A Generic Merge-Based Dynamic Indexing Framework for iMeMex

MASTER THESIS
Sandro Blum

Supervised by:
Dr. Jens-Peter Dittrich
Marcos Antonio Vaz Salles
Lukas Blunschki

Systems Group, http://systems.ethz.ch/
Department of Computer Science
ETH Zurich

August 11, 2008
Abstract

A Personal Dataspace Management System must be able to handle highly dynamic dataspaces. This requires index structures that are not only capable of efficiently speeding up queries on these dataspaces but that can also be updated on-the-fly. State of the art text-retrieval systems are typically based on inverted file indices that are updated either in-place or with a merge-based approach. In this thesis we present a merge-based dynamic indexing framework for the iMeMex Dataspace Management System. The framework we present is generic, extensible and is based on abstract sub components. We provide several sub component implementations and evaluate the system experimentally. Furthermore we have studied analytically the three best-known merge strategies: No merge, Immediate Merge and Logarithmic Merge. Based on this analysis we propose a cost model for determining the best strategy in a given scenario or as the basis for an adaptive merge strategy.
Acknowledgments
I would like to express my gratitude to my supervisors Dr. Jens-Peter Dittrich, Marcos Antonio Vaz Salles, and Lukas Blunschi for their provided help, guidance and continuous support through the course of this work. I would also like to thank them and the whole iMeMex group for the very friendly and helpful atmosphere.
Contents

1 Introduction 5
   1.1 About the iMeMex project 5
   1.2 Static vs. Dynamic Indexing 6
   1.3 Structure of this thesis 7

2 Background 9
   2.1 Text Retrieval 9
   2.2 Inverted Indices 9
   2.3 Static vs. Dynamic Text Collections 11
   2.4 Dynamic Indexing Techniques 11
      2.4.1 Document Additions 12
         2.4.1.1 In-Place Index Update 12
         2.4.1.2 Merge-Based Index Update 14
      2.4.2 Document Deletions 19

3 Generic Indexing Framework 21
   3.1 Basic Components 21
   3.2 Operations & Component Details 23
      3.2.1 Index Updates 23
      3.2.2 Merges 24
      3.2.3 Queries 26
      3.2.4 Example 26
   3.3 Concurrency & Synchronization 30
   3.4 Generic Data Types 33

4 Configurations 34
   4.1 Component Implementations Overview 34
      4.1.1 Static Indices 34
      4.1.2 Delta Indices 35
      4.1.3 Merge Strategies 35
   4.2 Evaluating Implementations 37
      4.2.1 Evaluation Criteria 37
      4.2.2 Experiments 38
4.2.2.1 Setup ............................................. 39  
4.2.2.2 Experiment 1 ................................. 40  
4.2.2.3 Experiment 2 ................................. 42  
4.2.2.4 Experiment 3 ................................. 44  
4.2.2.5 Experiment 4 ................................. 46  
4.2.2.6 Experiment 5 ................................. 48  
4.2.2.7 Experiment 6 ................................. 50  
4.2.2.8 Experiment 7 ................................. 52  
4.2.2.9 Experiment 8 ................................. 54  
4.2.3 Summary ......................................... 56  

5 Implementation ................................. 58  
5.1 Background: IMEMEX Specifics ............... 58  
5.2 In-Memory Inverted Index ................. 59  
5.2.1 General Architecture ....................... 60  
5.2.2 Update Operations ......................... 61  
5.3 On-Disk Inverted Index ....................... 68  
5.3.1 General Architecture ....................... 68  
5.3.2 Operations ................................. 70  
5.4 Merge Strategies ............................... 71  

6 Analysis .............................................. 73  
6.1 Merge Strategy Cost Model .............. 73  
6.2 Cost Functions ............................... 77  
6.2.1 No-Merge .................................... 78  
6.2.2 Immediate Merge ......................... 78  
6.2.3 Logarithmic Merge ......................... 78  
6.2.4 Combining Cost Functions ............. 80  
6.3 Comparing Merge Strategies ............. 82  
6.3.1 No-Merge vs. Immediate Merge .......... 82  
6.3.2 Immediate Merge vs. 2-Way Logarithmic Merge .......... 83  
6.3.3 b-Way vs. (b+1)-Way Logarithmic Merge .......... 86  
6.4 Relative Query Cost Factor Estimation .... 86  
6.4.1 Estimation of \( q_{abs}, v_{abs} \) Using Least Squares .......... 87  
6.4.2 Estimation of \( q_{abs}, v_{abs} \) By Direct Measurements .......... 89  
6.5 Prediction vs. Real Data ..................... 91  
6.5.1 Experiment Description ..................... 91  
6.5.2 Cost Model Based Predictions ........... 91  
6.6 Summary & Remarks ............................ 95  
6.6.1 Improving Cost Model ...................... 95
Chapter 1

Introduction

In this first chapter we would like to show the motivation behind the work of this thesis and how it contributes to the iMeMex project. We do that by shortly describing the main idea of iMeMex and illustrating the importance of dynamic indexing.

1.1 About the iMeMex project

iMeMex is an ongoing research project [8] and a first implementation of a Personal Dataspase Management System (PDMS) with the goal to handle the entire personal dataspace of a user. Such a dataspace usually consists of a very heterogeneous mix of all kinds of data. Typically a user’s local file system alone contains thousands of files of all kinds of types (PDFs, media files, text files, word documents). Besides that, a user may also access various remote data sources (IMAP, network drives). Without a PDMS the functionality of searching and managing that data is scattered among many different applications that all have their own data processing logic. For the user this means that he must have knowledge about the kinds of underlying data and formats in his personal dataspace and what applications/devices he can use to process and manage them. Furthermore, he has no unified view over all of his data, and he cannot perform complex queries or other advanced data management functionality (like backup or versioning) on it. Instead, he has to perform a number of manual tasks that are strictly tied to the underlying type of data and the applications used.

More formally, the key problems that a user is confronted with (without a PDMS) can be defined as follows:

- **Physical data dependence** (user must know about devices/formats used to store data)

- **Logical data dependence** (user cannot define user-centric view over data model that is used for data representation)
Building a platform that provides physical and logical data independence brings of course some challenges that must be solved. Two of them are defining a unified data model, that is powerful enough to represent a highly heterogeneous mix of data, and developing query capabilities on top of it.

In the context of iMeMex the following developments (among many others) were made to tackle the challenges described above:

- The iMeMex data model (iDM) [9], a powerful data model which differentiates clearly between the logical data model and its physical representation. iDM is able to represent a very wide range of structures (XML, file tree structures, cyclic graphs, infinite streams).
- The iMeMex query language (iQL) (shortly described in [8]), the query language to be used to query graphs represented in iDM.

For detailed information on the iMeMex project see [4, 8, 9].

1.2 Static vs. Dynamic Indexing

When thinking of a personal dataspace, it is important to note that it is not only a very heterogeneous but in most cases also a highly dynamic construct. By interacting with applications and devices a user continuously extends, deletes, and modifies data which his personal dataspace consists of. He may for example save a picture, edit his favourite song playlist, download a file from the Internet, rename a folder, or compress some files. Data sources may be added or removed on the fly, for example when plugging in a USB stick, and even when there is no user-interaction the personal dataspace usually does not remain in a static state. One can for example think of email clients automatically receiving emails, automated backup tasks copying around files in the background or shared network drives.

A PDMS that provides only static snapshots of the user’s personal dataspace and gets out-of-date is insufficient. Instead it must be able to automatically reflect all changes occurring in the user’s dataspace in more or less real-time. A user does not want to wait overnight before his PDMS is able to query a PDF file that he has just downloaded. In reality, he does not even want to wait minutes but will expect the system to be up-to-date within seconds.

The procedure that a PDMS must perform when a change in an underlying data source occurs can be broken down into three steps:

1. First a change in a data source must be detected, this can be divided into two sub steps.

   (a) Detecting that something has changed.
(b) Determining what has changed (for a certain level of granularity \(^1\)).

2. The PDMS must update its internal representation of the personal dataspace accordingly to the detected changes.

3. The index structures must be updated to reflect the updated internal representation of the personal dataspace. (These index structures are necessary in order to be able to efficiently answer queries on the personal dataspace.)

Of course, all of these steps have their own challenges. Note that the first two of them are not part of this thesis and have been at least partly solved already\(^2\).

Before this thesis, iMeMex was, however, missing to be able to perform the last of the steps presented above - fine grained updates to its index structures. Instead, only a static indexing approach was supported. A read-only index structure could be created once with no possibility to apply any subsequent updates. This thesis now fills this gap by providing iMeMex with a dynamic indexing framework able to handle the very dynamic nature of personal dataspaces.

### 1.3 Structure of this thesis

- Chapter 1 describes the basic idea behind iMeMex and motivates the need of a dynamic indexing framework.

- A short introduction to inverted indices (based on inverted files) is then given in chapter 2. We illustrate the problems one is confronted with when using inverted indices in a dynamic environment. Moreover, the most important dynamic indexing strategies in the context of inverted indices, as found in literature, are introduced.

- In chapter 3 we present our generic dynamic indexing framework used for indexing personal dataspace representations in iMeMex. This chapter is not about implementation details but about the interaction of the components of the system and the model of the framework in general.

- Chapter 4 then shows how this dynamic indexing framework can be configured. By presenting various experimental results we show how the properties of the dynamic indexing framework can be changed by choosing different components.

---

\(^1\) Taking a file change as example it is possible to think of various levels of granularity of detecting the change: It could be detected that something has changed in a certain directory / that only the name of a file has changed but its content is still the same / that the content of a file has changed / that 'hello' has been appended to the content of a file / ...

\(^2\) More information on how file changes are detected and handled in iMeMex under Windows can be found in the semester thesis [3].
• We then present some component implementations more in detail in chapter 5. We illustrate a few selected problems we were confronted with when implementing actual components for the dynamic indexing framework and what solutions we have come up with.

• In chapter 6 we study the No-Merge, Immediate Merge and Logarithmic Merge strategies from an analytical point of view. On the basis of this analysis we present a cost model determining the merge strategy to be used in a given scenario.

• A few topics for future research are proposed in chapter 7.

• Finally we shortly summarize our results in chapter 8.
Chapter 2

Background

In this chapter we introduce the concepts and terminologies used in the following chapters. Most of the concepts in this chapter have been described in scientific literature, references are given in the text.

2.1 Text Retrieval

Search engines that allow in one way or another to search for text are very common today. The most well-known examples are web search engines that index huge amounts of data, but also standard desktop applications like media managers (search by title, artist, album ..), document viewers, or file managers usually provide the functionality of searching for text. It is not surprising that text retrieval is an area that has been very well studied during the last few decades, and that much effort has been put into the analysis of various aspects of it.

When designing and implementing an indexing framework for a PDMS, one must luckily not reinvent the wheel. Textual data is certainly the predominant type of data to be indexed by a PDMS, even though it indexes other types of data (numbers, dates, ..) as well. It therefore makes sense to look at existing indexing techniques and adapt them to the needs of a PDMS.

2.2 Inverted Indices

The standard index structure typically used for text retrieval systems today is the inverted index based on an inverted file. In its simplest form, it is a mapping from terms to all documents that contain these terms. Depending on what the index is used for, it usually includes additional information. If the index must for example support phrase querying, the term positions within the respective document are also included in the index. Figure 2.1 shows an exemplary construction of an inverted index including term positions. This process involves at least the following steps:
Apples and Bananas and Cauliflower.

Good bananas taste good and fresh.

Figure 2.1: Inverted index construction steps

1. For every document, tokens must be extracted. This process is called **tokenization**.

2. All tokens of a document that are equal are consolidated to an element of the form (term, posting). The term is equal to the tokens value and the posting contains a reference to the document and, optionally, a calculated aggregate (e.g. term frequency or list of positions).

3. The consolidated elements of all documents are grouped by term and postings and are then combined to a posting list.

The final structure of the inverted index is depicted in Figure 2.2. It consist of two elements:

- A **vocabulary** mapping from a term to the offset of the term’s posting list in the **inverted file**.
- An **inverted file** containing contiguous posting lists for all terms.

Literature suggests that inverted indices have many advantages over other structures, like **suffix trees** or **signature files**, when used for general purpose text retrieval [16, 17]. Over the years many refinements and improvements have been devised, such that text retrieval systems using inverted indices today are able to index data in the order of gigabytes or even terabytes in a very reasonable amount of time. A today’s desktop computer can index one gigabyte of data in only a few minutes. At the same time queries are processed in only a fraction of a second. For a good overview on recent developments in the area of inverted indices, please refer to [16].
2.3 Static vs. Dynamic Text Collections

Much effort has been put into the improvement of indexing techniques based on inverted file indices. Aspects like index construction or query processing seem to be very well investigated and have been studied for many years. In contrast the aspect of index maintenance of inverted indices in a highly dynamic environment (like a file system) seems to be a relatively young discipline that has gained attention mainly during the last years, often in the context of desktop search systems ([5, 6, 7, 10, 11, 12, 13, 15]). In the next section, we give a short overview over some of the results that have been published and introduce the terminology used in that context.

2.4 Dynamic Indexing Techniques

A dynamic text collection has the following properties:

1. New documents may be added to the collection.
2. Existing documents may be deleted from the collection.
3. Existing documents in the collection may be modified.

Note that a modification of a document can also be modeled as a deletion followed by a reinsertion of a document. An indexing technique for dynamic text collections must therefore at least be able to handle document insertions and deletions.

The simplest approach of updating an index is just to re-index the whole collection whenever it changes. This strategy is referred to as re-build [12]. Re-indexing the whole collection for every single update is very costly and can hardly be used in a truly dynamic environment. This strategy may, however, still be used in a scenario where the index does not always need to be up-to-date and re-indexing is not required frequently. Think for example of an index for a web search engine. It does not get updated instantly whenever somebody
uploads or changes a website, but only after some time when the website gets re-indexed.

If we want the index always to be consistent with a dynamic collection, another approach than re-build is needed. In the literature several strategies are proposed that fall either into the category of in-place or merge-based update strategies, or into both, called hybrid strategies. Although based on different ideas, these strategies have one common characteristic: Updates are not applied one-by-one to the existing index structure. Instead, they are collected first in a separate structure and only at a certain point applied to the existing index as a whole. This potentially allows to amortize expensive update costs over many updates. This very basic concept is not new and has, in fact, already been applied to many problems in other domains [14]. A classic example, also used in [14], is that of an errata list for a book: Printing a new edition of a book whenever some text needs to be changed would be too expensive. Therefore all corrections are added to a list. Only when this list has reached a certain size, a new edition is printed. This analogy also shows that such an approach does not come entirely for free: The reader has to perform extra work, since he must also consult the errata list when reading the book. He has to do that because the book is not up-to-date (or stale). The reader could, however, also decide that it is not important for him to always be fully up-to-date and therefore not to consult the errata list at all and save some time.

2.4.1 Document Additions

When a document is added to the collection, it is first tokenized and postings are generated. As illustrated in the introduction of this section, the generated postings are not directly applied to the on-disk inverted index but rather accumulated in a separate structure in main-memory. Only when this in-memory structure has reached a certain size, it is combined with the on-disk index. The way in which this is actually done depends on the type of update strategy that is used. Such strategies are explained in the following subsections.

Also note that, in order to process a query, not only the on-disk index but also the accumulated postings in main-memory can/must be queried. All results are then found by combining the results of both structures.

2.4.1.1 In-Place Index Update

An in-place strategy [12, 15] appends new postings directly to the posting-lists of the existing on-disk index. This can, however, not always be done easily, since posting-lists must remain contiguous on disk in order to be queried efficiently. This problem can be reduced to a certain extent by over-allocating space for posting lists. Before adding postings to a posting list, it must be ensured that there is enough space available. If this is not the case, the old list must be relocated to a new location where the buffered in-memory posting can be
Figure 2.3: In-place index update
append. This introduces of course some additional complexity as it requires some kind of free space management. The main problem of this approach in terms of performance is the high costs associated with disk seeks caused by list re-locations.

An example of an in-place index update is shown in Figure 2.3. The “P”-boxes represent posting nodes, whereas the “F”-boxes represent free space. Note that the new “apples” posting is appended directly to the existing posting list because there is enough free space left. At the end of the “thread” posting list there is, on the other hand, not enough free space left for two new posting nodes. Therefore the “thread” posting list is relocated.

2.4.1.2 Merge-Based Index Update

A merge-based update approach (see Figure 2.4) creates an entirely new index on disk by merging the accumulated in-memory postings with the existing on-disk index. After that the old on-disk index, as well as the accumulated postings in memory, are deleted. In contrast to in-place update, the merge-based approach temporarily requires disk space for two index copies instead of only one. Intuitively, one may think that a merge-based update approach cannot perform well because the whole on-disk index is read and not only those posting lists that are subject to change. It is, however, important to note that the inverted index on disk is being processed sequentially during a merge. This reduces the number of disk seeks to a minimum. Furthermore, fragmentation does not occur and
It is important to understand that a merge-based update approach does not prevent the index from being queried at merge time. While a new index is being built up on disk during the merge, the old index and the accumulated in-memory postings are still available and can be queried.

Multiple On-Disk Indices. We have previously assumed that the accumulated in-memory structure is always immediately merged with the on-disk index whenever it has reached a certain maximum size. This is, however, only one strategy among many.

Immediate Merge. The strategy in which accumulated postings in memory are always immediately merged with the on-disk index is called Immediate Merge [5]. This is actually a very extreme strategy. Note that the inverted index on disk grows with the number of merges, whereas the in-memory structure is limited to a certain size. Eventually, this leads to the unfavourable situation where a very large on-disk index must be fully read, in order to be merged with a comparatively small in-memory structure. Postings are not written to disk just once when the in-memory structure has filled up, but read from and written to disk again with every following merge (see Figure 2.5).

No Merge. A strategy to avoid the problem of Immediate Merge, is not to merge at all. This means that a new, additional, on-disk index is created whenever the in-memory structure has filled up. Such a strategy is called No Merge [5]. It has the advantage that every posting is written to disk only once. This is, however, yet again a very extreme strategy, just at the other end of the spectrum. With every merge the number of on-disk indices is increased by
one (see Figure 2.5). This minimizes index construction costs but leads, on the other hand, to bad query performance. Not just one but many indices on disk must be queried.

**Balanced Strategies.** Both of the strategies presented above can be described as providing a highly unbalanced trade-off between costs for query processing and index maintenance. No Merge minimizes the index construction time but the query performance decreases with every merge. Immediate Merge on the other hand minimizes query processing time as there is only one on-disk index but brings along very high index maintenance costs.

At least two strategies have been proposed that provide a more balanced trade-off between update and query costs. They are both conceptually similar and are referred to as Logarithmic Merge [5] and Geometric Partitioning [11].

In the Logarithmic Merge strategy every index on disk is assigned a generation number $g$. A $b$-way ($b \geq 2$, $b \in \mathbb{N}$) Logarithmic Merge strategy is then performed according to the following rules:

- When an on-disk index is created exclusively from the in-memory structure, it is assigned $g = 0$.
- At most $b - 1$ indices of the same generation are allowed to co-exist. This condition is kept by always merging $b$ indices of generation $g = x$ to one new index of generation $g = x + 1$.

Figure 2.6 illustrates the procedure of a 2-Way Logarithmic Merge strategy that always merges two indices of the same generation. The procedure in Figure

---

**Figure 2.6: 2-Way Logarithmic Merge strategy (unoptimized)**
2.7 shows the same procedure but executed in an optimized way. Instead of always writing the in-memory structure to disk as a $g = 0$ index and then evaluating whether the merge condition applies, that condition can be evaluated in advance. The in-memory structure can then be directly merged with indices on disk accordingly. In other words, recursive cascades of merges are anticipated. Suppose for example that there are two indices $I_1, I_2$ on disk with $g_1 = 0, g_2 = 1$. If the in-memory structure $I_M$ is full, the standard procedure would be as follows: First a new index $I_3$ ($g_3 = 0$) on disk is created out of $I_M$, $I_1$ is merged with $I_3$ to a new index $I_4$ with $g_4 = 1$, $I_4$ is merged with $I_2$ to a new index $I_5$ with $g_5 = 2$. This is, however, not very clever. As this cascade of merges can be anticipated, $I_M$ can directly be merged with $I_1$ and $I_2$ to a new index $I_3$ with $g_3 = 2$. This leads to the same result as before but requires less data transferred from and to disk. Note, however, that the fan-in is not anymore restricted to $b$. In the following we refer to this strategy as optimized Logarithmic Merge.

Figure 2.8 shows the expected costs associated to index updates and queries for various strategies. Cost expectations are subject to the following assumptions: Query costs are in proportion to the number of indices on disk, merge costs are in proportion to the total size of the merged indices.

**Hybrid Index Update.** The previously described in-place and merge-based update approaches can be combined to an approach referred to as hybrid index update [6, 7]. It is motivated by the observation that a merge-based index update performs better than in-place update for short posting lists (because it does not require additional disk seeks). For long posting lists the situation is,
Figure 2.8: Expected merge/query costs
however, different. In that case the disk seeks necessary for the in-place update approach may be cheaper than having to sequentially process all data. This leads to the idea of a hybrid update approach using a mix of merge-based and in-place update depending on the situation.

### 2.4.2 Document Deletions

When a document is deleted, the index structures must be updated accordingly. With respect to the inverted index, this means that all postings related to a deleted document must be removed from all posting lists. Whereas new postings can just be appended at the end of an existing posting list, the positions of postings that are to be deleted must first be located within the posting list. In that sense document deletions are more difficult to handle than insertions of new documents.

Luckily document deletions must not always be applied instantly to the inverted indices on disk. Similar to aggregated index updates for document insertions, deletions can be aggregated in a data structure in memory and only be applied all-at-once after some time. This allows to amortize costs over a number of deletions. The downside of this approach is that new costs for query processing are introduced. In order to get correct results, all postings that belong to deleted documents must be post-filtered when the indices on disk are queried. Figure 2.9 illustrates the process of querying the in-memory structure and indices on disk. Postings belonging to deleted documents are filtered out.

All index maintenance strategies proposed in literature buffer document dele-
tions in memory and post-filter query results. Besides that, typically a garbage collection approach is used to remove postings belonging to deleted documents from the actual indices on disk from time to time. This is necessary in order to reduce index maintenance costs by reducing the size of the indices on disk and therefore speeding up operations like merging or posting list re-locations.
Chapter 3

Generic Indexing Framework

In this chapter we introduce the generic indexing framework that has been designed and implemented as part of this thesis. Note that implementation details are not covered in this chapter. Instead we illustrate the concept and explain the model of the framework.

3.1 Basic Components

Figure 3.1 shows a simplified illustration of the indexing framework broken down into its most basic components. The green box in the center is the system’s core. Its purpose is to coordinate and handle incoming updates (document insertions/deletions) and queries. It does that by interacting with several other components illustrated as blue boxes. Important about this interaction is that it is based on interfaces. The components illustrated in blue can therefore be described as abstract components. The core does not have knowledge about how these abstract components are actually implemented internally and they can therefore be easily switched or changed without affecting the core’s functionality.

Delta Index. The delta index component consists of an updatable index structure. Every time a document is added to or removed from the dynamic index the delta index is updated accordingly. Besides handling updates the delta index must, as its name suggests, also be able to process queries on the indexed documents. An implementation of a delta index usually operates on data structures in memory allowing updates to be processed more efficiently and more flexibly than on data structures on disk. The main idea of the delta index is to amortize update costs by accumulating them first and applying them in batch to the on disk structures. In practice the memory that can be used for building up the delta index is limited. Therefore typically a certain maximum delta index size is defined. When that maximum is reached the data is transferred to a static index (see next paragraph). After that process the delta index is empty again.
Static Indices. A dynamic index may also contain a number of static indices. In contrast to the delta index, a static index is bulk loaded only once and cannot be updated subsequently. It is loaded with data from the delta index and/or other static indices. More precisely, a static index is bulk loaded with either:

- Only the data of the delta index, or
- The data of the delta index and one or more static indices, or
- The data of two or more static indices

Like the delta index, a static index must be able to process queries on the indexed documents.

Terminology. In the following, the general process of creating a new static index and bulk loading it with data (of any kind) will be referred to as a merge. (In the context of the first of the cases listed above, this term may be a bit misleading because no data must be actually merged in that specific case. To keep things simple we will, however, not denominate this case specially and included it in the term merge.) Furthermore we are using the following terms:

- A delta merge includes data from the delta index. (One of the first two cases listed above.)
- A delta only merge includes only the data from the delta index.
- A static only merge includes only the data of static indices.
Besides that, a static index is in either an active or inactive state:

- **Inactive**: The index is currently being built up / bulk loaded, it can not (yet) be queried or merged.
- **Active**: The index has been completely built up (as the result of a merge process) and can be queried or merged.

**Merge Strategy.** The main purpose of a merge strategy is to provide the functionality of determining the following aspects:

1. When a merge should be initiated.
2. What data a merge should include. This can also be divided into two sub problems:
   
   (a) Should the data from the delta index be included in the merge?
   (b) Should data from static indices be included in the merge? If yes, from how many and from which ones?

3.2 **Operations & Component Details**

In section 3.1 we gave a simple overview over the basic components. In this section we present the basic operations of the dynamic index and, at the same time, explain components more in detail. Please consult Figure 3.2.

3.2.1 **Index Updates**

There are two types of index update operations: New documents can be inserted into the index and documents can be deleted from it. Note that document modifications can be modeled as a deletion followed by a re-insertion.

**Inserting New Documents.** The insertion of a new document into the dynamic index is simply forwarded to the delta index. Figure 3.2 shows the current and the previous delta index. Updates always go to the current one, new documents are therefore inserted into $\Delta_C$. (The role of the delta index labeled as previous is explained later.)

**Deleting Documents.** When a document is deleted, the dynamic index is provided with a document identifier, in the following referred to as oid, of the deleted document.

Similarly to a document insertion a document deletion is applied to the current delta index. The deletion is first applied to $\Delta_C$. Then the oid of the deleted document is added to $D_C$. The oids need to be collected because they are needed for the post-filtering of deleted documents from static indices.

23
3.2.2 Merges

The merge operation differs from the other operations (index updates and queries) in that it is not part of the interface provided by the dynamic index. Merge operations are executed internally and automatically by the core of the system.

Concurrent Merges. The dynamic index model that we present hereby requires and ensures that there is never more than one delta merge at a time. Additionally, however, an arbitrary number of static only merges can be executed concurrently. More information on issues related to concurrency and synchronization is given in 3.3.

Merge Process. As described in section 3.1, merging means that a new static index is created out of the data of a delta index and/or other static indices. We give now a description of the different steps of the merging process. To reduce the amount of text and to make things more understandable, the labels from Figure 3.2 are used. Note that an example of the merge process (and other operations) is given in subsection 3.2.4. This example should help to understand the statements in the following description more easily.

The merge process consists of the following three phases:
1. Initialization phase.

(a) A new, inactive, initially empty, static index $S_{\text{new}}$ is created.

(b) If the merge is a *delta merge*, then:
   i. $\Delta_C$, $D_C$ become $\Delta_P$, $D_P$
   ii. New $\Delta_C$, $D_C$ are created. They are initially empty.
   iii. $t_{\text{new}} = t_C + 1$

(c) If the merge is a *static only merge*, then:
   i. $t_{\text{new}} = t_C$

(d) For every active static index $S_i$ that is merged, mark $S_i$ as “reserved”, meaning that $S_i$ is already being merged and is not available anymore for other merges.

2. Merge phase.

(a) Incoming updates, during the merge phase, are applied to $\Delta_C$ and $D_C$.

(b) At the same time the new static index $S_{\text{new}}$ is bulk loaded with:
   i. Data from $\Delta_P$ (only if *delta merge*), and
   ii. Data from all static indices that are merged. Data belonging to deleted documents is post-filtered according to the following rule: For every static index $S_i$ with timestamp $t_i$ being merged, data is post-filtered if it is associated to an oid present in either $D_P$ or in any oid set in the *deleted oid store* associated with a timestamp $t$ with $t > t_i$ and $t \leq t_{\text{new}}$.

3. Finalization phase.

(a) If the merge is a *delta merge*:
   i. The current timestamp is increased: $t_C = t_{\text{new}}$.
   ii. $D_P$ is added to the deleted oid store with timestamp $t_{\text{new}}$.
   iii. $\Delta_P$, $D_P$ are deleted from the delta index. They are not used anymore.

(b) All static indices that were merged are deleted.

(c) Oid sets that are not used anymore are removed from the deleted oid store. These are all sets with a timestamp smaller than or equal to the minimal timestamp of all remaining static indices.

(d) $S_{\text{new}}$ is activated with timestamp $t_{\text{new}}$. From now on, it is available for queries or for being merged to another static index.
3.2.3 Queries

In order to get all results, the results of all of the following sub queries must be combined:

1. Query $\Delta_C$.

2. Query $\Delta_P$ and post-filter all data associated with documents with an oid present in $D_C$.

3. For every static index $S_i$ with timestamp $t_i$:
   
   (a) Query $S_i$ and post-filter all data belonging to documents with an oid present in either $D_C$, $D_P$ or in any oid set of the deleted oid store associated with a timestamp $t > t_i$.

3.2.4 Example

Figures 3.3 to 3.5 show an evolving dynamic index to which index update and merge operations are applied.

a) The dynamic index that is shown in the first step has already arisen from a number of operations. It started with the insertion of two documents followed by a delta only merge that lead to the creation of $S_0$. After that, having again an empty delta index, another document was inserted, and the previously inserted document 2 was deleted. That was followed again by a delta only merge leading to the creation of $S_1$. As part of that second merge the oid “d2” of the deleted document 2 was also added to the deleted oid store, associated with timestamp $t = 1$. In a last step document 2 was reinserted with a changed content and two new documents were added, again followed by a delta only merge. Note that the delta index is now empty because it was just merged in the last operation. The current timestamp is $t_C = 2$.

b) Starting from the dynamic index that was produced in a), two new documents are inserted and document 5 is deleted. These updates are applied to the current delta index as explained in 3.2.1.

c) Now a merge is initiated not only including the delta index but also the static indices $S_0$ and $S_2$. (There is no reason why exactly these two static indices are included in the merge, we have just chosen it to be so in this example.) In the initialization phase of the merge, the current delta index parts $\Delta_C$, $D_C$ become $\Delta_P$, $D_P$ and a new empty current delta index is created. Additionally a new (inactive) static index is initialized. Its timestamp is set to $t = t_C + 1 = 3$ because this is a delta merge.
Figure 3.3: Dynamic index operations example
Figure 3.4: Dynamic index operations example (continued)
MERGE: \[ \Delta + S_0 + S_1 \rightarrow S_3 \]

(finalization phase)

Figure 3.5: Dynamic index operations example (continued)
d) Now follows the merge phase. Concurrently to the merge process, document 6 is deleted and document 5 is reinserted with a changed content. These updates are directly applied to the current delta index and do not interfere with the merge process. The data from $\Delta_P$ is merged with the data from $S_0$ and $S_2$. Note that all data referring to the document with oid “d5” is directly filtered out from any static index because it is contained in $D_P$. The deleted oid store contains also a set containing the oid “d2”. Data belonging to that document is therefore also filtered out, but only (!) when the according static index has a timestamp $t < 1$. In our example data associated with the oid “d2” is therefore removed from $S_0$ but not from $S_2$.

e) In the finalization phase of the merge process all previously merged static indices are deleted. The same also applies to $\Delta_P$ and $D_P$. $D_P$ is, however, added to the deleted oid store and associated with the timestamp of the merge ($t = 3$) before it is deleted. Furthermore, $S_3$ is activated and can be queried subsequently. The timestamp $t_C$ is increased by one. From the deleted oid store the set associated with $t = 1$ is removed because it is not used anymore. For post-filtering, only those sets are needed that are associated with timestamps that are higher than the smallest timestamp of all static indices).

f) Shows the resulting dynamic index.

3.3 Concurrency & Synchronization

This section illustrates the issues coming up with concurrent data access and how we deal with them.

In a first step the resources that are subject to concurrent access are identified. We differentiate between reading and writing data and make the following assumptions:

1. When a resource is written to by an operation, no other operation can read from or write to that resource (mutual exclusion).
2. As long as a resource is not written to, an arbitrary number of operations can read concurrently from that resource.

Table 3.1 describes for every operation the resources it interacts with, as explained in the previous sections\(^1\). Note that the table contains only resources that are shared among different operations. When studying that table, and considering the above assumptions, a few things can be observed:

\(^1\)The “Static Indices $S_i$ (active)” column refers to accessing the actual data from active static indices. The column “Static Index Container” refers to static index meta data (what indices are active/inactive, what indices are reserved for a merge, ...).
Update operations exclude each other, one cannot insert a document and at the same time delete a document or insert multiple documents concurrently.

Queries can be run concurrently to other queries.

The actual merge phase in the merge process differs clearly from the initialization and finalization phase in that it does not require write-access to any resources. (The separation of the merge process into the three phases is actually motivated by this very difference.)

Update operations do not interfere with the merge phase of the merge process.

Queries can be processed concurrently to the merge phase of the merge process.

With respect to the merge process, it is important to note that the initialization and the finalization phases can be considered as setup and cleanup operations. They are very inexpensive compared to the merge phase where the actual work (reading, merging and writing of indices) is done. It is therefore important mainly that the merge phases of merge operations can be executed concurrently to other operations.

**Locking.** In order to avoid inconsistencies due to concurrent resource-access, it must be ensured that operations have exclusive access to resources that they write to. This is done by using a lock in the form of a *Binary Semaphore* (or *Mutex*). When we ignore the query operation for the moment (it is actually a special-case and is examined in the following paragraph) we can relatively simply manage access to resources in the following way:

<table>
<thead>
<tr>
<th>Operation</th>
<th>$\Delta_C$</th>
<th>$D_C$</th>
<th>$\Delta_P$</th>
<th>$D_P$</th>
<th>Static Indices $S_i$ (active)</th>
<th>Static Index Container</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert document</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>remove document</td>
<td>W</td>
<td>W</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>query</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>R</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>delta merge (init. phase)</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>W</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>delta merge (merge phase)</td>
<td>-</td>
<td>-</td>
<td>R</td>
<td>R</td>
<td>-</td>
<td>W</td>
</tr>
<tr>
<td>delta merge (final. phase)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>static only merge (init. phase)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
</tr>
<tr>
<td>static only merge (merge phase)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>R</td>
<td>-</td>
</tr>
<tr>
<td>static only merge (final. phase)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>W</td>
</tr>
</tbody>
</table>

Table 3.1: operations data access (simplified)
• There is one mutex lock (for all resources).

• All operations except merge operations in merge phase must acquire that lock before they are executed\(^2\).

Note that this is an approach based on a rather non fine-grained lock granularity. Nevertheless, it has the following advantages:

• Update operations conflict neither with each other nor with merge initializations/finalizations.

• During the expensive merge phases updates can still be executed.

• An arbitrary number of static only merges can be executed concurrently. Only their initializations and finalizations need to be synchronized.

• It is simple. (In the actual implementation, handling resource-accesses is actually more complex than shown in table 3.1.)

Lazy Query Evaluation. Directly returning all results of a query at once can be disadvantageous, especially if there is a large number of them or if not all results are actually needed but only the first few. A strategy often used is to return an iterator (or pull operator) that allows to fetch the results element-by-element. Unfortunately such an approach has the disadvantage of complicating resource access synchronization. Suppose for example that the delta index is currently being merged (in the merge phase) and at the same time a query is answered by returning an iterator. This iterator operates on data from various resources whenever a result element is fetched. Suppose now the merge phase ends and the finalization phase is executed. In that phase \(\Delta_P\) is deleted, but can this really be done safely? The answer to that question depends on whether the previously returned iterator still needs to operate on data from \(\Delta_P\) or not. In order to solve such problems the following mechanism is used:

• Whenever an iterator is returned, a global query counter is increased by one.

• When an iterator has fetched all results or is closed prematurely, the query counter is decreased by one.

• When an operation needs write-access to a resource that is used for query evaluation, it must first wait until the counter reaches zero. In order to avoid the problem of starvation new queries are temporarily blocked in such a situation.

\(^2\) Table 3.1 seems to suggest that the delta merge phase might conflict with a delta merge initialization/finalization phase because it does not acquire a lock. This is, however, not the case because there can not be more than one delta merge at a time.
3.4 Generic Data Types

In the last sections we have defined a *merge* as the process of creating a new static index out of the data of the delta index and/or other static indices. We also explained in the introduction that the indexing framework does not have knowledge about how components, like a delta index or a static index, are actually implemented or how they internally represent data. This leads to the question of how this can work with various implementations of delta indices and static indices that represent data internally in different ways.

One possible approach is to use a standardized format as shown on the left side of Figure 3.6. Such an approach has the advantage that any delta index implementation can be used together with any static index implementation. The downside of such an approach is, however, that all implementations must always convert their output to that specific format, regardless of how they store the data internally. Implementations are therefore very dependent on the standardized format. This makes it difficult, if not impossible, to implement solutions that perform well.

A more flexible solution is to use generic data types as illustrated on the right side of Figure 3.6. We are using that approach for our dynamic indexing framework. In that approach, a delta index implementation can provide its data in any form but can only be combined with a static index implementation that is able to handle that type of data.
Chapter 4

Configurations

In the previous chapter the dynamic indexing framework was presented independently from actual component implementations (static/delta indices, merge strategies). In this chapter we would like to present various configurations of the dynamic index that are based on actual component implementations. Note that the discussion of implementation details of the various components is not the purpose of this chapter. (Details for some implementations are given in chapter 5.) Instead we want to show how the properties of the dynamic index are affected by the choice of components. You will see that a certain choice of components often leads to favourable properties in respect to one criterion and at the same time to unfavourable properties in respect to another. Therefore the evaluation of a certain configuration depends heavily on the weighting of these criteria.

4.1 Component Implementations Overview

In this section we shortly introduce the most important component implementations. This is necessary because these implementations are the basic parameters of the experiments presented in this chapter.

4.1.1 Static Indices

In section 2.2 we explained that the inverted index is the data structure of choice to be used for text retrieval. In the context of this thesis, there is an implementation of a static index provided that is based on inverted indices. This implementation is a generic one that allows to vary various aspects like for example how positional information is stored in the inverted index. For more detailed information please refer to section 5.3.
4.1.2 Delta Indices

Log. Probably the most simple type of a delta index implementation is that of a log\(^1\). When a document is added to or removed from the collection, a reference to that document is simply added to the log. Documents are neither tokenized, nor analyzed in another way. Therefore an update to the log requires only a tiny amount of work and is done almost instantly. On the other hand, this also means that no index structure is available that allows to speed up queries. Instead all documents referenced in the log must be traversed and separately queried. This increases the query processing time substantially.

In-Memory Inverted Index. Another delta index implementation is that of an inverted index in memory. In contrast to an inverted index on disk, it can be updated in-place relatively easily because it operates on more flexible data structures in memory. When keeping an inverted index in memory, queries can be processed very efficiently. It could be argued that building up an inverted index in memory is quite costly, especially when compared to the previously mentioned log approach. It is, however, very important to understand that these costs are amortized when the in-memory inverted index is merged with an inverted index on disk because all of the work (tokenizing documents, creating posting lists, sorting by terms, etc..) has to be done at that point anyway.

4.1.3 Merge Strategies

The merge strategy implementation that is used in all of the following experiments is a kind of “meta” merge strategy consisting of a number of sub components. Each sub component handles one aspect of the merge strategy and for every type of sub component there exist various implementations. This allows to compose a large number of merge strategies by putting together different sub component implementations. In the following these types of sub components and some new terms are shortly introduced together with a few examples.

Merge Evaluation Preferences. These preferences control when it should be evaluated whether a merge has to be initiated or not. Various possibilities exist:

- Evaluate merge whenever a document has been inserted / deleted / ...
- Use polling. (Evaluate merge every \(x\) seconds.)
- Evaluate merge whenever an insertion is blocked.
- ...

\(^1\)In the context of a log the term “index” is actually not appropriate since a log does not provide an index over its referenced documents.
**Merge Condition.** When a merge evaluation is executed and this condition is met, a merge is initiated. Typical merge conditions are for example:

- Merge if the size of the delta index in memory is exceeding 50 MB.
- Merge always after 100 documents insertions.
- Never merge at all.

**Blocking Condition.** As long as this condition is met, new insertions are blocked. Typically, this condition is not met before the merge condition and its purpose is to set an upper limit on the available memory. In some of the following experiments the term *buffer size ratio* is used. This ratio describes the relation between the blocking condition and the merge condition. Assume for example that a merge is initiated whenever the delta index size has reached 10 MB and that new insertions are blocked as soon as the delta index size has reached 20 MB. In such a scenario the buffer size ratio corresponds to 0.5.

**Merge Planner.** The merge planner component of the merge strategy determines how many and which indices are merged. Examples of merge planners are the Immediate Merge / No-Merge / Logarithmic Merge strategies discussed in the second chapter.

**Synchronous vs. Asynchronous.** The merge strategies that have been presented in the previous chapters can be described as *synchronous* strategies. This means that there is always at most one delta merge and no other concurrent merges. A merge can only be initiated as soon as the previously initiated merge has finished. Therefore such a strategy is called *synchronous*.

Additionally we now introduce the concept of an *asynchronous* merge strategy. Such a merge strategy is also executing always at most one delta merge at once because the indexing framework does not allow concurrent delta merges. Additionally, however, an asynchronous strategy may initiate an arbitrary number of concurrently and asynchronously running static only merges. These static only merges can also be described as background merges as they operate only on static indices. Although the indexing framework does not restrict us in any way we are in the following always assuming that an *asynchronous* merge strategy plans delta merges according to a No-Merge strategy. In other words, the delta index is always just written to disk as a new static index without merging it with other static indices. The idea behind this strategy is to free memory as fast as possible. The created static indices are then processed by the background merges according to a defined strategy.

Figure 4.1 illustrates a synchronous Immediate Merge strategy and a similar asynchronous merge strategy. The asynchronous strategy operates like a No-Merge strategy for the delta merges. The delta index is always just written to
Figure 4.1: Synchronous vs. asynchronous merge strategy

disk without being merged with any static indices. The static only merges are planned according to a strategy that could be described as *merge all*. It takes all available static indices and merges them. Eventually the end product is the same for both types of merge strategies. The advantage of the asynchronous strategy is that delta merges are processed faster, on the other hand the number of static indices is not limited. Therefore merge costs may (temporarily) be higher for the asynchronous strategy than for the synchronous one.

### 4.2 Evaluating Implementations

#### 4.2.1 Evaluation Criteria

When evaluating and comparing different dynamic index configurations, we do that by considering the following criteria:

1. Index maintenance performance.
   
   (a) **Average time required per index update**: We use that criterion to measure the amortized update costs over many index updates. Note that the time required for processing a single update can differ significantly. One can for example think of the case where a single
update cannot be applied to the delta index because the delta index has reached its maximal size. In such a case, the delta index data must first be merged to a static index on disk before the update can be processed. This increases the time required to process that single update substantially.

(b) **Worst case time for a single index update:** This criterion is an indicator for the real-time capability of a dynamic index configuration. One can for example think of two different configurations: One configuration requires ten seconds for ten updates with one second per update. The other configuration also requires ten seconds for the same number of updates but requires nine seconds for one of those updates and one second divided among the other nine updates. In such a case, both configurations have the same average update rate. In the context of a dynamic index in a real-time environment, the first configuration would, however, probably be preferred because not only the amortized costs matter but also the worst case costs per update.

2. Query performance.

(a) **Average time required to process a query.**

Unfortunately these criteria are not independent from each other and depend on many factors like:

- Delta index insertion performance
- Delta index deletion performance
- The way merges are planned by the merge strategy
- Multi-threading capability of merge-strategy
- Buffer size ratio used for delta index
- ...

In the next subsection these dependencies are explored by a number of experiments.

### 4.2.2 Experiments

In the following subsections a few experiments are presented. For every experiment the purpose, parameters and the output is shown on the first page. The second page contains a short description of the process of the experiment and an interpretation of the output. For a summary of the results, see subsection 4.2.3.
4.2.2.1 Setup

We have performed our experiments on a machine with the following setup:

CPU 2 x AMD Opteron 280 (Dualcore), 2.4 Ghz, 2 x 1 MB L2 Cache, 2 x 128 KB L1 Cache

Memory 6 GB Ram (with ECC)

Harddisk 80 GB ATA, 7200 rpm, 2MB Cache

OS GNU/Linux 2.6.9-67.0.7.ELsmp
4.2.2.2 Experiment 1

Purpose. Compares log and in-memory inverted index performance with respect to document insertions.

![Update Progression Over Time](chart.png)

**Common Parameters**
- Static Index = Inverted Index
- Merge Strategy Type = Synchronous
- Delta Merge Planner = No-Merge
- Buffer Size Ratio = 1.0

**Specific Parameters**
- Delta Index (1) = Log
- Delta Index (2) = In-Memory Inverted Index
Description. This experiment shows the progression over time of one run of updating a dynamic index with either using a log or an in-memory inverted index as delta index. In both cases a merge is initiated whenever a certain number of documents have been inserted.

Interpretation. The points in time where a merge is executed can be clearly recognized in the curves as the merges are blocking updates for a certain time. The curves show that the total indexing time does not differ significantly for the two approaches. This means that the average time required per index update is the same for both approaches.

Furthermore it can be seen that new documents are inserted in the log almost instantly. On the other hand, executing merges requires a long time as the data in the log has not been preprocessed. Inserting a document into the inverted index takes longer than an insertion in the log, the effort is however amortized as soon as the delta index is merged. As work is distributed more uniformly among insertions and merge operations, the maximum blocking time (or time required for an update in the worst case) is much smaller for the inverted index than for the log (7 seconds vs. 50 seconds).
4.2.2.3 Experiment 2

Purpose. Compares query performance of log and in-memory inverted index.

![Query Processing Time Comparison](image)

<table>
<thead>
<tr>
<th>Common Parameters</th>
<th>Specific Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Index = Inverted Index</td>
<td>Delta Index</td>
</tr>
<tr>
<td>Merge Strategy = Never Merge</td>
<td>(1) = Log</td>
</tr>
<tr>
<td></td>
<td>(2) = In-Memory Inversion Index</td>
</tr>
</tbody>
</table>
Description. New documents are inserted into delta index (either log or in-memory inversion index). The delta index is never merged. This means that the dynamic index always consists of only the delta index and no static indices. The query processing time measured is for executing 5 unique keyword queries. Note that a logarithmic scale is used for the query processing time.

Interpretation. With an in-memory inversion index as delta index these queries are answered almost instantly. When using the log, query processing time increases dramatically even for very moderate amounts of data. This is because all documents in the log must be searched through for every single query. We conclude that a delta index implementation in the form of a log is impractical and should not be used when the delta index must support queries because query processing is simply too costly.
4.2.2.4 Experiment 3


**Common Parameters**
- Static Index = Inverted Index
- Merge Strategy = Never Merge

**Specific Parameters**
- Delta Index
- \(1\) = In-Memory Inversion Index
Description. The query processing time measured is for 5000 unique keyword queries. Merges are never initiated, therefore all data remains in the in-memory inverted index.

Interpretation. Query processing time increases linearly with the number of inserted documents and is below a second even for 5000 queries and more than 100 megabytes of indexed data. The difference between the query performance of an in-memory inverted index and a log is huge. In experiment 4.2.2.3 we have seen that a log requires about 10 seconds for processing 5 keyword queries for about 50 MB of indexed data. For the same amount of indexed data, an in-memory inverted index is able to process 5000 keyword queries in only 0.15 seconds.
4.2.2.5 Experiment 4

**Purpose.** Compare index maintenance performance (insertion performance only) for various merge planning strategies with respect to a growing document collection.

![Indexing Time Experiment](image)

**Common Parameters**
- **Static Index** = Inverted Index
- **Delta Index** = In-Memory Inversion Index
- **Merge Strategy**
  - **Type** = Synchronous
  - **Buffer Size Ratio** = 1.0
  - **Merge Condition** = size(ΔC) > 40 MB

**Specific Parameters**
- **Delta Merge Planner**
  - (1) = Immediate Merge
  - (2) = 2-Way Log. Merge
  - (3) = No-Merge
Description. For every data point measured, a dynamic index is instantiated into which data of the given size is inserted. Whenever $\Delta C$ of the delta index has become larger than 40 MB a merge is initiated. If/what static indices are included in the merge is determined by the merge planner.

Note that all three variants use the same delta index implementation. Therefore, the differences seen in indexing time are all caused by varying durations of the merge process.

Interpretation. The resulting curves show what can be expected by the descriptions given in 2.4.1.2. No-Merge performs best because data is written to disk only once and not included in any subsequent merges. Its indexing time is therefore linear to the indexed data size. Immediate Merge performs worst due to its quadratic time complexity (every merge includes the data from all previous merges as well) and the Logarithmic Merge is in between these two.

The indexing performance in general looks very promising. Indexing one gigabyte of pure text data requires about five minutes with the 2-way Logarithmic Merge strategy while requiring only 40 megabytes for the delta index.
4.2.2.6 Experiment 5

**Purpose.** Measures query performance over time for various merge planning strategies for a growing document collection.

<table>
<thead>
<tr>
<th>Query Processing Time (s)</th>
<th>Indexed Data Size (MB)</th>
<th>Query Rate Experiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMMEDIATE MERGE</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2-WAY LOGARITHMIC MERGE</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>NO-MERGE</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.5</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2.5</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

**Common Parameters**
- Static Index = Inverted Index
- Delta Index = In-Memory Inversion Index
- Merge Strategy
  - Type = Synchronous
  - Buffer Size Ratio = 1.0
  - Merge Condition = size(ΔC) > 10 MB

**Specific Parameters**
- Delta Merge Planner
  - (1) = Immediate Merge
  - (2) = 2-Way Log. Merge
  - (3) = No-Merge
Description. After every document insertion, 5000 unique keyword queries are executed and query processing time is measured. Measurements are averaged over three repetitions with one warm up run.

Interpretation. The curves clearly show the points in time where a merge is executed. The reason for this is that a merge has an effect on the number of static indices resulting in a change of the query processing time. It is interesting to see that query processing time is in fact very closely related to the number of static indices. This becomes very obvious when comparing the query processing time curves with those in Figure 4.2. For the No-Merge strategy, query processing time increases with every additional static index. The Immediate Merge strategy is not affected by this problem because it limits the number of static indices such that there is never more than one. The query rate therefore decreases only very slowly with the total index size. In the middle curve the (2-way) logarithmic pattern correlating to the number of static indices is evident.

Besides the effect of the number of static indices, all three curves are also linearly increasing with the total size of the dynamic index.
4.2.2.7 Experiment 6

Purpose. Shows trade-off between update and query costs for various merge planning strategies.

![Graph showing query/update trade-off for different merge strategies.](image)

**Common Parameters**
- Static Index = Inverted Index
- Delta Index = In-Memory Inversion Index
- Merge Strategy Type = Synchronous
- Buffer Size Ratio = 1.0
- Merge Condition = size($\Delta_C$) > 500 documents

**Specific Parameters**
- Delta Merge Planner
  - (1) = No-Merge
  - (2) = 8-Way Log. Merge
  - (3) = 4-Way Log. Merge
  - (4) = 2-Way Log. Merge
  - (5) = Immediate Merge
**Description.** For every data point measured a growing document collection is indexed. The documents are generated artificially and contain 40 unique string tokens. The whole collection consists of 20'000 documents. For every posting list update a certain number of queries (according to the predefined query/update ratio) is executed. The time measured is the total time needed for executing insertions, merges and queries.

**Interpretation.** When no queries are executed ($x = 0$), the No-Merge strategy performs best because it just generates a new static index out of the delta index without merging it with any existing static index on disk. In the scenario of this experiment, the 4-way and 8-way Logarithmic Merge strategies are however almost as fast. The 2-way Logarithmic Merge strategy requires more time but is still very close to No-Merge. When the query to update ratio is increased, the No-Merge strategy is soon outperformed by all strategies. This is due to the fact that for No-Merge the number of static indices is growing linearly with the number of merges. This increases query costs significantly. In the middle interval Logarithmic Merge is the best choice as it provides a good trade off between query and index maintenance costs. The smaller we choose the fan-in of the Logarithmic Merge strategy, the better it performs for high query to update ratios. For very high query to update ratios Immediate Merge performs best. This is because it minimizes query costs by having only one static index. (Even though index maintenance costs are very high for Immediate Merge, we can always find a query to update ratio for which these index maintenance costs are amortized.)

Note that this experiment shows only one scenario. See chapter 6 for a more thorough analysis of query versus update costs.
4.2.2.8 Experiment 7

Purpose. Shows performance comparison of an asynchronous and an equivalent synchronous merge strategy.

Common Parameters
- Static Index = Inverted Index
- Delta Index = In-Memory Inversion Index
- Merge Strategy
  - Merge Planner = Immediate Merge
  - Buffer Size Ratio = 1.0
  - Merge Condition $\text{size}(\Delta_C) > 1 \text{ MB}$

Specific Parameters
- Delta Merge Strategy Type
  - (1) = Synchronous
  - (2) = Asynchronous
  - (1 Static Only Merge Thread)
**Description.** For every data point measured a growing document collection (of a total size of about 110 MB) is indexed and for every document insertion a certain number of queries (according to query/insertion ration) is executed. The dashed curves show how much of the total time was used for query processing.

**Interpretation.** The asynchronous strategy performs much better than the synchronous one in respect to overall performance. This has two reasons:

- The asynchronous strategy is able to amortize merge costs by including multiple static indices (created by delta only merges) in one merge (see Figure 4.1).
- The asynchronous strategy makes better use of CPU resources in a multicore environment because the static only merge threads run in parallel with threads updating the delta index.

The dashed curves show that query costs are higher for the asynchronous strategy than for the synchronous one. This makes sense because an asynchronous strategy (independently from its merge planner) does not restrict the number of static indices that can coexist temporarily.
4.2.2.9 Experiment 8

Purpose. Compare inverted index sizes when two different approaches of storing component information are used:

1. Store component information in vocabulary.
2. Store component information in posting nodes.

Common Parameters
- Merge Strategy
  - Merge Cond. = Never Merge

Varying Parameter
- Static Index
  - (1) = Ext. Token Inversion Index
  - (2) = Stand. Token Inversion Index
- Delta Index
  - (1) = Ext. Token In-Mem. Inversion Index
  - (2) = Standard Token In-Mem. Inversion Index
**Description.** In both configurations all indexed data is stored in the delta index. The delta index is never merged. The size measured is the total size required by all structures of the delta index (vocabulary and posting lists). In the first approach, component information is stored directly in the posting nodes whereas it is stored in the vocabulary in the second one. For a more detailed explanation of the two approaches please refer to section 5.3.2.

**Interpretation.** The first approach requires more space than the second one when the indexed data size is very small. For even moderate data sizes, the second approach (storing component information in vocabulary) leads to a smaller index size. This is most likely due to the fact that storing component information in the vocabulary pays off more for long posting lists than for short ones (see Figure 5.10).
4.2.3 Summary

For the summary we are using the following abbreviations for the criteria given in subsection 4.2.1:

- $C_{up,avg}$: Average time required for an index update.
- $C_{up,worst}$: Time required for an index update ("blocking time") in the worst case.
- $C_{q,avg}$: Average time required for a query.

Conclusions from the experiments:

- An in-memory inverted index implementation has clear advantages over a log implementation for the use as delta index. In respect of $C_{up,avg}$, both solutions are equally competitive, with respect to $C_{up,worst}$ and especially $C_{q,avg}$, the in-memory inverted index performs much better.

- Merge strategies that keep the number of static indices small perform well with respect to $C_{q,avg}$ (Immediate Merge > 4-Way Logarithmic Merge > 2-Way Logarithmic Merge > No-Merge) but perform at the same time badly in respect of $C_{up,avg}$ (No-Merge > 2-Way Logarithmic Merge > 4-Way Logarithmic Merge > Immediate Merge). There is a clear trade-off between $C_{up,avg}$ and $C_{q,avg}$. The overall performance (with respect to $C_{up,avg}$ and $C_{q,avg}$) of a merge strategy depends on the ratio between updates and queries.

- Synchronous merge strategies (except No-Merge) perform badly with respect to $C_{up,worst}$ because new updates must be blocked, at least when the delta index is full, until the merge process has finished. Although Logarithmic Merge performs well in respect of $C_{up,avg}$, the worst case time required for an update is the same as for Immediate Merge.

- All asynchronous merge strategies perform well with respect to $C_{up,worst}$. They have the same performance as a synchronous No-Merge strategy under the assumption that static only merges do not decrease insertion performance.

- An asynchronous strategy can temporarily perform worse in respect of $C_{q,avg}$ than the according synchronous strategy because of a temporarily higher number of static indices.

These results are also roughly summarized in table 4.1. One can for example see in that table that for a configuration optimized for fast query processing a synchronous merge strategy should be used that keeps the number of static indices low (Immediate Merge) and uses a high buffer size ratio (1.0).
<table>
<thead>
<tr>
<th>parameters correlating with multiple criteria</th>
<th>$C_{up,avg}$</th>
<th>$C_{up,worst}$</th>
<th>$C_{q,avg}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>use asynchronous merge strategy (multi-core CPU)</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>use asynchronous merge strategy (single-core CPU)</td>
<td>- / +</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>use merge strategy that keeps the number of static indices small</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>use a high buffer size ratio</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 4.1: Evaluation criteria dependency
Chapter 5

Implementation

In this chapter we present a few selected component implementations more in detail. Note that it is not our intention to explain every single detail of every implementation. Instead we concentrate on a few interesting problems that have arisen during the development and show how we have addressed them.

5.1 Background: iMeMex Specifics

In the previous chapters we used the terms “indexing documents” or “indexing document collections”. In the context of iMeMex, there is some ambiguity in the use of these terms. In iMeMex, the indexing system does not operate directly on documents or files but on the logical representation of the personal dataspace. The data from all data sources is represented as a single graph consisting of elements called resource views (see Figure 5.1). In some simple cases, a document may be mapped to just one resource view. In general, however, a document - or another physical data entity - is represented as a sub-graph in the resource view graph. Regardless of the actual structure of that graph, the index must be updated whenever there is one of the following occurrences:

- A new resource view is added to the graph.
- An existing resource view is removed from the graph.
- An existing resource view is modified (this can however also be modeled as a deletion followed by a re-insertion).

Every resource view is identified by a unique oid. Although the process of indexing resource views is not so much different from that of text documents, there are some specialties.

1. A resource view consists of several components that all may contain tokens that must be indexed. As a consequence of that, the index must also
1. Introduction

1.1 Research Challenges

The Problem

In section 3.1 the delta index was defined as an updatable index structure that is able to process queries on the indexed data. Besides that, a

5.2 In-Memory Inverted Index

The in-memory inverted index presented in this section is an implementation of a delta index. In section 3.1 the delta index was defined as an updatable index structure that is able to process queries on the indexed data. Besides that, a

include the information in which component and at which position inside that component a token occurs (see Figure 5.3).

2. A resource view may contain tokens of various types (strings, numbers, dates, booleans, etc).

Implementations of both delta indices and static indices must take these two aspects into account.

Figure 5.1: Dataspace representation as resource view graph (from [9])

Figure 5.2: Delta index / static index interfaces (simplified)
The delta index must also be able to make its index data available in some form when it is merged. The left side of Figure 5.2 shows these specifications in form of a simplified interface.

### 5.2.1 General Architecture

**Multiple Token Types.** As previously mentioned, a resource view can contain data of various types that must be indexed. In the implementation presented in the following, this problem is solved by having a sub index for each data type (see 5.3).

**Vocabulary.** For a mapping from tokens to the corresponding posting lists, the TreeMap class from the Java API is used. It provides an implementation of a balanced binary tree guaranteeing logarithmic time complexity for updating or querying the vocabulary. If the vocabulary would only be used to answer point queries, it would of course be more efficient to use a hash map. In our case, sorting the tokens is, however, necessary because the index should also support efficient lookups for range queries.

**Posting List Structure.** A posting list contains all posting nodes associated with a certain token value. One can think of many data structures for storing a posting list. The most obvious ones are probably an array or a linked list of posting nodes. The linked list requires extra memory for the pointers but allows to add or remove posting nodes without changing the rest of the posting list. The array requires also extra space (depending on its growth strategy), and the whole posting list must be copied whenever the array has to be resized. In practice, both of these solutions suffer from the problem of a high memory...

---

Figure 5.3: Tokenizing resource views
overhead for objects. An object in Java typically requires 8 bytes of object overhead. For the reference to this object another 4 bytes are added, resulting in 12 bytes of memory overhead for every single posting node. Then it must also be considered that a posting node also contains the positions of the token occurrences. In the simplest case that information may be represented in an array, adding another 16 bytes of memory overhead, resulting in a (minimal) total overhead of 28 bytes per posting node. We are assuming here that the position of a token occurrence can be expressed with a primitive type and does not require another object, otherwise the object overhead is even higher. Eventually, this means that a high percentage of the memory used by the inverted index is attributed to rather object overhead and object references than to the actual data. This is undesirable because a high memory overhead means that the delta index reaches its maximum size earlier. This in turn increases the total number of merges that is required for indexing data of a given size. Depending on the merge strategy this eventually has the effect that more data has to be transferred from and to disk (Immediate Merge) and/or the number of static indices is increased (No-Merge). It is therefore important to keep the number of merges as small as possible.

To avoid the problem of memory overhead, posting nodes are not kept in the form of objects but are written in a serialized form to a byte array. When a posting list is read, it is reconstructed from its serialized form. To reduce memory consumption even further, posting nodes, or at least parts of it, are saved in a compressed form. This is done in two steps:

1. Oids stored in posting nodes are delta encoded. Instead of storing the actual oid only the difference to the previous oid is stored (within a posting list nodes are sorted by oid). The same applies to positional information stored in posting nodes.

2. As the delta encoded values from the previous step can be expected to be small, they are compressed by using a seven-bit compression scheme. Seven bits of a byte are used to store information, and one bit is reserved as “stop-bit”. This reduces data size considerably for small delta values. This scheme also has the advantage that it does not require much computational power as it operates on byte boundaries.

This is a typical case of a space-time trade-off. Serializing and compressing the index data requires additional processing time, on the other hand less memory is required to store the index. This in turn reduces the number of merges required for indexing data of a given size.

5.2.2 Update Operations

Inserting Resource Views. The general process of inserting a resource view consists of three steps:
1. Tokenization: Tokens are extracted from the resource view as seen in Figure 5.3 and automatically forwarded (according to a push model) to sub indices according to their type.

2. Aggregation: Tokens, together with the position information, are grouped by token value. This is done by using a hash-based early aggregation approach reducing the size of the data that has to be handled. The positions of all tokens of a group are aggregated, resulting in a posting node for every unique token value.

3. Whenever a new posting node is created, it is added to the posting list associated with the respective token value. The posting list is found by looking up the token value in the previously described binary tree. If no posting list associated with the token value exists, a new one is created and inserted into the tree.

In the following, the process of adding a new posting node to the posting list is further explained. Note that the posting list is always ordered by oid. We distinguish between two cases:

1. The oid of the posting node that is inserted is higher than the last posting node in the list.
2. The oid of the new posting node is smaller than the last posting node in the list.

We are always confronted with the first case when a new resource view is created (oids are assigned in an ascending order). Note that in such a case a posting node can be appended to the posting list without actually having to decompress any part of the posting list. This is achieved by storing the oid of the last posting node separately in a non-encoded form (see Figure 5.4).

In the second case, things get a bit more complicated. One is confronted with that case when an existing resource view is modified (modeled as a deletion followed by a reinsertion). Because the new posting node must be inserted somewhere in the posting list, the whole posting list must be decompressed and rebuilt. This can become expensive for long posting lists. To solve that problem, a strategy as depicted in Figure 5.5 could be used. A primary pointer array is introduced that contains references to posting list partitions. Whenever a partition has reached a certain size, it is split in two. This requires some book keeping overhead but has the following advantages:

- When a new posting node is inserted, only one partition must be decompressed and rebuilt. The corresponding partition can be found by using binary-search on the primary array. (Alternatively one could also use a tree instead of a pointer array.)

- When a posting node (with a certain oid) is deleted, only one partition must be decompressed and rebuilt. The posting node can also be found by using binary-search on the primary array.

- Without partitions, the whole posting list must be copied to a new memory location when the byte array has to be resized. When having partitions, a small new byte array is allocated, resulting in a new partition without having to touch the other partitions.

Deleting Resource Views. When a resource view has to be deleted, only its oid is provided to the indexing system (and not the deleted resource view itself).
ABBREVIATIONS:

R: resource view
B: bloom filter set
M: map (oid → bloom filter set)
D: set containing oids of (lazily) deleted resource views

Figure 5.7: Illustration of the process of deleting a resource view from the index
As a consequence, it is not known what tokens the deleted resource view had contained before it was deleted. This results in the problem that basically every posting list could be affected by the deletion. One could now just decompress all posting lists one after another and search for posting nodes to be deleted. With ten or even hundred thousands of posting lists this approach performs however very badly.

A first improvement is made by using a post-filtering approach. Instead of “physically” deleting resource views from posting lists a set is maintained that contains the oids of all deleted resource views, we refer to this approach as lazy deletion. When the inverted index is queried or all data is “bulk-read” the deleted posting nodes are post-filtered. This allows to massively speed up deletions for the price of a certain amount of wasted memory and a moderate increase in query costs. It is however important to understand that this works only as long as a resource view is not reinserted after it has been deleted. When a resource view is inserted that has been previously deleted (meaning its oid is in the set of deleted oids) that situation can be handled in at least two ways:

1. All posting nodes associated with the according oid are (physically) removed from all posting lists. Then the oid is removed from the set of deleted oids and the (new) resource view is reinserted.

   or

2. Posting nodes are assigned timestamps (or version numbers) when they are inserted, this allows to filter out posting nodes selectively without ever having to delete posting nodes physically.

We decided to stick with the first approach assuming that the second one wastes too much memory. Think for example of a resource view representing a large log text file. When that text file gets constantly modified and a timestamp/versioning approach is used, the delta index gets filled up with many different versions of that text file. This wastes a lot of memory. On the other hand, it may be worth to evaluate that approach again in future work in combination with a garbage collection approach that physically removes obsolete posting nodes when a certain threshold of deletions has been reached.

As previously illustrated, the post-filtering approach is helpful when a resource view is deleted once. The problem of having to process all posting lists, however, re-emerges as soon as the resource view is reinserted. One could now argue that it is not often the case that a resource view is deleted and reinserted again. Unfortunately, this is not true and reinsertions are expected to happen quite regularly. That is due to the fact that resource view modifications are modeled as deletions followed by reinsertions (see Figure 5.6).

The main reason why deletions are more difficult to handle than insertions is due to the fact that one cannot make use of the vocabulary that maps from tokens to posting lists. What is needed instead to support deletions is a mapping from oids to posting lists. Such a mapping can be provided by keeping a hash
map that maps from an oid to an array of posting list references. The number of additional posting list references introduced by such a mapping is the same as the total number of posting nodes. The memory required by such a structure can therefore be estimated as an additional 4 Bytes for every posting node. Depending on the average posting node size this can be a lot or not much at all.

In the in-memory inverted index implementation, we tried to further reduce the memory required by the structure mapping oids to posting lists. For that reason, instead of using arrays containing actual references to posting lists we have used sets based on Bloom filters [2]. For every inserted resource view a small Bloom filter is created into which the posting list references are inserted. When a resource view is deleted the Bloom filter based set is obtained by looking up the oid in a hash map. Then for every posting list it is determined whether it contains the said oid by querying the Bloom filter set. As a consequence, only those posting lists must be processed (and uncompressed) that really contain an entry associated with the deleted resource view. Due to the probabilistic nature of Bloom filters there will be a certain probability of false positives that depends on the parameters of the Bloom filters that are used. These cases however carry no weight when reasonable parameters are used. Without going into details of the parametrization of Bloom filters, it is obvious that different parameters are needed for resource views associated with a different number of posting lists. When a resource view is inserted the associated Bloom filter is then created in the following way:

1. Posting Nodes that have been created from the resource view are inserted into posting lists. During that process, the references to these posting lists are remembered.

2. When all posting nodes have been processed the Bloom filter is created. As the number of posting list references that will be added to the Bloom filter is already known, the parameters of the Bloom filter can be optimally set such that a certain false positive rate is acquired. (The more references are inserted into the Bloom filter the more memory is required to acquire a certain false positive rate.)

In the in-memory inverted index implementation the Bloom filter parameters are (by default) chosen such that an additional 2 Bytes are required per posting node (or posting list reference). This means only half the memory is required compared with the approach where the actual references are stored. Note that for that parameter setting the false positive rate is very low (< 1:2000). To get an idea of the actual memory requirement we have measured memory usage in the following scenario: Resource views corresponding to text files of a total size of 110 MB are inserted into the in-memory inverted index. This then leads to a total index size in memory of about 57 MB. Of that total size only 3.5 MB are used for Bloom filter sets (including the memory used by the hash map that
maps oids to the corresponding sets). Another positive effect of this approach is that the lookup in the hash map (that maps from an oid to a Bloom filter set) instantly gives us the information whether anything has to be deleted at all. (If the oid of a resource view is not present in the hash map then the index contains no data associated with that resource view.)

Figure 5.7 summarizes the deletion process as we have implemented it in our current in-memory inverted index implementation: Resource views are deleted lazily and only deleted physically from the posting lists when a resource view with the same oid is reinserted. Whenever a resource view is inserted into the index a Bloom filter set is created that speeds up the deletion process in case the resource view has to be deleted physically.

Handling Queries. As seen at the beginning of this section, the index is partitioned by data type. When querying the index, this has the positive side-effect that only the sub index corresponding to the data type of the query must be touched. The query operation is then straightforward and can be divided into the following steps:

1. Vocabulary Lookup: Tokens are looked up in the vocabulary that maps them to the according posting lists. Note that single keyword queries will only require one lookup whereas other types of queries (like for example range queries) require several lookups.

2. Processing Posting List(s): The posting lists that were identified in the vocabulary lookup must be uncompressed. Note that this will not be done in one large single step: As previously explained, the index answers queries by providing a pull operator that allows to fetch the result set element by element. Posting lists are therefore also uncompressed step by step whenever the next results must be delivered.

3. Post-Filtering: In the previous paragraph it was illustrated that resource views may be deleted lazily. Therefore, posting nodes belonging to resource views that are marked as deleted have to be filtered out. Note that some types of queries also require further post-filtering of certain nodes. Think for example of the query “What resource views have a name containing the term 'hello'?”. The posting list corresponding to the term 'hello' contains posting nodes associated with resource views containing that term in any component (for example in the content or in an attribute) and not only in the name. In such a case, posting nodes must therefore also be post-filtered by the positional information stored in the posting node.
5.3 On-Disk Inverted Index

In the previous chapter several configurations were presented that all make use of a static index implementation based on an on-disk inverted index. In this section some aspects of this implementation are presented. The functionality that this implementation must provide is summarized in the simplified interface shown on the right side of Figure 5.2.

5.3.1 General Architecture

As previously mentioned a resource view may contain tokens of various types. In the on-disk inverted index implementation, this fact is taken into account by having a sub index for each data type. This is the same approach as described for the in-memory inverted index implementation. The structure of a sub index is that of a typical inverted index as explained in 2.2 consisting of the following two parts:

- The inverted file is a file on disk that contains all posting lists in a sequential manner.
- The vocabulary provides a mapping from a term to the offset of the corresponding posting list within the inverted file.

Tobias Abt has provided an implementation of a generic external memory inverted index in his semester thesis [1]. It allows to build up a read-only inverted index by bulk loading. In the context of our (multi type) on-disk inverted index that code is reused as basis for the sub index implementation and has been adapted to our needs.

Inverted File. Posting lists within the inverted file are sorted lexicographically by the terms they are associated with. This allows to efficiently process range queries without introducing additional disk seeks. Moreover posting lists are written to the inverted file in a compressed form similarly as presented in the previous section for the in-memory inverted index implementation. Delta-encoding in combination with seven-bit compression is used to reduce the size of the data that has to be written to and read from disk (based on the assumption that CPU costs for compression are smaller than the saved IO costs).

Vocabulary. In general it can not be assumed that the vocabulary (or in our case vocabularies) will fit into main memory. Therefore, we have implemented a vocabulary based on a bulk loadable read-only B+ tree. To reduce the number of disk seeks the internal nodes of the tree are cached in memory. Moreover, leaf nodes are linked avoiding additional seeks for range lookups.
Figure 5.8: On-disk inverted index input data type comparison

Figure 5.9: Merging posting lists
5.3.2 Operations

Bulk Loading. In section 3.4 we have illustrated that a static index implementation can be used together with any delta index implementation as long as it is able to handle the output data format of that delta index. Figure 5.8 shows two examples of different delta index output data formats. In the first case, the static index implementation requires resource views as input. Such an input data format is chosen when the static index implementation is expected to be used together with delta index implementations that can output data in the form of resource views (like for example a log). In our case, however, we want the on-disk inverted index to perform well with delta index implementations that provide data already in the form of posting lists (like the in-memory inverted index implementation presented in the previous section). Such a scenario is shown in the second part of Figure 5.8. Note that this Figure illustrates only the data coming from a delta index. In reality, a static index may be created also by merging a delta index with one or many static indices (or by merging multiple static indices without the delta index). Figure 5.9 shows an example of merging two inverted indices (for simplicity only the string data type is illustrated). As in both indices posting lists are sorted lexically, they can be merged easily by traversing each index sequentially only once.

Handling Queries. Query handling is straightforward. First the B+ tree vocabulary is used to get the inverted file offsets of posting lists associated with the query terms. Then the posting lists are read from the inverted file. As mentioned before query results are not returned all at once. Instead a pull operator is returned that allows to fetch results one after another.

Moving Component Information Into Vocabulary. Resource views consist of various components. The index must therefore not only contain the information of what resource views a term or token occurs in but also where exactly within a resource view it occurs. One possibility is to store this information
in the posting nodes. The green index in Figure 5.10 illustrates this approach: As an example the first posting node of the posting list means that the term “hello” occurs in the component component of the resource view with oid 42 at the positions 2 and 52. The downside of this approach is that the component information is stored for every posting node even when most posting nodes refer to the same component.

The blue index in the same figure shows an alternative approach. Instead of storing the component information in the posting nodes, that information is combined with the token and stored in the vocabulary. This will lead to smaller posting lists. On the other hand, the vocabulary will use more space. In such an approach, there may be multiple entries in the vocabulary having the same term but different component information. The advantage is that component information must be stored only once for all posting nodes in a posting list. The downside is however that terms may be stored in the vocabulary multiple times.

Which of these two approaches should be used depends on the indexed data. The provided on-disk inverted index implementation is generic and supports both approaches. Although not mentioned previously the same applies also to the in-memory inverted index implementation presented in the previous section. In experiments of indexing textual data the second approach of storing component information in the vocabulary led to smaller index sizes (see experiment 4.2.2.9) and required less indexing time.

### 5.4 Merge Strategies

**Planning Merges.** Figure 5.11 shows a simplified version of the merge strategy interface as it is used in our dynamic index implementation:

- By providing *evaluation preferences* a merge strategy determines when or on what occasions it should be evaluated whether a merge should be initiated or not.
• A merge strategy must determine whether new insertions should temporarily be blocked. This is done based on statistics from the delta index.

• The most important aspect of a merge strategy is the planning of merges (in context of the interface planning also includes determining whether a merge should be initiated at all). In order to be able to plan merges a merge strategy is provided with:
  
  – Statistics from the delta index.
  – General statistics from the dynamic index.
  – A collection of all static indices that can currently be used for a merge. (Meaning all active static indices that have not yet been reserved for another merge.)

Statistics from the delta index contain information about the number of indexed resource views and the size of the delta index in memory. Statistics from the dynamic index include various information like for example the number of currently running merges or the number of currently blocked insertions.

**Composed Merge Strategy.** In subsection 4.1.3 we referred to a merge strategy implementation that is configured by choosing a combination of sub component implementations. We are calling this merge strategy implementation a *composed* merge strategy. We have already shown that it is useful in chapter 4 where we have reused various merge strategy sub components to compose merge strategies with various properties.
Chapter 6

Analysis

In this chapter we present our results from studying the No-Merge, Immediate Merge and $b$-Way Logarithmic Merge strategies from an analytical point of view. For all of these strategies we first derive cost functions. Based on these cost function we then propose a cost model. The cost model allows to determine which merge strategy, among the ones just mentioned, is the best for a given scenario.

6.1 Merge Strategy Cost Model

It is important to note that the purpose of our merge strategy cost model is not to determine the absolute total update and query costs of a merge strategy in a given scenario. Instead, the purpose of our model is to determine which merge strategy has the lowest costs in a given scenario (regardless of the absolute value of the costs). This difference is important because it means that only those costs need to be considered in the cost model that depend on the merge strategy. In other words, we are not interested in costs that are the same for all merge strategies because they do not give us any information that helps to determine the strategy with the lowest costs.

In our cost model we divide the indexing and querying process in a sequence of discrete steps. A step $n$ includes the following operations:

1. Data is inserted into the delta index until it is full. The maximum size $S_{\Delta}$ of the delta index is constant and the same for all steps. Note that the costs for inserting data into the delta index in step $n$ are always the same regardless of the chosen merge strategy. As these costs are independent from the merge strategy they do not give us any information about which strategy to prefer. Therefore, we do not need to consider them in our cost model.

2. A delta merge is executed. This merge may also include data from static indices as determined by the merge strategy. In our cost model, we assume
Figure 6.1: Illustration of $c_{U}(n)$, $c_{Q}(n)$ for immediate and No-Merge
Figure 6.2: Illustration of $c_U(n)$, $c_Q(n)$ for Logarithmic Merge (unoptimized)
Figure 6.3: Illustration of $c_U(n)$, $c_Q(n)$ for Logarithmic Merge (optimized)
that the costs of this merge process are proportional to the size of the
data involved in merges in that step. We define $c_U(n)$ as the function
returning the size of the data involved in merges in step $n$ expressed in
multiples of $S_\Delta$.

3. Queries are processed. The costs for processing queries in step $n$ consist
of:

(a) Costs for querying the delta index. Note that we do not need to
consider them in the cost model because they are the same for all
merge strategies.

(b) Costs for querying the static indices that are present in step $n$. We
can divide these costs into two types as follows:

i. Query costs that depend on the total size of all static indices
present in step $n$ (e.g. disk transfer cost, data processing cost).
These costs are again the same for all merge strategies. (In
every step, data of the size $S_\Delta$ is added to the dynamic index
regardless of the merge strategy.)

ii. Query costs that are proportional to the number of static indices
(e.g. cost for disk seeks). These costs have to be considered
in the cost model because the number of static indices in step
$n$ is determined by the merge strategy. We define $c_Q(n)$ as the
function returning the number of static indices in step $n$ (after
the merges have been processed).

6.2 Cost Functions

In the following paragraphs we introduce the cost functions that we have derived
for our cost model. They describe the progressions of update and query costs as
illustrated in the figures 6.1, 6.2 and 6.3. Note that we are using the following
abbreviations in formulas:

$NM$ No-Merge

$IM$ Immediate Merge

$LM[\alpha/u](b)$ $b$-way Logarithmic Merge (optimized / unoptimized)

$U$ Update

$Q$ Query

$\notin$ Intersecting
For every merge strategy $M$ we derive two functions that sum up update and query costs of all steps from $1..n$:

$$C_{U,M}(n) = \sum_{i=1}^{n} c_{U,M}(n)$$
respectively

$$C_{Q,M}(n) = \sum_{i=1}^{n} c_{Q,M}(n)$$

### 6.2.1 No-Merge

Figure 6.1 illustrates the cost progressions for the No-Merge strategy. The update costs are constant and query costs grow linearly with every step. This leads directly to the following cost functions. Note that we have used the arithmetic series formula for the sum of the query costs:

$$C_{U,NM}(n) = \sum_{i=1}^{n} 1 \quad (6.1)$$

$$C_{Q,NM}(n) = \sum_{i=1}^{n} n \quad (6.2)$$

(arithmetic series) \[ \frac{n^2 + n}{2} \]

### 6.2.2 Immediate Merge

The costs for the Immediate Merge strategy - illustrated in 6.1 - are the same as for No-Merge with the difference that update and query costs are swapped:

$$C_{Q,IM}(n) = \sum_{i=1}^{n} n \quad (6.3)$$

(arithmetic series) \[ \frac{n^2 + n}{2} \]

$$C_{U,IM}(n) = \sum_{i=1}^{n} 1 \quad (6.4)$$

$$= n$$

### 6.2.3 Logarithmic Merge

When analyzing the Logarithmic Merge strategy, we differentiate between the unoptimized and the optimized variant. When examining the numbers in the Logarithmic Merge figures 6.2 and 6.3 one can see that they are related to the following sequences:
• \((0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, 0, \ldots)\) is known as \textit{binary carry sequence}. The value at index \(n\) is given by the highest power of 2 dividing \(n\) without rest.

• \((1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 3, 4, 1, 2, \ldots)\) is known as \textit{binary digit sum}. The value at \(n\) is determined by counting the number of 1’s in the binary representation of \(n\).

The concepts of these two sequences can be generalized to base \(b\): The carry sequence value at index \(n\), with respect to \(b\), is the highest power of \(b\) that divides \(n\) without rest. The digit sum of \(n\) is the sum of all digits of the \(b\)-ary representation of \(n\). For these generalized number sequences we have derived the following explicit functions:

\[(\text{carry sequence}) \quad cs(n, b) = \sum_{i=1}^{[\log_b n]} \left\lfloor \frac{n}{b^i} \right\rfloor - \sum_{i=1}^{[\log_b n-1]} \left\lfloor \frac{n-1}{b^i} \right\rfloor \quad (6.5)\]

\[(\text{digit sum}) \quad ds(n, b) = n + (1 - b) \sum_{i=1}^{[\log_b n]} \left\lfloor \frac{n}{b^i} \right\rfloor \quad (6.6)\]

On the basis of 6.5 and 6.6 we formulate the cost functions for the unoptimized \(b\)-way Logarithmic Merge strategy as follows:

\[c_{U,LMu}(n, b) = \sum_{i=0}^{cs(n, b)} b^i \quad (6.7)\]

\[C_{U,LMu}(n, b) = \frac{b^{cs(n, b)+1} - 1}{b - 1} \quad (6.8)\]

\[c_{Q,LMu}(n, b) = ds(n, b) \quad (6.9)\]

\[C_{Q,LMu}(n, b) = \frac{b^{ds(i, b)+1} - 1}{b - 1} \quad (6.10)\]

The optimized Logarithmic Merge strategy has the same query costs as the unoptimized one. Update costs are, however, lower in those cases where
Figure 6.4: Continuous approximation of discontinuous update cost function of optimized Logarithmic Merge

cascaded merges are anticipated and reduced to a single merge. We have derived the following cost functions:

\[
c_U,LM_0(n, b) = b^{cs(n, b)} 
\]  
(6.11)

\[
C_U,LM_0(n, b) = \sum_{i=1}^{n} b^{cs(n, i)} 
\]  
(6.12)

\[
C_Q,LM_0(n, b) = C_Q,LM_u(n, b) 
\]  
(6.13)

\[
= \frac{n^2 + n}{2} + (1 - b) \sum_{k=1}^{n} \sum_{i=1}^{\lfloor \log_b k \rfloor} \left\lfloor \frac{k}{b^i} \right\rfloor
\]

6.2.4 Combining Cost Functions

We have defined two types of costs: query costs related to the number of static indices and update costs related to the data size involved in merges. When considering only one cost type, merge strategies can easily be compared by using the according formula. When both cost types are considered, we must specify how query and update costs relate to each other. We therefore introduce the relative query cost factor \( q \) and the relative update cost factor \( u \). As absolute values are not of interest we specify that \( q + u = 1 \) and we can therefore express the relative update cost factor as \( u = 1 - q \). More details are given later in section 6.4 including information about how this factor can be computed for a given system or scenario.
Based on the cost functions given in section 6.2 the total merge strategy costs according to our model can then be calculated with the following functions:

\[
C_{NM}(n, q) = (1 - q) n + q \left( \frac{n^2 + n}{2} \right) \tag{6.14}
\]

\[
C_{IM}(n, q) = (1 - q) \left( \frac{n^2 + n}{2} \right) + q n \tag{6.15}
\]

\[
C_{LMu}(n, b, q) = (1 - q) \sum_{i=1}^{n} \frac{k^{cs(i,b)}+1}{b-1} - 1 + q \left( \frac{n^2 + n}{2} + (1 - b) \sum_{k=1}^{n} \sum_{i=1}^{\left\lfloor \log_b k \right\rfloor} \left\lfloor \frac{k}{b^i} \right\rfloor \right) \tag{6.16}
\]

\[
C_{LMo}(n, b, q) = (1 - q) \sum_{i=1}^{n} b^{cs(n,i)} + q \left( \frac{n^2 + n}{2} + (1 - b) \sum_{k=1}^{n} \sum_{i=1}^{\left\lfloor \log_b k \right\rfloor} \left\lfloor \frac{k}{b^i} \right\rfloor \right) \tag{6.17}
\]

For a given number of merges \( n \) and a relative query cost factor \( q \) the best merge strategy can now be determined. This is done by comparing total costs as calculated by the above functions. A problem of the Logarithmic Merge cost functions 6.16 and 6.17 is, however, that they are not suited very well for further analysis. This is due to their rather complex discontinuous logarithmic behaviour (see for example figure 6.5). We addressed this problem by replacing these discontinuous functions with similar continuous ones. This approximation is based on the observation that the behaviour of the discontinuous logarithmic functions can be described more easily with respect to all domain values \( n = \)}
Figures 6.4 and 6.5 illustrate two approximation functions for \( b = 2 \). Note that all values \( C(n = b^i), i \in \mathbb{N} \) of these continuous functions exactly match the corresponding value of the discontinuous function. In-between these values, merge costs are strictly increasing. The formulas that we have derived for these continuous update and query cost functions are as follows:

\[
\begin{align*}
\tilde{C}_{U, LMu}(n, b) &= n (1 + \log_b n) \\
\tilde{C}_{U, LMo}(n, b) &= \frac{n + (b - 1) \log_b n}{b} \\
\tilde{C}_{Q, LM}(n, b) &= 1 + \frac{(b - 1) \log_b n}{2}
\end{align*}
\]

This results in the following formulas for total costs:

\[
\begin{align*}
\tilde{C}_{LMu}(n, b, q) &= (1 - q) n (1 + \log_b n) + q \left( 1 + \frac{(b - 1) \log_b n}{2} \right) \\
\tilde{C}_{LMo}(n, b, q) &= (1 - q) \frac{n + (b - 1) \log_b n}{b} + q \left( 1 + \frac{(b - 1) \log_b n}{2} \right)
\end{align*}
\]

### 6.3 Comparing Merge Strategies

In this section we use the formulas described in the previous section to compare merge strategies with each other with respect to our model.

#### 6.3.1 No-Merge vs. Immediate Merge

In a first step, we compare the Immediate Merge with the No-Merge strategy. We set the cost functions \( 6.14 \) and \( 6.15 \) equal and solve the resulting equation by \( q \):

\[
C_{NM}(n, q) = C_{IM}(n, q)
\]

\[\rightarrow \quad q_{NM|IM}(n) = \frac{1}{2}\]

This result is no surprise due to the symmetry of the two cost functions. Figure 6.6 shows the cost functions for \( n = 4 \) and \( n = 8 \) with respect to \( q \). Independently from the number of merges \( n \) the Immediate Merge strategy is always better than the No-Merge strategy for a scenario where \( q > \frac{1}{2} \) (and worse for \( q < \frac{1}{2} \)).
6.3.2 Immediate Merge vs. 2-Way Logarithmic Merge

In this paragraph, we study the relation between the Immediate Merge and the 2-way Logarithmic Merge strategy. We are interested in this combination because the 2-way Logarithmic Merge strategy is the one closest to Immediate Merge. As in the previous example, we want to find the value of \( q \) for which both strategies have equal costs. As before, we do that by setting the cost functions equal and solving the equation by \( q \). From a geometrical point of view this corresponds to finding the point of intersection of the lines given by the cost functions for a fixed value of \( n \). (Cost functions are linear with respect to \( q \).)

\[
C_{IM}(n, q) = \tilde{C}_{LMo}(n, 2, q)
\]
Figure 6.7: Intersecting immediate and 2-way Logarithmic Merge cost functions (for $n = 8$ and $n = 16$)

\[ n = 16, \text{im. merge} \]
\[ n = 16, b = 2 \log \text{merge} \]
\[ n = 8, \text{im. merge} \]
\[ n = 8, b = 2 \log \text{merge} \]

\[ q_{\text{IM} \not\text{LM}2o}(n) = \frac{n^2 - n \log_2 n - n}{n^2 - 3n + 2} \quad (6.23) \]

\[ q_{\text{IM} \not\text{LM}2o}(n \to \infty) = \lim_{n \to \infty} \frac{n^2 - n \log_2 n - n}{n^2 - 3n + 2} = 1 \quad (6.24) \]

\[ (\text{l'Hopital's rule}) = \lim_{n \to \infty} \frac{2n - (\log_2 n + \frac{1}{\ln 2}) - 1}{2n - 3} \]

Figure 6.7 shows the cost functions of the two strategies with respect to $q$ for $n = 8$ and $n = 16$. The intersection points in that figure match those in table 6.1 (calculated with formula 6.23). The figure also shows that the interval in which Immediate Merge performs better than the 2-way Logarithmic Merge strategy shrinks when the number of merges $n$ is increased. For (moderately) large values of $n$ that interval becomes very small and is close to $q = 1$. Note that a relative query cost factor of $q = 1$ corresponds to a scenario where merges do not cost anything at all.
### Table 6.2: Point of intersection for 'neighbour' Logarithmic Merge strategies

<table>
<thead>
<tr>
<th>Merge Fan-In $b$</th>
<th>Point of Intersection $q$ for $C_{LMo}(n, b, q) = C_{LMo}(n, b + 1, q)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0.377444</td>
</tr>
<tr>
<td>3</td>
<td>0.277004</td>
</tr>
<tr>
<td>4</td>
<td>0.214783</td>
</tr>
<tr>
<td>8</td>
<td>0.105719</td>
</tr>
<tr>
<td>32</td>
<td>0.020685</td>
</tr>
<tr>
<td>1024</td>
<td>0.000326</td>
</tr>
<tr>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 6.8: $b$-way Logarithmic Merge cost functions for $n = 64$ (blue), $n = 128$ (red), $n = 256$ (yellow)
6.3.3 b-Way vs. (b+1)-Way Logarithmic Merge

In the previous paragraph we have studied the right part of the spectrum of $q$ where query costs, as we have defined them in our model, are high in relation to update costs. In this paragraph we study the influence of increasing the fan-in of the Logarithmic Merge strategy. When the fan-in is increased, update costs are weighted less because there are fewer merges. On the other hand, query costs get more weight because of the increased number of static indices.

We use the following equation to find the point of intersection of the cost functions of two consecutive Logarithmic Merge strategies ($b$-way and $b+1$-way).

$$\tilde{C}_{LMo}(n, b, q) = \tilde{C}_{LMo}(n, b + 1, q)$$  \hspace{1cm} (6.25)

$$q_{LMo(b)\mid LMo(b+1)}(n) = \frac{2 \left( (b^2 - 1) \ln(b + 1) - b^2 \ln(b) \right)}{(b - 1) \left( b^2 \ln(b) - (b^2 - b - 2) \ln(b + 1) \right)}$$  \hspace{1cm} (6.26)

Interestingly the points of intersection do not depend on the number of merges $n$. With the help of formula 6.25 we can determine for every $b$-way Logarithmic Merge strategy the interval in which this strategy is better than all other (Logarithmic Merge) strategies. Table 6.2, for example, shows that the 4-way Logarithmic Merge strategy the best choice for a scenario where $0.214783 \leq q \leq 0.277004$.

Figure 6.8 shows the Logarithmic Merge cost functions for a range of fan-ins$^1$ for $n = 64$, $n = 128$, and $n = 256$. This figure illustrates that the interval in which a fan-in $b$ is the best choice gets smaller for large values and at the same time moves closer to $q = 0$. The 2-way Logarithmic Merge strategy is, for example, the best of all (Logarithmic Merge) strategies in the interval $0.377444 \leq q \leq 1.0$. The 1024-way Logarithmic Merge strategy, on the other hand, is the best choice only in the interval $0.0003260 \leq q \leq 0.0003263$.

6.4 Relative Query Cost Factor Estimation

We have explained in section 6.1 that the query and update costs, as defined in the cost model, are only those costs that are affected by the choice of the merge strategy. Based on the description of our cost model we can define the absolute query and update costs of a system or scenario as follows:

$q_{abs}$ Absolute costs for processing queries on one static index in-between two merges when only those query costs are considered that are proportional to the number of static indices.

$^1$The fan-ins for the cost functions shown in Figure 6.8 are $b = [2, 3, 4, 5, 6, 8, 12, 16, 32, 64, 128, 256]$. 
Absolute costs for merging data of the size of $S_\Delta$ when only those merge costs are considered that are proportional to the size of the data involved in the merge.

Furthermore we have explained in section 6.2.4 that the relative query cost factor $q$ describes the relation between update and query costs. It can be defined as:

$$q = \frac{q_{abs}}{q_{abs} + u_{abs}}$$  \hspace{1cm} (6.27)

The problem that remains now is to estimate values for $q_{abs}$ and $u_{abs}$ for a given system or scenario. In the following, we suggest two estimation approaches.

6.4.1 Estimation of $q_{abs}$, $u_{abs}$ Using Least Squares

In this approach we first model the absolute total time required for query processing and merging in step $n$, considering all costs including those that are independent of the merge strategy. In order to do that, we define the following parameters:

Parameters used for estimating $q_{abs}$:

$T_{Q,n}$ Total query processing time for all queries in step $n$

$N_n$ Number of processed queries in step $n$

$N_{avg}$ Average number of queries in one step.

$D_n$ Total size of static indices in step $n$

$S_n$ Number of static indices in step $n$

Parameters used for estimating $u_{abs}$:

$T_{U,n}$ Total time required for merges in step $n$

$M_n$ Total size of data included in merges in step $n$ (in multiples of delta index size $S_\Delta$)

Note that all these parameters are known or can be measured in every step $n$.

The total time required for query processing and merging in one step can then be modeled by the following equations:

$$T_{Q,n} = N_n(D_n x + S_n y + z)$$  \hspace{1cm} (6.28)

$$= N_n D_n x + N_n S_n y + N_n z$$

$$T_{U,n} = M_n v + w$$  \hspace{1cm} (6.29)
The values of $v$, $w$, $x$, $y$, and $z$ are unknown. In order to find these values we execute a few merge steps (using a 2-way Logarithmic Merge strategy) and measure the total query processing time and merge time for every step. We then use the gathered data to estimate the unknown values by a least-squares fit. After that, the values of $q_{abs}$ and $u_{abs}$ can be calculated by only considering that part of the total cost formulas that is determined by the merge strategy.

We have defined $q_{abs}$ as the costs for processing queries on one static index. Furthermore we have defined that only those costs should be considered that are proportional to the number of static indices. Similarly we have defined $u_{abs}$ as the costs for merging data of size $S_n$ when only those costs are considered that are proportional to the size of the data involved in the merges. Based on these definitions we can calculate $q_{abs}$ and $u_{abs}$ as follows:

$$q_{abs} = \frac{N_{avg}S_n y}{S_n} = N_{avg} y \quad (6.30)$$

$$u_{abs} = \frac{M_m v}{M_m} = v \quad (6.31)$$

With these values the relative query cost factor $q$ is then calculated with formula 6.27.

**Example.** The following example illustrates the formulas that we have introduced in this section.

Suppose a 2-way Logarithmic Merge strategy (optimized) is used initially\(^2\). A few merge steps are executed and for every step all parameters are recorded or measured. They are shown in table 6.3.

Based on these measurements we set up an overdetermined system of equations for each query costs and update costs:

$$10(x + y + z) = 16$$

Table 6.3: Estimating $q$ example - recorded/measured values

<table>
<thead>
<tr>
<th>n</th>
<th>$T_{Q,n}$</th>
<th>$N_n$</th>
<th>$D_n$</th>
<th>$S_n$</th>
<th>n</th>
<th>$T_{U,n}$</th>
<th>$M_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16 s</td>
<td>10</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>17 s</td>
<td>10</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>23 s</td>
<td>10</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>19 s</td>
<td>10</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>40</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^2\)The Immediate Merge and No-Merge strategies are not suited for our least squares estimation because we need a strategy for which all parameters in our measurements are linearly independent. In the Immediate-Merge strategy the parameter $S_n$ is linearly dependent with the constant factor 1 that is used for $z$. In the No-Merge strategy the parameter $M_m$ is linearly dependent with the constant factor 1 that is used for $w$.\n
88
\[10(2x + y + z) = 17\]
\[10(3x + 2y + z) = 23\]
\[10(4x + y + z) = 19\]

\[
v + w = 10 \\
2v + w = 20 \\
v + w = 10 \\
4v + w = 40
\]

Using the linear least squares approach, we find the values \(x = \frac{1}{10}, y = \frac{1}{2}, z = 1\) and \(v = 10, w = 0\).

With \(y\) and \(v\) we determine \(q_{\text{abs}}\) and \(u_{\text{abs}}\) using the formulas 6.30, 6.31:

\[
q_{\text{abs}} = N_{\text{avg}}y = 5 \\
u_{\text{abs}} = v = 10
\]

Eventually, this results in:

\[
q = \frac{q_{\text{abs}}}{q_{\text{abs}} + u_{\text{abs}}} = \frac{5}{5 + 10} = \frac{1}{3}
\]

Having calculated a relative query cost factor \(q = \frac{1}{3}\), we know from our analysis in the last sections that we should switch to the 3-way Logarithmic Merge strategy (see table 6.2). Note that this value depends on the average number of queries in-between two merges \(N_{\text{avg}}\). If \(N_{\text{avg}}\) changes over time we should also recalculate \(q\). Let us for example assume that \(N_{\text{avg}}\) increases after some time to a value of \(N_{\text{avg}} = 20\). Recalculating \(q_{\text{abs}}\) and \(q\) gives us \(q_{\text{abs}} = 20 \cdot \frac{1}{2} = 10\) and \(q = \frac{10}{10 + 10} = \frac{1}{2}\). The relative query cost factor \(q = \frac{1}{2}\) suggests that we should switch back to the 2-way Logarithmic Merge strategy.

### 6.4.2 Estimation of \(q_{\text{abs}}, u_{\text{abs}}\) By Direct Measurements

Another approach for estimating the values of \(q_{\text{abs}}\) and \(u_{\text{abs}}\) is to try to measure them directly. This is, however, not so easy because only those costs must be measured that fall into the definition of \(q_{\text{abs}}\) and \(u_{\text{abs}}\). In contrast to the least squares approach, one must also have knowledge about the implementation of the static indices. In the case where a static index implementation based on an on-disk inverted index is used, \(q_{\text{abs}}\) and \(u_{\text{abs}}\) could, for example, be measured as follows:

- \(q_{\text{abs}}\) is measured as the total time required for processing all queries in-between two merges divided by the number of static indices. However, not
the whole query processing time is measured but only the time used for vocabulary lookups and initial seeks to the correct offset in the inverted file. Time required for data processing or data transfer are not measured because they can not be influenced by the merge strategies and are the same for all strategies.

- $u_{abs}$ is the total time measured for the merge process (in one step) divided by the total data size involved in the merge in multiples of $S_\Delta$. The time measured for the merge process includes the time required for reading data from disk, writing data to disk, and processing data.
6.5 Prediction vs. Real Data

In this section we compare the predictions provided by our cost model with actual measurements from an experiment. The output of the experiment that we have performed is shown in Figure 6.9.

6.5.1 Experiment Description

The experiment shown in Figure 6.9 is similar to the one we have shown in subsection 4.2.2.7. For every measured data point a growing document collection is indexed. The collection consists of 277 plain text files with a total size of about 110 MB. After every document insertion a certain number of queries is executed. In the following, we denote the query/insertion ratio as $R_{Q/I}$. In figure 6.9 this ratio is shown on the x-axis. The size of the delta index is set to a maximum of 5 MB. The total number of merges that are performed during the indexing process is 29 (for all strategies). In the output of the experiment we see the total time that is required for indexing and queries. No-Merge performs best for a small when only few queries are executed. When $R_{Q/I}$ is increased, the 2-way Logarithmic Merge strategy gets better than No-Merge at some point. Eventually, Immediate Merge requires the least time when $R_{Q/I}$ is set to a very high value.

6.5.2 Cost Model Based Predictions

In order to be able to make predictions, we must first find an estimation for the relative query cost factor $q$ (as described in section 6.4). In the following, we do that by looking at the output of two additional experiments.

- Figure 6.11 shows the progression of the indexing time. The same document collection and delta index size is used as for the experiment shown in Figure 6.9. In the curve the time required for merging can be recognized. We have annotated the Figure with the number of static indices that are involved in the merges. The time required for the merge that involves 16 static indices is about 3.5 seconds. This allows us to estimate the time required for merging data of the size of one static index as shown in equation 6.32.

- For estimating a value for $q_{abs}$ we have reused the data from the experiment that we have shown in subsection 4.2.2.6. In that experiment the overhead for query processing per static index can be clearly recognized. In Figure 6.10 we see that this overhead is about 0.16 seconds. In that experiment, 5000 queries were executed. We can therefore estimate the overhead per query (and static index) as 0.16 seconds divided by 5000. This directly leads to equation 6.33.
Figure 6.9: Query/Update Trade-Off Between Immediate/Logarithmic/No-Merge
Figure 6.10: Estimating $q_{abs}$ graphically

$$u_{abs} \approx \frac{3.51}{16} \quad (6.32)$$

$$q_{abs} \approx \frac{0.16}{5000} \cdot N_{avg} \quad (6.33)$$

The average number of queries in our experiment (Figure 6.9) depends on the number of queries per document insertion $R_{Q/I}$. We know that the total number of documents is 277. The number of merges during the indexing process is 29. With this information we can calculate the average number of queries in-between two merges as follows:

$$N_{avg} = \frac{277 \cdot R_{Q/I}}{29} \quad (6.34)$$

With the estimations 6.32 and 6.33 we can now determine the relative query cost factor $q$:

$$q = \frac{q_{abs}}{q_{abs} + u_{abs}} = \frac{0.000306 \cdot R_{Q/I}}{0.000306 \cdot R_{Q/I} + 0.219} \quad (6.35)$$

For our predictions we solve the equation 6.35 by $R_{Q/I}$:

$$R_{Q/I}(q) = \frac{716.5 \cdot q}{1 - q} \quad (6.36)$$
In subsection 6.3, we have derived functions for calculating $q$ for the point of intersection in the cost functions of two merge strategies. In other words, they calculate the value of $q$ where two merge strategies perform equally well. For a number of merges $n = 29$ we get the following values for $q^3$:

$$q_{NM\not\mid LM2o}(29) = 0.1623$$
$$q_{NM\not\mid IM}(29) = 0.5$$
$$q_{IM\not\mid LM2o}(29) = 0.8877$$

If we plug in these values in the formula 6.36, we get the following estimated values for $R_{Q/I}$:

$$R_{Q/I}(0.1623) = 138.8 \quad (6.37)$$
$$R_{Q/I}(0.5) = 716.5 \quad (6.38)$$
$$R_{Q/I}(0.8877) = 5665.0 \quad (6.39)$$

This means, according to our cost model and our estimation of $q$, we can make the following predictions:

---

*Footnote: The function $q_{NM\mid LM2o}(n)$ is not given in section 6.3. It is derived by setting the functions 6.14 and 6.22 equal and solving the resulting equation by $q$.***
Competing Strategies & $R_{Q/I}$ Prediction & $R_{Q/I}$ Measured \\
| No-Merge | 2-Way Logarithmic Merge | 138.8 | 152 \\
| No-Merge | Immediate Merge | 716.5 | 709 \\
| 2-Way Logarithmic Merge | Immediate Merge | 5665.0 | 5683 \\

Table 6.4: Comparison predicted vs. measure cost intersections

- No-Merge is the best of the three strategies used in the experiment for a value of $R_{Q/I}$ in the interval $[0 \ldots 138.8]$.
- 2-way Logarithmic Merge is the best strategy for a value of $R_{Q/I}$ in the interval $[138.8 \ldots 5665.0]$.
- Immediate Merge is the best strategy for a value of $R_{Q/I}$ in the interval $[5665 \ldots \infty]$.
- No-Merge is better than Immediate Merge for values of $R_{Q/I}$ in the interval $[0 \ldots 716.5]$, for higher values of $R_{Q/I}$ Immediate Merge is the better of the two strategies.

In Figure 6.9, we can see that the predictions are very close to the actual values measured in the experiment. For a comparison of the predictions with the measurements also see table 6.4. Although the results are very promising, we must also consider that we have used the Figures 6.10 and 6.11 that allowed us to make a very accurate estimation of $q$.

6.6 Summary & Remarks

In this chapter, we have shown how to describe the relation of query and update costs of a system or scenario by estimating a relative query cost factor $q$. Knowing that factor allows to chose the merge strategy (among No-Merge, Immediate Merge and k-way Logarithmic Merge strategies) that has the lowest total costs with respect to the presented cost model. In the case where $q$ is in the range of the Immediate Merge strategy it might also be necessary to estimate the number of expected merges $n$.

6.6.1 Improving Cost Model

A disadvantage of the cost model as it is formulated in this chapter is that it does not consider the origin of the data included in a merge: It is not differentiated between data that stems from the delta index and data from static indices. In reality, data from static indices must first be read from disk before it can be merged (and written back to disk) whereas the data from the delta index can be read directly from memory. Fortunately, the cost model can be adapted
relatively easily to take this aspect into account as well. In the following, the steps necessary in order to adapt the model are explained shortly:

- $u_{abs}$ is redefined as the absolute costs for reading or writing data of the size of $S_\Delta$ from/to disk as part of a merge process (assuming that reading and writing data do not differ with respect to update costs).

- It can be observed that the size of the data that is read from disk $S_R$ during a merge is the same as the data $S_W$ that is written to disk minus the size of the delta index $S_R = S_W - S_\Delta$. This applies for all merge strategies. Based on this observation update cost functions can be redefined as follows:

\[
\begin{align*}
c_{U,\text{new}}(n) &= 2c_U(n) - 1 \\
C_{U,\text{new}}(n) &= 2C_U(n) - n
\end{align*}
\]

- On the basis of the updated cost functions, merge strategy cost functions can be recalculated and set equal as shown in section 6.3.
Chapter 7

Future Work

Many aspects of the indexing framework that we have presented in this thesis can be improved. In the following, we propose a few topics that we consider to be among the more important ones:

7.1 Adaptive Merge Strategy.

In chapter 6 we have shown how to describe a given system or scenario by a relative query cost factor $q$. On the basis of this factor and our cost model the merge strategy with the smallest total costs (out of Immediate Merge, No-Merge, and $b$-way Logarithmic Merge) can be chosen.

In a dynamic system it is very likely that $q$ changes over time. This is, for example, the case when the query rate changes. It would therefore make sense to implement and study a merge strategy that is based on the presented cost model and adapts its behaviour dynamically when the scenario changes. This can be achieved by constantly re-estimating $q$ and switching to the according merge strategy if necessary. However, special care must be taken when switching the merge strategy. We have explained earlier that the Logarithmic Merge strategy is based on the concept of assigning generation numbers to static indices. Unfortunately the concept of statically assigning generation numbers as shown in 2.4.1.2 is not flexible enough to allow switching from one Logarithmic Merge strategy to another with a different fan-in. When switching the merge strategy, it must be ensured that the generation numbers are also adapted accordingly. Note, for example, that a generation 1 static index that was created by the 2-way Logarithmic Merge strategy is only half the size of a generation 1 static index created by the 4-way Logarithmic Merge. This problem can be solved by computing the generation number of a static index dynamically based on its size. (We also recommend the use of dynamic generation numbers to solve the problem mentioned in 7.3)
7.2 Garbage Collection & Merging Compressed Data.

In our current implementation of the indexing framework, data in a static index belonging to deleted resource views is post-filtered when that static index is merged. We have argued that this makes sense because the data has to be read anyway at that point. We must, however, also take into consideration that the data of a static index may be written to disk in a compressed form (as it is the case with our on-disk inverted index implementation). Post-filtering data belonging to deleted resource views requires that data to be decompressed and therefore increases the CPU workload. An idea, proposed in [5], is to use a garbage collection approach such that the data belonging to delete resource views is not post-filtered in every merge but only when a certain deletion threshold has been reached. When such an approach is used, static indices need not necessarily be decompressed in every merge. In order to reduce CPU workload, as little data as possible should be uncompressed (and recompressed) in merges where data is not post-filtered. The way in which this is achieved depends on the static index implementation. Figure 7.1 illustrates the process of merging static indices based on on-disk inverted indices containing compressed posting lists. For the sake of simplicity every inverted index contains only one posting list. In the first case a) posting lists need not be uncompressed because they are associated with different terms. In the second example b) both posting lists are relating to the same term. However, posting lists do not overlap and can therefore be merged in compressed form. In the third example c), posting lists overlap and must therefore be uncompressed before they are merged. The resulting posting list must then be recompressed.

7.3 Size Balanced Merges.

The Logarithmic Merge strategy is based on generation numbers that determine when and which indices have to be merged. This strategy is based on the assumption that all static indices of the same generation are of the same size. This assumption is, however, not true anymore when resource views are deleted and the corresponding data is post-filtered from static indices. This leads to a degradation of the merge performance when indices are merged that differ in size significantly. This problem can be solved by implementing a new merge strategy (the indexing framework does not need to be changed) that assigns generation numbers dynamically based on the size of a static index. Estimating the size of a static index is, however, not so easy because some part of the data associated with deleted resource views may be post-filtered during the merge.

Another approach, similar to the concept of dynamic generation numbers, is that of a dynamic balanced tree as proposed in [10]. It is also based on the idea of arranging sub indices in such a way that they do not differ significantly in size when they are merged. (The proposed dynamic balanced tree is also very
Figure 7.1: Merging compressed posting lists
interesting as it provides a generalized merge strategy concept.)

7.4 Improving Performance of Component Implementations.

Another topic is that of further evaluating and improving performance of the component implementations. For the in-memory inverted index we already suggested a few ideas in section 5.2:

- It could be evaluated if insertion and/or deletion performance can be improved when the compressed posting lists are split up in parts (as seen in Figure 5.5).

- The post-filtering approach we use for speeding up deletions could be generalized by attaching timestamps to posting nodes. This would free us from the necessity of having to physically remove posting nodes from posting lists. A garbage collection approach could still be used to remove obsolete posting nodes when the fraction of garbage postings becomes too large. This would allow to amortize the expensive deletion costs in the in-memory inverted index in a similar way as we do it in a more general way for index updates in the dynamic index.
Chapter 8

Conclusions

In this master thesis we have presented a generic and extensible merge-based indexing framework for iMeMex. It provides dynamic indexing support and supersedes the previously used static re-index approach by allowing to apply fine-grained updates to the index structures. It is not anymore necessary to wait until index structures are built-up before they can be queried. Instead, index structures are dynamically updated whenever the underlying data changes and can be queried right from the start.

We have designed the indexing framework to be configurable and extensible. It can be used with any combination of component implementations (static index, delta index and merge strategy implementations). The only condition is that the static index implementation can handle data as provided by the delta index implementation. This is ensured by a generic parameter allowing maximum flexibility for future component implementations.

Furthermore, we have provided several implementations for each type of component. By conducting various experiments we have evaluated strengths and weaknesses of these implementations with respect to index maintenance cost, query cost and real-time capability. We have shown that there is often a trade-off between these criteria for different configurations of components.

A lot of effort has been put into improving the performance of our component implementations. This is especially true for our delta-index implementation based on an in-memory inverted index. Our experimental results show that a dynamic index with a configuration based on this in-memory inverted index, an on-disk inverted file and a balanced merge strategy like Logarithmic Merge has a very good overall performance. One gigabyte of plain text data can be indexed on our system in only a few minutes even when only a few tens of megabytes of main memory are used. Because we have used optimizations like in-memory compression and early aggregation, indexing performance has significantly improved compared to our previously used approach of bulk loading a static on-disk inverted index (based on external-memory sort). Furthermore our indexing framework allows to automatically take advantage of multiprocessor
environments. An arbitrary number of merges can be executed in parallel and queries run in separate threads providing an easy way of parallelization on the framework level.

Finally, we have studied the No-Merge, Immediate Merge and Logarithmic Merge strategies analytically. On that basis we have provided a cost model that determines which of these strategies is the best for a given scenario.
Bibliography


