	strict	strict	vectors	yes
TJStrictPre	Sect. 4	GN+pre(+post)	strict	strict	vectors	yes

Figure 5: Previous combinations of prefix path and subtree filtering. Intermediate result storage order given by last item in “filtering order”. GN is the node order returned by the getNext input stream merger.

3 Premises for Performance

To make algorithms that are both fast in practice and worst-case optimal, we need an understanding of how filtering strategies and data structures impact performance.

For any graph G, let $V(G)$ be its node set and $E(G)$ be its edge set. Let a *matching problem* $\mathfrak{M}$ be a triple $\langle Q, D, I \rangle$, where Q is a query tree, D is a data tree, and $I \subseteq V(Q) \times V(D)$ is a relation such that for $q \mapsto q' \in I$ the node label of q equals the node label of q'. Each edge $\langle p, q \rangle \in Q$ has a label $L(\langle p, q \rangle) \in \{$“A–D”, “P–C”$\}$, specifying an ancestor–descendant or parent–child relationship. Let a *node map* for $\mathfrak{M}$ be any function $M \subseteq I$. Assume a given $\mathfrak{M} = \langle Q, D, I \rangle$ when not otherwise specified.

Definition 1 (Weak/strict edge satisfaction)**.** *The node map M weakly satisfies a downward edge $e = \langle p, q \rangle \in E(Q)$ iff $M(p)$ is an ancestor of $M(q)$, and strictly satisfies e iff $M(p)$ and $M(q)$ are related as specified by $L(e)$.*

Definition 2 (Match). *Given subgraphs $Q'' \subseteq Q' \subseteq Q$, the node map $M : V(Q') \to V(D)$ is a weak (strict) match for Q'' iff all edges in Q'' are weakly (strictly) satisfied by M. If $Q'' = Q$ we call M a weak (strict) full match.*

Where no confusion arises, the term weak (strict) match may also be used for $M(Q)$.

We denote the set of unique strict full matches by O. As is common, we view the size of the query as a constant, and call a twig join algorithm linear and optimal if the combined data and result complexity is $\mathcal{O}(I + O)$ [3].[1]

The results presented in the following all apply to both weak and strict matching, unless otherwise specified. The following lemma implies that we can use filtering strategies that only consider parts of the query.

Lemma 1 (Filtering). *If there exists a $Q' \subseteq Q$ containing q where no match M' for Q' contains $q \mapsto q'$, then there exists no match M for Q containing $q \mapsto q'$.*

Proof. By contraposition. Given a match $M \ni q \mapsto q'$ for Q, for any $Q' \subseteq Q$ containing q, the match $M \setminus \{p \mapsto p' \mid p \notin Q'\}$ matches Q' and contains $q \mapsto q'$. $\square$

3.1 Preorder Filtering on Matched Paths

Many current algorithms use the getNext input stream merge component [2], which returns data nodes in a relaxed preorder, which only dictates the order of matches for query nodes related by ancestry. This is not detailed in the original description [2] and is easy to miss.[2] The TwigMix algorithm incorrectly assumes strict preorder [10] (See Appendix E).

Definition 3 (Match preorder). *The sequence of pairs $q_1 \mapsto q'_1, \ldots, q_k \mapsto q'_k \in V(Q) \times V(D)$ is in global match preorder iff for any $i < j$ either (1) q'_i precedes q'_j in tree preorder, or (2) $q'_i = q'_j$ and q_i precedes q_j in tree postorder. The sequence is in local match preorder if (1) and (2) hold for any $i < j$ where $q_i = q_j$ or $\langle q_i, q_j \rangle \in E(Q)$ or $\langle q_j, q_i \rangle \in E(Q)$.*

The following definition formalizes a filtering criterion commonly used when processing data nodes in preorder, local or global.

Definition 4 (Prefix path match). *M is a prefix path match for $q_k \in Q$ iff it is a match for the (simple) path $q_1 \ldots q_k$, where q_1 is the root of Q.*

To implement prefix path match filtering, preorder algorithms maintain the set of open nodes, i.e., the ancestors, at the current position in the tree. Most algorithms have one stack of open data nodes for each query node, and given a current pair $q \mapsto q'$ pop non-ancestors of q' from the stacks of q and its parent [2, 8, 10]. *Weak* filtering can then be implemented by checking if (i) q is the root, or (ii) the stack for the parent of q is

[1] For Twig[2]Stack the combined data, *query* and result complexity is $\mathcal{O}(I \log Q + I b_Q + OQ)$, where b_Q is the maximum branching factor in the query [3]. The TJStrict algorithms we present in Section 3 have the same complexity when using a heap-based input stream merger, and $\mathcal{O}(IQ + OQ)$ complexity when using a getNext-based input stream merger. Note that the total number of data nodes references in the input and output are $|I|$ and $|O| \cdot |Q|$, respectively.

[2] To be precise, getNext also returns matches for sibling query nodes in order.

non-empty. If q' is not filtered out, it is pushed onto the stack for q, and added to the intermediate results. This can be extended to *strict* checking of P–C edges by inspecting the top node on the parent's stack. Strict prefix path matching is rarely used in practice, as can be seen from the fourth column in Figure 5.

The implementation of prefix path checks is the reason for the secondary ordering on query node postorder for match pairs in Definition 3. Without the secondary ordering problems arise when multiple query nodes have the same label: A data node could be misinterpreted as a usable ancestor of itself when checking for non-empty stacks, or hide a proper parent of itself when checking top stack elements.

Algorithms storing intermediate results in postorder use a global stack for the query [3, 11], and inspection of the top stack node cannot be used to implement prefix path matching when the query contains A–D edges, as ancestors may be hidden deep in the stack. Extending these algorithms to implement prefix path filtering requires maintaining additional data structures.

The choice of preorder filtering does not influence optimality, as illustrated by the following example.

Example 1. *Assume non-branching data generated by $/(\alpha_1/)^n \dots (\alpha_m/)^n \beta/\gamma$, and the query $//\alpha_1// \dots //\alpha_m/\gamma$, where $\alpha_1, \dots, \alpha_m, \beta, \gamma$ are all distinct labels. Unless it is signaled bottom-up that the pattern α_m/γ is not matched below, the result enumeration phase will take $\Omega(n^m)$ time, because all combinations of matches for $\alpha_1, \dots, \alpha_m$ will be tested.*

3.2 Postorder Filtering on Matched Subtrees

The ordering of match pairs required by most bottom-up algorithms is symmetric with the global preorder case:

Definition 5 (Match postorder)**.** *A sequence of pairs $q_1 \mapsto q'_1, \dots, q_k \mapsto q'_k$ is in match postorder iff for any $i < j$ either (1) q'_i precedes q'_j in tree postorder, or (2) $q'_i = q'_j$ and q_i precedes q_j in tree preorder.*

The second property is required for the correctness of both Twig^{2}Stack [3], where a data node could hide proper children of itself, and TwigList [11], where a node could be added as a descendant of itself.

Definition 6 (Subtree match)**.** *M is a subtree match for $q \in Q$ iff it is a match for the subtree rooted at q.*

Example 1 also illustrates why strict subtree match checking is required for optimality, because no node labeled γ is a direct child of a node labeled α_m in the data. As described in Section 2, Twig^{2}Stack and TwigList respectively implement strict and weak subtree match filtering, and for these algorithms strict filtering is required for optimality. TwigList could be extended to strict filtering by traversing descendant intervals to look for direct children, but this would have quadratic cost if implemented naively, as descendant intervals may overlap.

In algorithms storing data in preorder, nodes are added to the intermediate results after passing preorder checks. If nodes are stored in arrays, later *removing* a node failing a postorder check from the middle of an array would incur significant cost. Note that many of the algorithms listed in Figure 5 inherit weak subtree match filtering from the input stream merger used, as described later in Section 4.4.

3.3 Result Enumeration

Even if the strict subtree match checking sketched for TwigList above could be implemented efficiently, results would still not be enumerated in linear time, as usable child nodes may be scattered throughout descendant intervals, as shown by the following example:

Example 2. *Assume a tree constructed from the nodes* $\{\mathtt{a}_1, \ldots, \mathtt{a}_n, \mathtt{b}_1, \ldots, \mathtt{b}_{2n}\}$*, labeled* `a` *and* `b`*. Let each node* $\mathtt{a}_i$ *have a left child* $\mathtt{b}_i$*, a right child* $\mathtt{b}_{n+i}$*, and a middle child* $\mathtt{a}_{i+1}$ *if* $i < n$*. Given the query* $/\!/\mathtt{a}/\mathtt{b}$*, each node* $\mathtt{a}_i$ *is part of two matches, one with* $\mathtt{b}_i$ *and one with* $\mathtt{b}_{n+i}$*, but to find these two matches,* $2n - 2i$ *useless* `b`*-nodes must be traversed in the descendant interval. This gives a total enumeration cost of* $\Omega(n^2)$*.*

The following theorem formalizes properties of the intermediate result storage in Twig^{2}Stack that are key to its optimality.

Theorem 1 (Linear time result enumeration [3])**.** *The result set* O *can be enumerated in* $\Theta(O)$ *time if (i) the data nodes* d *such that* $root(Q) \mapsto d$ *is part of a strict full match can be found in time linear in their number, and (ii) given a pair* $q \mapsto q'$*, for each child* c *of* q*, the pairs* $c \mapsto c'$ *that are part of a strict subtree match for* q *together with* $q \mapsto q'$ *can be enumerated in time linear in their number.*

3.4 Full Matches

When different types of filtering strategies are combined, it may be interesting to know when additional filtering passes will not remove more nodes.

Theorem 2 (Full Match)**.** *(i) A pair* $q \mapsto q'$ *is part of a full match iff (ii) it is part of a prefix path match that only uses pairs that are part of subtree matches.*

sketch. ($i \Rightarrow ii$) Follows from Lemma 1. ($i \Leftarrow ii$) Let $\mathfrak{M} = \langle Q, D, I \rangle$ be the initial matching problem, and $\mathfrak{M}' = \langle Q, D, I' \rangle$ be the matching problem where I' is the set of pairs that are part of subtree matches in $\mathfrak{M}$. The theorem is true for pairs with the query root, as for the query root a subtree match is a full match. Assume that there is a prefix path match $M_{\downarrow q} \ni q \mapsto q'$ for q in $\mathfrak{M}'$, and that p is the parent of q. By construction, $M_{\downarrow q} \ni p \mapsto p'$ is also a prefix path match for p. We use induction on the query node depth, and prove that if $p \mapsto p'$ is part of a full match M_p for p, then $q \mapsto q'$ must be part of a full match for q. Let Q_q be the subtree rooted at q, and $Q_p = Q \setminus Q_q$. Let $M'_p = M_p \setminus \{r \mapsto r' \mid r \in Q_q\}$. By the assumption $q \mapsto q' \in I'$, there exists a subtree match $M_{\uparrow q} \ni q \mapsto q'$ for q. Then the node map $M_q = M'_p \cup M_{\uparrow q}$ is a full match for q, because (1) $p \mapsto p' \in M'_p$ and $q \mapsto q' \in M_{\uparrow q}$ must satisfy the edge $\langle p, q \rangle$ as they are used together in $M_{\downarrow q}$, and (2) Q_p and Q_q can be matched independently when the mapping of p and q is fixed. □

In other words, if nodes are filtered first in postorder on strictly matched subtrees, and then in preorder on strictly matched prefix paths, the intermediate result set contains *only* data nodes that are part of the *final* result. The opposite is not true: In the example in Figure 1, the pair $\mathsf{c} \mapsto \mathsf{c}_3$ would not be removed if strict prefix path filtering was followed by strict subtree match filtering.

Note that strict full match filtering is not necessary for optimal enumeration by Theorem 1, and that the optimal algorithms we present in the following do not use it. They use prefix path match filtering followed by subtree match filtering, where the former is only used to speed up practical performance. On the other hand, the input stream merge component we introduce in Section 4.4 gives inexpensive *weak* full match filtering.

4 Fast Optimal Twig Joins

In this section we create an algorithmic framework that permits any combination of preorder and postorder filtering. First we introduce a new data structure that enables strict subtree match checking and linear result enumeration.

4.1 Level Split Vectors

A key to the practical performance of TwigList and TwigFast is the storage of intermediate nodes in simple vectors [11], but this scheme makes it hard to improve worst-case behavior.

To enable direct access to usable child matches below both A–D and P–C edges, we *split* the intermediate result vectors for query nodes below P–C edges, such that there is one vector for each data tree level observed, as shown in Figure 6. Given a node in the intermediate results, matches for a child query node below an A–D edge can be found in a regular descendant interval, while the matches for a child query node below a P–C edge can be found in a *child interval* in a child vector. This vector can be identified by the depth of the parent data node plus one.

In the following we assume that level split vectors can be accessed by level in amortized constant time. This is true if level split vectors are stored in dynamic arrays, and the depth of the deepest data node is asymptotically bounded by the size of the input, that is, $d \in \mathcal{O}(I)$. If this bound does not hold, which can be the case when $|I| \ll |D|$, expected linear performance can be achieved by storing level split vectors in hash maps.

When nodes are added to the intermediate results in postorder, checking for non-empty descendant and child intervals inductively implies strict subtree match filtering. This is illustrated for our example in Figure 6. As each check takes constant time, the intermediate results can be constructed in $\Theta(I)$ time, as for TwigList [11]. Result enumeration with this data structure is a trivial recursive iteration through nodes and their child and descendant intervals, which is almost identical to the enumeration in TwigList. The difference is that no extra check is necessary for P–C relationships, and that the result enumeration is $\Theta(O)$ by Theorem 1 when strict subtree match filtering is applied.

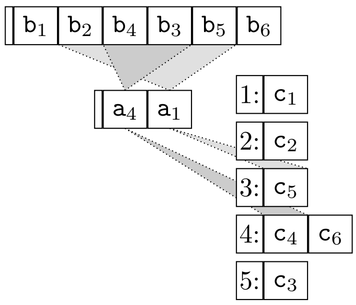


Figure 6: Intermediate data structures using level split vectors and strict subtree match filtering. As opposed to in Figure 4b, $\mathtt{a}_2$ does not satisfy.

4.2 The TJStrictPost Algorithm

Algorithm 1 shows the general framework we use for postorder construction of intermediate results, extending algorithms like TwigList [11]. It allows using any combinations of the preorder and postorder checks described in Section 3, from none to weak and strict, and allows using either simple vectors or level split vectors. A global stack is used to maintain the set of open data nodes, and if prefix path matching is implemented, a local stack for each query node is maintained in parallel. The input stream merge component used is a priority queue implemented with a binary heap. The postorder storage approach used here requires global ordering, and cannot read local preorder input (see Appendix E).

Algorithm 1 Postorder construction.

```
While ¬Eof:
    Read next $q \mapsto d$.
    While non-ancestors of $d$ on global stack:
        Pop $q' \mapsto d'$ from global and local stack.
        If $q' \mapsto d'$ satisfies postorder checks:
            Set interval end index for $d'$
            in the vector of each child of $q'$.
            Add $d'$ to intermediate results for $q'$.
    If $d$ satisfies preorder checks:
        For each child of $q$, set interval start index for $d$.
        Push $q \mapsto d$ on global stack.
        Push $d$ on local stack for $q$.
Clean remaining nodes from the stacks.
Enumerate results.
```

The correctness of Algorithm 1 follows from the correctness of the filtering strategies described in Sections 3.1 and 3.2, and the correctness of TwigList [11], with the enumeration algorithm trivially extended to use child intervals when level split vectors are used.

We now define the TJStrictPost algorithm, which builds on this framework. Detailed pseudocode can be found in Appendix A. The algorithm uses level split vectors, and, as opposed to the previous twig join algorithms listed in Figure 5, it includes *strict* checking of both matched prefix paths and subtrees. The former is implemented by checking the top data node on the local stack of the parent query node, while the latter is implemented by checking for non-empty child and descendant intervals. A $\Theta(I + O)$ running time follows from the discussion in Section 4.1.

4.2.1 A note on TwigList:

As noted in the original description, chaining nodes with the same tree level into linked lists inside descendant intervals can improve practical performance in TwigList [11]. However, as the first child match with the correct level must still be searched for, further changes are needed to achieve linear worst-case evaluation. This can be implemented by maintaining a vector for each query node with the *previous* match on each tree level at any time. A node must then be given pointers to such previous matches as it is pushed on stack in TwigList. When the node is popped off stack, it can then be checked if any children have been found, and intermediate results can be enumerated in linear time, assuming that $d \in \mathcal{O}(I)$, as for our solution.

4.3 The TJStrictPre Algorithm

Algorithm 2 shows the general framework we use to construct intermediate results in preorder, extending algorithms like TwigFast [10]. It supports any combination of preorder and postorder filtering, simple or level split vectors, and input in global or local preorder. As opposed to with postorder storage, nodes are inserted directly into intermediate result vectors after they have passed a prefix path check. Local stacks store references to open nodes in the intermediate results. If strict subtree match filtering is required, or weak subtree match filtering is not implied by the input stream merger, intermediate results are filtered bottom-up in a post-processing pass.

TwigFast is reported to have faster average case query evaluation than previous twig joins [10], and we hope to match this performance in a worst-case linear algorithm. The TJStrictPre algorithm is similar to TJStrictPost, and uses strict checking of prefix paths and subtrees, and stores intermediate results in level split vectors. See detailed pseudocode in Appendix B. TJStrictPre uses the getPart input stream merger, which is an improvement of getNext described in Section 4.4. If the post-processing pass can be performed in linear time, then the algorithm can evaluate twig queries in $\Theta(I + O)$ time, by the same argument as for TJStrictPost.

The filtering pass is implemented by cleaning intermediate result vectors bottom-up in the query, in-place overwriting data nodes not satisfying subtree matches, as described in detail in Appendix D. The indexes of kept nodes are stored in a separate vector for each query node, and are used to translate old start and end indexes into new positions.

Algorithm 2 Preorder construction.

```
While $\neg$EOF:
    Read next $q \mapsto d$.
    For the stack of both $q$'s parent and $q$ itself:
        Pop non-ancestors of $d$,
        and set their end indexes.
    If $d$ satisfies preorder checks:
        For each child of $q$, set interval start index for $d$.
        Add $d$ to intermediate results for $q$.
        Push reference to $d$ on stack for $q$.
Clean stacks.
Clean intermediate results with postorder checks.
Enumerate results.
```

To achieve linear traversal, the intermediate result vector of a node is traversed in parallel with the index vectors of the children after they have been cleaned. For level split vectors, there is one separate index vector per used level, and a separate vector iterator is used per level when the parent is cleaned. Also, there is an array giving the level of each stored data node in preorder, such that split and non-split child vectors can then be traversed in parallel. Start values are updated as nodes are pushed onto a stack in preorder, while end values are updated as nodes are popped off in postorder.

4.4 The getPart Input Stream Merger

The getNext input stream merge component implements weak subtree match filtering in $\Theta(I)$ time, and is used to improve practical performance in many current algorithms using preorder storage [8,10]. Assume in the following discussion that there is one preorder stream of label-matching data nodes associated with each query node. The input stream merger repeatedly returns pairs containing a query node and the data node at the head of its stream, implementing Comp. 2 in Figure 3.

The getNext function processes the query bottom-up to find a query node that satisfies the following three properties: (1) when its stream is forwarded at least until its head follows the heads of the streams of the children in postorder, it still precedes them in preorder, (2) all children satisfy properties 1 and 2, and (3) if there is a parent, it does not satisfy 1 and 2. Property 2 implies that weak subtree filtering is achieved, and Property 3 implies that local preorder by Definition 3 is achieved.

The procedure is efficient if leaf query nodes have relatively few matches, which can be the case in practice in XML search when all query leaf nodes are selective text value predicates. However, if the internal query nodes are more selective than the leaf nodes, or if not all leaves are selective, the overhead of using the getNext function may outweigh the benefits.

To improve practical performance we introduce the *getPart* function, which requires the following property in addition to the above three: (4) if there is a parent, then the current head of stream is a descendant of a data node that was the head of stream for the parent in some previous subtree match for the parent. This inductively implies that nodes returned are also weak prefix path matches, and from the ordering of the filtering steps, the result is weak *full* match filtering by Theorem 2. To allow forwarding streams to find such nodes, the algorithm can no longer be stateless, as shown by the following example:

Example 3. *Assume that the heads of the streams for query nodes* $\texttt{a}^1$ *and* $\texttt{b}^1$ *in Figure 1 are* $\texttt{a}_3$ *and* $\texttt{b}_2$*, respectively. Then it cannot be known by only inspecting heads of streams whether or not any usable ancestors of* $\texttt{b}_2$ *were seen before* $\texttt{a}_3$*, and* $\texttt{b}_2$ *must be returned regardlessly.*

Property 4 is implemented in getPart by maintaining for each query node, the data node *latest* in the tree *postorder* that has been part of a weak full match. This value is updated when a query node is found to satisfy all four properties. To ensure progress, streams are forwarded top-down in the query to match the stored value or the current head for the parent node. Note that multiple top-down and bottom-up passes may be needed to find a satisfying node, but each such pass forwards at least one stream past useless matches. See detailed pseudocode in Appendix C.

5 Experiments

The following experiments explore the effects of weak and strict matching of prefix paths and subtrees, different input stream merge functions, and level split vectors.

We have used the DBLP, XMark and Treebank benchmark data, and run the commonly used DBLP queries from the PRIX paper [12], the XMark queries from the XPathMark suite part A [5] (except queries 7 and 8, which are not twigs), and the Treebank queries from the TwigList paper [11]. In addition, we have created some artificial data and queries. Details can be found in Appendix F. The experiments were run on a computer with an AMD Athlon 64 3500+ processor and 4 GB of RAM. All queries were warmed up by 3 runs and then run 100 times, or until at least 10 seconds had passed, measuring average running-time.

All algorithms are implemented in C++, and features are turned on or off at compile time to make sure the overhead of complex methods does not affect simpler methods. Feature combinations are coded with 5 letter tags. We use **H**eap, get**N**ext and get**P**art for merging the input streams, and store intermediate results in pr**E**order or p**O**storder. We use no (**-**), **W**eak or **S**trict prefix path match filtering, and no (**-**), **W**eak or **S**trict subtree match filtering. Intermediate results are stored in simple vectors (**-**) or **L**evel split vectors. The previous algorithms TwigList and TwigFast are denoted by `HO-W-` and `NEWW-`, respectively, while TJStrictPost and TJStrictPre are denoted by `HOSSL` and `PESSL`. Note that filtering checks are not performed in intermediate result construction if the given filtering level is already achieved by the input stream merger. Strict subtree match filtering is implemented by descendant interval traversal when not using level split vectors. With preorder storage an extra filtering pass is used to implement subtree match filtering. Worst-case optimal algorithms match the pattern `***SL`.

We present no performance comparisons with the Twig^{2}Stack algorithm because it does not fit into our general framework. Getting an accurate and fair comparison would be an exercise in independent program optimization. TwigList is previously reported to be 2–8 times faster than Twig^{2}Stack for queries similar to ours [11].

5.1 Checked Paths and Subtrees

Figure 7 shows results for running the XMark query //`annotation`/`keyword`, with cost divided into different components and phases. Filtering lowers the cost of building intermediate data structures because their size is reduced, and the cost of enumerating results because redundant traversal is avoided. Note that this query was chosen because it shows the potential effects of prefix path and subtree match filtering, and it may not be representative.

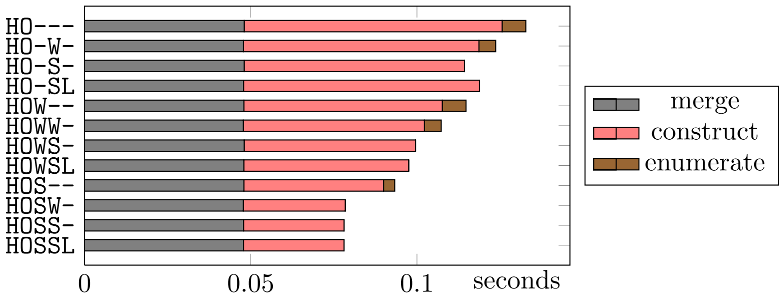


Figure 7: Query `//annotation/keyword` on XMark data. Cost divided into merging input, building intermediate results, and result enumeration.

Figure 8 shows the effects of prefix path vs. subtree match filtering averaged over all queries on DBLP, XMark and Treebank. Heap input stream merging was used because it allows all filtering levels, and postorder storage was used to avoid extra filtering passes. Each timing has been normalized to 1 for the fastest method for each query. Raw timings are listed in Appendix G. As opposed to in Figure 7, there is on average little difference between the methods using at least *some* filtering both in preorder and postorder. The benefits of asymptotic constant time checks when using level split vectors seems to be outweighed by the cost of maintaining and accessing them, but only by a small margin.

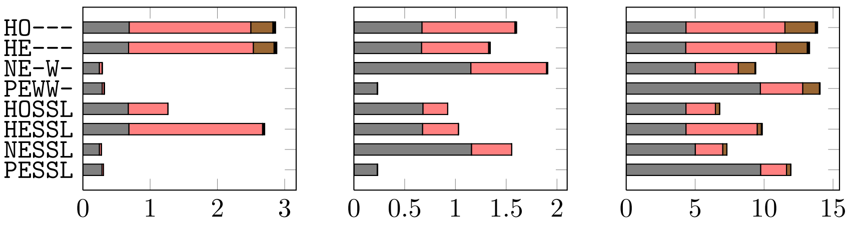
5.2 Reading Input

Figure 9 shows the effect of using different input stream mergers. The labels in the artificial data tree used are Zipf distributed ($s = 1$), with `a`, `b`, `y` and `z` being the labels on 30%, 13%, 1.0% and 1.0% of the nodes, respectively. The data and queries were chosen to shed light on both the benefits and the possible overhead of using the advanced input methods.

For the first query, //`a`/`b`[`y`][`z`], the leaves are very selective, and both getNext and getPart very efficiently filter away most of the nodes. The input stream merging is slightly

		Prefix path							
		-		W		S			
		avg	max	avg	max	avg	max	avg	max
Subtree	--	1.84	3.5	1.29	1.61	1.23	1.62	1.45	3.5
	W-	1.24	1.45	1.03	1.13	1.01	1.06	1.09	1.45
	S-	1.24	1.46	1.03	1.10	1.02	1.08	1.10	1.46
	SL	1.32	1.55	1.09	1.19	1.06	1.20	1.16	1.55
		1.41	3.5	1.11	1.61	1.08	1.62	1.20	3.5

Figure 8: The effect of parent match filtering. vs. child match filtering. Running DBLP, XPathMark and Treebank queries. Normalizing query times to 1 for the fastest method for each query. Showing arithmetic mean and maximum for normalized time.



(a) (b) (c)

Figure 9: Running queries on Zipf data. (a) Selective leaves. (b) Selective internal nodes. (c) No selective nodes.

more efficient for the simpler getNext. In the second query, //y/z[a][b], the internal nodes are selective, while the leaves are not. Here getPart efficiently filters away many nodes, while getNext does not, making it even slower than the simple heap, due to the additional complexity. The third query shows a case where getPart performs worse than both the other methods. In this query, //a[a[a][a]][a[a][a]], all query nodes have very low selectivity, and are equally probable. The filtering has almost no effect, and only causes overhead. Note the cost difference between HOSSL and HESSL, which is due to the additional filtering pass over the large intermediate results.

5.3 Combined Benefits

Figure 10 shows the effects of combining different input stream mergers and additional filtering strategies. The same queries as in Figure 8 are evaluated, and the first column shows the same tendencies: There is not much difference between the strategies as long as you do at least weak match filtering on both prefix path and subtree.

		Input stream merger									
		HO		HE		NE		PE			
		avg	max	avg	max	avg	max	avg	max	avg	max
Match filtering	---	7.0	33	6.6	30					6.8	33
	-W-	4.5	23	7.5	34	3.7	19			5.2	34
	-S-	4.5	23	7.5	34	3.7	18			5.2	34
	-SL	4.8	25	8.3	38	3.8	19			5.6	38
	W--	4.9	24	4.9	23					4.9	24
	WW-	3.7	19	5.4	25	3.2	15	1.02	1.11	3.3	25
	WS-	3.7	19	5.4	26	3.2	15	1.04	1.15	3.3	26
	WSL	3.9	20	5.7	27	3.2	15	1.08	1.22	3.5	27
	S--	4.8	24	4.8	24					4.8	24
	SW-	3.7	19	5.2	26	3.2	15	1.03	1.12	3.3	26
	SS-	3.7	19	5.1	26	3.2	15	1.05	1.17	3.3	26
	SSL	3.9	20	5.5	27	3.2	15	1.05	1.20	3.4	27
		4.4	33	6.0	38	3.4	19	1.04	1.22	4.1	38

Figure 10: Input mergers vs. filtering strategies.

In the second column all methods using any subtree match filtering are more expensive, because with preorder storage, subtree match filtering is performed in a second pass over the intermediate results. A second pass is also used for for subtree match filtering in the third and fourth columns, but in practice the getNext and getPart components have already filtered away more nodes, the intermediate results are smaller, and the second pass is less expensive.

Note the difference between using getNext and getPart. The new method is more than three times as fast on average, and is more than one order of magnitude faster for queries where only some of the leaf nodes are selective. The getPart function also fast forwards through useless matches for the leaves that are not selective, while getNext passes all leaf matches on to the intermediate result construction component. Also note that the maximum overhead of using `PESSL`, the fastest worst-case optimal method, is at most 20% in any benchmark query tested.

6 Related Work

This work is based on the assumption that label-partitioning and simple streams is used. Orthogonal previous work investigates how the underlying data can be *accessed*. If the streams support *skipping*, both unnecessary I/O and computation can be avoided [7]. Our getPart algorithm, which is detailed in Appendix C, can be modified to use any underlying skipping technology by changing the implementation of FwdToAncOf() and FwdToDescOf(). *Refined partitioning* schemes with structure indexing can be used to reduce the number of data nodes read for each query node [4, 9]. Our twig join algorithms are independent of the partitioning scheme used, assuming multiple partition blocks matching a single query node are merged when read. Another technique is to use a node encoding that allows reconstruction of data node ancestors, and use *virtual streams* for the internal query nodes [14]. Our getPart algorithm could be changed to generate virtual internal query node matches from leaf query node matches, as complete query subtrees are always traversed. For a broader view on XML indexing see the survey by Gou and Chirkova [6]. XPath queries can be rewritten to use only the axis *self*, *child*, *descendant*, and *following* [16]. To add support for support for the *following*-axis, we would have to add additional logic for how to forward streams, and modify the data structures to store start indexes for the new relationship.

7 Conclusions and Future Work

In this paper we have shown how worst-case optimality and fast evaluation in practice can be combined in twig joins. We have performed experiments that span out and extend the space of the fastest previous solutions. For common benchmark queries our new and worst-case optimal algorithms are on average three times as fast as earlier approaches. Sometimes they are more than an order of magnitude faster, and they are never more than 20% slower.

In future work we would like to combine the new techniques with previous orthogonal techniques such as skipping, refined partitioning and virtual streams. Also, it would be interesting to see an elegant worst-case linear algorithm reading local preorder input and producing preorder sorted results, that does not perform a post-processing pass over the data, and does not need the assumption $d \in \mathcal{O}(I)$.

Acknowledgments This material is based upon work supported by the iAd Project funded by the Research Council of Norway, and the Norwegian University of Science and Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the funding agencies.

References

[1] Radim Bača, Michal Krátký, and Václav Snášel. On the efficient search of an XML twig query in large DataGuide trees. In *Proc. IDEAS*, 2008.

[2] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD*, 2002.

[3] Songting Chen, Hua-Gang Li, Junichi Tatemura, Wang-Pin Hsiung, Divyakant Agrawal, and K. Selçuk Candan. Twig^{2}Stack: bottom-up processing of generalized-tree-pattern queries over XML documents. In *Proc. VLDB*, 2006.

[4] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proc. SIGMOD*, 2005.

[5] Massimo Franceschet. XPathMark web page. `http://sole.dimi.uniud.it/~massimo.franceschet/xpathmark/`.

[6] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *Knowl. and Data Eng.*, 2007.

[7] Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *Proc. VLDB*, 2003.

[8] Zhewei Jiang, Cheng Luo, Wen-Chi Hou, and Qiang Zhu Dunren Che. Efficient processing of XML twig pattern: A novel one-phase holistic solution. In *Proc. DEXA*, 2007.

[9] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *Proc. SIGMOD*, 2002.

[10] Jiang Li and Junhu Wang. Fast matching of twig patterns. In *Proc. DEXA*, 2008.

[11] Lu Qin, Jeffrey Xu Yu, and Bolin Ding. TwigList: Make twig pattern matching fast. In *Proc. DASFAA*, 2007.

[12] Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using Prüfer sequences. In *Proc. ICDE*, 2004.

[13] Mirit Shalem and Ziv Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *Proc. ICDE*, 2008.

[14] Beverly Yang, Marcus Fontoura, Eugene Shekita, Sridhar Rajagopalan, and Kevin Beyer. Virtual cursors for XML joins. In *Proc. CIKM*, 2004.

[15] Chun Zhang, Jeffrey Naughton, David DeWitt, Qiong Luo, and Guy Lohman. On supporting containment queries in relational database management systems. *SIGMOD Rec.*, 2001.

[16] Ning Zhang, Varun Kacholia, and M. Tamer Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *Proc. ICDE*, 2004.

A TJStrictPost Pseudocode

Algorithm 3 shows more detailed pseudocode for the TJStrictPost algorithm described in Section 4.2. Tree positions are assumed to be encoded using BEL (*begin*, *end*, *level*) [15]. The function GetVector(q, *level*) returns the regular intermediate result vector if q is below an A–D edge, or the split vector given by *level* if q is below a P–C edge.

Algorithm 3 TJStrictPost

```
function EvaluateGlobal():
    $(c, q, d) \leftarrow$ MergedStreamHead()
    while $c \neq$ EOF:
        ProcessGlobalDisjoint($d$)
        if Open($q, d$):
            Push($S_{global}$, $(d.end, q)$)
        $(c, q, d) \leftarrow$ MergedStreamHead()
    ProcessGlobalDisjoint($\infty$)

function ProcessGlobalDisjoint($d$):
    while $S_{global} \neq \emptyset \wedge$ Top($S_{global}$)$.end < d.end$:
        $(end, q) \leftarrow$ Pop($S_{global}$)
        Close($q$)

function Open($q, d$):
    if CheckParentMatch($q, d$):
        $u \leftarrow$ new Intermediate($d$)
        MarkStart($q, u$)
        Push($S_{local}[q], u$)
        return true
    else:
        return false

function Close($q$):
    $u \leftarrow$ Pop($S_{local}[q]$)
    MarkEnd($q, u$)
    if CheckChildMatch($q, u$):
        Append(GetVector($q, u.d.level$), $u$)

function MarkStart($q, u$):
    for $r \in q.children$:
        $u.start[r] \leftarrow$ GetVector($r, u.d.level + 1$)$.size + 1$

function MarkStart($q, u$):
    for $r \in q.children$:
        $u.end[r] \leftarrow$ GetVector($r, u.d.level + 1$)$.size$

function CheckParentMatch($q, d$):
    if Axis($q$) = "//":
        return IsRoot($q$) or $S_{local}$[Parent($q$)] $\neq \emptyset$
    else:
        if IsRoot($q$): return $d.level = 1$
        else: return $S_{local}$[Parent($q$)] $\neq \emptyset \wedge d.level =$ Top($S_{local}$[Parent($q$)])$.level + 1$

function CheckChildMatch($u$):
    for $r \in u.q.children$:
        if $u.end[r] < u.start[r]$: return false
    return true
```

B TJStrictPre Pseudocode

Algorithm 4 shows more detailed pseudocode for the TJStrictPre algorithm described in Section 4.3.

Algorithm 4 TJStrictPre

```
function EvaluateLocalTopDown():
    $(c, q, d) \leftarrow$ MergedStreamHead()
    while $c \neq$ EOF:
        if $\neg$IsRoot($q$):
            ProcessLocalDisjoint(Parent($q$), $d$)
        ProcessLocalDisjoint($q, d$)
        Open($q, d$)
        $(c, q, d) \leftarrow$ MergedStreamHead()
    for $q \in Q$:
        ProcessLocalDisjoint($q, \infty$)
    FilterPass($q.root$)

function ProcessLocalDisjoint($q, d$):
    while Top($S_{local}[q]$).$d.end < d.end$:
        Close($q$)

function Open($q, d$):
    if CheckParentMatch($q, d$):
        $u \leftarrow$ new Intermediate($d$)
        $V \leftarrow$ GetVector($q, d.level$)
        if $\neg$IsLeaf($q$):
            MarkStart($q, u$)
            Push($S_{local}[q]$, $(V, V.size)$)
        Append($V, u$)
        return $\neg$IsLeaf($q$)
    else:
        return false

function Close($q$):
    if $\neg$IsLeaf($q$):
        $(V, i) \leftarrow$ Pop($S_{local}[q]$)
        MarkEnd($q, V[i]$)
```

C GetPart function

Pseudocode for the getPart function is shown in Algorithm 5, where what is conceptually different from the previous getNext function is colored dark blue.

GetPart forwards nodes both to catch up with the parent and child streams, whereas getNext only does the latter. The getNext algorithm is completely stateless, and only inspects stream heads. When a match for a query subtree is found, the stream for the subtree root node is read and forwarded. Then it is not possible to know in the next call on this point in the query, whether the child subtrees were once part of a match or not. In the getPart function we save one extra value per query node, stored in the M array, namely the latest match in the tree postorder which was part of a weak match for the *entire query*. When considering a query subtree, the currently interesting data nodes are those that are either part of a match using a previous head in the parent stream, or part of a new match using the current head in the parent stream (see Lines 9-13).

Algorithm 5 GetPart

```
 1: function MergedStreamHead():
 2:     while true:
 3:         $(c, d, q) \leftarrow$ GetPart($Q.root$)
 4:         if $c \neq$ MisMatch:
 5:             if $c \neq$ Eof:
 6:                 Fwd($q$)
 7:             return $(c, d, q)$

 8: function GetPart($q$):
 9:     if $\neg$IsRoot($q$):
10:         $p \leftarrow$ Parent($q$)
11:         FwdToDescOf($q$, $M[p]$)
12:         if $\neg$Eof($q$) $\wedge$ $\neg$Eof($p$) $\wedge$ $M[p].end < \mathrm{H}(q).end$:
13:             FwdToDescOf($q$, H($p$))
14:     if IsLeaf($q$):
15:         if Eof($q$): return (Eof, $q$, $\perp$)
16:         else: return (Match, $q$, H($q$))
17:     $(d_{min}, q_{min}) \leftarrow (\infty, \perp)$ ; $(d_{max}, q_{max}) \leftarrow (0, \perp)$
18:     for $r \in q.children$:
19:         $(c_r, d_r, q_r) \leftarrow$ GetNext($r$)
20:         if $c_r \neq$ Eof:
21:             if $c_r =$ MisMatch: flag MisMatch
22:             elif $q_r \neq r$: return (Match, $d_r$, $q_r$)
23:             if $d_r.begin < d_{min}.begin$: $(d_{min}, q_{min}) \leftarrow (d_r, q_r)$
24:             if $d_r.begin > d_{max}.begin$: $(d_{max}, q_{max}) \leftarrow (d_r, q_r)$
25:         else:
26:             FwdToEof($q$)
27:     if $q_{min} = \perp$: return (Eof, $\perp$, $q$)
28:     FwdToAncOf($q$, $d_{max}$)
29:     if flagged MisMatch:
30:         if Eof($q$): return (Eof, $\perp$, $q$)
31:         else: return (MisMatch, $\perp$, $q$)
32:     if $\neg$Eof($q$) $\wedge$ H($q$)$.begin < d_{min}.begin$:
33:         if IsRoot($q$) $\vee$ H($q$)$.end < M[p].end$:
34:             if $M[q].end <$ H($q$)$.end$: $M[q] \leftarrow$ H($q$)
35:         return (Match, H($q$), $q$)
36:     else:
37:         if $d_{min}.begin < M[q].end$:
38:             return (Match, $d_{min}$, $q_{min}$)
39:         else:
40:             if Eof($q$): return (Eof, $\perp$, $q$)
41:             else: return (MisMatch, $\perp$, $q$)

42: function FwdToEof($q$):
43:     while $\neg$Eof($q$): Fwd($q$)

44: function FwdToDescOf($q$, $d$):
45:     while $\neg$Eof($q$) $\wedge$ H($q$)$.begin \leq d.begin$: Fwd($q$)

46: function FwdToAncOf($q$, $d$):
47:     while $\neg$Eof($q$) $\wedge$ H($q$)$.end \leq d.begin$: Fwd($q$)
```

The forwarding of streams based on child stream heads is very similar to in getNext (Lines 17-28). Unless the search is short-circuit (Line 22), the stream is forwarded at least until the head is an ancestor of all the child heads (Line 28). The query node itself is returned if an ancestor of the child heads was found, and unless the previous M value is an ancestor of the current head, it is updated. When a child query node is returned, it is known whether or not it is part of a match.

D Extra Filtering Pass

Algorithm 6 gives pseudocode for the extra filtering pass used to obtain strict subtree match filtering when using preorder storage in TJStrictPre.

Algorithm 6 FilterPass

```
 1: function CleanUp($q$):
 2:     if Axis($q$) = "//":
 3:         CleanUpVector(GetVector($q, \cdot$), $C[q]$)
 4:     else:
 5:         for $h \in$ used levels:
 6:             CleanUpVector(GetVector($q, h$), $C_h[q]$)

 7: function CleanUpVector($V, C$):
 8:     $i \leftarrow j \leftarrow 0$
 9:     while $i < V.size$:
10:         if CheckChildMatch($q, V[i]$):
11:             $V[j] \leftarrow V[i]$
12:             Append($C, i$)
13:             $j \leftarrow j + 1$
14:         $i \leftarrow i + 1$
15:     Resize($V, j$)

16: function FilterPass($q$):
17:     if IsLeaf($q$): return
18:     for $r \in q.children$:
19:         FilterPass($r$)
20:     if NonLeafChildren($q$) $\neq \emptyset$:
21:         for $u \in$ AllNodes($q$):
22:             FilterPassPost($u.d$)
23:             for $r \in$ NonLeafChildren($q$):
24:                 $u.start[r] \leftarrow$ FwdIter($r, u.start[r], u.d$)
25:             Push($S_{local}[q], u$)
26:         FilterPassPost($\infty$)
27:     CleanUp($q$)

28: function FilterPassPost($q, d$):
29:     while $S_{local}[q] \neq \emptyset \wedge$ Top($S$)$.end < d.end$:
30:         $u \leftarrow$ Pop($S$)
31:         for $r \in$ NonLeafChildren($q$):
32:             $u.end[r] \leftarrow$ FwdIter($r, u.end[r], u.d$)

33: function FwdIter($q, pos, d$):
34:     if Axis($q$) = "//":
35:         while $I[q] < C[q].size \wedge C[q][I[q]] < pos$:
36:             $I[q] \leftarrow I[q] + 1$
37:         return $I[q]$
38:     else:
39:         $h \leftarrow d.level + 1$
40:         while $I_h[q] < C_h[q].size \wedge C_h[q][I_h[q]] < pos$:
41:             $I_h[q] \leftarrow I_h[q] + 1$
42:         return $I_h[q]$
```

During the clean-up (Line 1-15), nodes failing checks are overwritten, and it is stored in the C vectors which values were not dropped. The query nodes are visited bottom-up by the FilterPass function, updating the vectors of one query node at a time, based on the cleanup in non-leaf child query nodes. The FilterPass and FilterPassPost functions go through all data nodes in preorder and postorder respectively, updating interval start and end indexes.

The AllNodes call returns a special iterator to all intermediate data nodes for a query node, sorted in total preorder. For query nodes with an incoming P–C edge and level split vectors, the order in which nodes were inserted on different levels was recorded during construction in TJStrictPre. Details are omitted in Algorithm 4, where an extra statement must be added after line 22, storing a reference to the used vector.

The FwdIter function contains the logic for updating the start and end indexes. Each query node has an iterator for each vector, which is utilized when traversing the matches for the parent query node. In essence, the segments of child and descendant intervals which contain references to nodes which were not saved during a cleanup pass are discarded.

E GetNext and Postorder

Many algorithms use the getNext function [2] for merging the input streams instead of a heap or linear scan [8, 10], because it cheaply filters away many useless nodes by implementing weak subtree match filtering. In this Appendix we show why using getNext with postorder intermediate result construction gives problems regardless of whether local or global stacks are used.

The getNext function does not return the data nodes in strict preorder, as assumed in the correctness proof for the TwigMix algorithm [10], but in local preorder (see Definition 3). As explained in Section 3.1, the top-down algorithms using getNext maintain one stack or equivalent structure for each internal query node. When a new match for a given query node is seen, the typical strategy [2] is popping non-ancestor nodes off the parent query node's stack, and the query node's own stack.

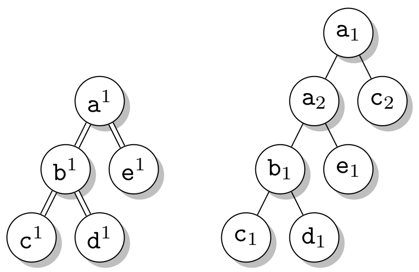
If a global stack is combined with using getNext input, errors may occur, as for the example query and data in Figure 11. With local preorder the ordering between the nodes not related through ancestry is not fixed. Assume that the ordering is

$$\langle \mathtt{a}^1 \mapsto \mathtt{a}_1, \mathtt{a}^1 \mapsto \mathtt{a}_2, \mathtt{b}^1 \mapsto \mathtt{b}_1, \mathtt{c}^1 \mapsto \mathtt{c}_1, \mathtt{d}^1 \mapsto \mathtt{d}_1, \mathtt{c}^1 \mapsto \mathtt{c}_2, \mathtt{e}^1 \mapsto \mathtt{e}_1 \rangle.$$

Then $\mathtt{c}^1 \mapsto \mathtt{c}_2$ will pop $\mathtt{a}^1 \mapsto \mathtt{a}_2$ off stack before $\mathtt{e}^1 \mapsto \mathtt{e}_1$ is observed, and $\mathtt{e}^1 \mapsto \mathtt{e}_1$ will never be added as a descendant of $\mathtt{a}^1 \mapsto \mathtt{a}_2$.

But using local stacks and a bottom-up approach also gives errors, because data nodes are added to the intermediate structures as they are popped off stack when using postorder storage, which is too late when using getNext and local stacks. If the typical approach is used, $\mathtt{c}^1 \mapsto \mathtt{c}_2$ will only pop $\mathtt{b}_1$ off the stack of $\mathtt{b}^1$ and $\mathtt{c}_1$ off the stack of $\mathtt{c}^1$. Then $\mathtt{d}^1 \mapsto \mathtt{d}_1$ will never be added to the child structures of $\mathtt{b}^1 \mapsto \mathtt{b}_1$, because it is popped to late.

It may be possible to modify the local stack approach to work with postorder storage and getNext input, but this would require carefully popping nodes on ancestor and descendant stacks in the right order.



(a) Query (b) Data

Figure 11: Problematic case with local preorder input and postorder storage.

F Benchmark Data and Queries

Figure 12 gives some details on the benchmark data used in our experiments, and Figure 13 lists the queries we have used.

Name	**Size**	**Nodes**
Source		
DBLP	676 MB	35 900 666
`http://dblp.uni-trier.de/xml`		
XMark	1 118 MB	32 298 988
`http://www.xml-benchmark.org`		
Treebank	83 MB	3 829 512
`http://www.cs.washington.edu/research/xmldatasets`		

Figure 12: Benchmark datasets used in experiments.

#	data	Hits	Source	xpath
D1	DBLP	6	[12]	//inproceedings[author/text()="Jim Gray"][year/text()="1990"]/@key
D2	DBLP	21	[12]	//www[editor]/url
D3	DBLP	13	[12]	//book/author[text()="C. J. Date"]
D4	DBLP	2	[12]	//inproceedings[title/text()="Semantic Analysis Patterns."]/author
X1	XMark	40 726	[5]	/site/closed_auctions/closed_auction/annotation/description/text/keyword
X2	XMark	124 843	[5]	//closed_auction//keyword
X3	XMark	124 843	[5]	/site/closed_auctions/closed_auction//keyword
X4	XMark	40 726	[5]	/site/closed_auctions/closed_auction[annotation/description/text/keyword]/date
X5	XMark	124 843	[5]	/site/closed_auctions/closed_auction[.//keyword]/date
X6	XMark	32 242	[5]	/site/people/person[profile/gender][profile/age]/name
T1	Treebank	1 183	[11]	//S/VP//PP[.//NP/VBN]/IN
T2	Treebank	152	[11]	//S/VP/PP[IN]/NP/VBN
T3	Treebank	381	[11]	//S/VP//PP[.//NN][.//NP[.//CD]/VBN]/IN
T4	Treebank	1 185	[11]	//S[.//VP][.//NP]/VP/PP[IN]/NP/VBN
T5	Treebank	94 535	[11]	//EMPTY[.//VP/PP//NNP][.//S[.//PP//JJ]//VBN]//PP/NP//_NONE_

Figure 13: Benchmark queries used in experiments.

G Extended Results

Figure 14 shows the timings that the aggregates in Section 5 are based on.

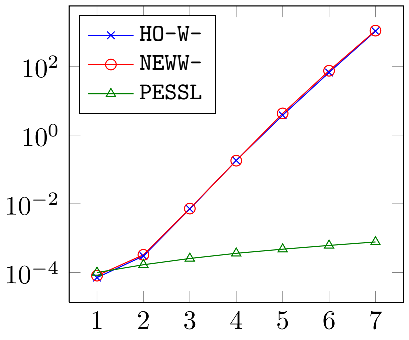
	D1	**D2**	**D3**	**D4**	**X1**	**X2**	**X3**	**X4**	**X5**	**X6**	**T1**	**T2**	**T3**	**T4**	**T5**
`HO---`	2.53	0.43	1.07	1.59	0.57	0.11	0.11	0.75	0.25	0.25	0.31	0.30	0.40	0.47	0.50
`HO-W-`	1.42	0.25	0.44	1.12	0.43	0.10	0.11	0.60	0.24	0.21	0.18	0.17	0.24	0.32	0.32
`HO-S-`	1.42	0.26	0.44	1.11	0.43	0.10	0.11	0.61	0.24	0.21	0.17	0.18	0.24	0.32	0.33
`HO-SL`	1.70	0.28	0.48	1.22	0.46	0.10	0.11	0.65	0.26	0.23	0.18	0.19	0.24	0.34	0.33
`HOW--`	1.96	0.31	0.32	1.18	0.34	0.08	0.09	0.49	0.19	0.25	0.22	0.22	0.32	0.42	0.42
`HOWW-`	1.26	0.19	0.33	0.92	0.32	0.08	0.08	0.46	0.19	0.21	0.16	0.17	0.22	0.31	0.30
`HOWS-`	1.34	0.19	0.32	0.91	0.31	0.08	0.08	0.45	0.19	0.21	0.16	0.16	0.21	0.30	0.32
`HOWSL`	1.31	0.20	0.31	0.96	0.31	0.09	0.09	0.46	0.20	0.23	0.17	0.17	0.23	0.33	0.32
`HOS--`	2.03	0.31	0.32	1.17	0.32	0.08	0.08	0.47	0.19	0.25	0.21	0.19	0.27	0.35	0.40
`HOSW-`	1.27	0.20	0.31	0.91	0.30	0.08	0.08	0.44	0.19	0.21	0.16	0.15	0.21	0.29	0.29
`HOSS-`	1.27	0.21	0.32	0.92	0.30	0.08	0.08	0.44	0.19	0.21	0.17	0.15	0.21	0.30	0.30
`HOSSL`	1.32	0.20	0.31	0.97	0.30	0.08	0.08	0.46	0.19	0.23	0.17	0.18	0.23	0.34	0.31
`HE---`	2.46	0.40	1.07	1.48	0.55	0.09	0.09	0.70	0.20	0.23	0.30	0.29	0.36	0.45	0.48
`HE-W-`	2.73	0.40	1.25	1.67	0.72	0.09	0.10	0.89	0.21	0.27	0.35	0.34	0.43	0.50	0.54
`HE-S-`	2.74	0.40	1.25	1.66	0.72	0.09	0.10	0.89	0.21	0.27	0.34	0.34	0.43	0.49	0.54
`HE-SL`	3.14	0.42	1.47	1.83	0.80	0.09	0.10	0.97	0.23	0.32	0.37	0.40	0.45	0.54	0.58
`HEW--`	1.97	0.31	0.34	1.13	0.34	0.08	0.08	0.48	0.19	0.24	0.23	0.22	0.28	0.42	0.42
`HEWW-`	2.13	0.32	0.34	1.24	0.38	0.08	0.09	0.53	0.19	0.27	0.29	0.28	0.35	0.45	0.49
`HEWS-`	2.13	0.32	0.34	1.24	0.39	0.08	0.09	0.52	0.20	0.28	0.29	0.28	0.35	0.44	0.49
`HEWSL`	2.33	0.33	0.35	1.31	0.42	0.08	0.09	0.57	0.20	0.32	0.28	0.30	0.37	0.48	0.51
`HES--`	1.99	0.31	0.34	1.16	0.33	0.08	0.08	0.47	0.19	0.24	0.22	0.19	0.28	0.33	0.40
`HESW-`	2.15	0.32	0.34	1.26	0.36	0.08	0.09	0.51	0.20	0.28	0.25	0.21	0.31	0.35	0.45
`HESS-`	2.14	0.33	0.34	1.24	0.37	0.08	0.09	0.51	0.20	0.29	0.24	0.21	0.31	0.35	0.45
`HESSL`	2.35	0.33	0.36	1.33	0.40	0.08	0.09	0.55	0.21	0.34	0.26	0.24	0.36	0.38	0.48
`NOWW-`	0.96	0.22	0.09	0.71	0.49	0.11	0.15	0.85	0.34	0.30	0.08	0.08	0.16	0.34	0.22
`NE-W-`	1.03	0.25	0.08	0.93	0.59	0.12	0.15	0.99	0.40	0.33	0.08	0.09	0.17	0.34	0.27
`NE-S-`	1.03	0.25	0.08	0.89	0.68	0.12	0.16	1.10	0.39	0.35	0.08	0.09	0.17	0.34	0.29
`NE-SL`	1.06	0.26	0.08	0.92	0.77	0.12	0.16	1.20	0.42	0.36	0.09	0.09	0.17	0.34	0.29
`NEWW-`	0.94	0.23	0.08	0.72	0.51	0.11	0.14	0.87	0.35	0.31	0.07	0.07	0.15	0.31	0.23
`NEWS-`	0.95	0.23	0.08	0.72	0.54	0.11	0.15	0.90	0.36	0.32	0.08	0.08	0.15	0.32	0.24
`NEWSL`	0.95	0.23	0.08	0.73	0.55	0.11	0.15	0.93	0.37	0.33	0.08	0.08	0.15	0.32	0.24
`NESW-`	0.95	0.23	0.08	0.74	0.49	0.11	0.14	0.87	0.36	0.31	0.07	0.07	0.15	0.30	0.23
`NESS-`	0.93	0.23	0.08	0.72	0.51	0.11	0.15	0.89	0.36	0.32	0.08	0.07	0.15	0.31	0.23
`NESSL`	0.94	0.23	0.08	0.73	0.54	0.11	0.15	0.92	0.37	0.33	0.08	0.08	0.15	0.31	0.24
`PEWW-`	0.22	0.04	0.08	0.05	0.33	0.06	0.09	0.43	0.16	0.20	0.06	0.06	0.04	0.13	0.10
`PEWS-`	0.22	0.04	0.08	0.05	0.34	0.06	0.09	0.44	0.16	0.21	0.06	0.06	0.04	0.13	0.10
`PEWSL`	0.22	0.04	0.08	0.05	0.36	0.06	0.09	0.49	0.18	0.22	0.06	0.06	0.05	0.13	0.10
`PESW-`	0.22	0.04	0.08	0.05	0.31	0.06	0.09	0.45	0.18	0.20	0.06	0.06	0.04	0.12	0.10
`PESS-`	0.22	0.04	0.08	0.05	0.33	0.07	0.09	0.43	0.16	0.21	0.06	0.06	0.04	0.13	0.10
`PESSL`	0.22	0.04	0.08	0.05	0.35	0.06	0.10	0.44	0.17	0.22	0.06	0.06	0.05	0.13	0.10

Figure 14: Time for all tested methods on all benchmark queries. All queries were warmed up by 3 runs and then run 100 times, or until at least 10 seconds had passed, measuring average running-time.

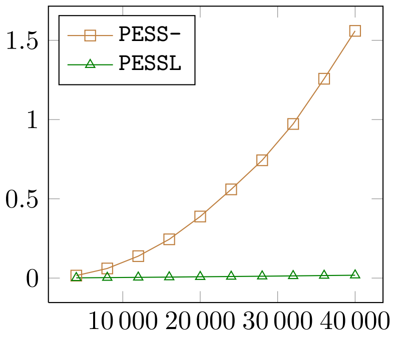
H Bad Behavior

In this appendix we list some experiments showing the super-linear behavior of previous twig join algorithms.

Figure 15a shows the *exponential* behavior of TwigList and TwigList (`HO-W-`) and TwigFast (`NEWW-`) with the data and query from Example 1. The data is $/(\alpha_1/)^n \ldots (\alpha_m/)^n \beta/\gamma$ with $m = 10$ and $n = 100$, and the query is $/\!/\alpha_1/\!/ \ldots /\!/\alpha_k/\gamma$, with k varying from 1 to 7.



(a) Varying query parameter k.



(b) Varying total nodes.

Figure 15: (a) Exponential behavior without strict matching. (b) Quadratic behavior without optimal enumeration. Query time in seconds.

Figure 15b shows the results of an experiment based on Example 2 with varying $n = 10\,000$. The simple query $/\!/$`a`/`b` has quadratic cost even when strict prefix path and subtree match filtering is used, if P–C child matches are not directly accessible. Many of the `a` nodes in the data are nested, and have a small number of `b` children, but a large number of `b` descendants. For approaches using simple vectors, overlapping descendant intervals are scanned for direct children, and this results in quadratic running time.

Appendix B.5.2

Paper VI

Linear Computation of the Maximum Simultaneous Forward and Backward Bisimulation for Node-Labeled Trees

Nils Grimsmo, Truls A. Bjørklund and Magnus Lie Hetland

XML Symposium (XSym), workshop at VLDB
2010

Abstract

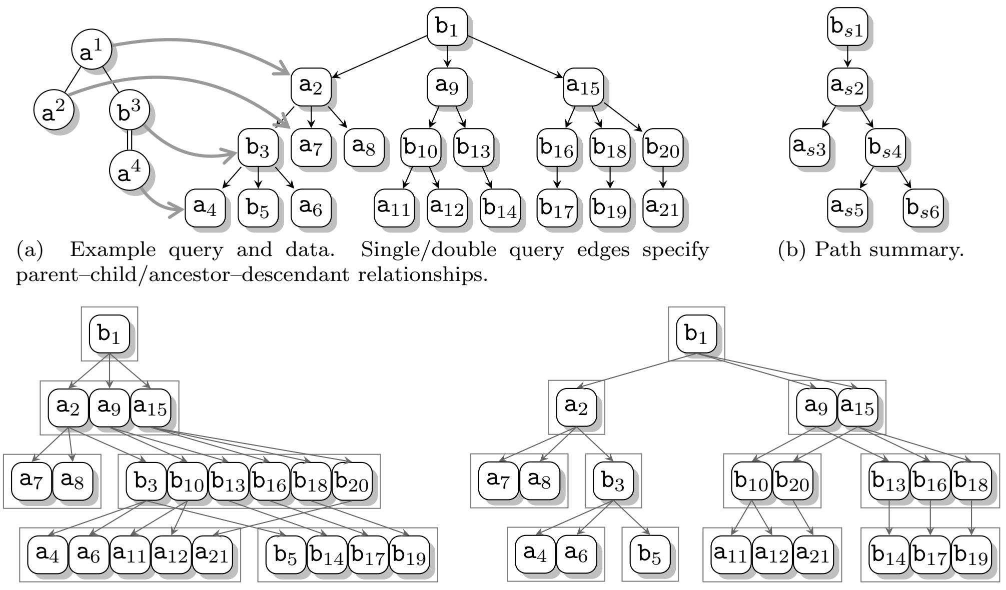
The F&B-index is used to speed up pattern matching in tree and graph data, and is based on the maximum F&B-bisimulation, which can be computed in loglinear time for graphs. It has been shown that the maximum F-bisimulation can be computed in linear time for DAGs. We build on this result, and introduce a linear algorithm for computing the maximum F&B-bisimulation for tree data. It first computes the maximum F-bisimulation, and then refines this to a maximal B-bisimulation. We prove that the result equals the maximum F&B-bisimulation.

1 Introduction

Structure indexes are used to reduce the cost of pattern matching in labeled trees and graphs [5, 14, 9], by capturing structure properties of the data in a *structure summary*, where some or all of the matching can be performed. Efficient construction of such indexes is important for their practical usefulness [14], and in this paper we reduce the construction cost of the *F&B-index* [9] for tree data.

In a structure index, data nodes are typically *partitioned* into *blocks* based on properties of the surrounding nodes. A structure summary typically has one node per block in the partition, and edges between summary nodes where there are edges between data nodes in the related blocks. Matching in structure summaries is usually more efficient than partitioning the data nodes on label and using structural joins to find full query matches [14, 9].

In a *path index*, data nodes are classified by the labels on the paths by which they are reachable [5, 14]. For tree data this equals partitioning nodes on their label and the block of the parent node. Figures 1c and 1b show path partitioning and the related summary for the example data in Figure 1a. With path indexes, non-branching queries can be evaluated without processing joins [14, 18].


(a) Example query and data. Single/double query edges specify parent–child/ancestor–descendant relationships.

(b) Path summary.

(c) Path partition.

(d) F&B partition.

Figure 1: Partitioning strategies. Superscripts and subscripts give node identifiers.

A natural extension of a path index is the F&B-index, where nodes are partitioned on both their label, the partitions of the parents, *and* the partitions of the children, as shown in Figure 1d. This gives an index where more of the pattern matching can be performed on the summary, and also branching queries can be evaluated without processing joins [9].

The focus of this paper is efficient computation of the maximum *simultaneous forward and backward bisimulation* (F&B-bisimulation), which is the underlying concept used to partition nodes in the F&B-index. It can be computed in time loglinear in the number of edges in the graph [9]. A linear construction algorithm for directed acyclic graphs (DAGs) has been presented recently [12], but we show that it is incorrect. On the other hand, there exists a correct algorithm which can compute *either* the maximum *forward bisimulation* (F-bisimulation) *or* the maximum *backward bisimulation* (B-bisimulation) in linear time for DAGs [3]. We extend this algorithm to compute the maximum F&B-bisimulation in linear time for tree data. This has relevance for applications where the underlying data is known to be tree shaped, such as in many uses of XML [6].

2 Background

In this section we present different types of bisimulations, and show how they can be computed by first partitioning on label, and then *stabilizing* the graph.

We use the following notation: A directed graph $G = \langle V, E \rangle$ has node set V and edge set $E \subseteq V \times V$. Let $n = |V|$ and $m = |E|$. For $X \subseteq V$, $E(X) = \{w \mid \exists v \in X : vEw\}$ and $E^{-1}(X) = \{u \mid \exists v \in X : uEv\}$. Each node $v \in V$ has label $L(v)$. A *partition* P of V is a set of *blocks*, such that each node $v \in V$ is contained in exactly one block. For an equivalence relation $\sim \subseteq V \times V$, the equivalence class containing $v \in V$ is denoted by $[v]_\sim$. The equivalence relation arising from the partition P is denoted $=_P$. A relation R_2 is a *refinement* of R_1 iff $R_2 \subseteq R_1$. A partition P_2 is a refinement of the *coarser* P_1 iff $=_{P_2} \subseteq =_{P_1}$. Let the *contraction graph* of a partition P be a graph with one node for each equivalence class of $=_P$, and an edge $\langle [u]_{=_P}, [v]_{=_P} \rangle$ whenever $\langle u, v \rangle \in E$.

The structure summary built for a structure index is typically isomorphic with the contraction graph for the data partition. For a partitioning to be useful, it must yield a summary that somehow simulates the data, such that pattern matching in the summary gives the same results as pattern matching in the data, or at least no false negatives. If nodes are partitioned into blocks where nodes in some way simulate each other, then the contraction graph also simulates the data in the same way.

2.1 Bisimulation and Bisimilarity

Broadly speaking, bisimulations relate nodes that have the same label and related neighbors. We use the following properties of a binary relation $R \subseteq V \times V$ to formally define the different types of bisimulation:

$$\begin{aligned}
vRv' &\Rightarrow L(v) = L(v') && (1)\\
vRv' &\Rightarrow (uEv \Rightarrow \exists u' : u'Ev' \wedge uRu') \wedge \\
&\quad\; (u'Ev' \Rightarrow \exists u : uEv \wedge uRu') && (2)\\
vRv' &\Rightarrow (vEw \Rightarrow \exists w' : v'Ew' \wedge wRw') \wedge \\
&\quad\; (v'Ew' \Rightarrow \exists w : vEw \wedge wRw') && (3)
\end{aligned}$$

Definition 1 (Bisimulations [13, 9])**.** A relation R is a B-bisimulation iff it satisfies (1) and (2) above, an F-bisimulation iff it satisfies (1) and (3), and an F&B-bisimulation iff it satisfies (1), (2) and (3).

For each type, there exists a unique maximum bisimulation, of which all other bisimulations are refinements [13, 9]. We say that two nodes are *bisimilar* if there exists a bisimulation that relates them, i.e., they are related by the maximum bisimulation. Since bisimilarity is an equivalence relation, it can be used to partition the nodes [13, 9]. When nodes u and v are backward, forward, and forward and backward bisimilar, we write $u \sim_B v$, $u \sim_F v$ and $u \sim_{F\&B} v$, respectively. Figure 2 illustrates the different types of bisimilarity.

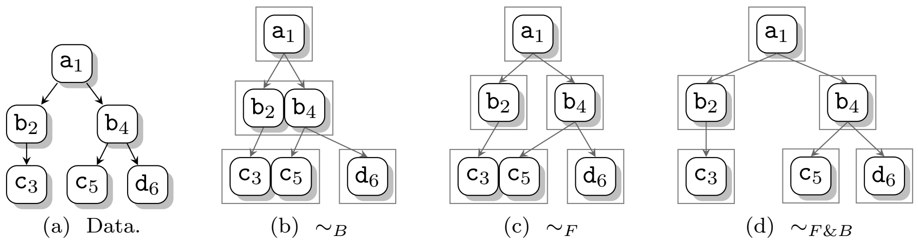


Figure 2: Partitioning on different types of bisimilarity.

Two *graphs* are said to be bisimilar if there exists a total surjective bisimulation from the nodes in one graph to the nodes in the other. For a given graph, the smallest bisimilar graph is unique, and is exactly the contraction for bisimilarity [3]. The F&B-bisimilarity contraction is the basis for the F&B-index [9].

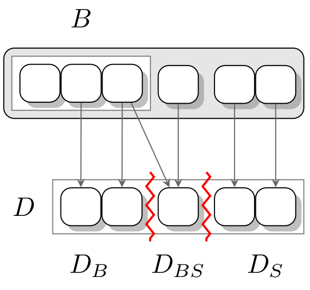
2.2 Stability

The different types of bisimilarity listed in the previous section can be computed by first partitioning the data nodes on label, and then finding the coarsest *stable* refinements of the initial partition [15, 4, 9]. A partition is *successor stable* iff all nodes in a block have incoming edges from nodes in the same set of blocks, and is *predecessor stable* iff all nodes in a block have outgoing edges to the same set of blocks [9]. The coarsest successor, predecessor, and successor and predecessor stable refinement of a label partition, equal a partition on B-bisimilarity, F-bisimilarity and F&B-bisimilarity, respectively [4, 9].

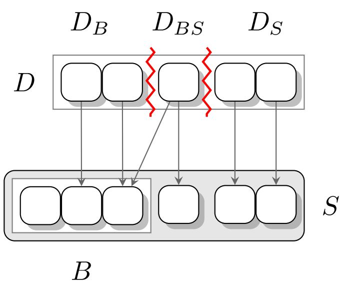
Definition 2 (Partition stability [15, 9])**.** Given a directed graph $G = \langle V, E \rangle$, then $D \subseteq V$ is *successor stable* with respect to $B \subseteq V$ if either all or none of the nodes in D are pointed to from nodes in B (meaning $D \subseteq E(B)$ or $D \cap E(B) = \emptyset$), and D is *predecessor stable* with respect to B if either none or all of the nodes in D point to nodes in B (meaning $D \subseteq E^{-1}(B)$ or $D \cap E^{-1}(B) = \emptyset$).

For any combination of successor and predecessor stability, a partition P of V is said to be stable with respect to a block B if all blocks in P are stable with respect to B. A partition P is stable with respect to another partition Q if it is stable with respect to all blocks in Q. P is said to be stable if it is stable with respect to itself.

Figure 3 shows cases where a block can be split to achieve different types of stability. The block D is not stable with respect to B, but we can split it into blocks that are: Assume that D is stable with respect to a union of blocks S such that $B \subset S$. We can split D into blocks that are stable with respect to both B and $S \setminus B$, shown as D_B, D_{BS} and D_S in the figure. Stabilizing also with respect to $S \setminus B$ is crucial for obtaining a $\mathcal{O}(m \log n)$ running time in the partition stabilization algorithm explained in the next section.



(a) Succ-stability.



(b) Pred-stability.

Figure 3: Refining D into blocks that are stable with respect to B and $S \setminus B$.

2.3 Stabilizing graph partitions

We now go through Paige and Tarjan's algorithm for refinement to the coarsest predecessor stable partition [15], extended to simultaneous successor and predecessor stability by Kaushik et al. [9], as shown in Algorithm 1. The input to the algorithm is a partition P and the set of flags *Directions* $\subseteq \{\textsc{Succ}, \textsc{Pred}\}$, which specifies whether P is to be successor and/or predecessor stabilized.

Figure 4 illustrates an example run of the algorithm with *Directions* $= \{\textsc{Succ}, \textsc{Pred}\}$. In addition to the current partition P, the algorithm maintains a partition X, where the blocks are unions of blocks in P. Initially X contains a single block that is the union of all blocks in P, and the algorithm maintains the loop invariant that P is stable with respect to X by Definition 2. The algorithm terminates when the partitions P and X

Algorithm 1 Graph partition stabilization.

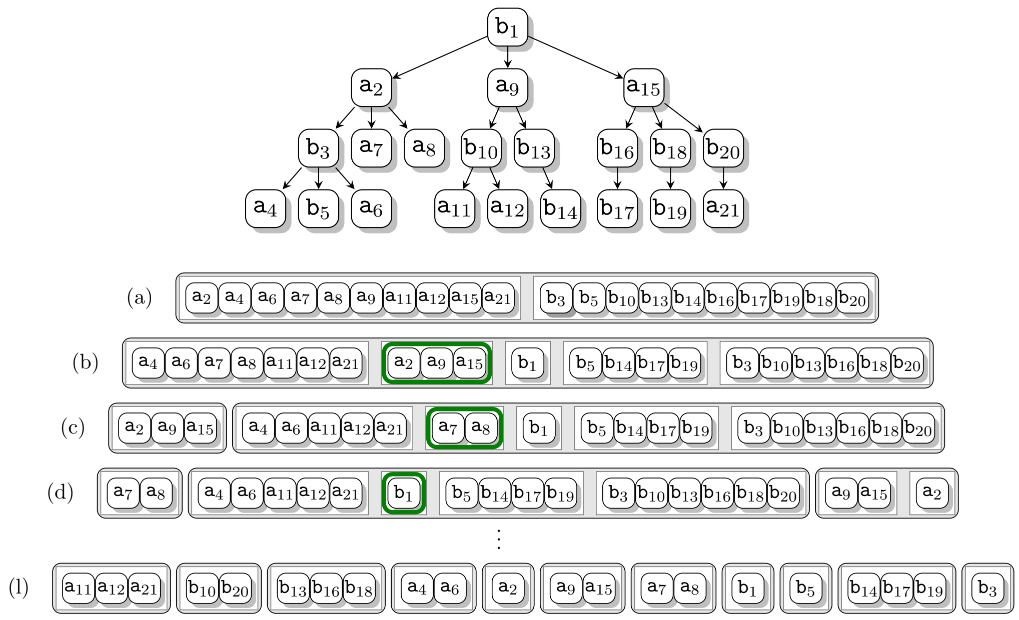
```
1: ▷ $P$ is the initial partition.
2: function StabilizeGraph($P$, $Directions$):
3:     for $dir \in Directions$:
4:         InitialRefinement($P$, $dir$)
5:     $X \leftarrow \{\bigcup_{B \in P} B\}$
6:     while $P \neq X$:
7:         Extract $B \in P$ such that $B \subset S \in X$ and $|B| \leq |S|/2$.
8:         Replace $S$ by $B$ and $S \setminus B$ in $X$.
9:         for $dir \in Directions$:
10:            StabilizeWRT(copy of $B, P, dir$)
```

are equal, which means P must be stable with respect to itself. But the loop invariant may not be true for the given input partition initially: Blocks containing both roots and non-roots are not successor stable with respect to X, because non-roots have incoming edges from the single block $S \in X$, while roots do not. Similarly, blocks containing both leaves and non-leaves are not predecessor stable with respect to X. In Algorithm 1 initial stability is achieved by calls to InitialRefinement(), which splits blocks in a simple linear pass. Initial splitting is illustrated by the step from line (a) to line (b) in Figure 4.

The algorithm repeatedly selects a block $S \in X$ that is a *compound* union of blocks from P, and selects a *splitter block* $B \subset S$ with size at most half of S. Then S is replaced by B and $S \setminus B$ in X, as shown when extracting $B = \{\mathsf{a}_2, \mathsf{a}_9, \mathsf{a}_{15}\}$ between lines (b) and (c) in Figure 4. The call to StabilizeWRT() uses the strategies depicted in Figure 3 to stabilize P with respect to both B and $S \setminus B$, to make sure P is stable with respect to the new X. It is important to use a *copy* of B as splitter, as the stabilization may cause B itself to be split. The step from line (b) to (c) in the figure shows that a block of nodes labeled a is split when successor stabilizing with respect to $B = \{\mathsf{a}_2, \mathsf{a}_9, \mathsf{a}_{15}\}$.

Efficient implementation of the above requires some attention to detail [15]: The partition X can be realized through a set $\mathcal{X}$ containing sets of pointers to blocks in P, such that for each $S \in X$ we have $S = \bigcup_{B \in \mathcal{S}} B$ for the related $\mathcal{S} \in \mathcal{X}$. To extract a $B \subset S \in X$ in constant time, we also maintain the set of compound unions $\mathcal{C} = \{\mathcal{S} \in \mathcal{X} \mid 1 < |\mathcal{S}|\}$. A block $B \subset S$ such that $|B| \leq |S|/2$ can be found by choosing the smalllest of the two first blocks in any $\mathcal{S} \in \mathcal{C}$. Note that we can check if $P = X$ by checking whether $\mathcal{C}$ is empty, and neither P, X nor $\mathcal{X}$ need to be materialized, as they are never used directly. You only need to maintain $\mathcal{C}$, and a $\mathcal{P} \subseteq P$ containing the blocks in P not in some $\mathcal{S} \in \mathcal{C}$. For inserting and removing elements in constant time, the sets are implemented as doubly linked list. In addition, each $v \in B$ has a pointer to B, and each $B \in \mathcal{S}$ has a pointer to $\mathcal{S}$. This allows keeping the data structures updated throughout the evaluation.

As all operations in the while-loop excluding the calls to StabilizeWRT() are constant time operations on linked lists, the complexity of the loop is bounded by the number of splitter blocks selected, which again is bounded by the number of times a node may be part of such a splitter. Splitter blocks at most half the size of a compound union are



b1
a2 a9 a15
b3 a7 a8 b10 b13 b16 b18 b20
a4 b5 a6 a11 a12 b14 b17 b19 a21
(a) a2 a4 a6 a7 a8 a9 a11 a12 a15 a21 | b3 b5 b10 b13 b14 b16 b17 b19 b18 b20
(b) a4 a6 a7 a8 a11 a12 a21 | a2 a9 a15 | b1 | b5 b14 b17 b19 | b3 b10 b13 b16 b18 b20
(c) a2 a9 a15 | a4 a6 a11 a12 a21 | a7 a8 | b1 | b5 b14 b17 b19 | b3 b10 b13 b16 b18 b20
(d) a7 a8 | a4 a6 a11 a12 a21 | b1 | b5 b14 b17 b19 | b3 b10 b13 b16 b18 b20 | a9 a15 | a2
⋮
(l) a11 a12 a21 | b10 b20 | b13 b16 b18 | a4 a6 | a2 | a9 a15 | a7 a8 | b1 | b5 | b14 b17 b19 | b3

Figure 4: Algorithm 1 doing successor and predecessor stabilization on a label partition of the data from Figure 1a. The white boxes are the current blocks in P, while the gray boxes are the current blocks in X. Line (a) shows initial label partition. Step (a)–(b) shows refinement separating roots from non-roots and leaves from non-leaves, and steps (b)–(l) show simultaneous predecessor and successor stabilization.

always selected, and no node in this block is part of a splitter again before the block itself has become a compound union. This means that the number of times a given node is part of a splitter block is $\mathcal{O}(\log n)$, and that the total number of splitter blocks used is $\mathcal{O}(n \log n)$ [15].

Algorithm 2 shows how all blocks in a partition P can be successor (or predecessor) stabilized with respect to a block $B \in P$ and $S \setminus B$, where $B \subset S \in X$, in time linear in the number of edges going out from (or into) B [15]. For successor stability, only blocks D pointed to from B are affected, and they are stabilized with respect to both B and $S \setminus B$ without involving $S \setminus B$ directly. This is done by maintaining for each node $v \in V$, the number of times it is pointed to from each set $S \in X$, and storing references to these records from the related edges. We can then differentiate between nodes pointed to from B, pointed to from both B and $S \setminus B$, and pointed to only from $S \setminus B$. Nodes from the first two categories are moved into new blocks, while the rest are untouched.

As the cost of a single call to StabilizeWRT() is bounded by the number of nodes in the splitter block and the number of outgoing (or incoming) edges, the total cost for the calls is $\mathcal{O}((m+n)\log n)$, as a given node or edge is used for splitting $\mathcal{O}(\log n)$ times. Assuming $n \in \mathcal{O}(m)$, the cost of Algorithm 1 is $\mathcal{O}(n)$ for the initial refinement, $\mathcal{O}(n \log n)$ for the

Algorithm 2 Stabilizing with respect to a block

```
1:  function StabilizeWRT($B, P, d$):
2:      Assume $dir$ = Succ.                                   (or Pred)
3:      for $D \in P$ pointed to from $B$:                     (or pointing into $B$)
4:          Initialize $D_B$ and $D_{BS}$ and associate with $D$.
5:      for $v \in D \in P$ pointed to from $B$:               (or pointing into $B$)
6:          if $v$ is pointed to only from $B$:                (or pointing only into $B$)
7:              $D' \leftarrow D_B$
8:          else:
9:              $D' \leftarrow D_{BS}$
10:         if $D' \notin P$: Insert $D'$ after $D$ in $P$
11:         Move $v$ from $D$ to $D'$.
12:         if $D = \emptyset$: Remove $D$ from $P$.
```

while-loop excluding StabilizeWRT(), and $\mathcal{O}(m \log n)$ for the StabilizeWRT() calls, giving a total of $\mathcal{O}(m \log n)$ [15, 9].

3 Linear Time Stabilization

Linear time computation of F&B-bisimilarity for DAG data has been attempted earlier. The SAM algorithm [12] partitions the data separately on B-bisimilarity and F-bisimilarity, and then combines the partitions by putting two nodes in the same final block iff they are in the same blocks in both partitions. It builds on the following theorem, which is stated without proof or reference: "*Node n_1 and node n_2 satisfy F&B-bisimulation if and only if they satisfy F-bisimulation and B-bisimulation.*" The *only if* part is of course true, but the *if* part is not, as can be seen from the partitioning of a tree with six nodes in Figure 2. Here $c_3 \sim_B c_5$ and $c_3 \sim_F c_5$, but $c_3 \not\sim_{F\&B} c_5$, because for the parent nodes $b_2 \not\sim_{F\&B} b_4$. Also note that it is assumed for the running time of the SAM algorithm that the number of edges to and from each node can be viewed as a constant.

3.1 Stabilizing DAG partitions

We now present an algorithm for refining a partition of the nodes in a DAG *either* to successor stability *or* to predecessor stability. It is based on two different previous algorithms: Paige and Tarjan's loglinear algorithm for stabilizing general graphs [15], and Dovier, Piazza and Policriti's algorithm for computing F-bisimilarity on unlabeled graphs [3], which has linear complexity for DAG data. A difference between these two algorithms is that the former is given an initial partition as input, which is then *refined*, while the latter starts with the set of singleton blocks, from which the final partition is *constructed*. These are called *negative* and *positive* strategies, respectively [3]. Dovier et al. describe how their algorithm can be extended to compute F-bisimilarity for *labeled* data, but when developing an algorithm for refining to simultaneous successor and predecessor stability in the next section, we use the result of a predecessor stabilization as input to a successor stabilization, and hence cannot use a positive strategy.

Dovier et al.'s algorithm initially computes the *rank* of each node in the DAG, which is the length of the longest path from the node to a leaf. We extend the notion of rank to both directions in the DAG:

Definition 3 (Rank)**.** In a DAG G, the successor and predecessor rank of $v \in V(G)$ is defined as:

$$rank_{\text{SUCC}}(v) = \begin{cases} 0 & \text{if } v \text{ is a root in } G \\ 1 + \max_{\langle u,v\rangle \in E(G)} rank_{\text{SUCC}}(u) & \text{otherwise} \end{cases}$$

$$rank_{\text{PRED}}(v) = \begin{cases} 0 & \text{if } v \text{ is a leaf in } G \\ 1 + \max_{\langle v,w\rangle \in E(G)} rank_{\text{PRED}}(w) & \text{otherwise} \end{cases}$$

Algorithm 3 shows our modification of Paige and Tarjan's algorithm [15] based on Dovier et al.'s principles [3]. It refines a partition of a DAG *either* to predecessor *or* to successor stability, and runs in linear time, due to the order in which splitter blocks are chosen.

Algorithm 3 DAG partition stabilization.

```
1: ▷ Assume sets are ordered.
2: function StabilizeDAG($P$, $dir$):
3:     RefineAndSortOnRank($P$, $dir$)
4:     $X \leftarrow \{\bigcup_{B \in P} B\}$
5:     while $P \neq X$:
6:         Extract first $B \in P$ such that $B \subset S \in X$.
7:         Replace $S$ by $B$ and $S \setminus B$ in $X$.
8:         StabilizeWRT($B$ copy, $P$, $dir$)
```

A run of the algorithm with *dir* = PRED is illustrated in Figure 5. Instead of only separating between leaves and non-leaves (or roots and non-roots) as in Algorithm 1, blocks are initially split such that predecessor (or successor) rank is uniform within each block, and sorted, such that the rank is monotonically increasing in the partition. This is done in the function RefineAndSortOnRank(), which is described later in this section. An initial refinement and sorting on predecessor rank is shown when going from line (a) to line (b) in Figure 5.

In a partition that respects a given type of rank, let the rank of a block be equal to the rank of the contained nodes. The following lemma implies that the initial refinement on rank does not split blocks unnecessarily:

Lemma 1. *Given nodes $u, v \in V(G)$ and a partition P of G, if P is successor stable then $[u]_{=_P} = [v]_{=_P} \Rightarrow rank_{\text{SUCC}}(u) = rank_{\text{SUCC}}(v)$, and if P is predecessor stable then $[u]_{=_P} = [v]_{=_P} \Rightarrow rank_{\text{PRED}}(u) = rank_{\text{PRED}}(v)$.*

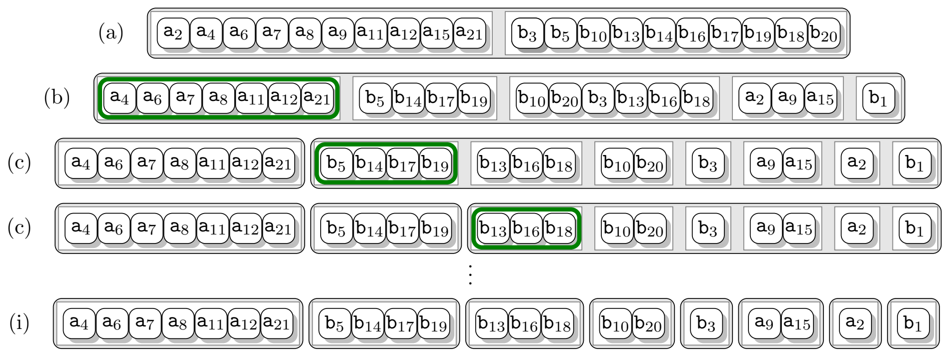

(a)
a_2 a_4 a_6 a_7 a_8 a_9 a_{11} a_{12} a_{15} a_{21}
b_3 b_5 b_{10} b_{13} b_{14} b_{16} b_{17} b_{19} b_{18} b_{20}
(b)
a_4 a_6 a_7 a_8 a_{11} a_{12} a_{21}
b_5 b_{14} b_{17} b_{19}
b_{10} b_{20} b_3 b_{13} b_{16} b_{18}
a_2 a_9 a_{15}
b_1
(c)
a_4 a_6 a_7 a_8 a_{11} a_{12} a_{21}
b_5 b_{14} b_{17} b_{19}
b_{13} b_{16} b_{18}
b_{10} b_{20}
b_3
a_9 a_{15}
a_2
b_1
(c)
(i)

Figure 5: Using Algorithm 3 to predecessor stabilize a label partition of the data from Figure 1a. The first step shows refinement on predecessor rank.

Proof. For F-bisimilarity $[u]_{\sim_F} = [v]_{\sim_F} \Rightarrow rank_{\text{PRED}}(u) = rank_{\text{PRED}}(v)$ [3]. If P is predecessor stable, then $=_P$ is an F-bisimulation [4], and therefore a refinement of partitioning on $\sim_F$, such that $[u]_{=_P} = [v]_{=_P} \Rightarrow [u]_{\sim_F} = [v]_{\sim_F}$. The case for successor stability is symmetric. $\square$

The next lemma implies that if blocks are chosen as splitters in order of their rank, a node will be part of a block that is used as a splitter at most once. This property is used to achieve a linear complexity in our algorithm.

Lemma 2 ([3])**.** Given a DAG G, a partition P of G that respects predecessor rank and a block $B \in P$, predecessor stabilization of P with respect to B only splits blocks D where $rank_{\text{PRED}}(D) > rank_{\text{PRED}}(B)$.

This is symmetric for successor stabilization and successor rank, as a reversed DAG is also a DAG.

In Algorithm 3 the blocks in the current partition P are maintained ordered on rank. This is implemented through a detail in our method for stabilization with respect to a block in Algorithm 2, which is different from the original description [15]: The new blocks D_B and D_{BS} are inserted into P on the position after the old block D, and not at the end of P. The sets of blocks which make up the unions in X are also ordered such that their concatenation yields an ordered list of blocks. This is maintained by inserting B followed by $S \setminus B$ on the original position of S in X. Notice how blocks never change positions during the stabilization shown in lines (b)–(i) in Figure 5.

In Dovier et al.'s algorithm, rank is computed by performing a topological traversal of the DAG depth first [3]. Because we need to *refine* a given partition on rank, as opposed to *constructing* a partition on rank from scratch, the problem is slightly more involved. Algorithm 4 shows how a partitioning can be refined and sorted on successor or predecessor rank in a single pass. The algorithm traverses the DAG with a hybrid between a topological sort and a breadth first search, implemented using edge counters and a queue. Blocks are refined and sorted on the fly.

Algorithm 4 Refining and sorting on rank.

```
function RefineAndSortOnRank($P$, $dir$):
    Assume $dir = \textsc{Succ}$                                   (or $dir = \textsc{Pred}$)
    $Q$ is a queue.
    for $v \in V$:
        $v.count \leftarrow |\{x \mid \langle x, v\rangle \in E\}|$          (or $|\{x \mid \langle v, x\rangle \in E\}|$)
        if $v.count = 0$:
            $v.rank_{dir} \leftarrow 0$
            PushBack($Q, v$)
    for $B \in P$:
        $B.currRank \leftarrow -1$
    while $Q$:
        $v \leftarrow$ PopFront($Q$)
        Let $B \ni v$.
        if $v.rank_{dir} \neq B.currRank$:
            $B.rankedB \leftarrow \{\}$
            Append $B.rankedB$ at the end of $P$.
            $B.currRank \leftarrow v.rank_{dir}$
        Move $v$ from $B$ to $B.rankedB$
        Remove $B$ from $P$ if empty.
        for $x$ where $\langle v, x\rangle \in E$:                    (or $\langle x, v\rangle \in E$)
            $x.count \leftarrow x.count - 1$
            if $x.count = 0$:
                $x.rank_{dir} \leftarrow v.rank_{dir} + 1$
                PushBack($Q, x$)
```

Lemma 3. Algorithm 4 refines and orders P on successor (or predecessor) rank in $\mathcal{O}(m+n)$ time.

Proof (sketch). Because the queue is initialized with the roots, and a node is added to the queue when the last parent is popped from the queue, the nodes are queued and popped in order of successor rank, and this distance is calculated from the parent node with the greatest successor rank. As the successor rank of the nodes that are moved to a new block grows monotonically, only one associated block $B.rankedB$ is created per successor rank found in block B, and all such blocks are appended to P in sorted order. As a node is only queued once, and the cost of processing a node is proportional to the number of outgoing edges, the total running time of the algorithm is $\mathcal{O}(m+n)$. The case for predecessor rank is symmetric. □

Theorem 1. Algorithm 3 yields the coarsest refinement of a partition of a DAG that is successor (or predecessor) stable.

Proof (sketch). For predecessor stability, the only differences from Paige and Tarjan's algorithm are the initial refinement and the order in which splitter blocks are chosen.

By Lemma 1 blocks are not refined unnecessarily when refininig on rank, and the order of split operations is not used in the correctness proof for the original algorithm [15]. Successor stability is symmetric. □

To implement Algorithm 3 we use the same underlying data structures as for Algorithm 1: doubly linked lists $\mathcal{C}$ and $\mathcal{P}$ realizing X and P. In Algorithm 3, the extract operation is implemented by removing the first $B \in \mathcal{S}$ from the first $\mathcal{S} \in \mathcal{C}$. The replace operation is implemented by moving B from $\mathcal{S}$ to the end of $\mathcal{P}$, and if only one block B' is left in $\mathcal{S}$, this B' is also moved from $\mathcal{S}$ to $\mathcal{P}$, and $\mathcal{S}$ is removed from $\mathcal{C}$.

Theorem 2. The running time of Algorithm 3 is $\mathcal{O}(m+n)$.

Proof (sketch). We analyze the cost of StabilizeWRT() separately. Outside the while loop, the call to RefineAndSortOnRank() has cost $\mathcal{O}(m+n)$ by Lemma 3, and the construction of $\mathcal{C}$ and $\mathcal{P}$ has cost $\mathcal{O}(|P|) \subseteq \mathcal{O}(n)$. As splitters are chosen in order of their rank, by Lemma 2 a splitter block is not later split itself. This means that each node is only part of a splitter once, and that the while-loop is run $\mathcal{O}(n)$ times. The loop condition is implemented by checking if $\mathcal{C} \neq \emptyset$. As all the operations on linked lists inside the loop have complexity $\mathcal{O}(1)$, the total cost of the while-loop is $\mathcal{O}(n)$. The StabilizeWRT() function is called $\mathcal{O}(n)$ times, and the cost of one call is linear in the number of nodes and edges used [15]. As nodes are only part of a splitter block once, edges are also only used for splitting once, and the total cost is $\mathcal{O}(m+n)$. □

3.2 Stabilizing Trees

We now present an algorithm for finding the coarsest successor *and* predecessor stable refinement of the nodes in a tree. It uses the solution for DAGs from the previous section to refine a partition *first* to the coarsest predecessor stable refinement, and *then* to the coarsest successor stable refinement, as shown in Algorithm 5. For trees this yields a partition that is still predecessor stable, as we prove in the following.

Algorithm 5 Stabilization for trees

```
1: function StabilizeTree($P$, $Directions$):
2:     if Pred $\in$ $Directions$:
3:         StabilizeDAG($P$, Pred)
4:     if Succ $\in$ $Directions$:
5:         StabilizeDAG($P$, Succ)
```

Figure 6 shows with a continuation of Figure 5 how Algorithm 5 is used to find a successor and predecessor stable refinement. The starting point in the figure is a predecessor stable refinement found after calling StabilizeDAG(P, Pred). This partition is then successor stabilized by calling StabilizeDAG(P, Succ), which first refines and sorts on successor rank, shown between lines (i) and (j) in the figure, and then uses the blocks in the current P as splitters in order, shown in lines (j)–(t). Compare this partition with the F&B-bisimilarity partition in Figure 1d.

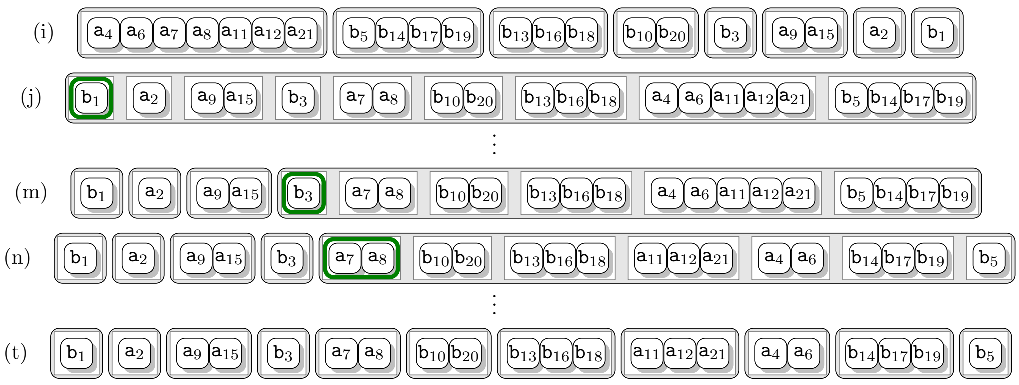


Figure 6: Continuing Fig. 5: Line (i) shows predecessor stable partition, step (i)–(j) shows successor rank refinement, and steps (j)–(t) show successor stabilization.

Theorem 3. If a predecessor stable partition of the nodes in a tree is refined to successor stability, the resulting partition is still predecessor stable.

Proof (sketch). Blocks are split in three ways: To refine on rank, with respect to a block $B \in P$, or as a side effect with respect to $S \setminus B$ in Algorithm 2. By Lemma 1, the first type of split does not cause any split that would not eventually be caused by an algorithm that iteratively refines P with respect to a random block $B \in P$, and from the correctness proof of Paige and Tarjan's algorithm [15], neither does splitting with respect to $S \setminus B$.

We now use induction on the refinement steps, and show that the partition P remains predecessor stable. It is true initially by assumption. The induction step is to split a block $D \in P$ on successor stability with respect to a block $B \in P$.

B will split D into two parts D_B and D_S, containing the nodes pointed to and not pointed to from B, respectively. The splitting of D may only affect the predecessor stability of the new blocks D_B and D_S with respect to their descendants, and of the set of blocks $\mathcal{B}$ pointing into D with respect to D_B and D_S. After the split, $B \subseteq E^{-1}(D_B)$ and $B \cap E^{-1}(D_S) = \emptyset$, and for all other $B' \in \mathcal{B}$ we have that $B' \cap E^{-1}(D_B) = \emptyset$ and $B' \subseteq E^{-1}(D_S)$, because these B' by assumption have pointers into D, but by the fact that the data is a tree, do not have pointers into D_B.

For any block G pointed to from some node in D, $D \subseteq E^{-1}(G)$ by the initial assumption of predecessor stability, meaning all nodes in D point into G. This means that all nodes in D_B and D_S also point into G, and thus D_B and D_S are predecessor stable with respect to G. $\square$

Figure 7a illustrates this theorem. By contrast, assuming the data was not a tree, the blocks $B' \in \mathcal{B}$ would not necessarily be predecessor stable after splitting D, as shown in Figure 7b.

Theorem 4. Algorithm 5 finds the coarsest refinement of a partition of the nodes in tree that is successor and predecessor stable in $\mathcal{O}(n)$ time.

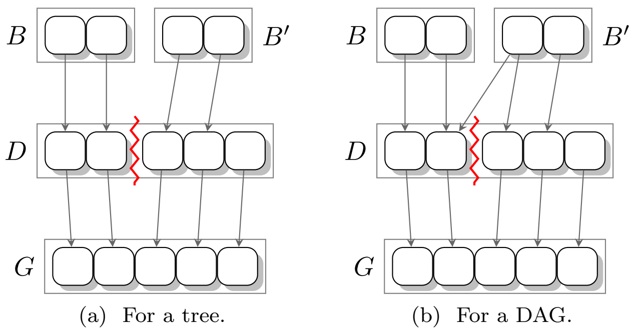


Figure 7: Splitting D for successor stability w.r.t. B in a partition P. This does not impact predecessor stability for any block given tree data, but may break predecessor stability in a DAG.

Proof. From Theorems 1, 2 and 3, and the fact that $m \in \mathcal{O}(n)$ for tree data. □

Corollary 1. The F&B-index can be built in $\mathcal{O}(n)$ time for tree data using Algorithm 5.

Proof (sketch). The coarsest successor and predecessor stable refinement of a partitioning on label gives the maximum F&B-bisimulation [9], and the summary structure can be constructed from the contraction graph [9]. □

4 Related Work

There are many variations of structure summaries for graph data, such as partitionings on similarity [8, 14], the F+B-index [9], the A(k) index [11], and the D(k)-index [2]. Note that an implication of Theorem 3 is that the F+B-index and the F&B-index are identical for tree data. The cost of matching in the summary can be reduced by label-partitioning it and using specialized joins [1], or using multi-level structure indexing [17]. For queries using ancestor–descendant edges on graph data, different graph encodings offer trade-offs between space usage and query time [20]. For a general overview of indexing and search in XML see the survey by Gou and Chirkova [6]. There is previous research on updates of bisimilarity partitions [10, 19, 16]. Some of these methods trade update time for coarseness, as refinements of bisimilarity may be cheaper to compute after a data update. Single directional bisimulations are used in many fields, such as modal logic, concurrency theory, set theory, formal verification, etc. [3], but to our knowledge, F&B-bisimulation is not frequently used outside XML search.

5 Conclusions and Future Work

In this paper we have improved the running time for refining a partition to the coarsest simultaneous successor and predecessor stability for tree data from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$, and with that the computation of F&B-bisimilarity, and construction of the F&B-index.[1] An incorrect linear algorithm for DAGs has been presented recently [12], and it would be interesting to know whether the problem is actually solvable in linear time for DAGs.

A natural extension of our work would be to reduce the cost of updates in F&B-bisimilarity partitions for trees. A particularly interesting direction would be to improve indexing performance for typical XML document collections, where there is a large number of small independent documents. It may be possible to iteratively add documents to the index with (expected) cost dependent only on the size of the documents.

Acknowledgments

This material is based upon work supported by the iAd Project funded by the Research Council of Norway and the Norwegian University of Science and Technology. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the funding agencies.

References

[1] Radim Bača, Michal Krátký, and Václav Snášel. On the efficient search of an XML twig query in large DataGuide trees. In *Proc. IDEAS*, 2008.

[2] Qun Chen, Andrew Lim, and Kian Win Ong. D(k)-index: an adaptive structural summary for graph-structured data. In *Proc. SIGMOD*, 2003.

[3] Agostino Dovier, Carla Piazza, and Alberto Policriti. A fast bisimulation algorithm. In *Proc. CAV*, 2001.

[4] Jean-Claude Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Sci. Comput. Program.*, 13(2-3), 1990.

[5] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proc. VLDB*, 1997.

[6] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *Knowl. and Data Eng.*, 2007.

[7] Nils Grimsmo, Truls Amundsen Bjørklund, and Magnus Lie Hetland. Linear computation of the maximum simultaneous forward and backward bisimulation for node-labeled trees (extended version). Technical Report IDI-TR-2010-10, NTNU, Trondheim, Norway, 2010.

[8] Monika R. Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *Proc. FOCS*, 1995.

[9] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *Proc. SIGMOD*, 2002.

[1]See the extended version of this paper for some performance experiments [7].

[10] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Pradeep Shenoy. Updates for structure indexes. In *Proc. VLDB*, 2002.

[11] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *Proc. ICDE*, 2002.

[12] Xianmin Liu, Jianzhong Li, and Hongzhi Wang. SAM: An efficient algorithm for F&B-index construction. In *Proc. APWeb/WAIM*, 2007.

[13] Robin Milner. *Communication and concurrency.* Prentice-Hall, Inc., 1989.

[14] Tova Milo and Dan Suciu. Index structures for path expressions. *Proc. ICDT*, 1999.

[15] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 1987.

[16] Diptikalyan Saha. An incremental bisimulation algorithm. In *Proc. FSTTCS*, 2007.

[17] Xin Wu and Guiquan Liu. XML twig pattern matching using version tree. *Data & Knowl. Eng.*, 2008.

[18] Beverly Yang, Marcus Fontoura, Eugene Shekita, Sridhar Rajagopalan, and Kevin Beyer. Virtual cursors for XML joins. In *Proc. CIKM*, 2004.

[19] Ke Yi, Hao He, Ioana Stanoi, and Jun Yang. Incremental maintenance of XML structural indexes. In *Proc. SIGMOD*, 2004.

[20] Jeffrey Xu Yu and Jiefeng Cheng. Graph reachability queries: A survey. In *Managing and Mining Graph Data*, Advances in Database Systems. Springer US, 2010.

Appendix C

Errata

After the publication of the papers in this thesis some errors have been identified. The errors are mainly typographical and do not have any impact on the conclusions of the respective papers nor the thesis as a whole. However, to avoid that the errors disturb the reader's understanding of the papers, an overview of the known errors is provided below.

Paper II:

In Figures $2(a_1)$, $2(a_4)$, $2(b_1)$, $2(b_4)$ in the published version, and Figures 4 and 7 in the submitted version, the x-axes read 2^1, 10^1, 10^3, 10^2, 10^4, and so on. This unintuitive ordering is a result of a typographical error, and the correct labeling should be 2^1, 10^1, 10^2, 10^3, 10^4, and so on. The underlying data follows the correct labeling.

Paper V:

The last sentence in the first paragraph of Section 4.4 ends with "data node at the head of its stream, implementing Comp. 2 in Figure 3.". This sentence should have referred to Comp. 1 in Figure 3 instead.

There are a few errors in the pseudo code in the Appendix of Paper V. Rather than providing a detailed list of the different errors, the corrected pseudo code is shown in Algorithms 1, 2 and 3. The functions with corrections are included, and the corrected lines have red font. The correction at Line 23 in Algorithm 2 fixes a bug in the original version that potentially could end processing before all results were found. To ensure that this special case plays no particular role in our test data, we have rerun the experiments after fixing the bug. Unfortunately, the compiler on our test computer has been updated since the original experiments were run, and the results are therefore not identical to those reported in the paper. However, the only significant differences are that the methods using getNext and those using getPart run slightly faster (typically 10-20%), while the other methods have similar results. The relative differences between getNext and getPart remain the same, and the conclusions in the paper are therefore not affected. All the other errors are typographical, and have no impact on the results.

Algorithm 1 TJStrictPost

```
function MarkEnd($q, u$):
    for $r \in q.children$:
        $u.end[r] \leftarrow$ GetVector($r, u.d.level + 1$)$.size$

function CheckChildMatch($q, u$):
    for $r \in q.children$:
        if $u.end[r] < u.start[r]$: return false
    return true
```

Algorithm 2 GetPart

```
function GetPart($q$):
    if $\neg$IsRoot($q$):
        $p \leftarrow$ Parent($q$)
        FwdToDescOf($q, M[p]$)
        if $\neg$Eof($q$) $\wedge \neg$Eof($p$) $\wedge$ $M[p].end < $ H$(q).end$:
            FwdToDescOf($q$, H($p$))
    if IsLeaf($q$):
        if Eof($q$): return (Eof, $q, \perp$)
        else: return (Match, $q$, H($q$))
    $(d_{min}, q_{min}) \leftarrow (\infty, \perp)$ ; $(d_{max}, q_{max}) \leftarrow (0, \perp)$
    for $r \in q.children$:
        $(c_r, d_r, q_r) \leftarrow$ GetPart($r$)
        if $c_r \neq$ Eof:
            if $c_r =$ MisMatch: flag MisMatch
            elif $q_r \neq r$: return (Match, $d_r, q_r$)
            if $d_r.begin < d_{min}.begin$: $(d_{min}, q_{min}) \leftarrow (d_r, q_r)$
            if $d_r.begin > d_{max}.begin$: $(d_{max}, q_{max}) \leftarrow (d_r, q_r)$
        else:
            FwdToEof($q$)
    if $q_{min} = \perp$: return (Eof, $\perp, q$)
    FwdToAncOf($q, d_{max}$)
    if flagged MisMatch:
        return (MisMatch, $\perp, q_r$)

    if $\neg$Eof($q$) $\wedge$ H$(q).begin < d_{min}.begin$:
        if IsRoot($q$) $\vee$ H$(q).end < M[p].end$:
            if $M[q].end <$ H$(q).end$: $M[q] \leftarrow$ H($q$)
        return (Match, H($q$), $q$)
    else:
        if $d_{min}.begin < M[q].end$:
            return (Match, $d_{min}, q_{min}$)
        else:
            if Eof($q$): return (Eof, $\perp, q$)
            else: return (MisMatch, $\perp, q$)
```

Algorithm 3 FilterPass

```
1:  **function** FilterPass($q$):
2:      **if** IsLeaf($q$): **return**
3:      **for** $r \in q.children$:
4:          FilterPass($r$)
5:      **if** NonLeafChildren($q$) $\neq \emptyset$:
6:          **for** $u \in$ AllNodes($q$):
7:              FilterPassPost($q, u.d$)
8:              **for** $r \in$ NonLeafChildren($q$):
9:                  $u.start[r] \leftarrow$ FwdIter($r, u.start[r], u.d$)
10:             Push($S_{local}[q], u$)
11:         FilterPassPost($q, \infty$)
12:     CleanUp($q$)

13: **function** FilterPassPost($q, d$):
14:     **while** $S_{local}[q] \neq \emptyset \wedge$ Top($S_{local}[q]$)$.end < d.end$:
15:         $u \leftarrow$ Pop($S_{local}[q]$)
16:         **for** $r \in$ NonLeafChildren($q$):
17:             $u.end[r] \leftarrow$ FwdIter($r, u.end[r], u.d$)
```