Master Thesis

Performance tradeoffs in write-optimized databases

Author(s):
Hildenbrand, Stefan

Publication Date:
2008

Permanent Link:
https://doi.org/10.3929/ethz-a-005747645

Rights / License:
In Copyright - Non-Commercial Use Permitted
Performance Tradeoffs in Write-Optimized Databases

Stefan Hildenbrand

Master’s Thesis

Systems Group
Department of Computer Science
ETH Zurich

http://www.systems.ethz.ch/

March 2008 - August 2008

Supervised by:
Dr. Jens Dittrich
Lukas Blunschi
Marcos Antonio Vaz Salles
Abstract

Monitoring applications like environmental monitoring are write-intensive applications. Stream processing techniques are currently used to cope with the high update rates in these scenarios because standard database techniques are not suitable, e.g., a $B^+$-Tree cannot sustain a high update rate and still provide good query performance because the tree has to be reorganized constantly when updates arrive. But current stream processors only support continuous queries over the data. These techniques have difficulties to support ad hoc queries and to represent the current state of the database. Nevertheless, in some write-intensive scenarios the capabilities of a database would be helpful, for example to answer queries on the current state of the monitored environment. Other approaches traded query performance for update performance or vice versa, for example R-Trees improve query performance for spatial data (e.g., for a moving objects scenario) but are more expensive to update than a $B^+$-Tree.

We can improve the query performance of a $B^+$-Tree under high update load if we use a (small) write-optimized delta to collect updates and a (large) read-optimized index to answer the queries. The delta is merged with the index from time to time. This is the main idea of delta-indexing.

Delta-indexing is currently applied in read-mostly environments where indexes are used that are expensive to maintain (e.g., data warehouses). In this setup, delta-indexing turned out to be very helpful. Previous work on delta-indexing queried the delta during query processing in order to provide up-to-date query results and therefore traded query performance for update performance.

This thesis applies the idea of delta-indexing to write-intensive scenarios. We want to provide both good query performance and good update performance. To achieve this, we allow a limited amount of staleness in the query answers. For example in environmental monitoring the data showed as the current state may be a few seconds old if this staleness enables the operator to do ad hoc queries on the data. Since we do not have to provide up-to-date answers, our prototype does not have to query the delta and thus achieves higher performance. In order to keep the staleness low, we try to rebuild the read-optimized index as often as possible.

We show that delta-indexing can achieve an up to four times higher update rate than a standard $B^+$-Tree while sustaining the same or even a slightly higher query rate. Our experiments show that this approach introduces a maximum amount of staleness of about one second for an index that maps a key to a single value and contains 8 million elements.
# Contents

1 Introduction .......................................................... 7  
   1.1 Motivation & Goals ............................................ 7  
   1.2 Contributions & Thesis Organization ....................... 9  

2 Delta-Indexing ....................................................... 11  
   2.1 Analogy & Main Idea ......................................... 11  
   2.2 Related Work .................................................. 15  
      2.2.1 Related Work ............................................. 15  
      2.2.2 Application Areas ....................................... 17  
      2.2.3 Differences Between Related Work and This Thesis .. 17  

3 Organization of the Delta .......................................... 19  
   3.1 Log-based Delta ............................................... 20  
   3.2 Tree-structured Delta ....................................... 20  
   3.3 Hash-based Delta ............................................. 21  
   3.4 Partitioning Delta ........................................... 21  
   3.5 Partitioned Hash Delta ..................................... 21  
   3.6 Implementation Issues ...................................... 22  
      3.6.1 Memory Layout .......................................... 22  
      3.6.2 Sorting .................................................. 25  
      3.6.3 Organization of Data into Chunks ....................... 28  

4 Experiments ................................................................ 33  
   4.1 Workload .......................................................... 33  
   4.2 Setup .............................................................. 35  
   4.3 Measuring & Methodology ..................................... 36  
   4.4 Parameter Selection for the Index Structures .............. 37  
   4.5 Parameter Selection for the Deltas ......................... 40  
   4.6 Maximum Update Rate Vs. Delta Size Experiment ........... 43  
   4.7 Query Rate Vs. Update Rate Experiment ..................... 46  
   4.8 Staleness .......................................................... 50  

5 Conclusions .................................................................. 59  
   5.1 Conclusion ......................................................... 59  
   5.2 Future Work ....................................................... 60  

A Source Code Overview .............................................. 63
1 Introduction

1.1 Motivation & Goals

Information systems have to fulfill many purposes and satisfy many often conflicting requirements. Their main tasks are to store data and to answer queries on the collected data. In the past these two simple tasks seemed conflicting: We can support high update rates by optimizing the storage for writes, e.g. we can simply append the updates to a sequential file (so called log-based approach), but then queries are expensive because we have to scan the whole file to find an element. Or we provide a good query performance by inventing complex index structures, but these indexes are expensive to maintain and thus have poor update performance. There are many other approaches in between these two extreme cases, but they always trade query performance for update performance or vice versa. One solution is that we add another dimension to the picture in order to allow other tradeoffs: For example stream processing techniques can be applied to provide both good update performance and good query performance, but these techniques currently only provide support for continuous queries, e.g. event detection.

In scenarios like environmental monitoring or moving objects we want to support as high update rates as stream processors but still provide the query capabilities of a database for example to support ad hoc queries on the current state of the database. Standard techniques in databases like B⁺-Trees are not suitable, because with the current strategy to maintain these indexes, update-in-place, the B⁺-Tree has to be reorganized constantly if many updates arrive. Thus it cannot answer many queries simultaneously.

To help a database sustain the high update rates of such a monitoring application, we can think of a combination of write-optimized and read-optimized structures that achieve a better overall performance. This is the main idea of what Severance and Lohman [33] call differential files. In this thesis we will call it delta-indexing: A (small) write-optimized delta collects the updates and a (large) read-optimized index answers the queries. From time to time the index and the delta are merged to build a new index. This process is repeated infinitely. The algorithm is explained in more detail in Chapter 2.

Delta-indexing is used in read-optimized databases and turned out to be a good solution. The complex index structures that are build for example in a data warehouse are expensive to maintain. For this reason, data warehouses usually do not work with actual data but are loaded periodically (for example overnight) and then not updated until the next load. In many application scenarios (e.g. a world-wide company that does not have a period of low activity where the data warehouse could be loaded) we still want to update the indexes. Delta-indexes are used to solve this problem.

Our interest in delta-indexes is their untapped potential to improve write-optimized
Introduction

Figure 1.1: Other solutions trade update performance for query performance or vice versa. Our delta-indexing approach provides high performance for both tasks but has some staleness in the data.

In many applications (e.g. monitoring applications or management of history tables and logs) we are interested in a high update performance to collect as much data as possible while still providing reasonable query performance. Some delta-index approaches like the LSM-Tree [26] or the original approach of Severance and Lohman trade query performance for update performance just like the other approaches because in order to get an up-to-date query result, we have to consider both the index and the delta which is expensive. But if we allow a limited staleness in the query answer, delta-indexing can achieve both high query performance and high update performance. Since we do not require up-to-date query results, we only have to query the read-optimized index. Thus we can exploit the full query performance of the read-optimized index and still have the update performance of the write-optimized delta. Figure 1.1 illustrates the tradeoffs of this idea.

In such write-intensive applications, it is known that we cannot sustain the high update rate if we have to use external storage like disks (see for example [35]). But servers nowadays contain many gigabytes of fast random access memory, thus the idea of databases that fit completely into main memory is not impossible any more. A lot of decisions in the past were made to avoid random access to the disk. These decisions have to be rethought in the context of the vast amount of main memory at hand (see for example [25]). Delta-indexing was a good idea when data basically had to be written sequentially, for example to a tape or a disk. This thesis tries to figure out whether delta indexing still works in a main memory oriented environment.
1.2 Contributions & Thesis Organization

The main contributions of this thesis are the following:

**implementations of delta-indexes** We present different implementations of delta-indexes with a focus on the implementation of the write-optimized delta. We evaluate the tradeoffs between update performance, query performance and staleness of the data (i.e. the dimensions shown in Figure 1.1) for these implementations of delta-indexes. Our implementation of delta-indexing uses several deltas to allow concurrent processing on modern hardware.

**cost model and experimental results for staleness** Our implementation of delta-indexes does not query the delta and therefore does not provide up-to-date answers. We present a cost model to estimate the staleness of the data introduced by this approach and compare it to experimental results. Our experiments show that we can achieve a maximum staleness of 1 second for an index size of 8 million elements while providing a much higher update rate than a standard B⁺-Tree.

**experimental results** We further present a series of experiments that show that delta-indexing works well in main memory and with relational data. Our experiments show that delta-indexing can achieve an up to four times higher update performance than a standard B⁺-Tree while still maintaining the same query rate.

The rest of this thesis is organized as follows: Chapter 2 explains the idea of delta-indexing in detail and reviews related work. In Chapter 3 we present the different delta implementations we considered. The experiments are covered in Chapter 4. We conclude in Chapter 5. Appendix A contains a detailed description of the source code used for this thesis.
2 Delta-Indexing

In this chapter we explain the main idea of delta-indexing with a simple analogy and then present an abstract view of the version we implemented for our experiments. The second part of this chapter reviews some related work and gives a quick overview of possible application scenarios. A summary of how this thesis differs from previous work concludes the chapter.

2.1 Analogy & Main Idea

The main idea of delta-indexing is similar to erratas of books, in particular of encyclopedias (i.e. the index). The content of an encyclopedia is subject to change. New terms might become interesting, information needs to be kept up to date. But it is very expensive to print a new edition of such a book. Therefore only a small addendum containing the changes (i.e. the delta) is published from time to time. This reduces the production costs but the reader has to consider both the addendum and the main book to get the actual content. After a while, when enough changes have been collected, a new edition of the encyclopedia with all updates processed is printed. Severance and Lohman [33] presented a similar analogy in 1976 and applied this idea to databases: changes are accumulated in a differential file while the main database is not touched. When the differential file grows sufficiently large, the database is reorganized to include the collected changes. In order to get an up-to-date view of the database, the differential file is consulted as a first step when a query is processed.

In the context of modern computer systems we can do slightly different than in the presented analogy or as Severance and Lohman did:

- modern systems include much more main memory than before, thus we can keep everything in main memory and may observe other tradeoffs than Severance and Lohman did.

- Some application scenarios allow for a limited staleness of the data. In our experiments we therefore query only the index instead of both index and delta. Since we hopefully can do very fast rebuilds, this will only introduce a limited amount of staleness in the data. We can process much more queries and updates since we completely avoid the lookup in the (write-optimized) delta.\(^1\)

\(^1\)Severance and Lohman noticed, that the lookup in the delta is expensive. They suggest different techniques (e.g. Bloom filters) to avoid unnecessary lookups. See Section 2.2 for more details.
We also want to get advantage from the multi-threading capabilities of modern systems. We therefore use several deltas to allow parallel execution of different tasks (i.e. rebuilding the index, answering queries and processing updates).

An abstract view of our implementation of a delta-index is shown in Figure 2.1. Drawn in red on the left side is the actual read-optimized index. Queries are sent to this index. As explained above, the answer might not be up-to-date since not all updates are in the index yet. We explicitly allow this staleness in the data. Many application scenarios are insensitive to a limited amount of staleness like for example stock market analysis which usually does not work on real-time data.

On the right side of the figure in blue is the current write-optimized delta. Updates are stored in this delta. When the delta contains a predefined number of elements\(^2\),

\(^2\)There are other strategies to choose the point of time for the rebuild (e.g. during low system load). We chose this strategy since we are interested in high update rates and assume that the delta will be filled up very fast. For some delta implementations we will use this parameter to tune them. For example an array storing the updates will only reserve as much memory as needed to store the given number of elements.
2.1 Analogy & Main Idea

```java
while (run) {
    while (!delta.isFull()) {
        if (request.isQuery()) {
            index.answerQuery(request);
        } else {
            delta.processUpdate(request);
        }
        request = streamOfRequests.getNext();
    }
    QueueFullDeltas.put(delta); // synchronized
    // blocks until an empty delta is available
    delta = QueueEmptyDeltas.get();
}
```

Listing 2.1: pseudo-code for processing queries and updates

```java
while (run) {
    // blocks until a full delta is available
    delta = QueueFullDeltas.get();
    // do the actual merge
    temp = merge(index, delta);
    // switch to the new index
    // no problems of concurrent access
    // the other thread is only reading the index
    index = temp;
    delta.clean();
    QueueEmptyDeltas.put(delta); // synchronized
}
```

Listing 2.2: pseudo-code for performing the merges

Severance and Lohman would reorganize the database and reuse the same delta. In our approach, we use several deltas to exploit the multi-threading capabilities of modern systems. The full delta is put into the queue for full deltas and an empty one is fetched from the queue for empty deltas. These two queues decouple receiving queries and updates from performing the merges. The thread handling the merge step (depicted as ‘merger’ in the figure) receives full deltas from the queue and merges them with the actual index. We are building a new index based on the old one and the delta. The result replaces the old index while the processed delta can be reused for new updates and is put into the queue for empty deltas. Listings 2.1 and 2.2 show pseudo-code for the explained algorithm.

Although in all our experiments the mappings in delta and index are the same (i.e. we map from a primary key to the data), this is not necessary. The delta can be organized as a mapping from primary key to the data and the index can be organized as a mapping
from any attribute to the data or the primary key (for example as a secondary index on some attribute).

Depending on the implementation of the three basic parts (read-only index, delta, merger), different effects on the performance result. The focus of this thesis lies on the implementation of the deltas. Chapter 3 explains the covered variants in detail. As a read-optimized index any index structure that supports efficient bulkloading can be used. If the structure supports fast sequential access of the data, it is even more suitable for our needs. Since the focus of this thesis is on relational data we selected three general index structures that do not make any assumptions on the data except for a partial order on the keys. We consider the following implementations:

**Sorted Array** The baseline for an index is a sorted array. Queries are performed using binary search. All other index structures usually start with the sorted data and build an extra structure on top of it. The time it takes to set up this structure has to be justified by a speed-up in queries. This index uses no overhead at construction time (except sorting, but that is needed for all other indexes as well) and provides a lookup time of $O(\log n)$.

**B$^+$-Tree** The second structure we take into consideration is a full blown B$^+$-Tree as suggested by Knuth [14] (the name B$^+$-Tree was introduced in [3]). This type of index is widely used in databases. It takes $O(\log n)$ to bulkload a B$^+$-Tree from a sorted array. It provides the same asymptotic lookup time as a sorted array but has better cache performance and less computational overhead than binary search on a sorted array.

**CSS-Tree** Since we focus our experiments on main memory processing of the queries, we selected the CSS-Tree by Rao and Ross [30] as our third index structure. The idea is to set up a directory similar to a B$^+$-Tree structure on top of the sorted array. The structure does not support updates and has to be bulkloaded to incorporate changes. Since we are rebuilding the index in delta-indexing anyway, this structure seems very interesting for our experiments. A CSS-Tree has the same asymptotic performance as the B$^+$-Tree but due to better cache performance and memory layout it can outperform the B$^+$-Tree both in bulkloading performance and query performance.

The three index structures differ in their balance between build time and access performance. The sorted array only sorts at build time and is expected to perform very well if much more updates than queries arrive. The CSS-Tree does some additional processing at build time and is expected to perform well in mixed workloads. According to [30] it is expected to perform better than a B$^+$-Tree when many more queries than updates arrive.
2.2 Related Work

In this section, we review related work and present possible application scenarios. A summary of the differences between the presented work and this thesis follows at the end.

2.2.1 Related Work

Delta-indexing traces back to the early seventies (for example [31, 32, 36]). Severance and Lohman [33] present ten arguments why differential files particularly for the maintenance of very large databases should be used more often. In order to provide up-to-date query results, they query the delta as well. The application of a Bloom filter is suggested to reduce the number of unnecessary lookups in the deltas.

Some papers improved different aspects of the differential file approach shortly after it was published and attacked some of the drawbacks of the differential file approach as pointed out by Lorie [24]. For example Kollias et al. [15] suggest to keep only the inserts in a separate structure. With this variation a query needs to search the delta only if the entry was not found in the read-optimized index. Since a lookup in the read-optimized index is supposed to be fast and there are less unnecessary lookups in the delta this improves the query performance. For the same reason Gremillion [9] worked on an improved version of the Bloom filter used in [33].

Our approach is not to trade in query performance by having to consider the delta during a lookup but allowing some staleness in the query answer. We therefore can exploit the full performance of the read-optimized index and do not have to worry about query performance of the write-optimized delta.

The question of when to reorganize the index received some attention. A lot of research has been done on the optimal policy and the intervals of batch processing in databases in general and index rebuild in particular. For example [23] develops a general model for these “how often to perform x” problems like when to do a backup and shows that reorganizations of files are a special case of that model. The same considerations can be applied to the question of when to merge a delta with the read-optimized index.

In the late Nineties the idea of delta-indexing was applied to B-Trees and their variants. Lang, Driscoll and Jou [17] point out that the drawback of time consuming merges as presented by Lorie [24] is the most serious one and suggest an algorithm to efficiently perform batch insertions into tree structured files. Essentially their suggestion is not delta-indexing anymore because they are applying the updates as a batch to an existing tree instead of building a new tree. Basically they push the updates from the root of the tree to the corresponding leaves. The performance gain results from the fact that each node is only read and written once for all updates that go to that leaf instead of following the whole root-leaf-path for each update.

About at the same time, Srivastava and Ramamoorthy [34] presented similar ideas to merge an existing B⁺-Tree with a batch of updates. They call this process grouped
updates. They propose two algorithms to efficiently merge an update log with an existing B$^+$-Tree and analyze the time savings. One algorithm is very similar to the approach used in [17] and pushes the updates from the root to the leaves. The second algorithm scans the tree at the leaf-level by exploiting pointers to the right sibling. They restructure the tree after each insert if needed.

We stick to the original proposal of Severance and Lohman [33] and build a completely new index from the old index and the delta.

Currently three tendencies in the research on delta-indexing and batched updates can be identified:

1. modify the merge process to allow concurrent queries
2. organize the deltas in a hierarchical way to further improve update performance
3. buffer at node-level instead of tree-level

As representatives for 1) we quickly discuss the research done by Pollari-Malmi, Soisalon-Soininen et al. (e.g. [29, 28]). The basic idea here is to make the merge process concurrent with other operations (i.e. search). Concurrency control mechanisms are improved to lock only small parts of the index when the updates are applied in a batch to the existing index. Rebalancing of the tree is either deferred or done in a way that a concurrent search can proceed after the rebalancing was applied in order to produce consistent and up-to-date results. Another idea investigated in this direction is the idea of relaxed data structures that do not have as strict constraints on the invariants of the data structure (see for example [18]). We do not need any expensive concurrency control mechanisms in our approach since we do not modify the index but build a new one while the old index is still ready to answer queries.

Two representative examples for 2) are the LSM-Tree by O’Neil et al. [26] and the idea of stepped-merge and stepped-hash by Jagadish et al. [13]. The core idea here is to maintain several indexes at different levels. Indexes of the same level are then merged together to an index of the next level ([26] mentions merge sort as an analogy). Their main concern is the I/O performance since they store large parts of the deltas and indexes on external memory like disks. This thesis does not consider external memory and assumes that the entire data fits into main memory.

The buffer tree by Arge [1] is a representative example for 3). The main idea is to associate buffers in main memory for the nodes of a large (external) data structure. Instead of buffering all updates to the structure in one delta, the changes are collected on a per-node base.

The goal of all these approaches basically is to provide higher update rates while not loosing too much of the query performance. Graefe [7] summarizes some of the ideas presented so far that are applicable to B-Trees and their variants.
2.2 Related Work

2.2.2 Application Areas

The idea of delta-indexing has a range of application scenarios. The focus of this thesis is on relational data. A key maps to a tuple of data (or a pointer to such a tuple). Delta-indexing works well in read-mostly environments (see for example [12]). The question is whether such indexes can sustain a high update rate.

Delta-indexes are state-of-the-art in enterprise search. For example TREX from SAP uses this technique to allow fast updates to the indexes [20].

In text retrieval systems indexes play an important role and the maintenance of these large indexes received some interest. See for example [21] for a survey of some techniques used in that area. Although they claim that “update techniques from other areas cannot be readily adapted to text retrieval systems” [21], the basic ideas are very similar. Büttcher and Clarke [2] use a hierarchical organization of what they call sub-indexes to improve update performance of an index. This resembles the ideas from the LSM-Tree [26] and consecutive work.

Another field where high update rates for indexes are investigated is the tracking of moving objects. Most of this work is on data structures that support dimensional data (like R-Trees for example) and cannot be applied to relational data in general. See for example [27] for a somewhat outdated survey.

In general, delta-indexing may be interesting in application scenarios where high update rates are required while still maintaining some index structure is desirable. Wherever maintaining an index is expensive but bulkload strategies for that index are available and either a limited amount of staleness can be tolerated or queries may become a bit slower, delta-indexing can be used to improve performance.

2.2.3 Differences Between Related Work and This Thesis

In summary, this thesis differs in the following points from the related work presented so far:

**Limited staleness instead of up-to-date answers** Concerning the query strategy we allow some staleness in the data. This strategy is similar to what is used in read-optimized application scenarios like search engines and data warehouses. We only query the read-optimized index, which will produce an answer that reflects the state at the point in time when the last merge started. Other work always produced up-to-date results by querying the delta as well (and therefore trading in some query performance). In our approach we do not have to query the delta and therefore do not need to trade in query performance but pay with a limited amount of staleness.

**Main memory instead of external memory** Other solutions until now kept at most the differential part of the index in main memory. All other parts (i.e. the read-optimized index) were stored on external memory. The main argument for delta-indexing was the reduced I/O-cost. For all experiments in this thesis all data is kept in main memory.
Re-build instead of re-merge Another main difference between our work and the work presented is that usually the batch of updates is applied to an existing index (i.e. the index is modified during that process). Our approach is to build a completely new index from the delta and the read-optimized index as it was originally suggested by Severance and Lohman [33]. In that way the read-optimized index is never changed and always available to answer queries. This is a workaround for the disadvantage of time consuming merges of the index pointed out in [24]. Modifying an existing index either needs some sort of locking or a sophisticated strategy to make sure a query that ended up in a part of the structure that is modified concurrently is answered correctly. Both approaches are not suitable for high update rates. Locking is a huge overhead in any case and the escape strategies produce unnecessarily complex data structures.

Focus on delta implementation Lorie [24] points out that updates might affect an element already modified and this has to be taken into account. One can either modify the merge process to make it aware of such cases or use a delta that only keeps the latest update. In general our work focuses much more on the implementation of the delta than any of the presented work so far.
3 Organization of the Delta

A delta-index consists of three basic parts: a read-optimized index, a write-optimized delta and a merge strategy. In this chapter, we discuss how the delta can be implemented. The selection of read-optimized indexes and the merge strategy were presented in Section 2.1.

A write-optimized delta basically has to fulfill three tasks:

- collect updates
- prepare the collected updates for merge (i.e. bring them to an intermediary representation in sorted manner)
- handle updates for the same key (sometimes called duplicates): The last update that was collected has to be reflected in the read-optimized index. Either the delta keeps only one update per key (i.e. early aggregation) or it ensures that the order of the updates does not change.

In our experiments we consider the following data structures as deltas (Table 3.1 contains a qualitative comparison):

**log-based delta** We use a simple log where new entries are appended at the end. This solution provides very fast collection but is expensive to sort and the handling of duplicates is difficult.

**tree-structured delta** We use a B⁺-Tree to store the updates. With this solution, sorting and handling of duplicates is done by the tree but the insertion of new elements is expensive because the order of the elements is always ensured.

**hash-based delta** We use a hash table as a delta that distributes the elements according to their hash code to different buckets. This solution provides efficient handling of duplicates and fast collection but it is expensive to sort.

**partitioning delta** We use a table which distributes the entries into partitions. This provides fast collection and helps with sorting (since only the element within one partition have to be sorted) but duplicates are somewhat difficult to handle.

**partitioned hash delta** We combine the hash table and the partition table to a delta that partitions on the first level to improve sort performance and uses hashing within the partitions to improve performance of duplicate handling.
Table 3.1: Qualitative comparison of different delta implementations.

<table>
<thead>
<tr>
<th></th>
<th>collect updates</th>
<th>prepare merge</th>
<th>handle duplicates</th>
</tr>
</thead>
<tbody>
<tr>
<td>log-based delta</td>
<td>⊕⊕</td>
<td>⊗</td>
<td>⊗⊕</td>
</tr>
<tr>
<td>tree-structured delta</td>
<td>⊗⊕</td>
<td>⊕⊕</td>
<td>⊕⊕</td>
</tr>
<tr>
<td>hash-based delta</td>
<td>⊗</td>
<td>⊗⊕</td>
<td>⊗⊕</td>
</tr>
<tr>
<td>partitioning delta</td>
<td>⊗</td>
<td>⊗</td>
<td>-</td>
</tr>
<tr>
<td>partitioned hash delta</td>
<td>⊗</td>
<td>⊗</td>
<td>⊗</td>
</tr>
</tbody>
</table>

We first present the algorithm and then cover some implementation issues like memory layout for arrays and sort algorithms that will be used for the different deltas (especially for the log-based delta). We further present the idea of chunk-based tables, which is a memory allocation strategy we will use for the hash-based, partition-based and partition-hash-based deltas (we call these three deltas chunk-based deltas).

### 3.1 Log-based Delta

We can use an array to keep the updates and simply append new tuples to the end of the array and rebuild the index if the reserved space is filled. For inserting, this approach has the least computational overhead since we do not aggregate updates on the same key or do any preprocessing of the data. For extracting the data (i.e. bring it to a sorted intermediary representation) we need a lot of work and we have to make sure that the last update wins. Since it is quite expensive to do early aggregation (i.e. detect duplicates on every insert), it is necessary that, if the merging sorts the elements in the array, this is done in a stable way, which means that elements with the same key have to remain in the same relative order.

### 3.2 Tree-structured Delta

Instead of a simple log-based delta, we can use a tree for the delta. A tree takes care of a lot of problems we had to handle ourselves with the log-based approach. A tree removes duplicates (i.e. the last update wins) and sorts the input. If we link the leaves together in order to enable sequential access, merging is straightforward. We use the same implementation of the B⁺-Tree as we use for the read-optimized index but tune the parameters more towards a write-optimized structure.

If the mapping of the index is the same as in the delta, you might ask why we use a read-optimized tree as index anyway. We could send the queries directly to the delta. For the B⁺-Tree we know that inserts get slower the larger the tree grows (since we need more levels in the tree). It may pay off to insert only up to a certain number of elements into a small delta tree and then bulkload a large index tree from the delta and the old index tree. For other delta implementations it may be useful to build a read-optimized index because the read-optimized index supports other types of queries. For example if
we use a hash table as delta it makes sense to build a tree as read-only index because the tree supports range queries which are very expensive on a hash table.

3.3 Hash-based Delta

Another way to organize the delta is to hash. Inserting is very fast but since hashing does not introduce an order in the data that can be exploited for sorting, we will have to sort the whole delta in the end. We use a simple chaining variant of hashing and organize the data in so called buckets (sometimes called slots). Therefore we have to handle smaller chunks of memory and can eliminate duplicates more efficient than in a log-based delta.

3.4 Partitioning Delta

Since sorting consumes most of the time during the merge-step of the log-based approach, we investigated the idea of partitioning the incoming data while inserting it into the delta. We are basically doing the first step of a QuickSort (the ‘divide’) in advance. Another analogy would be a BucketSort. Obviously this solution may suffer with skewed data.

While the partitioning may help with sorting, we have to take care of duplicates. Either we use a stable sort, use timestamps to enforce stability or we remove duplicates while inserting. In our experiments we considered one implementation that removes duplicates by using a scan over all elements in the same partition and another implementation that keeps duplicates but employs a stable sort.

3.5 Partitioned Hash Delta

The partitioning delta described above performs very well with uniform data. But as soon as there is skew in the data the performance gets very bad. If we do not remove duplicates the performance is not affected by the skew anymore but that is basically just postponing the work that has to be done anyway.

The problem with duplicate elimination lies within the implementation. To remove duplicates we use a simple scan through all elements in a specific partition. With skewed data, there are cases where many elements fall into the same partition and the lists to scan get very long. To overcome this limitation, the idea is to hash within a partition in order to reduce the number of elements that have to be scanned. Figure 3.1 illustrates this idea.

The solution should be as fast as the partitioning solution with uniform data and as fast as the hashing solution with skewed data. This means that we need an adaptive solution that changes its behavior depending on the number of elements that fall into one partition or bucket. If too many elements fall into one partition, we allocate several hash buckets to handle the overflowing elements (step 2 in the figure). If one of the hash buckets overflows, we again allocate a set of hash buckets to handle that (step 3 in the figure). We do not use more than two levels of hash buckets because each level
Figure 3.1: Implementation of “Partitioned Hash Delta”: In step 1 the data is partitioned (i.e. letters a-f, g-l, m-r and s-z). In steps 2 and 3 the data is hashed and no order is introduced on the letters (for example letter q is not in the same bucket as p and r). We have 5 elements in one partition and only 1 or 2 in the other partitions, which is easily possible with skewed data, but the hash function did distribute the elements pretty bad into the different buckets, which is highly unlikely and shown only for the sake of argument. In step 3 one bucket remains empty while the other bucket overflows and we have to escape to a list of buckets.

of indirection needs a lot of memory and our experiments showed that an overflow after two levels of hashing is very rare. If still an overflow happens, we escape to a list of buckets (step 4 in the figure).

3.6 Implementation Issues

This section covers some implementation issues that arose during the development of our prototypes. We describe how we allocate memory and do some experiments on sorting.

3.6.1 Memory Layout

All prototypes used in this thesis are written in Java. Developing in Java is easy and it allows for rapid prototyping. The compiler takes care of the memory management and a well established runtime environment is available. But all these nice facilities and the simplicity of developing in Java do not deter us from thinking about some issues
of memory layout. In all scenarios we are working with key-value-pairs or even key-value-vectors (for example if we want to store an entire row of values together with its key). So how do we store these tuples? At first sight and as Java is an object-oriented language, one might be tempted to store the tuples as objects. But since these tuples are only used for data storage and retrieval we can cut out the object overhead and go with multi-dimensional arrays. Each row of the array corresponds to one tuple, the first column contains the key, the other columns contain the values.

Unfortunately Java has its own idea of multi-dimensional arrays and implements these as array of arrays. Instead of a continuous block of memory divided into rows and columns, we have an array of pointers that point to other arrays (see for example Figure 3.2a). Traversing such an array row-wise always requires to follow one pointer and load another piece of memory to the caches. This implementation allows for jagged arrays, which means the rows do not need to have the same length. For our experiments we cannot exploit this advantage because we always know how the rows look like and all the rows have the same length since we have relational data in mind that usually has a fixed schema. We have to pay extra time to follow the pointers that are used to implement multi-dimensional arrays.

In the case of relational data, we have to store key-value mappings, but the value may be a single integer (in case we only store a row-ID or one column in the index) or a whole row of a table. In our experiments we simplify the second case to a vector of integers of fixed length (all data can be encoded in this manner). Figure 3.2 shows what the memory layout looks like when we store key-value-pairs and entire rows with a key in the two array layouts.

The following simple experiment should illustrate the differences between the two array layouts. Assume a table that contains a row id and ten integer values. We want to sum up one column of this table. We measure the time it takes to do this in a multi-dimensional array and compare that to the time it takes to do this in a linearized array. Linearized means that we allocate a one-dimensional array and store the tuples one after another in this array. Since we assumed a table with a fixed schema, we know that each tuple has a fixed length (in our example 11) and we can do offset computations to access the different “rows” of the array. This is very similar to what the C compiler does when using multi-dimensional arrays. The results are shown in Table 3.2. We see that it takes about two times longer to process the same amount of data if the data is stored in a multi-dimensional array instead of a linear array.

This experiment may be very artificial and not applicable to real workloads and access

<table>
<thead>
<tr>
<th>number of tuples</th>
<th>linear [ms]</th>
<th>2'500'000</th>
<th>5'000'000</th>
<th>10'000'000</th>
<th>20'000'000</th>
</tr>
</thead>
<tbody>
<tr>
<td>multi-dimensional [ms]</td>
<td>78.9</td>
<td>166.8</td>
<td>320.1</td>
<td>629.7</td>
<td></td>
</tr>
<tr>
<td>factor ( \frac{\text{multi-dimensional}}{\text{linear}} )</td>
<td>2.0</td>
<td>2.1</td>
<td>2.0</td>
<td>2.0</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Time to sum up one column of a table.
Figure 3.2: Different memory layouts for arrays. Highlighted is the second tuple in each layout.
patterns. For this reason we investigate the task of sorting the tuples by key in the next
subsection and reconsider the memory layout again.

3.6.2 Sorting

Many algorithms to merge two data structures work on sorted input. It is therefore
crucial for the performance of our prototypes to use a fast sort algorithm.

We started with the implementations that are offered by the Java library. There are
two variants: MergeSort for object arrays and QuickSort for native typed arrays (e.g.
`int[]`). Both variants are optimized and use several tricks to gain performance (like
InsertionSort on small arrays, clever choice of pivot element in QuickSort, . . . ).

Since we will need a stable sort algorithm\(^1\) we have to adapt QuickSort. We do this by
adding a timestamp to each element in the data structure and compare the timestamps
if two keys are otherwise identical.

We reconsider the question of memory layout and experiment with both MergeSort
and QuickSort for both multi-dimensional arrays (for Java this is an array of objects
and the default sort algorithm in this case is MergeSort) linearized arrays as presented
in the last subsection. We tweak the library version of MergeSort by hardcoding the
comparison (instead of using a comparator object). For the linearized version we inves-
tigate one implementation that merges the key and the timestamp to one
`long` value that can be compared with one processor instruction instead of fetching and comparing
the timestamps if the keys match.

Since the access time for the implementations is different (for example the QuickSort
methods have to add a timestamp, which is not needed in MergeSort, or the implement-
ation with `longs` has to merge key and timestamp and to extract the key afterwards)
the experiment measures the time it takes to insert the data into the data structure, sort
the structure and bulkload a B\(^+\)-Tree from it. Figure 3.2 shows how the memory layout
looks like when we do not use a timestamp, Figure 3.3 shows how we extend the struc-
ture with the timestamp. These two figures also introduce the terms used in Figures 3.4
and 3.5. Figure 3.4 shows the breakdown of the time for the different tasks: insert, sort
and access for a trace containing only globally unique keys. Figure 3.5 shows the results
for a trace that contains 50% duplicates (which means that QuickSort needs to consider
the timestamp and some elements will get skipped in the bulkloading process).

We see that linear arrays are much faster than multi-dimensional arrays for smaller
tuple sizes, which is what we expected. There is a point where the overhead of copying
the complete payload instead of just changing a pointer gets larger than the overhead
of following a pointer to access the sort key. On our test machine the linear arrays
are faster than multi-dimensional arrays for up to 32 integers payload. This is quite
interesting since this means that copying that much data (32 integers that are 4 bytes
each makes 128 bytes) is still faster than following a pointer for each comparison. In
further experiments we use the linear approach since for large payloads we can use the

\(^1\)If a tuple is updated more than once, we want the last update to win, thus the order of updates on
the same key must not change or duplicates have to be eliminated before sorting.
Figure 3.3: Different memory layouts for arrays when we store a timestamp in order to stabilize the sorting. Highlighted is the second tuple in each layout.
3.6 Implementation Issues

Time to access data and bulkload a Tree
Time to sort
Time to insert

payload: 1 int
payload: 10 int
payload: 40 int

Figure 3.4: Experiments with different array organizations and sort algorithms. The trace contains 5 million unique elements.

Figure 3.5: Experiments with different array organizations and sort algorithms. The trace contains 5 million elements with 50% duplicates.
The question whether QuickSort or MergeSort should be used is not that easy to answer from the results. The overhead of adding a timestamp when inserting the tuples into the array is negligible. When it comes to sorting, it is as usual: QuickSort wins. But the advantage of faster sorting is paid by a slower access time. Without a timestamp in the structure, we can directly copy the data into the leaves of the tree and do not have to use special code to get rid of the timestamp. Overall the differences between the two methods are very small. Since the code gets simpler if we do not have to carry a timestamp, we will use MergeSort where a stable sort is needed and QuickSort otherwise.

The performance gain of using longs is noticeable but not that large. We will not use it in further experiments since it uses more memory, it does not scale well (i.e. it is already slower with a payload of 10 integers) and the handling is more difficult.

3.6.3 Organization of Data into Chunks

We still need an efficient way to implement the partitions and buckets of the partition-based, hash-based and partition-hash-based deltas. We can use chunks of memory. How the data is distributed over these chunks depends on what kind of delta we implement and is explained later. Let us first take a look at how these chunks are organized: We start with \( n \) such chunks that can contain \( k \) elements each. Usually we choose \( k \) as \( 2^c/n \) where \( c \) is the capacity of the delta because we want the chunks to be filled about 50% in the average case. This is similar to the split and merge conditions of a B^+-Tree: if a node gets full, it is split into two half full nodes. If a node contains less than that number of elements, it is merged with another node.

To reduce the overhead of objects and with the results of linear vs. multi-dimensional arrays in mind, we aligned these chunks continuously in memory. Since we know how large one chunk is, we can address the chunks with offsets in a large array. To link chunk \( x \) to chunk \( y \), we simply store the offset of chunk \( x \) in a dedicated field of chunk \( y \). The instantiation of a chunked table is shown in Listing 3.1.

If the distribution function distributes the tuples evenly over the chunks, everything is fine. But there may always be workloads that will lead to an overflow of one or more chunks. We have to link other chunks to the overflowing one. Depending on how the data is distributed over the chunks, different linking strategies can be used: either we just append another chunk to the overflowing one (like in a linked list) or we allocate multiple chunks and distribute the data over these new chunks.

hash-based delta We implemented a minimal variant of a hash table using the chunked memory layout to avoid the object overhead of the library variant from Java. The first \( m \) chunks of the chunked table (that has \( n \) chunks, \( n > m \)) represent the

---

2 We can store the data in a large linear array and extract the key and an offset in that array to a separate structure. That structure has then a tuple size of 2 (key and offset) and can be sorted using the investigated methods. This is similar to multi-dimensional arrays except that we have the keys right at hand and can sort as fast as with normal linear arrays but still have less copying overhead. Accessing the data is a bit slower since we have to follow the offsets into the array.
3.6 Implementation Issues

// number of updates in this delta
int c = 5000;

// length of one tuple (i.e. key, timestamp, values)
int tupleSize = 2;

// number of buckets
int n = 100;

// number of tuples per bucket
int k = 2 * c / n;

// worst case: all tuples fall into the same bucket
// i.e. c tuples need (c / k) chunks
// and for each of the n buckets we reserve one chunk
int numChunks = (c / k) + n;

// each chunk needs two extra ints for book keeping
int[] delta = new int[numChunks * ((k * tupleSize) + 2)];

Listing 3.1: Implementation of a chunked table.

$m$ buckets of the hash table. The bucket where a tuple falls into is selected based
on the hash value\(^3\) of the key. If the chunk representing a bucket overflows we just
take another chunk that is not yet allocated and link it to the previous one.

Figure 3.6 shows an example: the table has two buckets (i.e. $m = 2$). To illustrate
the overflow we chose that one bucket may contain $k = 3$ elements. The second
chunk is already full and we linked another chunk to that bucket. In the example
chunk #1 points to chunk #2. Since we have only one extra chunk ready and we
have to be prepared for the worst case (i.e. all elements fall into the same bucket),
we may only insert six elements.

We assume that the hash function is good enough such that overflow only hap-
pens in rare cases and therefore did not implement a more sophisticated overflow
strategy. For some ideas how the hash table could be improved see the end of this
section.

**partitioning delta** We use the chunk-based memory allocation in a similar way as for
the hash-based delta. The first $m$ chunks of the chunked table (that has $n$ chunks,
$n > m$) correspond to $m$ partitions. If one of the chunks representing a partition
overflows, we link another chunk to this partition.

**partitioned hash delta** For the first step of the distribution (i.e. partitioning, see Fig-
ure 3.1) we allocate $m$ chunks for $m$ partitions. As long as the elements fit into one
chunk, we store them as usual. If we need to allocate another chunk of memory
due to overflow we do not just append a new chunk but use the first chunk to hold
pointers to $k$ (the number of elements that fit into one chunk) other chunks. We

\(^3\)We use the hash function from the Java library: $h = x \oplus (x \gg 20) \oplus (x \gg 12) \oplus (x \gg 7) \oplus (x \gg 4)$, where
$\oplus$ denotes the bitwise exclusive or operator and $\gg$ denotes the logical shift right operator.
move the tuples that were in the first chunk to a new chunk according to their hash value. New tuples now are hashed to one of these \( k \) chunks that are pointed to by the first chunk. If there are even more elements that fall into the same chunk, we add one other indirection of hashing. The algorithm then continues to link new chunks together as in the normal partitioning delta approach. With this layout we likely can access \( k \) elements directly in the less frequently used buckets while cutting the number of elements to scan by \( k \) or even \( k^2 \) in the frequently overflowing buckets.

The only problem that chunked tables have is the high memory requirement. Since we want to store all the data in one large linear array and only work with offsets in that array, we need to reserve a piece of memory in advance that is large enough to keep all incoming data even in the worst case. Depending on how the chunks are allocated, that may require a different number of chunks. For example in the partitioned hash delta, the worst case happens when we have to allocate both levels of hashing in many partitions. Assume we use chunks that can contain \( k = 4 \) elements. If we insert 5 elements that fall all in the same partition and map to the same hash bucket in both levels of hashing, we need 10 chunks to store these 5 elements: one chunk containing pointers to 4 chunks in the level of hashing from which 3 chunks are empty and the forth chunk contains pointers to another 4 chunks from which again 3 are empty and the forth contains 4 elements. Since we have 5 elements, we need another chunk to store the fifth element, i.e. we need \( 2k + 2 \) chunks to store \( k + 1 \) elements. Adding more elements in this partition does not make the situation worse since we need at least \( k \) elements to produce another overflow. But we can do the same in any other partition which then leads us to the worst case of two chunks to store one element.

For our case of delta-indexing, one workaround for this problem is to signal that the delta is full if the reserved amount of memory is exceeded. The delta is then scheduled...
for rebuilding and we start collecting updates in a fresh delta. Therefore we can calculate the memory requirements for the average case and do not have to worry for worst case scenarios. However, in a very unbalanced allocation of the chunks we might have to rebuild the deltas more often than expected when we reserve not enough memory.

Our implementation of a hash table is not very sophisticated. We did not improve it, since the experiments show that the current implementation is already quite fast with most workloads. Ideas on how to improve it can be found in the following papers: A survey of dynamic hashing can be found in [5]. The technique used in our implementation uses some ideas from Extendible Hashing [6]. The ideas presented in [4] sound very promising. They claim to efficiently handle non-uniform data which would allow the use of an order-preserving hash function. Such a function could again simplify sorting. Other approaches based on Linear Hashing [22] that support sequential access were proposed by Hachem and Berra ([10, 11]). Larson extended the idea of Linear Hashing with ideas from dynamic hashing in [19].
4 Experiments

This chapter presents the main experiments. We give details about the experimental setup, i.e. the systems we use for our experiments as well as what and how we measure. The workloads used for our experiments are explained in detail. We further present the following experiments:

**parameter selection for the index structures** We evaluate different parameter settings for our index structures (sorted array, B⁺-Tree and CSS-Tree) to get the best performance for our workload. An index has to be fast on answering queries and cheap to bulkload.

**parameter selection for the deltas** We evaluate different parameter settings for our delta implementations (log based, tree-structured, hash-based, partition-based and hash-partition-based) to get the best performance for our workload. A delta has to support high insert rates and be cheap to bring into a sorted intermediary representation.

**maximum update rate vs. delta size** Using the parameters selected in the previous two experiments, we evaluate the influence of the delta size on the update performance.

**query rate vs. update rate** We compare our implementations of delta-indexes with other indexes and evaluate how the different indexes perform under different mixes of queries and updates.

**staleness** We measure how long it takes for one update to reach the read-optimized index (i.e. until it is considered in the query result). This mainly depends on the number of updates that are applied in one merge.

Based on these experiments, we provide a cost-model for estimating the staleness of the data introduced by our delta-indexing approach.

4.1 Workload

There are four types of operations that can be done on an index: We can query for an element, we can update or delete an existing entry or we can insert a new entry. For our experiments we focused on queries and updates.

There are different ways to generate data for experiments. In the case of indexes we considered three distributions: a) uniform, b) skewed and c) sequential (i.e. as skewed as possible if unique keys are required). Figure 4.1a shows examples for these distributions. The uniform distribution contains roughly the same number of elements...
in each partition of the key domain. The skewed distribution contains a huge number of elements in the first part of the key domain. The first partition contains even more tuples than unique numbers since the lowest values appear several times in this distribution. The sequential distribution contains as many tuples as there are unique numbers in the first few partitions. It contains only numbers from the lower end of the key domain. This explains the sharp edge in the figure.

For some traces we would like to have data that follows a certain distribution but does not contain duplicates (for example to populate the index with a given number of elements before the experiment starts). In the uniform case we just use a random number generator that generates uniform distributed data. Duplicates are just skipped. In the sequential case we use a simple counter. If we want to exclude a set of values (i.e. if we want to create a trace containing only inserts, we have to exclude the values that are already preloaded in the index), we simply skip that value of the counter.

In the skewed case, things get a little more difficult. Since we cannot simply drop numbers without changing the distribution, we have to come up with another solution. If duplicates are allowed (as in an update-only workload), we simply use a 80-20-distribution generator as described in [8]. If no duplicates are allowed we would still like to get close to a 80-20-distribution. Since such a distribution will generate many duplicates (a small set of numbers will occur very often), this is not that easy. Just skipping duplicates

---

We did not consider a Zipf-distribution because the initialization of a random number generator for a Zipf-distribution is linear in the size of the value range and takes too long since we want to generate keys in the whole range of positive 32-bit integers.
and asking the generator for new numbers until enough numbers are generated is one option, but then the data is not as skewed as it should be. To approximate the aimed distribution we use sequential numbers at the beginning of the trace and switch to a less dense load when we do not need all values in a partition anymore. Since a skewed distribution contains one value very often and this value is usually the smallest one in the domain, we have to start with that value.

Assume we want to generate \( n \) numbers between 0 and \( 2^{32} \). Then the original trace contains many (say \( k \)) zeros, thus our approximation should contain \( k \) numbers as close to zero as possible. Therefore we put the numbers from 0 to \( k \) in the approximate trace. But since the original distribution will contain some of these numbers already (with a 80-20-distribution there will be lots of ones for example), we have to enlarge the sequential part of the approximation. The sequential part has to be long enough that it can contain as many unique numbers as the original trace contained numbers (counting all numbers) in the same range, i.e. \( l = \text{count}_{s}(\text{origTrace}[s] < l) \).

We can calculate \( l \) by iteration starting with \( l = 1 \), counting all zeros in the original trace, set \( l \) to that number, calculate the number of elements below \( l \), and continue until \( l \) does not grow anymore. Figure 4.1b shows the distributions of the three generation methods (unprocessed 80-20, approximation and naive duplicate-drop). The approximation contains as many elements as possible in the first few buckets to compensate for the skew (due to duplicates the original trace can contain more elements in the first few buckets than there are unique numbers) and then quickly returns to the expected distribution. We see that the naive approach has less skew than expected and the constructed trace compensates the skew as early as possible and then follows the aimed distribution.

### 4.2 Setup

All experiments were performed on servers having each two 2.4 GHz Dual Core AMD Opteron 280 processors, i.e. four cores in total, and 6 GB of main memory. If not stated otherwise, the OS running on the server is a single-core variant of RedHat Enterprise Linux with kernel 2.6.9. With this setup the results of the experiments are more comparable since no multi-core issues have to be considered. Some of the delta-indexing implementations are multi-threaded and would have an advantage over the standard B+-Tree implementation when running on a system with more than one core. The focus of this thesis is solely on main memory performance of the investigated techniques thus no swapping or paging of any data to secondary storage was allowed (i.e. swap was turned off) during all our experiments.

All prototypes are written in Java and compiled with version 1.6.0_02 of Sun’s compiler. The runtime environment was the server VM from Sun in the same version. Below is the complete version information.

```
$ java -version
java version "1.6.0_02"
Java(TM) SE Runtime Environment (build 1.6.0_02-b05)
Java HotSpot(TM) 64-Bit Server VM (build 1.6.0_02-b05, mixed mode)
```
The compiler did not produce any debug information for the experimental runs (compiler-flag -g:none). The VM was created with an initial heap size of 5 GB and the maximum heap size was set to the same amount to reduce memory management overhead.

4.3 Measuring & Methodology

In the investigated scenarios, basically two things can be measured. Either we measure the time a given data structure needs to process a given workload trace (i.e. measure time to process) or we measure the number of operations a given data structure is able to perform within a fixed amount of time (i.e. measure throughput).

The first setup, measure time to process, is quite easy to perform and interpret. We prepare a workload trace of a certain length with the desired operation types and data distribution and feed the data structure with that trace. If the length of the trace is a multiple of the size of the delta we can control the number of merge phases that are executed during the experiment. This ensures that all updates in the trace reached the read-optimized index and all deltas are clean. In the end every method has performed the same amount of work and all indexes are equivalent (i.e. they contain the same mappings).

In the second setup, measure throughput, things are a little more difficult. Since the experiment runs for a fixed amount of time, some things are out of control. The data structure will perform merges when needed and there is no guarantee that all updates reached the index when the time is up. To workaround this limitation, we set the runtime high enough to make sure there are several rebuilds during one run of the experiment. Despite of these shortcomings, this setup seems more realistic and more natural to perform. People are interested in high throughput over an extended period of time and not some quick effort over a short trace. Moreover this setup allows to plot the query rate versus the update rate.

For all experiments we first do a warm-up run. This run is ignored. After $n = 3$ runs we estimate the standard deviation $s$ using the sample standard deviation formula shown below [16]. If it is below 1% of the average $\bar{X}$, we stop. If $s$ is too high, we continue. After each run, $s$ is re-estimated and compared to the average.

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (X_i - \bar{X})^2} \quad \leq 0.01 \bar{X}$$

All time measurements are performed using the Java method System.nanoTime(). This returns the current value of the most precise available system timer, in nanoseconds. For the throughput measurements a timer thread is used that calls Thread.sleep(time) and stops the experiment when the time is up. Although this call only gives millisecond precision and there are no guarantees that this thread is scheduled exactly after the desired time, this is good enough for the experiments and the repetitions take care of the deviations.
4.4 Parameter Selection for the Index Structures

We investigate three index structures that work well with delta-indexing: sorted array with binary search, CSS-Tree and B$^+$-Tree. We complement our selection of indexes with a hash table (the same implementation that we use for the delta) to see how delta-indexing performs compared to conventional indexes. The following experiments aim to select the variable parameters of these data structures. The sorted array does not have any tuning parameters. For both CSS-Tree and B$^+$-Tree we can vary the size of the nodes. For the hash table we can vary the number of buckets. For the index structure it is important to have an excellent query performance and a reasonable bulkload time, we therefore measure these factors.

Figure 4.2 shows the results for (a) querying the index using an uniform trace, (b) querying the index using a skewed trace (i.e. many queries go to a small number of nodes), and (c) building the index structure when we vary the node size for B$^+$-Tree and CSS-Tree. We run 5 million queries on an index containing $2^{23}$ (about 8 million) elements and plot the average time for one single query versus the (log-scaled) node size. For the building experiment we measure the time it takes to bulkload $2^{23}$ elements from a sorted array into the index. We plot the average time for one single element versus the (log-scaled) node size.

The plots show that any size between 16 and 64 elements per node seems reasonable for the CSS-Tree. For the B$^+$-Tree a node size of 128 elements is optimal. For the minimum node size the dominating factor is the size of one cache line. When that portion of data is brought to the CPU, we should use all of the available information and not just a part of it. Especially in the case of the B$^+$-Tree the sharp edge right at the beginning is caused by the cache line effect.

The maximum node size is difficult to interpret. Many factors play a role there. On one hand the height of the tree gets smaller with larger node sizes but on the other hand the binary search within one node is more expensive if the nodes are large.

For further experiments we fix a node size of 32 for CSS-Trees and of 128 for (read-optimized) B$^+$-Trees.

We run a similar experiment as above but we vary the number of buckets for the hash table instead of the node size for the B$^+$-Tree and CSS-Tree. The setup is the same (i.e. 5 million queries on an index containing $2^{23}$ elements respective bulkloading an index from $2^{23}$ elements). The results are shown in Figure 4.3.

As expected, the hash table has a very good query performance that gets better the more buckets we use. Since there is no way to bulkload a hash table, the performance in the load experiment is very bad. We will see how the hash table performs when we do immediate insert later. For now we note that we should use as many buckets as possible to get the best performance from a hash table.
Figure 4.2: a), b) Average query time (5 million queries on an index containing 8 million elements, time for one query); c) Average build time (loading 8 million elements). We vary the node size of the B+-Tree and the CSS-Tree.
4.4 Parameter Selection for the Index Structures

Figure 4.3: a), b) Average query time (5 million queries on an index containing 8 million elements, time for one query); c) Average build time (loading 8 million elements). We vary the number of buckets for the hash table.
4.5 Parameter Selection for the Deltas

The different implementations of the deltas have different tuning parameters. We look at each of these in turn and measure the time to insert an element into the structure and the time it takes to produce a sorted array (i.e. an intermediary representation from which the desired read-optimized index will be built).

**array-based log** No parameters can be varied with this implementation. The selection of the sort algorithm and the memory layout have an effect on performance but these were already discussed and chosen in section 3.6.

**$B^+\text{-Tree}$** As pointed out before, we can vary the node size of a $B^+\text{-Tree}$. The experiments in the previous section aimed for good query performance. Now we are interested in good insert performance. Figure 4.4 shows the average time per element for (a) insert with an uniform trace, (b) insert with a skewed trace and (c) extract the data (i.e. bring it in sorted order to an array). We measure how long it takes to insert 5 million elements into a delta that already contains 8 million elements. The other experiment measures how long it takes to extract the data from the delta and bring it to an intermediary structure (i.e. a sorted array). We vary the node size of the $B^+\text{-Tree}$ while the other parameters remain fixed.

The results show that there is an optimal node size at 64 elements per node. If we have less elements per node, the tree gets too high. If we pack more elements into one node, the handling within one node gets too expensive. This is because binary search within one node takes more time and keeping the node sorted involves a lot of copying when the node contains many elements. For extracting the data it is better to have large nodes because we just scan at leaf level. If we have to change the leaf less often, we have less overhead. The benefit is not that large if we make the nodes larger than 64 elements but the overhead of inserting gets larger, thus we fix that node size for (write-optimized) $B^+\text{-Trees}$ in further experiments.

**chunk-based deltas** For the chunk-based deltas (i.e. partitioned log, hash table and partitioned hash table) the important parameter is the number of buckets. The more buckets we have, the simpler and faster insertion gets because we have less overhead (e.g. duplicate elimination, overflow chains) if less elements fall into one bucket. On the other hand it might get more difficult to collect and sort the data if it is distributed over many buckets. We do the same experiments as above (i.e. insert and extract) except that we vary the number of buckets and keep the other parameters stable. Figure 4.5 shows the experimental results.

As expected, inserting gets cheaper if we have more buckets. When we insert skewed data in a partitioned log with early aggregation which suffers very much

---

2Of course we need to reserve more memory for the delta if we want to have more buckets. But since we implement the buckets with chunks we need to reserve more chunks for buckets initially but the other chunks will only be allocated when required. Therefore the required space does not grow linearly with the number of buckets.
4.5 Parameter Selection for the Deltas

Figure 4.4: a), b) Average insert time (5 million inserts into a delta containing 8 million elements, time for one insert); c), d) Average extract time (extracting 8 million elements). We vary the node size of the $B^+$-Tree.
Figure 4.5: a), b) Average insert time (5 million inserts into a delta containing 8 million elements, time for one insert); c), d) Average extract time (extracting 8 million elements). We vary the number of buckets.
4.6 Maximum Update Rate Vs. Delta Size Experiment

from skewed data, the task gets only feasible with many buckets but the same
tendencies are visible for all deltas that use buckets.

The plots for extracting show opposite tendencies. If we use more buckets than the
delta contains elements, the performance suffers very much. In this case extracting
gets very expensive because we pay the overhead of processing a bucket but in
many cases do not get an element out of it. The results show that the optimum
is somewhere in the middle of the tested value domain when about 4 elements fall
into one bucket on average.

Since inserting is the more expensive task and should be weighted more, we will
use half as many buckets as the delta is supposed to contain elements in further
experiments and for all variants of chunked deltas. Although the partitioned log
with early aggregation has a good extract performance, it will not be used any
more since the insert performance in the skewed case is just too bad.

4.6 Maximum Update Rate Vs. Delta Size Experiment

Now that we selected reasonable parameters for both deltas and indexes we should take a
look at the relationship between these two building blocks of a delta-index. The number
of rebuilds is an important factor and this factor depends heavily on the size of the delta.
If the delta is small, we need more rebuilds but these rebuilds can be done faster. In
addition to that, most deltas tend to have bad insert performance if they grow too large.
It is important to find the right balance between these effects. Two examples should
illustrate the idea: Consider an array-based log as a delta. If we use a small delta, we
have many merges that can only be amortized over a small number of updates. On
the other hand the effects that inserts get more expensive when the array is large are
negligible, therefore we can make the delta as large as we want. The main limit is the
amount of staleness of the data in the read-optimized index that we want to allow. As
a second example consider a tree-based delta. The larger the tree gets, the costlier is
an insertion. The delta should not be too large. On the other hand, if the delta is too
small, we have many rebuilds that are costly.

The experiment works as follows: We preload the index with $2^{23}$ elements and measure
the time it takes to run a trace containing $2^{23}$ updates and report the update rate. The
experiment is finished if all updates are written to the read-optimized index (i.e. the
delta is empty and the rebuilding finished). If the delta can hold all updates, there will
be only one rebuild, if the delta can only hold for example a quarter of the updates,
there will be four rebuilds.

The results for a trace that contains updates for uniformly selected keys from an
uniformly selected keyspace (i.e. the index is loaded with uniformly selected keys, from
these keys the update keys are selected uniformly) are shown in Figure 4.6. Figure 4.7
shows the results for the same experiment when both the keys in the index and the
update keys taken from those are selected with skew.

For the batched $B^+$-Tree it seems that the size of the delta does not matter very much.
The larger the delta is, the more data we have to sort at once but we can profit more
Figure 4.6: Update rate for processing $2^{23}$ updates (uniform trace) on an index containing $2^{23}$ elements. We vary the delta size.
4.6 Maximum Update Rate Vs. Delta Size Experiment

Figure 4.7: Update rate for processing $2^{23}$ updates (skewed trace) on an index containing $2^{23}$ elements. We vary the delta size.
Experiments

from the better cache performance when we apply the data in larger batches to the tree. The overhead of having to sort larger deltas is a little higher when we use a partitioned hash table as delta. It seems reasonable to use a delta that has space for one eighth of the elements in the index (in our case 1 million elements).

For the other three indexes: sorted array, B⁺-Tree and CSS-Tree the differences are comparable no matter which delta we use. With a small delta size we have to do many rebuilds of the index and the B⁺-Tree gets worse because it takes longer to bulkload a B⁺-Tree than a CSS-Tree. The larger the deltas get, the smaller the differences between different index implementations. The time for the rebuild is not the dominating factor any more but the efficiency to collect and to sort large amounts of data.

Log-based deltas scale very well independent of the number of elements to be sorted. The experiment shows that we should make the log-based deltas as large as possible. For the other two implementations of deltas, the partitioned hash-table and the partitioned log, a similar tendency in favor of large deltas exists. The size of the delta should be at least a quarter of the size of the index. For larger deltas, the overhead of sorting many elements starts to kick in. The results are not very clear there. Depending on the workload, the effects vary. With uniform workload seems good to have a delta size of half or even quarter the size of the index. In the skewed case larger delta sizes are better. For the next experiments we will use a delta that has a quarter of the size of the index because that seems a reasonable number in both cases and forces more rebuilds which in turn will reduce the staleness of the data in the read-optimized index.

4.7 Query Rate Vs. Update Rate Experiment

The main focus of this thesis is the trade-off between query and update performance. As mentioned at the beginning, faster query performance involves some cost to set up an external structure or to optimize the existing structure. Depending on the ratio between updates and queries, different approaches are more cost-effective. The experiment works as follows: we preload the index with \(2^{23}\) elements. For a fixed amount of time we run a predefined mix of queries and updates against it. We selected 100 seconds as runtime, because in that time any setup will have to do a reasonable number of rebuilds but the experiment still finishes in reasonable time. We measure how many queries and how many updates we can process and plot this data. The plot (Figures 4.8 and 4.9) is a bit difficult to read because the datapoints resulting from the same setup do not have the same \(x\) coordinates as usual but they lay on a straight line going through the origin (i.e. the line where the ratio between query rate and update rate is the same). As a guideline the plot contains lines at the ratios we measured: 3:1, 1:1, and 1:3. We do the experiment this way, because it is very difficult to maintain a fixed rate for queries or updates (especially close to the limits of the data structure). Maintaining a fixed ratio between queries and updates on the other hand is very easy.

Based on the results from the experiments presented in Section 4.4 we selected the following implementations of read-optimized indexes to test:

- sorted array
- **B**⁺-Tree with a node size of 128 elements
- **CSS-Tree** with a node size of 32 elements

We combine these indexes with the following implementations of write-optimized deltas that have capacity for a quarter of the elements in the index (i.e. 2 million elements in this experiment). The selection of the parameters is based on the results from the experiments presented in Section 4.5:

- a log
- a partitioned log without early aggregation with half as many partitions as it can contain elements (i.e. 1 million partitions).
- a partitioned hash-table with half as many buckets as it can contain elements (i.e. 1 million buckets).

In addition to these nine combinations, we evaluate the performance of the following stores:

- **B**⁺-Tree with a **B**⁺-Tree as delta. The merge happens when the delta contains one eighth of the number of elements in the index.
- **B**⁺-Tree with immediate insert (i.e. no delta-indexing), with a node size of 32 tuples.
- batched **B**⁺-Tree (buffer the updates but apply the updates in sorted order to the tree instead of rebuilding the tree) with a node size of 32 elements. As buffer we use the three implementations of deltas that we selected for the delta-indexes with the same parameters but with a capacity of one eighth of the index size (i.e. 1 million elements).

For better readability of the plot we dropped several possible combinations based on the experiments in Sections 4.4 and 4.5. The results for an uniform trace are shown in Figure 4.8, the results for a skewed trace are shown in Figure 4.9.

As the other experiments foreshadowed, a **CSS-Tree** as read-optimized index in combination with a partitioned hash table as delta performs very well. When only updates are sent to the delta-index, it does not matter much what the read-optimized index is. Even the batched **B**⁺-Tree with a partitioned hash table as delta is competitive when only updates are processed. On the other hand, it does not matter what delta is used when only queries are used, a **CSS-Tree** will answer the queries very fast. It is then not surprising that the combination of the best delta and the best index performs so well. If only queries are processed, the combination of **CSS-Tree** and partitioned hash table is about twice as fast as a conventional **B**⁺-Tree. The more updates are to be performed, the more performance can be gained by using that delta-index. If for example 75% of the incoming workload are updates (i.e. ratio 3:1), the delta-index performs about three times better than the standard **B**⁺-Tree.
Figure 4.8: Query rate versus update rate (uniform trace). The index contains $2^{23}$ elements. We measure for 100 seconds. The black dotted lines show where the ratio between update rate and query rate is 3:1, 1:1 respective 1:3.
Figure 4.9: Query rate versus update rate (skewed trace). The index contains $2^{23}$ elements. We measure for 100 seconds. The black dotted lines show where the ratio between update rate and query rate is 3:1, 1:1 respective 1:3.
Even with a partitioned log as delta and a \( B^+ \)-Tree as index we can achieve twice the update performance of a standard \( B^+ \)-Tree. Delta-indexing works very well with many updates while still allowing a reasonable query performance. Depending on the implementation of the delta, the performance gain with mixed workloads (especially when there are 50% or more queries) is not very high. It may not be high enough to tolerate the staleness in the data that is introduced by our implementation of delta-indexing.

The tradeoffs between different index implementations are interesting as well. These tradeoffs can be seen best in the plot showing the results for the partitioned hash table as a delta with the uniform trace (Figure 4.9c). They are similar in all other setups but not so distinct. If we only perform updates, it does not make sense to build any index. This is why the delta-index with a sorted array as index has the highest update rate. But as soon as we have a very small number of queries in the workload, it pays off to build a CSS-Tree. But as long as we do not process more than 50% queries, a sorted array as an index is still better than a \( B^+ \)-Tree (the exact ratio depends on the implementation of the delta, if we use for example a log-based delta the additional cost to build a \( B^+ \)-Tree is not that large compared to the cost for sorting the delta and it pays off to build a \( B^+ \)-Tree when only 25% of the workload are queries ). Surprisingly a sorted array can even answer more queries than a standard \( B^+ \)-Tree if we only process queries and use the uniform trace. This can happen because the \( B^+ \)-Tree is not optimized and may have a lot of nodes that are only half full. A \( B^+ \)-Tree that was bulkloaded and has its nodes packed to the limit easily outperforms the sorted array. Even the batched \( B^+ \)-Tree seems to achieve a better organization of its nodes since the workload arrives in sorted order and the tree can answer more queries. In the skewed case on the other hand the \( B^+ \)-Tree can profit from cache effects more than the sorted array and achieves better query performance.

### 4.8 Staleness

Staleness is an important factor in delta-indexing but the experiments described so far did not measure staleness explicitly. We derive a cost model for the staleness based on the results from other experiments.

First, we have to define what we mean by staleness in our setup. Figure 4.10 shows an example for a timeline. Since we do not query the deltas we do not get up-to-date answers but get an answer that reflects the state of the database at the time when the latest rebuild started. The updates collected in the delta that was merged with the index in the latest rebuild will be present in the new index while the updates collected during the rebuild will be visible in the index only after the next rebuild. The time it takes until an update is reflected in the read-optimized index is the amount of staleness.

The example in Figure 4.10 shows how Index i1 is merged with Delta d1 when d1 is full. During that merge, the updates are collected in Delta d2 while the queries still go to Index i1. When the merge is complete, the Index i1 is replaced with Index i2 which now contains the updates from Delta d1 and the content of Index i1. As soon as
Figure 4.10: Timeline for a delta-index.
Delta d2 is full and the merge finished, another merge of i2 and d2 starts while updates are collected in Delta d3.

The staleness of the data depends on the time between two merges (i.e. period). It takes between one and two periods for an update to reach the index. If the update is the first to be written in a new delta it will only make it to the index after two periods because it has to wait until the delta is scheduled for merge (which will take one period because a merge just started) and then it takes another period until the merge finished and the index is replaced. If the update is the last to be written in a delta before the merge starts, it takes only one period (i.e. the time it takes to merge) to reach the index.

We see that the time between two merges is determined by the merge time if many updates are to be processed because the delta will then be filled very fast and we do as many merges as possible. This is the case in all our experiments. If we do not process a stream of updates that fills up the deltas quickly and forces us to do as many merges as possible, the time between two merges is determined by the time it takes until the delta is full. Our current implementation just waits for the delta to get full which in turn means that we get a high staleness because the time between two merges gets larger and therefore it takes more time until an update is reflected in the index. Our implementation was built with high update rates in mind but the problem can easily be solved by forcing merges after a fixed amount of time even if the delta is not full yet. The staleness is then limited by that parameter (if it is not smaller than the merge time).

The time for one period can be calculated using the different components of a delta-index. One period includes the time to insert all updates into the delta, sorting the delta, merging the data from the delta and the index and to build the index on top of the data. If we want to process queries in parallel, we have to take into account the time spend for answering the queries. This can be expressed as a formula as follows:

\[ N_U (T_I + T_E) + (N_U + N_I)T_M + N_I'T_L + N_QT_Q \]

where \( N_U \) is the number of updates in the delta, \( N_I \) is size of the index before the rebuild, \( N_I' \) is the size of the index after the rebuild and \( N_Q \) the number of queries that have to be processed in parallel\(^3\). The variables \( T \) capture the time per element used for different parts of the process:

- **\( T_I \) time to insert per element:** This is the time it takes to insert a single update into the delta (e.g. append it to the log).

- **\( T_E \) time to extract per element:** This is the time it takes to get the content of the delta into a sorted intermediary representation, for example read it from the log and sort the data. We take the total time of these two operations and divide it by

\(^3\)You might wonder why everything is linear in the number of elements, especially since sorting usually is in the order of \( O(n \log n) \). Our sort algorithm works in \( O(n) \) since we are partitioning the data into \( n \) buckets of constant size that can be sorted independently.
the number of elements to find the time per element.\footnote{We do not separately measure the two tasks ‘get’ and ‘sort’ because the deltas may combine these two tasks to get better performance. For example with a B\textsuperscript{+}-Tree the data does not need to be sorted and is directly read from the leaves of the tree. Opposed to that, the content of a hash table has to be copied from the different buckets into a compact array and needs to be sorted afterwards.}

- \(T_M\) time to merge per element: This is the time it takes to merge the sorted intermediary representation of the delta with the data from the index. This task was not evaluated experimentally until now and will be investigated below.

- \(T_L\) time to load per element: This is the time it takes to build the index on top of the sorted and merged data per element. This could be for example the time to bulkload a B\textsuperscript{+}-Tree.

- \(T_Q\) time to answer a query: This is the time it takes to get the answer for one query from the read-optimized index.

Since we are only processing updates and thus the size of the index does not change (i.e. \(N_I = N_I'\)) and if we ignore for a moment that we can process queries in parallel (i.e. \(N_Q = 0\)), we can simplify the formula to\footnote{We only consider the uniform case here because with skewed updates there are many updates that go to the same key and therefore the performance of the hash table is difficult to estimate, since the effect of the early aggregation is difficult to capture. When the trace contains many duplicate keys we cannot assume that the same number of updates that was inserted will be extracted when we do early aggregation. We then would need to use different variables for the number of updates inserted and the number of updates extracted.}:

\[
N_U(T_I + T_E + T_M) + N_I(T_M + T_L)
\]

We need an experiment to estimate \(T_M\) since that was not separately measured by the other experiments yet. We take a sorted array containing 8 million unique elements (representing the index) and merge that with a sorted array of variable length (representing the delta of different sizes). In one setup, the array representing the delta contains uniformly selected elements from the array representing the index, in the other setup the elements are selected with skew (i.e. lots of duplicates that can be skipped during the merge). Figure 4.11 shows the results.

We see that the time to merge is linear in the number of elements merged and therefore the time per element remains constant for uniform traces. In the skewed case the time to merge is about the same with small traces since only few duplicates can be skipped. If the array representing the delta gets larger, it contains more duplicates and the average time per element decreases.

Based on this experiment and the experiments described in Sections 4.4 and 4.5 we can estimate the variables as presented in Table 4.1.

Using these estimations of the variables, we can now predict the time it takes to process a certain number of updates when the index contains a fixed number of elements. We compare these predictions to an experiment that varies the number of updates processed during one merge (i.e. the delta size) while keeping the index size stable. Figure 4.12
Figure 4.11: Time to merge two sorted arrays. One has a fixed size of 8 million entries, the other size is varied.

<table>
<thead>
<tr>
<th></th>
<th>log</th>
<th>hash</th>
<th>partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_I$ [ns]</td>
<td>15</td>
<td>393</td>
<td>332</td>
</tr>
<tr>
<td>$T_E$ [ns]</td>
<td>602</td>
<td>138</td>
<td>183</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>array</th>
<th>B$^+$-Tree</th>
<th>CSS-Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_L$ [ns]</td>
<td>0</td>
<td>73</td>
<td>27</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>intermediary representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_M$ [ns]</td>
<td>22</td>
</tr>
</tbody>
</table>

Table 4.1: Estimations for the variables to calculate the staleness based on the experimental results from Sections 4.4, 4.5 and 4.8.
shows the results of the experiment. Figure 4.13 shows a comparison of the experimental results with the estimation from our cost model for two selected combinations of delta and index.

The results from the experiments show that we can process about 1 to 2 million updates within one second when the index contains 8 million elements. A delta size of 1 million elements seems a reasonable size because when we process less updates per merge the overhead of rebuilding the index for the 8 million elements from the index is kicking in. With larger deltas the staleness grows very fast to several seconds.

Compared to the estimation based on the cost model we see that for the sorted array index with a partition-based delta the model is quite accurate. The sorted array as index behaves very predictable, which makes the estimation easier.

The plot shows a $B^+$-Tree as index with a log-based delta as well. The estimation is quite accurate for large delta sizes, but our cost model overestimates the effect of rebuilding the tree for the elements that are already in the index (i.e. the part of the model multiplied by $N_I$, the number of elements in the index) and thus the estimation is not very accurate for small index sizes. One reason for this may be that $T_M$, the time it takes to actually merge the intermediary representation from the delta with the content of the index, and $T_L$, the time to load the index, are not separately measurable in a $B^+$-Tree because we do not extract the data from the tree to an intermediary representation since we do not have to sort it. Our cost model does not take this into account properly.

The combination that performed best in the other experiments, a CSS-Tree as index with a partitioned hash-table as delta is compared to the prediction from our cost model as well. The shape of the line seems to be matched very accurately, our model is just a constant offset away from the experimental results. This is most likely due to cache effects that are not captured by measuring the different times independently.
Figure 4.12: Time for one processing cycle (i.e. staleness). The index contains 8 million elements.
Figure 4.13: Comparison of experimental results for staleness with estimation from cost model.
5 Conclusions

5.1 Conclusion

The basic idea of delta-indexing is to keep updates in a small write-optimized delta and answer queries using a large read-optimized index. The two structures are merged from time to time and a new read-optimized index is built. We applied this idea to relational data and experimentally evaluated the performance in main memory.

We presented several implementations for the write-optimized delta with different characteristics. A simple log-based delta has very high insert performance but it is expensive to sort. A $B^+$-Tree as a delta already sorts the input while inserting but that makes inserting expensive. A hash table allows to quickly eliminate duplicates and we therefore can use a faster sort algorithm because the sort does not have to be stable, but the performance gain is not as large compared to a simple log. Partitioning the data while inserting it into the delta brings a performance gain in the sort step but suffers with skewed workloads. A partitioned hash table reduces the penalty of skewed data while not loosing too much performance of the partitioning delta.

Our experiments included three types of read-optimized indexes: a sorted array with binary search is quickly built but does not bring much performance gain when querying. A $B^+$-Tree boosts queries reasonably well but takes some time to build. A CSS-Tree achieves even better query performance as the $B^+$-Tree and takes less time to build.

The number of updates that are processed in one merge step is an important factor. Our strategy is to merge when the delta collected a fixed number of updates, therefore we can directly control that factor. Our experiments showed that we should collect at least one quarter of the number of elements in the read-optimized index to achieve good performance.

Delta-indexing for relational data in main memory seems to work. Our experiments showed that even with a simple log-structured delta and a standard $B^+$-Tree we can achieve twice the update performance of a $B^+$-Tree without delta-indexing. If we put some more effort into the organization of the delta we can gain another factor of performance.

Our implementation of delta-indexing does not consult the delta when answering a query. Thus, the query result is not up-to-date. Our experiments showed that the staleness of the data in the read-optimized index is not higher than one second if we carefully select the size of the delta. Our proposed cost model for the staleness is able to predict the amount of staleness quite accurate.
5.2 Future Work

**other workloads** This thesis focused on queries and updates. The experiments we did can be adapted for other tasks like inserts and deletes. Some attention has to be paid when the index is growing since many of the factors we measured are influenced by the size of the index.

The workloads used in this thesis were produced with random generators. Although we had relational data in mind when we constructed the traces, we did not use a real benchmark. An interesting experiment would be to run a benchmark like TPC-C\(^1\) where the conventional indexes are replaced by delta-indexes. Since TPC-C requires up-to-date answers and does not tolerate staleness, the delta-indexes should be improved to query the deltas in addition to the index.

**query the deltas** As mentioned above, it would be interesting to measure how much performance is lost, if we query the deltas in order to produce up-to-date answers. Although there are a lot of papers on delta-indexing that queried the delta, it is still an open question how that performs in main memory since other work experimented on external memory.

**range queries** The indexes we investigated are well prepared to answer range queries. Delta-indexing is very useful when the read-optimized index provides some functionality that other data structures, for example hash tables, do not have. If just point queries are sent to the index, an immediate hash table outperforms our implementations of delta-indexes. An experiment with range queries that really exploit the functionality of B\(^+\)-Trees would be interesting.

**multi-core machines** Our experiments always run on a single core because we are not interested in absolute performance but wanted to analyze the relative performance of delta-indexing compared to other solutions like standard B\(^+\)-Trees. Nevertheless, our implementation is ready to be run on multiple cores without any changes since the processing of updates and queries happens in a different thread than the merge. The merge step could easily be parallelized even more by using distributed sorting algorithms and the like. Especially the partitioned log and the partitioned hash table can easily be processed with multiple threads since the data is already partitioned and can be distributed to different threads that handle the sorting within the partitions and buckets.

In order to compare a multi-core-variant of a delta-index to other (not multi-threaded) indexes, one could use the ideas presented by Stonebraker et al. \cite{Stonebraker00} and partition the data over multiple instances of indexes (e.g. B\(^+\)-Trees) to exploit the available cores.

\(^1\)See [http://www.tpc.org/tpcc/](http://www.tpc.org/tpcc/).
Acknowledgements

I would like to express my gratitude to my supervisors Dr. Jens Dittrich, Lukas Blunschi and Marcos Antonio Vaz Salles for their provided help, guidance and continuous support through the course of this work. I would like to thank them and the whole DBIS group for the very friendly and helpful atmosphere in the office.

Special thanks go to my fellow students who read drafts of this thesis and provided me with additional feedback, and to my friends and family for being around.
A Source Code Overview

This Appendix presents in some more detail the source code used for this thesis. We use the structure of the Java packages as guideline and explain the code for the deltas, the indexes and the mergers first. Then we cover additional packages that contain the data generation classes and other tools as well as the classes for running the actual experiments.

A.1 Package deltas

The package *deltas* contains the classes representing different implementations of deltas. There are three subpackages *log*, *chunked* and *tree* that divide the deltas according to the underlying memory allocation scheme. Figure A.1 shows the hierarchy of the different delta implementations.

Each delta implements the methods defined in the interface *Delta*. The main functionality is to store elements, the methods *log* and *logArray* take a key and a value respective an array of values and return true if the mapping was successfully put into the delta. The method *reset* cleans the delta and makes it available for reuse. The method *getNumberOfElements* returns the number of elements that are stored in the delta (if the delta does early aggregation and stores only one update for each key, this is the number of keys in the delta) and *isFull* returns true if the delta is full (according to the defined capacity). In order to avoid the expensive *instanceOf* operator, each delta has to implement a method that returns true if the returned result contains a timestamp that needs special handling. The methods *getNext* and *setNext* take care of the queue functionality for deltas. The class *AbstractDelta* provides an implementation for the methods needed for the queue functionality and defines the default answer for *hasTimestamp*. All deltas extend this abstract class.

The class *DeltaQueue* represents a thread-safe implementation of a queue that contains deltas. It uses the methods *getNext* and *setNext* of the deltas and takes care of the locking. The methods *get* and *put* are the usual queue functions. The methods *pause* and *resume* are used to control the blocking behavior of the queue. After a call of *pause* the queue will return all remaining deltas it contains in subsequent calls to *get*. If all deltas are processed, it returns null instead of blocking as usual. A call to *resume* will switch back to the normal behavior. This functionality can be used to let the queue run empty, for example to ensure that all elements are written to the read-optimized index.

The interface *DeltaFactory* defines the method *getDelta* for a factory producing deltas. Each delta provides a static method that returns such a factory which can then be used to produce deltas to initially fill the queue.
A.1.1 Subpackage log

The subpackage log contains the different implementations of deltas that are based on a large array and store updates sequentially in that array. There are two characteristics to distinguish these log-based deltas: the structure of the array and the usage of a timestamp. We can use linear arrays (i.e. the classes called Linear...Delta) where we align all data sequentially in a continuous block of memory and use offset computation to access the data or we can use multi-dimensional arrays (i.e. the classes called MultiDim...Delta) where each update is stored as a row in a two-dimensional array. Since we might want to sort the content of the deltas with a stabilized QuickSort, we need variants of the deltas that write a timestamp to the data that forces a total order on the elements. These variants contain the infix ...TS... in their classname.

The class LinearLongDelta is a bit a special case. It stores the data all as longs in a linear array but the first element is a combination of the key and a timestamp. Since the key has only 32 bits we can use the other 32 bits of the long to force the total order on the keys by gluing them together (i.e. \( m = (k \ll 32) | t \), where \( \ll \) denotes the shift left operation and | denotes the bitwise or operation). This manipulation simplifies the stabilized QuickSort since we need only one CPU operation (on a 64 bit machine) to compare each element instead of having to consider a separate timestamp if two keys otherwise match when we store the data as ints. The problem with this implementation of course is that the key has to be extracted from the combination first to be further used. The experiments will have to show whether the cheaper comparison is worth the more expensive extraction. The other problem is that all data will be stored as longs which needs more memory than storing it as ints or if we combine two values from the payload to one long we need special treatment to extract the values.
All log-based deltas work basically the same. The data is appended to the log (if necessary a timestamp is added) and a counter is updated that can be used to know where to append. This counter can be used to know when the delta is full. Each implementation provides a method to get the content of the delta. Since the content will always have another type, this could not be defined in an interface. Constructors take an argument that defines the capacity of the array and therefore the capacity of the delta. The size of one tuple can be set at construction time.

A.1.2 Subpackage chunked

The subpackage chunked contains deltas that are based on chunks of memory. The abstract class AbstractChunkedDelta contains the logic to handle these chunks. It provides a method insert that inserts the given key-value-mapping into a specified bucket. This method will take care of chaining more chunks together to make enough storage for a given bucket. The class contains a method toLinearArray that will extract the data from the chunks and write sequentially to a linear array. The abstract methods handleBucket and handleResult are called when all chunks of one bucket respectively of the whole delta were written to the array. Descendants can then do the necessary post-processing.

In the class HashTableChunked the bucket is chosen according to the hash value of the key while in the class PartitionTableChunked the buckets are partitioning the key domain. The constructors of these classes define how many buckets should be used and how large the capacity of the delta should be. Since some knowledge about the key domain can improve performance, there is a flag that indicates whether the delta will only contain positive keys or any key. The constructor for class PartitionTableChunked has an additional flag that indicates whether updates on the same key should be aggregated early (by using a scan) or not.

The class PartitionHashTableChunked does a more sophisticated allocation of the chunks and the tuples are distributed over the buckets in several levels. For an abstract view see section 3.5. The class overrides the methods insert and toLinearArray from AbstractChunkedDelta to implement that more complicated allocation.

A.1.3 Subpackage tree

The subpackage tree contains a delta that uses a B+-Tree as storage. It is basically an adapter that adapts the methods from the interface Delta to the B+-Tree from the package indexes below.

A.2 Package indexes

The package indexes contains different implementations of indexes. We set up a hierarchy of interfaces and abstract classes to distinguish the different implementations. Figure A.2 shows the hierarchy of the index implementations.
Every index can answer queries. This functionality is defined in the interface `Index`. Then indexes can be either mutable or immutable. The interface `ImmutableIndex` is just a tagging interface for immutable indexes that do not provide any further functionality. The interface `MutableIndex` defines the methods that allow updating elements in an index.

### A.2.1 Subpackage immutable

The subpackage `immutable` contains the indexes that are immutable, i.e. that can only be bulkloaded.

The class `SortedArray` takes a sorted linear array and uses binary search to answer queries. No additional logic is used there.

The class `CSSTree` is a translation of the code originally written in C by Rao and Ross [30]. Based on a sorted array it sets up a directory that speeds up queries. It is like a B+ Tree that does not store the data in the leaves but in an array. Since the nodes of the directory are aligned in one array only offsets in that array have to be used. The structure uses no pointers and other object overhead whatsoever.

### A.2.2 Subpackage mutable

The subpackage `mutable` contains the indexes that can process updates. We differentiate mutable indexes again in immediate indexes (i.e. change is immediately represented in the index) and deferred indexes (i.e. delta-indexes). The interfaces `ImmediateIndex` (only a tagging interface) and `DeferredIndex` (contains `pause` and `resume` functionality) take care of this separation.
A.3 Package mergers

Subpackage immediate

The subpackage immediate contains indexes that apply updates without delay to their main structure. The subpackage contains an implementation of a hash table that basically extends the hash table implementation used as a delta and adapts it to the interface of a mutable index and adds the query functionality. The second immediate index is a B+-Tree and is contained in its own package since a lot of helper classes are used.

Subpackage deferred

The subpackage deferred contains the delta-indexes. The abstract class AbstractDeferredIndex takes care of common functionalities (like get from index, put into delta, queue management, etc.) for delta-indexes. For each implementation of the read-optimized index exists a subclass of AbstractDeferredIndex that defines how the merge happens.

A.3 Package mergers

The package mergers contains the classes handling the merging of indexes and deltas. We spent these classes an extra package to have a better overview but the functionality could easily be moved to the delta and deferred index classes so no separate merge classes would be necessary. In order to make the structure of a delta-index more obvious from the package hierarchy, we added this package.

The abstract class AbstractMerger contains the switch statement to distinguish between different delta implementations and to chose the right access method for the delta. It gets the data from the delta in the corresponding intermediary representation (i.e. linear or multi-dimensional array with or without timestamp, or tree) and calls the corresponding abstract method for the actual merge.

The concrete classes ArrayMerger, BTreeMerger and CSSTreeMerge contain the corresponding code for merging different intermediary representations with the respective indexes.

A.4 Package data

The package data contains the class DataGenerator which produces traces according to different random distributions and requirements. It can produce traces according to an uniform distribution, a skewed 80-20-distribution or a sequential trace. The traces contain only unique keys. It can produce traces that do not contain certain elements (for example to produce a trace that contains only inserts, we need keys that are not yet in the index). The third option is to produce traces that contain only keys from a certain dataset (for example to produce traces that contain updates). Again these keys can be selected to the three distributions but they may contain duplicates.
A.5 Package tools

The package tools contains a set of additional tools like random number generators, implementations of sorting algorithms and threading tools that are used by the other classes.

A.5.1 Subpackage random

The subpackage random contains random number generators.

The classes MersenneTwister and MersenneTwisterFast produce uniformly distributed random numbers using the Mersenne Twister algorithm\(^1\). The implementation is taken from Sean Luke and the ECJ project\(^2\).

The class EightyTwentyDistribution produces random numbers according to a skewed distribution. For more details see [8].

A.5.2 Subpackage sorting

The subpackage sorting contains classes that sort different types of arrays with different sorting algorithms.

The classes QuickSort and MergeSort adapt the respective implementations from the Java library to multi-dimensional and linear arrays. The class StabilizedQuickSort extends the normal QuickSort to use a timestamp in order to stabilize the sort.

A.5.3 Subpackage threading

The subpackage threading contains some helpful classes to control threading. The class ThreadingTools conveniently wraps some system calls like sleep and gc. The interface Resumable and the abstract class ResumableRunnable extend the notion of a thread from the library with functions for stopping and resuming a thread.

A.6 Package experiments

The package experiments contains the classes the run the actual experiments. They combine the indexes and deltas, stress them with a certain workload and measure different aspects.

The abstract class AbstractExperiment contains methods that can be helpful for all experiments, like preparing files to write out the results, printing status information to the console or calculating averages.

The experiments are divided into several subpackages that are described below.

---

\(^1\)http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
\(^2\)http://cs.gmu.edu/~eclab/projects/ecj/
A.6 Package experiments

A.6.1 Subpackage memoryLayout

The subpackage memoryLayout contains experiments to investigate the performance of different memory layouts.

The class HashTablePerformance compares our implementation of a hash table with the implementation from the Java library.

The class LinearVsMulti compares linear arrays to multi-dimensional arrays by summing up one column of a table. This experiment is used to produce table 3.2. The main method takes the number of elements (i.e. rows) in the table as an argument.

A.6.2 Subpackage sortExperiments

The subpackage sortExperiments contains experiments to investigate the sort performance of different implementations.

The main experiment is written in class TimeToProcessCompleteArray. This experiment runs all steps involved after each other and measures the time for each step. This experiment is used to produce the numbers that Figures 3.4 and 3.5 are based on. If the main method is called with no argument, it will use a trace that contains only unique keys, if it is called with the string duplicates as argument, it will use a trace that contains 50% duplicates.

The other classes TimeToInsertArray, TimeToSortArray and TimeToAccessArray only measure a single step of the whole process and are superseded by the experiment TimeToProcessCompleteArray.

A.6.3 Subpackage distributions

The subpackage distributions contains classes that analyze the distribution of the used workload traces.

The class DistributionAnalysis is used to produce Figure 4.1a. The class DistributionAnalysisSkew is used to produce Figure 4.1b.

A.6.4 Subpackage deltaParameter

The subpackage deltaParameter contains experiments to measure the performance of the different delta implementations.

There exist two sets of experiments, one measures the insert performance and the other measures the extract performance (i.e. bring content in an intermediate representation suitable for merging with the index). Each of the two sets contain three experiments: one varying the number of elements, one varying the number of buckets and one varying the node size. All experiments take one argument that describes the distribution of the workload that should be used: uniform for an uniformly distributed trace, skewed for an 80-20-distribution.

These experiments are used to produce Figures 4.4 and 4.5.
A.6.5 Subpackage indexParameter

The subpackage `indexParameter` contains experiments to measure the performance of the different index implementations.

There exist two sets of experiments, one measures the query performance and the other measures the time to build the index. Each of the two sets contain three experiments: one varying the number of elements, one varying the number of buckets and one varying the node size. All experiments take one argument that describes the distribution of the workload that should be used: `uniform` for an uniformly distributed trace, `skewed` for an 80-20-distribution.

These experiments are used to produce Figures 4.2 and 4.3.

A.6.6 Subpackage deltaIndexingExperiments

The subpackage `deltaIndexingExperiments` contains the experiments for Sections 4.6, 4.7 and 4.8.

The class `TimeToIndexVsDeltasize` measures how long it takes to process a trace of fixed length varying the delta size. It takes three arguments:

1. a string determining the type of workload: either `updates` for a workload only updating the existing elements in the index or `inserts` for a workload that only inserts new elements into the index.

2. a string determining the distribution of the keys that are preloaded in the index before the experiment starts. This can be either `uniform` or `skewed`.

3. a string determining the distribution of the keys of the workload. The distribution depends on the distribution of the elements in the index. If the index is updated, the keys for the update trace are selected from the preloaded keys with the given distribution. If new elements are inserted, the keys from the insert trace may not be contained in the index already. These two restrictions make the distribution dependent on the keys in the index. This argument can be either `uniform` or `skewed`.

The class `QueryRateVsUpdateRate` uses mixed workloads and measures the update rate and query rate. It takes two arguments determining the distribution of the preload trace and the workload traces as described above for arguments 2 and 3 of class `TimeToIndexVsDeltasize`.

The class `Staleness` measures the time it takes to process a certain number of updates. These numbers are used to derive the staleness of the data. It takes the same arguments as the `TimeToIndexVsDeltasize` class. It essentially does the same as `TimeToIndexVsDeltasize` but instead of always processing a fixed number of updates with different delta sizes it measures the time to process different numbers of updates.
A.7 Helper Scripts: Plotting

The folder output contains the scripts to draw graphs from the data produced by the experiments.

The file linestyles.conf contains a set of predefined line styles for GNUPlot that is used in the plot scripts. The file bargraph.pl is a script used to make bargraphs with GNUPlot. The file epstopdf is a script used to produce pdf files from the eps graphics produced by GNUPlot.

The other scripts prepare the input for GNUPlot or bargraph.pl to produce the figures used in this thesis. They gather the latest versions of the needed data from the directory ./plot and write the results to the same directory. The scripts take parameters similar to the experiments. For more details see the headers of the scripts.

A.8 Helper Scripts: Configuration, Compiling and Execution of Experiments

In the root of the directory, a set of BASH script can be found. The simplify common tasks for building and running the project.

The script config.sh contains parameters for the Java compiler and the virtual machine. It is used by other scripts to set the necessary parameters like classpath or heap size.

The script compile.sh takes care of compiling the whole project.

The script run.sh runs the experiment specified by the first argument. The experiment has to be specified as when invoking java. Other parameters given to run.sh will be passed on to the experiment.

The scripts batch.sh take care of running multiple variants of experiments. They pass all possible combinations of arguments to the respective experiments and execute them as a batch. For example batchDeltaParameters.sh runs both the insert and extract experiments on the deltas while varying all parameters (i.e. number of elements, number of buckets and node size) and using both distributions (i.e. uniform and skewed).

---

3Note that the plot scripts assume that epstopdf is available somewhere in the execution path of the operation system.
Bibliography


