Universiteit Utrecht

High Performance String Searching in Main Memory Databases

Author:
Matthijs Meulenbrug[1]

Supervisor:
drs. Hans Philippi

June 26, 2012

[1] Student number: 3154459

Abstract

There are many applications that need to search a small database containing small records. Often this database has to be queried thousands of times a second. Traditional full text search engines have been created for searching huge databases and large individual records. It is common that only a part of the index resides in memory. The bulk however is stored on disk. As each read operation from disk is an order of magnitude slower than a read operation from memory we like to have a tool specifically designed for use with keyword matching in small database containing small records. The key is speed and a high number of searches per second. In this thesis we inspect existing full text tools and then show how we implemented a completely new tool designed specifically for this task. We describe how we used specialized data structures to make optimal use of main memory and in particular cache memory, and what problems we have overcome.

Acknowledgments

I would like to thank Utrecht University for giving me a better and deeper understanding of Computer Science, but also for the great people and atmosphere which made it a joy to attend.

In particular I would like to thank my supervisor Hans Philippi for guiding me while working on my Master of Science project. Especially for his patience during the project as my own company took up a lot of time and stretched the time it took me to finish this project!

I would also like to thank my colleagues at Nova Group B.V. for their support.

Contents

Chapter 1

Introduction

More and more information becomes available each day and in many cases this information needs to be searchable within an acceptable timeframe. Full text search systems have been created to facilitate existing database systems in doing so. These full text search systems have improved over time handling terabytes of data and returning relevant results in several tens of a second.

1.1 User expectations/interaction

When using a full text search systems the users expect to see certain results according to past experiences. They are used to the search engines they use every single day and expect a search box on other websites to behave similarly. The results have to be consistent. When the character '*' can be used to do prefix search, adding '*' to the front of a query word instead of the back should trigger a postfix search. Users without knowledge of the inner workings of DBMSs cannot see the technical difficulties that are involved in postfix search in contrast to prefix search.

Search companies like Google put the quality search results bar very high for smaller websites where search functionality is a feature to facilitate easy site navigation. It is up to the programmers of a website to make sure the search functionality does not differ too much from what the users are used to.

The same holds for large search portals that handle millions of queries a day. One particular brand of large search portals are sites like the Internet Movie Database, more commonly known as IMDb (http://www.imdb.com) and until recently the download search engine Mininova (http://www.mininova.org) These sites have to handle hundreds or thousands of queries a second, yet have only a relatively small database, at most a couple million records. Also the type of queries for these sites are small, users will look for titles of movies, software, music and more.

This is a big contrast with what all the existing implementations of full text search engines are developed for today.

1.2 Related Work

Full text search has become more relevant in recent years, not just for web applications, but also desktop computers searching ones own files. Research into the field of full text search has been conducted for the last 40 years. As described before most of that work focusses on large collections where the bulk of the corpus was stored on disk as memory was still very expensive, especially in the early years of information retrieval. This research can roughly be divided into two separate fields. The first focussing on string matching, and approximate string matching. The second, broader field, on storing, retrieving, and performing subsequent operations like intersection on posting lists.

Zobel et al. [22] in their paper "Inverted files for text search engines" created an extensive overview of the work conducted until 2006. It becomes clear that a lot of work has been done on large databases where compression plays an important part of the work. This is needed to keep the index manageable and in main memory as much as possible.

1.2.1 Data structure

There are several possible ways to implement a string search engine. Depending on the implementation a structure is necessary to quickly retrieve posting lists. Structures like a suffix array and an inverted file can be used. Puglisi et al. [18] states that the inverted file appears to be the better choice in most cases, albeit mainly for a compressed index. Zobel et al. [22] support this statement, also with uncompressed indexes, but expresses concern for space usage for these (uncompressed) indexes.

Stored in this structure are often n-grams, as presented by Cavner et al. [1] in 1994. Generating the n-grams from a keyword is done by generating every n character string from the keyword provided. For example for the string "Utrecht" the following 3-grams are generated:

```
('$Ut', 'Utr', 'tre', 'rec', 'ech', 'cht', 'ht$')
```

The $ sign is used to delimit the start and end of the string. An approach derived from n-grams is the use of variable length n-grams called VGRAMS [14]. Using variable length n-grams can improve performance by carefully selecting relevant grams.

Posting lists

Posting lists do not necessarily store only the document IDs. It is not uncommon to also store frequency and/or ranking information alongside

the document ID. These lists are most commonly resizable lists or linked lists consisting of smaller arrays [22].

Depending on the kind of structure and database there are three possible update strategies:

- Rebuild the entire index

  The index itself is immutable or insertion is so slow that rebuilding is the best posibility.

- Delta index

  A secondary index is kept to enable to store changes.

- Incremental index

  Changes are propagated to the index immediately.

1.2.2 Ranking

When presenting results to the user the most relevant result should be on top. Algorithms that are often used to score results are term frequency and inverse document frequency (TF/IDF) [7, 20]. TF/IDF is a weighted similarity function used to determine the similarity between the query and the records being searched. The term frequency uses the notion that if a keyword occurs more often in a single record it is more important. So if a word occurs twice within a record its score is doubled. However words that are frequent in the entire corpus, like for example "the" or keywords relevant for each record, will have a negative impact on the scoring as irrelevant records will score highly due to these words. The inverse document frequency decreases the importance of keywords relative to the frequency of the keyword in the entire corpus, thereby increasing the relevance of unique low frequency words.

1.2.3 Approximate String Matching

Approximate string matching can be implemented cheaply when implemented in combination with TF/IDF [8]. Using n-grams and the TF/IDF of the n-grams in combination with the query n-grams similar strings can be identified relatively cheap. Other approaches to approximate string matching use edit distance between two strings to identify possible relevant records [11].

Chapter 2

Existing implementations

To add search functionality to ones website there are several existing implementations available. Many of these systems are an addition to the existing database infrastructure. The full text index for such an external system is created by doing a full table scan (query) on the source data and inserting each record into the full text index with its associating primary key. Then, when a full text query is performed, the search system returns a list with primary keys which are in turn are retrieved from the DBMS.

With database implementations that include full text search functionality it is often merely added as a proof of concept instead of a full-fledged feature to be used in a production environment.

Of all the full text search implementations available at this time, most are aimed towards searching huge databases of data stored on disk. As described above these implementations are not great for high volume sites with small datasets.

We looked at several implementations that are widely used in order to determine their strengths and weaknesses.

- Sphinx
- Mysql Full Text Search
- Lucene

2.1 Sphinx

Sphinx[1] has quickly risen to become one of the most popular solutions for full text-search and is used by many websites including Mininova. Sphinx is a standalone full text search engine aimed to quickly search gigabytes of data with a lot of features including infix searching, relevance scoring and more. It is being actively developed by its main creator Andrew Aksyonoff.

[1] More information can be found at http://www.sphinxsearch.com

Sphinx is diverging to becoming a full featured database that has very good full text search capabilities.

2.1.1 Features

Sphinx has too many features to list here, so below is a short overview of what we believe to be the most important ones.

High indexing speed In excess of 10 MB/s on modern CPUs

High search speed Can search up to 100 GB of text in sub-second response times

Advanced search options Ability to do word proximity, boolean and phrase queries

MySQL integration Acts as a mysql storage engine, making sphinx virtually compatible with every popular programming language

2.1.2 Rotating Indexes

Due to the nature of Sphinx' implementation, the index is virtually immutable and instead of modifying or inserting new records in the existing index it would often be equally as fast or faster to regenerate a new index from scratch instead. The solution Sphinx provides is to use a delta index, instead of rebuilding a new index every time a new records needs to be inserted, or regenerating the index several times a day, a secondary small index is used to store new records. Deletions and updates are handled using a kill-file. The document IDs in the main index that are to be removed or updated are put in a kill-file. This files' document IDs are suppressed in the results from the main index and, in case of an update, the results from the delta index are taken. Because this index is so small it can be regenerated quickly, leaving only a short amount of time where new records would not be reflected in the search index.

Depending on the insertion rate of new records this delta-index has to be merged with the main index at some point in time. The more insertions the more often it has to be merged. Even though this works quite well it might still take a considerable amount of time for the delta index to be regenerated. This could potentially confuse users that will find records manually that they cannot locate using the provided search functionality. The search results need to be up to date at all times.

2.1.3 Main memory properties

Sphinx will cache, when possible, dictionary files to memory. Several other small structures are cached in main memory as well, the postings list on

the other hand will be kept on disk. The structures that will be cached to memory are all optimized to save (disk) space, as described in the section below. They therefore increase query speed when stored on disk, however when kept in memory this is only useful with large databases where a dictionary file otherwise might now have fit in memory. This is reflected in the configuration options where the user has little control over what will be kept in memory or not. Most options related to memory use are memory constraints when assembling query results, not so much the ability to set caching options.

It is clear that its strength lies in its ability to quickly search huge collections on disk, which is exactly what it was designed to do.

2.1.4 Implementation

The implementation of Sphinx is in its core an inverted file. The three files that make up this inverted file is a dictionary list file (`.spi`), a document list file (`.spd`) and its associated hit list file (`.spp`).

The dictionary file contains all the keywords, stored as an *id*, followed by a position in the document list file. This document list file, in turn, holds all the documents in which the query keyword occurs one or more times. Lastly, the hit list file holds all the positions where the keyword occurs within one document.

These files use several compression methods in order to make them as small as possible.

- Delta Encoding

  Compression method where the only the difference between the current and previous value is stored. This is reset after a number of values in order to minimize the need to reconstruct the value from "scratch".

- Variable length byte string

  Instead of fixed 32 or 64 bits integer values only 7 bits are used, the 8th bit is used to indicate that the next byte is also part of this value, therefore minimizing space needed to store (especially smaller) values.

The implementation is rather straightforward, for infix searching for example, Sphinx indexes every possible substring including the word to be indexed. But due to the amount of features sphinx consists of a large code base of over 45.000 lines of code.

Besides the search-daemon Sphinx also supports quite a few other ways to communicate with the outside world. It can be installed as a MySQL storage engine and even, included in the last release, act as a mysql server itself.

2.2 MySQL

MySQL is one of the most well known DBMSes available. It is open source and being actively developed by dozens of developers around the world. Many large companies rely on its stability and speed, and have done so for years. Even though it is free, easy to set up and easy to learn, MySQL has many powerful features like replication and stored procedures. MySQL supports two storage engines out of the box: InnoDB and MyISAM. Both engines first and foremost job is to be a relational database system, it focusses on retrieving data quickly from tables and joining them through use of a query execution plan. InnoDB, which is a newer engine than MyISAM, has transaction and foreign key support. The older MyISAM does not have transaction support, but in most cases is a little faster than InnoDB (mainly due to the lack of transactions) and has built-in support for full text search.

The easiest way to do infix search queries in both engines is to use the `LIKE` keyword as follows:

```
SELECT id, name FROM Users WHERE name LIKE "%Brown%"
```

This might be easy to use, but will result in a full table scan, therefore making it practically useless in an environment where quick results are vital.

2.2.1 Full text search capabilities

The MySQL full text index is a vast improvement over the `LIKE` keyword. It was first introduced in MyISAM in late 1999, since then quite a few parts have been rewritten and improved. In 2001 boolean full text search was added and since 2003 it also support unicode.

The full text capabilities of MyISAM consists of a specialized "full text index" to be generated over a column or set of columns and the associated SQL keywords to trigger the use of this index. The syntax is as follows:

```
MATCH (col1,..) AGAINST (expression [, mode])
```

Full text search in MySQL has three different types of queries:

1. Natural language search
2. Query expansion search
3. Boolean search

2.2.2 Natural language

Using the natural language search mode the user can enter a search term as one would expect to ask a question. This means no words can be excluded, but also that there are no special operators. If words are too small or are part of the stop-word list they will be ignored. Also words that occur in more than 50% of the rows are ignored, meaning that if a dataset is small enough it could become very difficult to get results.

2.2.3 Query expansion search

Query expansion search mode extends a query to include words that might be relevant to the search at hand. A query for the word "car" may be extended to a query for not only "car" but also "Volvo", "BMW" and "Audi". The user would assume that the search system has knowledge of the query and might therefore return more relevant results. It works by doing a second pass with words that occur most often in the first pass. Therefore results might be returned that do not include the word "car" but only "Audi".

A downside to query expansion is the possibility that it might insert a lot of noise into the query result set.

2.2.4 Boolean search

In order to use boolean search query, the mode parameter has to be set to `IN BOOLEAN MODE`. This allows the user to add characters with a special meaning to search queries. For example the `+` sign means the word must appear in each row returned, the − sign, on the other hand, means a word must *not* appear any of the rows returned. Besides these trivial cases there are also other characters with special meaning like the `<` and `>` characters that will decrease or increase the importance of a word (to be used with relevance ranking). The asterisks character (∗) is a wildcard operator, but unfortunately, can only be used for prefix searching.

Boolean search mode requires the users to be educated on the use of the special operators. Because the operators are similar to those used by Google and Bing, search engines used by many people on a daily basis, most users will already be familiar with a large subset of the operators. A specialized "advanced search" page will likely be sufficient to help users construct more complex queries.

2.2.5 Advantages

Because it is part of MySQL, as long as you use MyISAM, the user does not have to install a secondary product to do the searches. Additionally the search results can be instantly retrieved from the database, there is no need to do a secondary query in the DBMS to retrieve the records with the primary keys returned by the external search product.

Another advantage is it is never necessary to update the index at a certain interval as the index is dynamic. New records are available instantly.

2.2.6 Limitations

Arguably the biggest limitation of MySQL's full text search is it is lacking the ability to do postfix queries. Only prefix and infix search queries are possible. One method to circumvent this is to add a second field to the

table and store the original search field reversed. Then for a search query requiring postfix search the second (reversed) field is used to match the reversed query. Even though this works quite well, this is far from ideal. The data is redundant as it is being stored twice and therefore it also needs to be searched twice increasing the query time considerably. There is also more room for error as every change has to be reflected to the reversed field as well. As MySQL does not support triggers the reversed field cannot be updated automatically.

Another disadvantage of using MySQL is the poor scaling. It cannot handle a lot of queries and large database sizes (mysql works with relatively small tables and short strings, beyond a certain threshold the performance drops considerably). This is due to the nature of the dynamic index. When a modify query occurs mysql would need to recalculate the weights for all terms, as this would take too much time this change is only calculated on a "local" level, the weights on a global level are more dynamic. This will degrade the speed of the queries considerably with a lot of terms. An exact explanation of the MySQL internals can be found on the MySQL Internal Algorithms page.

Additionally MySQL does not support stemming by default. Recently the MySQL team added a feature where custom parsers can be created for full text search. Some implementations already exist to add stemming. mnoGoSearch for example has a simple stemming implementation that can be found at http://www.mnogosearch.org/. However the user has to depend on third party tools and extensions without any guaranties for support.

2.2.7 Main memory properties

Index sizes for full text indexes cannot exceed 4GB when the user wants to keep it in memory, even if the available memory is sufficient. This means with indexes larger than 4GB the disk will still be hit, even though this is not necessary as it could be stored in memory.

2.2.8 Implementation

The full text index consists of a two level B-tree [9], the secondary B-tree is created with the list of IDs where the word occurs.

Boolean search benefits from this B-tree structure in several ways. Firstly because it allows for the easy addition of proximity searches due to the nature of B-trees. Secondly because the "embedded" B-tree will avoid an index scan with a query with multiple terms. For example a search query like `"+reserved +for"` where the `reserved` has lesser results than `for`. Now instead of intersecting the two lists it is more efficient to use the secondary B-tree to find the rows for `reserved` in the `for` list. The scoring algorithms within MySQL is based on term frequency and inverse document frequency

(TF/IDF) [20].

2.3 Apache Lucene

Lucene [16] is created by the Apache Software Foundation and is, unlike Sphinx, not a complete package: it is a full text search library written in Java. It has a large active development team and community.

The features set is quite similar to Sphinx; performance wise Lucene's indexer is quite slow, indexing on a Pentium M 1.5 Ghz will go at about 20 MB a minute according to the Lucene website[2]. Even though this is slow compared to Sphinx, a significant advantage of the indexing is that it is able to perform immediate propagations. This means only the initial index would take a lot of time to create, then if there are only a small number of changes or new records, the incremental updating would be more important than the slow speed. In addition to the incremental updating Lucene also has a low memory footprint and good source data to index compression ratio.

Searching the index is very quick. In some cases it is comparable or even faster than Sphinx depending on the index and query. It also has a rich feature set like proximity queries, phrase queries and range queries. Because it is written in Java it can be deployed to virtually any platform.

2.3.1 Usage

Since Lucene is a library it can be used in the following ways:

1. Use the library directly (Java interface)
2. Use a 3rd party library (available for many languages and different interfaces)
3. Use a standalone server

Using the library directly will be the most likely choice when there are specific wishes regarding the interface with the Lucene. However most people using different languages will use one of the many libraries already available. For example the PHP programming language has a PEAR module to directly use Lucene without much hassle.

The last option is to use Solr, a sub-project of Lucene, which is a standalone (web-)server with many interfaces to query the database and get results back in a particular format (e.g. XML, JSON). Solr is a servlet which has to run in a Java server like Tomcat.

[2] http://lucene.apache.org/java/docs/features.html

2.3.2 Internals

Most full text search engines use the same internal algorithm which is an inverted index. Lucene is no different, it uses an inverted index, yet it is quite different from the for example Sphinx. The biggest difference between Sphinx and Lucene is that it lacks the same feature as MySQL; its ability to do postfix and infix search queries, which is a great disadvantage.

Even though Java makes it easier to extend Lucene than C++ it would also means that it Lucene runs in the JVM and the programmer has less influence over memory usage and behavior.

Chapter 3

Goals

The goal of this thesis project was to create a full text search engine with a focus on speed, for a relatively small dataset of about 1 to 2 million records. This fits the description of sites like Mininova. For Mininova each record consists of two fields, which is at most 104 bytes. The first field is a unique ID (32 bits number) and the second field a title or description of at most 100 `ASCII` characters. The source corpus therefore would be a maximum of almost 100MB with 1 million records. Most titles however will use only part of the available 100 bytes as the following histogram shows. The histogram is of Mininova's actual database of about 1.3 million records.

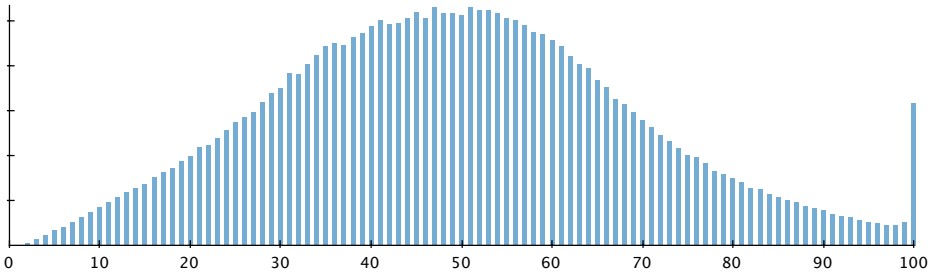

Figure 3.1: Mininova.org record lengths

Most records are around 50 characters, with a large peak at 100 characters as a number of records were truncated at insertion time.

Traditional full text search engines

With traditional full text search engines created for searching huge databases, only a part of the index resides in memory, the bulk however is stored on disk. As each read operation from disk is an order of magnitude slower than a read operation from memory, it is therefore key to limit the roundtrips to disk as much as possible [4]. Due to this huge difference in access times

optimizations for the main memory structures with regards to the cache are often overlooked. Therefore the full text search engine created for this thesis would preferably need an index structure that could fit entirely in memory to avoid roundtrips to disk. Another advantage of storing the index in memory is the ability to move large chunks of data around using pointers and references. This gives us the ability to modify otherwise "immutable" indexes and perform immediate propagation.

Which leads us to the following goals:

Speed Should be able to handle a lot of queries per second.

Mutability The index should be able to handle updates with a minimal decrease in queries per second

3.1 Problem Description

Creating such a full text search engine is no easy task, and to achieve the goals defined above there are a number of problems that have to be solved.

When the entire index is resident in main-memory retrieving something from main-memory is now our slowest operation, and likely to becomes the bottleneck [12]. The index should therefore try to utilize the cache as often as possible in order to speed up queries, and minimize memory roundtrips.

A problem that coincides with cache behavior is the usage of pointers. Having the entire index structure in main-memory allows us access every memory position, ignoring cache behavior, in constant time. Pointers will give us flexibility, but this is not always without cost. Pointers will often degrade the caching behavior. A balance has to be found between mutability and good cache behavior of the index.

3.1.1 Tuning Intersections

One of the problems we will be focussing on in this thesis is at the end of search pipeline. Once the query has been analyzed it often results in an intersection (join) of two or more of the posting lists. In order to do efficient updates all posting lists are stored as a tree or other data structure of some sort. However traversing a tree is inefficient, and will have a great impact on the overall query time. It might be better to store smaller trees as arrays, or a similar structure that is efficient to sequentially retrieve all document IDs.

Search in postings list

Another option would be to find and use some threshold from which we change from using a normal array structure to a tree. This could yield good results with queries that contain both small, often occurring, query words

with a large posting lists, like for example the word: `an`, and one or more query words that will result in a much smaller posting list. In that case, instead of having to skip thousands of IDs, we can search each ID from the small array in the large postings list. Since this large postings list is a tree we can likely minimize cache misses, but at some point the number of cache-misses will become larger than when a join is performed.

3.2 Questions

- What are limitations of an index stored in memory?

  What are bottlenecks when an index is stored in memory. How might L{1,2,3} caching influence accessing different parts of memory when randomly accessing parts of the index. To a lesser extend: how does memory behave on Linux; what part is actually available (high memory and low memory)? What could be potential pitfalls when programming.

- How can an index be efficiently stored and updated in memory?

  This question is an extension of the first. What would be an efficient structure to quickly retrieve the records from memory taking the first questions findings into account? Will certain kind of queries perform better than others, and how can we take advantage of main memory properties.

Chapter 4

Implementation

4.1 Implementation Approach

As described before there are two possible approaches to creating a full text search engine. The first is altering an existing implementation to take advantage of the in-memory indexes. The second is to start with a clean slate, and program the tool from scratch.

4.1.1 Modify an existing implementation

After reviewing the existing full text search programs, the most likely candidate to use as a starting point is Sphinx. Sphinx' current implementation comes closest to what we are trying to achieve. The fact that it is written in C++ gives us great control over how memory is used. Also due to the nature of Sphinx being a full text search engine the query parser would require very little work. However, upon closer inspection of the codebase, it became clear that the bloated codebase would require significant work before getting started. To begin with, the current codebase would have to be studied and prepared. This means getting familiar with the programming style and codebase as well as gaining a deeper understanding on how all the different parts interact. Next, unused features will have to be removed or disabled, all disk optimized compression methods need to be removed and rewritten, and all indexes will have to be modified to be entirely stored in memory. This is no small task, the worst case scenario would mean an entire rewrite of the core functionality and would not guarantee that the existing codebase will have unforeseen influences on the main memory behavior at a later stage.

4.1.2 New implementation

Another alternative would be to start programming from scratch. This means a lot of support code would have to be written to get started. Even

though starting with a new implementation would mean more work in the beginning it would give greater flexibility in the end. The entire index can be optimized for being stored in main memory. Since there are no previous design decisions that can potentially limit the index design, the possibility of running into major problems at a later stage is smaller. Still parts from existing implementations might be used where applicable, for example query parsing.

4.1.3 Conclusion

Using an existing implementation might avoid having to write some boilerplate code, however it would force us to work around certain design decisions and might not yield the optimal solution. Therefore creating an entirely new implementation would be the best choice. Writing it in C gives us the greatest flexibility on manipulating the memory and setting flags on how the cache should be used.

4.2 Architecture

In this section we will give an overview of different components working together in our implementation and the flow of the operations.

When simplifying the architecture the application consists of three main components working together.

Query parser

The first component is the query parser. This will parse the queries, sanitize the input, and transform it into a structure that can be used when querying the inverted file.

Inverted file

The next component is an inverted file. We use this to find the postings lists associated with n-grams.

The inverted file itself is implemented as a hash table. Therefore retrieving a postings list, and thus also a single term query, can be done in constant time: O(1). However the hash table is not perfect and therefore collisions are inevitable. Before returning the postings list we need to ensure we have the correct list by verifying the n-gram associated with the postings list. This results in a little overhead, however as collisions are not very common it usually results in a single comparison.

Postings list

The last components are the postings lists. The postings lists are data-structures to quickly test whether or not a document ID is present in the list and to do unions and intersections between two or more of these postings lists.

The postings list can be one of multiple data structures, like an array or a CST-Tree. These structures will be discussed below.

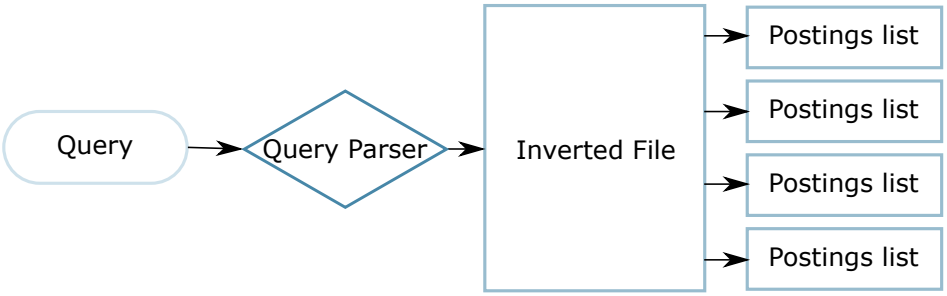

Figure 4.1: Overview of the architecture

To give a better understanding of the flow of operations we describe two use cases, indexing and performing a query, in more detail. We will start by describing all actions for indexing.

4.2.1 Indexing

The indexing operation is executed upon starting the application, however a large part of the indexing operation is also used when updating the index with new entries and removing obsolete entries. The main difference between the two is that the first time all data is loaded into the postings lists using a bulk-loading method instead of the normal insertion function. Depending on the underlying structure we can then build the postings lists in a single operation. For example with a tree structure no rotations are needed, and therefore do not have to move data around. This will save us a considerable amount of time with large postings lists. The only precondition for bulk-loading is that all records are provided in order from lowest to highest ID.

The source input data itself contains tuples which consist of an id (integer) and a title (string) or a maximum of 100 characters. When inserting we need to process this data such that it is a useful format to be inserted in our index.

1. For example when we start out with the following tuple for inserting into our index:

```
(1, "Ubuntu Desktop Live CD-amd64")
```

2. First we need to sanitize the input string. This means removing all non alphanumerical characters and replacing them with a space. In this example we only need to remove the '-'. Additionally we convert all uppercase characters to lowercase.

```
(1, "ubuntu desktop live cd amd64")
```

3. Next the string is split by a space such that we have a list of separate words.

```
(1, ["ubuntu", "desktop", "live", "cd", "amd64"])
```

4. Finally we need to generate every possible substring, the n-grams, for each of the individual words in the list.

```
(1, [["ubuntu", "buntu", ..., "ub"], ["desktop", ....], ...])
```

The reason for adding every possible substring to our index is speed. This means we can perform any prefix or postfix query in O(1) time. As memory is very cheap and the source database is relatively small we do not have to compromise and can simply put every n-gram in our index. A 6 byte word will result in 50 bytes stored in memory, that is excluding overhead of the data structure. However by just using pointers and offsets to a single string we can minimize the storage needed by quite a considerable amount.

Another consideration was whether to use some kind of ranking algorithm like inverse document frequency to score the results. As all records are titles of music, movies, and other products this has very little meaning and thus effect on the results of the query. It would be a better idea to use all metadata associated with the record to score the results and ordering. For example the number of downloads.

With a normal insertion we retrieve the postings lists for every substring using the inverted file (figure 4.1) and insert the document ID into these postings lists. The postings lists itself can be a number of data structures, from an array to some kind of tree structure. This depends on a number of factors like the number of document IDs in the postings list. More information regarding these structures can be found in section 4.3.

Generating all these substrings and inserting them one by one results in a lot of work when inserting and removing records, however it will significantly reduce the amount of work we have to do for queries. As querying will be the most common operation we want to do as little work as possible for this operation.

4.2.2 Querying

When performing a simple query we now only have to do a minimal amount of work depending on the length of the query (figure 4.2). Similar to indexing the query has to be sanitized and split into words, additionally query

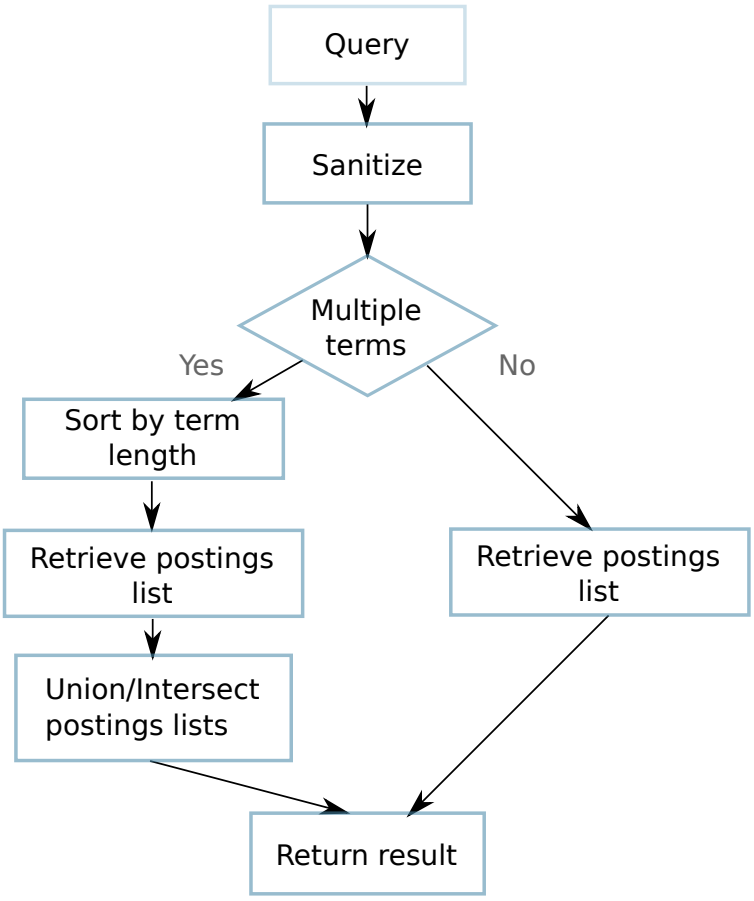

Figure 4.2: Query operation in the index

modifiers such as - and + have to be extracted and taken into consideration when performing the query.

With a single word query we simply retrieve the postings list and can instantly return the result. Depending on the query type with more than one term in the query multiple postings lists have to be retrieved, combined and returned.

Multiple terms

When the query consists of two or more terms in an "and query" we need to combine (intersect) the resulting postings lists. By sorting the terms by length we try to minimize the amount of work that has to be done. Small n-grams will be much more common than large n-grams. We found in our test corpus that this especially holds for 2-grams. Most of the 2-gram postings lists contained tens of thousands of records in our test corpus. Additionally we found, in general, the longer the substrings the smaller the postings lists.

Since we need to do an intersection of all the postings lists we start with the longer terms. This gives us in general the smaller postings lists first. If one of the n-grams returns an empty postings lists we can abort the query and return an empty result set. The same holds for an intersection as $\emptyset \cap A = \emptyset$. As soon as an intersection returns an emtpy set we can instantly return an empty result set to the user. This could potentially save a lot of work not having to intersect several lists with each more than a couple thousand document IDs.

Other query types require different strategies. For example an 'or' query will result in unions instead of intersections.

4.3 CST-Tree

4.3.1 Description

The CST-Tree [10, 15] stands for Cache Sensitive T-Tree. The T-Tree [13] is a data structure proposed as a better main memory conscious alternative for the AVL-Tree [13]. The difference with an AVL-Tree is that where an AVL-Tree only holds a single value per node, the T-Tree holds many. This is results in significantly less nodes than a normal AVL-Tree, depending on the number of values per node.

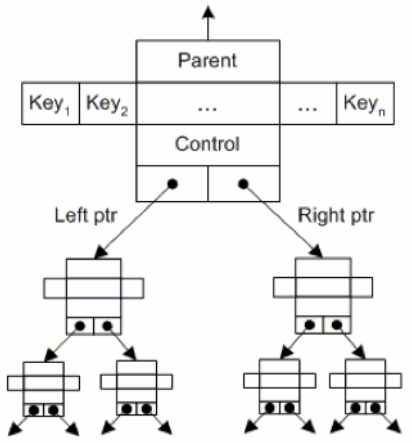

Figure 4.3: A T-Tree node

When traversing the tree both the minimum and maximum values of the node will be used to determine whether or not to go to the left or right subtree. In case the value is bound by the minimum and maximum key the node itself will be searched, resulting in less nodes to visit in a search. In turn this also leads to less rotations.

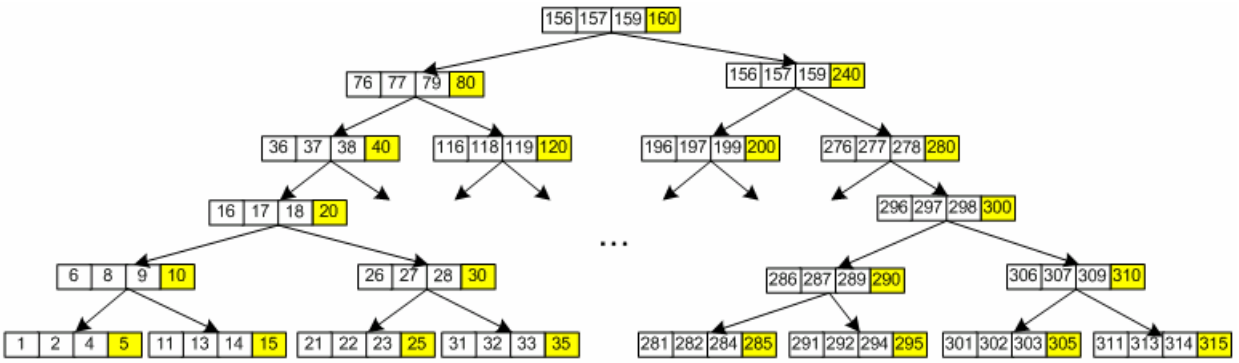

Figure 4.4: A Full T-Tree (Source [10])

Memory

In the past this would speed up the performance of the tree, as memory access time (latency) and processor clock speeds were very close. With modern processors and memory that has changed. The clock speed of processors has increased at a much higher rate than the memory latency. This means that memory access has become a lot more expensive. This has changed to such an extend that a B+-Tree, a real "disk" based data structure, has become faster than a T-Tree [15].

Cache

Therefore processor caches became more important in main memory algorithms [17]. If more data is stored in cache less data has to be retrieved from memory, which results in less cpu cycles waiting for data from memory. Processors in general have two or three layers of cache called L1, L2 and L3 cache. Each step from L1 to L3 decreases in speed, but increases in size. But still the L3 cache is an order of magnitude faster than main memory.

All processor caches consist of cache lines. These are the units in which they are retrieved from main memory and stored in cache. The size of a cache line varies by processor, but currently the most common size is 64 bytes. This means that algorithms optimized for cache lines will perform better than algorithms that only use one value from each cache line retrieved from memory. Every value outside of the cache line will result in a cache-miss and has to be retrieved from main memory.

This is the main reason the T-Tree is slower than a B+-Tree. For every step the T-Tree makes to a new subtree, a new node from which only two values will be used, has to be accessed and will likely result in a cache-miss. For a B+-Tree multiple values within a node are likely to be inspected. Additionally the height of a B+-Tree is much lower than a T-Tree thus

resulting in less cache-misses.

Structure

The CST-Tree is in many ways different from a T-Tree.

The first is the tree itself. Instead of using T-Tree nodes, and therefore storing all values in an array, a CST-Tree uses only the maximal value for each T-Tree node. Each maximal value will be stored in one large binary search tree. Looking at figure 4.4 the tree will only consist of the yellow values in the T-Tree.

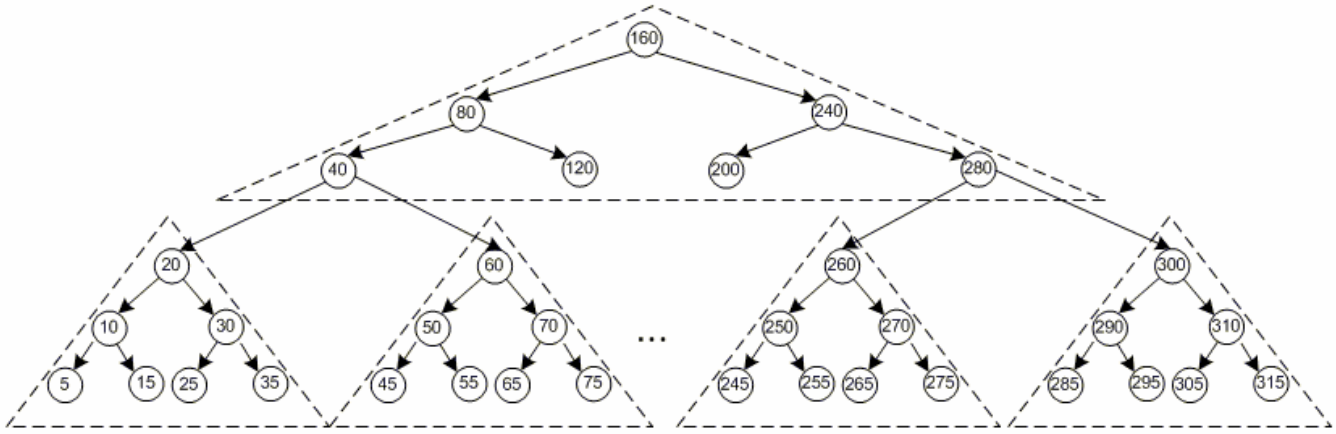

Figure 4.5: The binary tree split up to fit in a cacheline. (Source [10])

This binary search tree is split up such that each smaller part of the tree, as can be seen in figure 4.5, will fit in a cache line. To be as efficient with space as possible trees are encoded as an array, where the left branch is the array index * 2 and the right branch is the array index * 2 + 1 (figure 4.6). Note that the item with array index 0 is used to store the parent pointer. To go back to the parent the index is divided by two as an integer. As the binary tree will fit in a cache line we can descend the height of the small tree without a cache miss.

For each of these values in the maximal values trees there is an associated data node. This node holds all the other values from the corresponding T-Tree node, including the value in the array encoded binary tree. These data nodes together with the binary tree array are called a node group.

A node group will, among some other control values, consist of part of the binary tree and the associated data nodes. In order to traverse the tree it also contains pointers to the child-node groups. An example is given in figure 4.7.

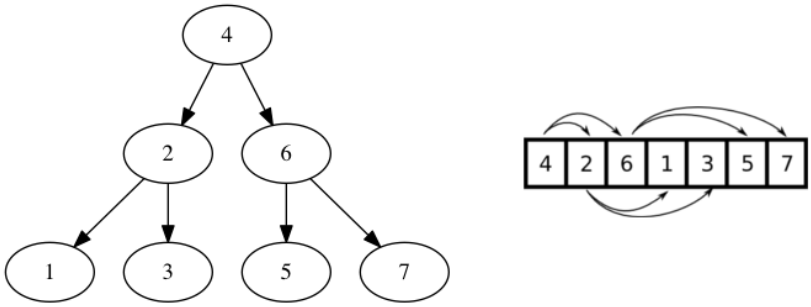

Figure 4.6: Translation from tree to array

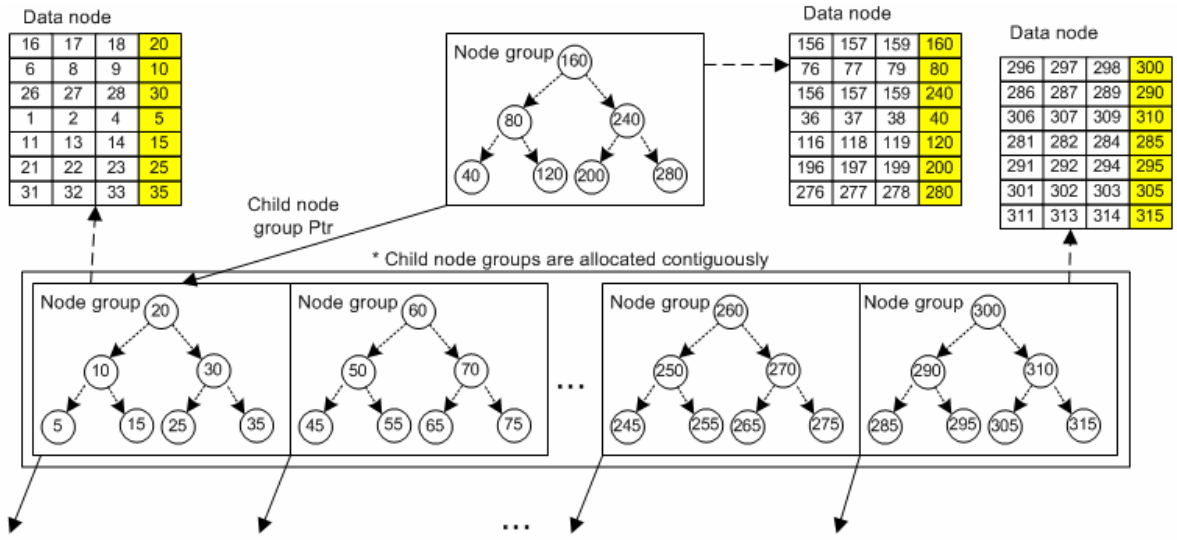

Figure 4.7: A full CST-Tree node. (Source [10])

4.3.2 Operations

Search

Starting with the root node group the internal binary search tree is descended. While doing so we keep track and mark the last move to a left subtree. This move means that the value we seek is smaller than the current node value and that therefore the value could possibly be in the data node associated with that particular node.

When the end of the node group tree is reached we check to see if there is a child node group "attached" to the branch we are trying to descend, if so, we continue with that node group. If we reach the end without a single node larger than the value we seek, the value is not present in the tree, else we search the marked data node of the last move to the left subtree. The pseudocode based on [10] for the CST-Tree search operation can be found

in listing 4.1.

Listing 4.1: Searching a CST-Tree

```
CST_Search( key, tree)
  //key: a key to find, tree: a CST-tree
  //RID: a record ID to be found compareKey = get the
    first key to compare in the root node group of tree;
  // 1st step: traverse node groups
  while ( compareKey != NULL ) {
    if ( key <= compareKey ) {
      lastMarkedNode = data node corresponding
                       to current key;

      compareKey = get the key of left sub-tree;
    }
    else
      compareKey = get the key of right sub-tree;
  }
  // 2nd step: binary search in a data node
  if ( lastMarkedNode != NULL) {
    dataNode = get the data node from lastMarkedNode;
    RID = binary search in the dataNode;
    return RID;
  }
  else
    return NOTFOUND;
```

For example trying to find the value 287 in the tree in figure 4.5 would mean that first we need to determine the binding node (figure 4.8). We start with 160 until we reach the end of the node group. We then reach the node with value 300. As the value is smaller than 300, and can therefore reside in the data node associated with node 300, it is marked. We then traverse to the left once more as the value 287 again is smaller than 290. Our last marked node is replaced with 290. When we reach node 285 the only possible position 287 can reside in is the data node associated with 290. Next the identified data node is searched for the value 287.

Insertion

An insertion starts similar to a search. We start by finding a node that binds the value we want to insert. Meaning that the smallest value of the data node is smaller than, and the largest value of the data node is larger than, the value we want to insert. If the binding data node has a free space left to store the value it is added to the data node. If not the smallest

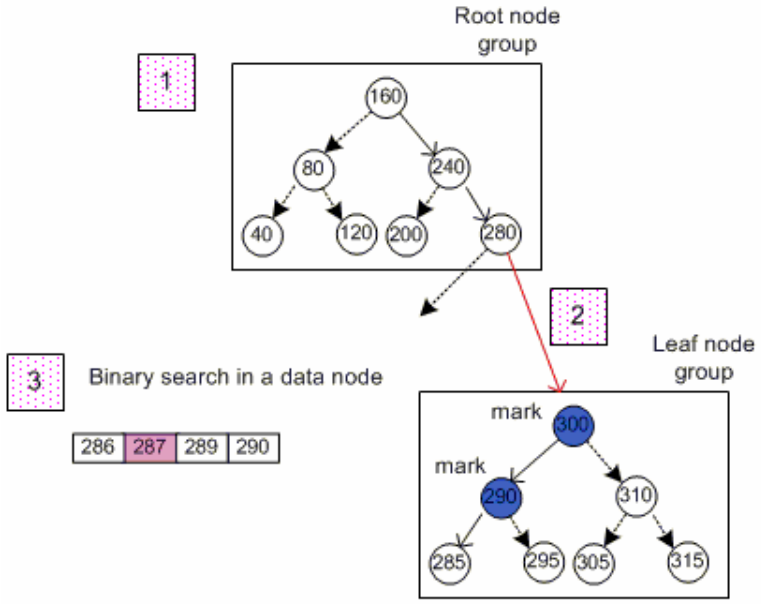

Figure 4.8: Searching for value 287 in a CST-Tree

value overflows the data node and is removed and inserted in the most left subtree. This can cause another overflow which makes repeating this process necessary.

When no binding node is found the value is inserted in the last node that was reached while finding the binding node, the binary search tree is also updated with the new value. When necessary a new node group is constructed. By avoiding this a much as possible the tree is kept as small and efficient as possible.

Finally the binary search tree is balanced, or in case of a new node group the node groups are balanced. A detailed description of this operation can be found in [10].

For example using the tree defined in figure 4.5 and 4.4 we can insert the value 288 (figure 4.9). Again we traverse the tree till we reach the binding node 290. We can then insert the value in the data node associated with node 290. Inserting 288 will overflow the node, leaving node 286. This node will be put in the left sub-tree, however this will cause another overflow, leaving node 281. As there is no left sub-tree a new child node group will be created holding only node 281.

Traversal

Another important operation that will be used a lot is in-order traversal of the tree. This is needed when merging and intersecting two lists.

We need to be able to get the values from the CST-Tree one by one as

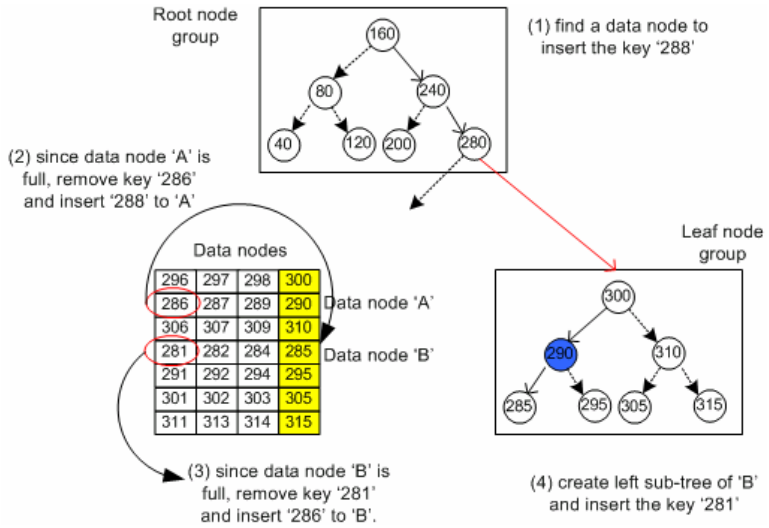

Figure 4.9: Inserting value 288 in a CST-Tree

we might not need every value when intersecting. Getting each value would then waste valuable processor time and memory. We need to traverse the tree using an iterator and therefore we cannot simply use recursion to get all values in a single function call.

We use the iterator data structure to keep state. Within this data structure we keep track of the current node group, the position within that node group we are traversing, and the direction we are moving. We start by initializing the iterator with the smallest value and the exact node containing that value.

Once a value is requested from the iterator, the data node belonging to the smallest binary tree value is emptied. Once all values from a single data node are returned we check what the last motion was, and move accordingly to the pseudocode in listing 4.2.

Listing 4.2: Traversing a CST-Tree using an iterator

```
// LeftUp: Moved from the left branch to the parent
// RightUp: Moved from the right branch to the parent

initializeIterator()
  lastMotion = LeftUp
  state = Emptying

  currentNode = leftUntilNoFurther(rootNode)

nextValue()
  // "Empty" all values from the data node
```

```
if state == Emptying && values left in data node:
  return nextValue from datanode
else:
      state = None

if lastMotion == LeftUp:
  if exists right branch:
    currentNode = right(currentNode)
    currentNode = leftUntilNoFurther(currentNode)
  else:
    lastMotion = RightUp
    return nextValue()

  state = Emptying
  return nextValue()

if lastMotion == RightUp:
  while motion(parent(currentNode)) == RightUp
        && parent(currentNode) != rootNode:
        currentNode = parent(currentNode)

  if parent(currentNode) == rootNode:
    return NULL
  else:
    // Go to the parent one more time from the left.
    // The right node is the next to be inspected.
    currentNode = parent(currentNode)
    lastMotion = LeftUp
    state = Emptying

    return nextValue()
```

Cache Behavior and Modifications

A number of the CST-Trees optimizations to increase cache performance involve allocating large blocks of memory that will be later used to store more information. By doing so we can avoid using pointers to different blocks (to for example nodes), which means an extra cache miss. Instead offsets are used to calculate the next piece of memory we need to access.

The problem with this method quickly becomes apparent when using tens of thousands of CST-Trees. All this unused memory quickly adds up and keeps you from storing valuable data.

One of these places is the storage of the child node group pointers. To

avoid having to allocate all child node groups as a single block we use an array of pointers that point to the individual node groups.

Speed

The changes specified above will obviously have a negative effect on the speed of the cst-tree. This slowdown of the cst-tree is mainly due to the additional cache misses of the pointers added to keep the tree flexible and to minimize the memory used by the blocks allocated in advance. We trade off speed for flexibility and memory space.

4.3.3 Prefetching

Even though modern compilers are able to optimize a great deal they can only go so far with a static analysis. Compilers can use very specific processor information to rewrite and reorder instructions to use the available processor resources as best as possible. However in certain situations the programmer knows exactly what data will be needed in the near future. Having his data available in the cache at the appropriate time can save a considerable amount processor idle time waiting for the data to be retrieved from the main memory.

Software Prefetching

Most processors have specific instructions that can be used by the programmer to help the processor with caching. This is called a prefetching instruction. A prefetching instruction will request the specified memory location from the memory but does not wait for the data to arrive in cache, thus not blocking the execution of the program. The data requested is then loaded in, most often, the L2 or L3 cache memory. If the request is made enough in advance the data will be available by the time it is needed or, at least, the time waiting for the data will be considerably less than if not prefetched. With algorithms that follow a specific pattern this can save a considerable amount of time.

Even though prefetching is relatively easy to use it should not be used without careful consideration. Even a trivial prefetching "fix" can interfere with the compiler optimizations or the processor caching algorithms, and off-balancing caching behavior to such an extend that it will negatively impact performance.

Hardware Prefetching

Besides the ability for the programmer and the compiler to give hints to the processor about what data should reside in cache at a specific moment in time the processor will also try to prefetch relevant pieces of data to the

cache. Depending on the processor different kinds of algorithms are used. For example one of the algorithms used is based on the assumption that often data is read sequentially, therefore resulting in not only putting the data required in the cache, but also the memory location directly following the one requested. Other prefetching and caching algorithms analyze the data access patterns and try to predict the next memory location that will be needed in a computation. The exact algorithms used depend on the processor architecture and the compiler.

GCC

The GNU Compiler Collection (GCC), which is the compiler used for this project, has a single built in function to tell the processor what memory location should be put in the cache [5]:

```
__builtin_prefetch(address, rw, locality);
```

address The address of the memory location to be prefetched.

rw Will the memory location be used as "read only" or "read and write"

locality This is a value from 0 to 3 defining how long the memory location should be kept in the cache.

Using the parameters of the internal prefetch function we have some control over how the data should be stored in the cache. The use of the locality parameter enables us to keep the data in cache for as short a time as possible, therefore having more space available for other data. For example we can specify that the data will only be used once (locality set to 0), or whether or not it has a low/moderate temporal locality (locality set to 1 or 2), or if the processor should try to keep the data in memory for as long as possible (locality set to 3).

Prefetching in a CST-Tree

Due to our changes to the CST-Tree the child node group locations are not calculated using an offset to the child node pointer we are currently using, but are stored as a separate pointer to a new memory location. This gives us more flexibility and is more memory efficient since we only allocate each leaf node when they are needed instead of an entire level of leaf nodes (which are child node groups) as we might only use a single memory location from the entire block (figure 4.10). Downside of this change is that we lose some of the caching advantages as we will have to retrieve the data nodes separately, using a pointer, and cannot just calculate the child node group memory location using the tree location.

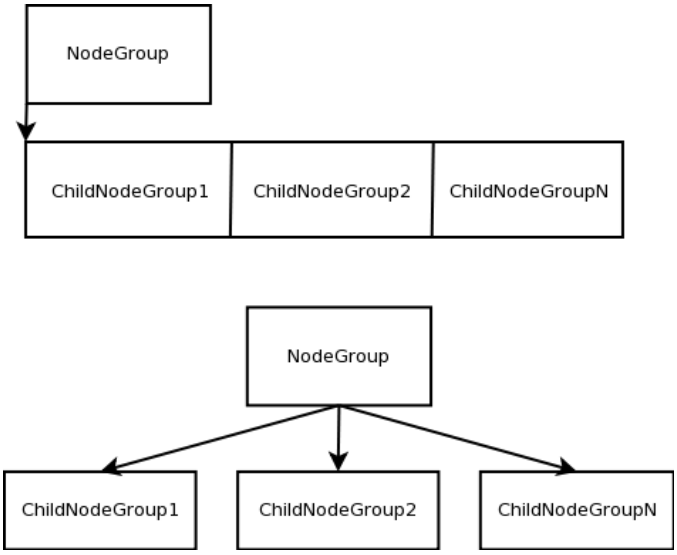

Figure 4.10: Changes to the CST-Tree

We tried minimizing this disadvantage by using prefetching to ensure that all the memory locations are already in memory at, or as close as possible at, the time we need to retrieve the next node.

Approaches

Using Cachegrind [21], and keeping the changes we made in mind, we identified two points in the CST-Tree where it should be possible to benefit from prefetching.

Data nodes

The first of which is the prefetching of the data node corresponding to a node from the internal binary tree. By prefetching the data array each time we traverse to the left subtree we could potentially save time when we reach the end of the binary tree and need to inspect the data node associated to the last left traversal. Each left traversal means the value we are looking for is smaller than the binary tree node and therefore could resides in the data node associated with this binary tree node.

Listing 4.3: Prefetching the data node

```
CST_Search( key, tree)
  //key: a key to find, tree: a CST-tree
  //RID: a record ID to be found compareKey = get the
    first key to compare in the root node group of tree;
```

```
while ( compareKey != NULL ) {
  if ( key <= compareKey ) {
    lastMarkedNode = data node corresponding
                     to current key;

    prefetch lastMarkedNode

    compareKey = get the key of left sub-tree;
  }
  else
    compareKey = get the key of right sub-tree;
}

if ( lastMarkedNode != NULL) {
  dataNode = get the data node from lastMarkedNode;
  RID = binary search in the dataNode;
  return RID;
}
else
  return NOTFOUND;
```

By prefetching `lastMarkedNode` we should save time transferring the data from memory to cache. By the time the system is ready to do a binary search in the `dataNode` the data should be on its way to the cache, or in the best case scenario already available in cache.

After decompiling the modified program to assembly we verified that the change resulted in a single `prefetcht0` instruction to be added to the program.

Unfortunately benchmarking the modified code resulted in a slowdown of the `CST_Search` function. It is hard to determine the exact cause for this slowdown, even with all the existing benchmarking and analysis tools, as we only added a single instruction which should not block the execution of our program.

Programs like Cachegrind give you a good impression of where memory access takes a long time in your code, but is does not accurately specify the line causing the slowdown, and often only gives you an indication of where to look.

In this case it is clear what line causes the slowdown; our added prefetching instruction, yet it is hard to give a definitive answer why this is the case. One scenario is that the hardware prefetcher already ensures that the data is in cache by the time we need it. This would result in our prefetching instruction only being overhead instead of an improvement, or possibly even disrupting the caching behavior causing an even bigger slowdown. The second scenario is that the data is not prefetched enough in advance and the

small overhead of the prefetch instruction causes the slowdown.

Child node groups

The second possibility where we might benefit from prefetching is the moment right after traversing to a new node group. When `compareKey` points to a new node group we prefetch all possible child node groups. As we always traverse to the bottom of the tree at least one of the child node groups will be needed. As the most common cacheline size currently is 64 bytes our tree will at most have 16 different child node groups. This will be able to fit in most caches with ease.

While we are processing the current node group the child node groups will be fetched, without blocking, from memory to cache. Once we reach the end of the node group we instantly have access to the next child node group. Then we repeat this step, prefetching the child node groups belonging to the node group.

This change yielded the same result as first prefetching implementation. It had a small negative impact on performance. We verified that only a single prefetching instruction was added and that none of the other instructions were executed in a different order in the compiled version.

Conclusion

We learned that manually influencing caching behavior is a very delicate process that can easily do more harm than good. Unless developing for a very specific architecture it will be difficult not to interfere with existing caching algorithms from both the compiler and processor.

4.4 Bitmap index and B+-Tree

Beside a CST-Tree we also implemented and tested several other data structures like the B+-Tree and a bitmap index.

4.4.1 Bitmap index

Bitmap indexes perform very well when performing binary operators like union and intersection as with each processor instruction 32 or 64 (the number of bits processor) records can be combined. Because of this property they are used in a number of DBMSs. However one of the downsides of a bitmap index is that each new record potentially adds another bit to every single bit array.

Sparse posting lists, which are common with longer words, would then use up a lot of unnecessary space. Take for example a posting list containing only one ID pointing to the 1 millionth record. Which would result in 1

million bits to be stored for that single list without compression. First 999.999 zeros followed by a single 1. Our tool uses hundreds of thousands of posting lists where the maximum record ID is currently 13.5 million. This would result in gigabytes of wasted memory.

As there are only 2.5 million records in the database a mapping can be made from record ID to the position in the bitmap index. However this would result in added complexity and considerably more memory usage compared to when a database is used with less deleted records yet the same size.

An alternative to avoid the bad space complexity is a compressed bitmap index. The bit arrays are decompressed and subsequently compressed whenever they are needed. In certain cases, depending on the implementation, only part or nothing has to be decompressed. For example simply testing whether a value is present in a posting list. Only the position, or block containing the position, has to be decompressed to return a result. Compression and decompression however does not come without cost.

lemurbitmapindex

For preliminary tests to determine whether or not a bitmap index was a viable option for our tool we used the lemurbitmapindex [3]. During these tests it became apparent that compressing and decompressing results in a lot of overhead. The preliminary tests were considerably worse compared to the CST-Tree and other structures that we abandoned the bitmap index.

4.4.2 B+-Tree

The B+-Tree [2] is widely used for many different applications including many DBMSs. Inside a B+-Tree all records are stored in the leaf nodes. All parent nodes therefore consist only of keys also present in the leaf nodes. In posting lists the key and value will be the same. Additionally all leaf nodes are linked using a pointer (figure 4.11). This makes B+-Trees ideal for performing range queries. Once the minimum value has been found the pointers can be followed till the maximum value has been reached.

When benchmarking the B+-Tree against the CST-Tree the results were as expected. The B+-Tree could process less queries per second than the CST-Tree as can be seen in figure 4.12. Inserting on the other hand is quicker with a B+-Tree as the value only has to be appended and possibly propagated upwards to the root, whereas the CST-Tree might have to perform rotations when overflows occur.

By using a node size that which fits inside a cacheline we tried to optimize for good cache behavior. This helps because (as described in section 4.3.1) data from a single node, which fits in its entirety in a single cacheline, might all be used when traversing down the tree.

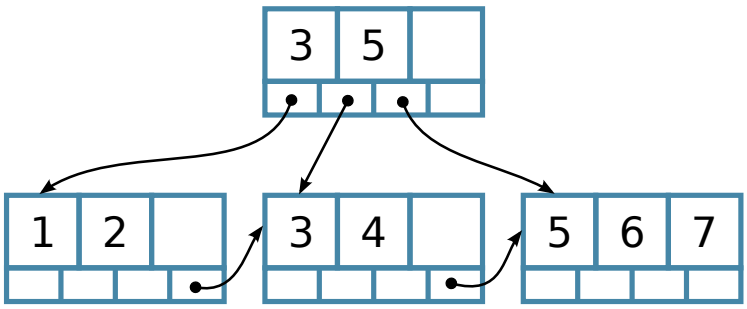

Figure 4.11: B+-Tree data structure

CSB+-Tree

Same as with the CST-Tree the B+-Tree also has a cache sensitive B+-Tree counterpart. The CSB+-Tree was introduced in 2000 by Roa et al. [19]. Unfortunately due to time constraints in combination with the complexity of the CSB+-Tree were unable to take this data structure into consideration. However several papers [10,12] show that a CST-Tree is considerably quicker in a variety of benchmarks.

4.5 Array or CST-Tree

Both an array and a CST-Tree have different behavior depending on the situation they are used in. For example there is a big difference in performance when incremental insertion IDs are used opposed to random values. In traditional web based applications data is stored in some kind of SQL database where new records are assigned an incremental primary key. With these kinds of traditional web applications the scenario of random insertion IDs is not very likely. However other types of applications might use some kind of hash to generate a record id. For example when the hash of an image is used as a primary key and a short description of the image needs to be searched. Both scenarios might require a different technique to achieve optimal results.

Another aspect that influences the efficiency of the data structures is whether or not the data structure will be used for a union operation or intersection when combining multiple postings lists.

4.5.1 Insertion

As described above especially insertion greatly depends on the situation the store is being used in.

For example adding incremental values to a sorted array is trivial. Simply append it to the end of the array and double the size when it is full and

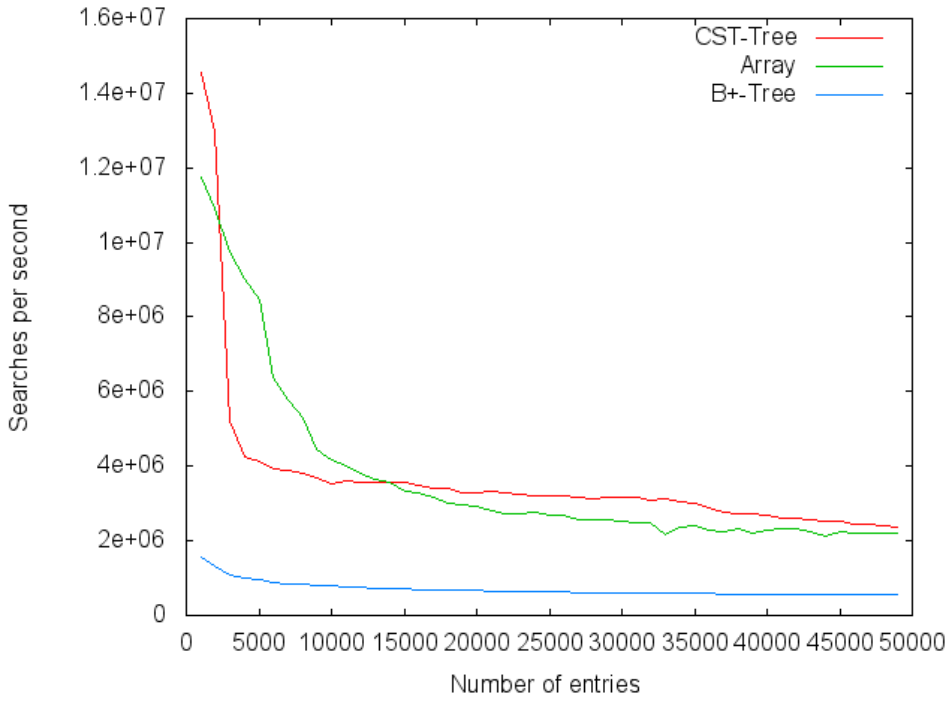

Figure 4.12: Benchmarking the B+-Tree

more space is needed. Adding these values to some kind of tree structure will require more work as rotations and reordering of values is needed in order to keep the tree balanced.

However when new values are added in a random order the tables are turned. When using an array every insertion value smaller than the maximum value in the array will require a lot of work reordering. All values larger than the insertion value have to be moved one position in memory. In this situation the array becomes unusable when the number of values exceed a certain threshold.

4.5.2 Search

While searching the global state, meaning the number of data structures and the number of entries per datastructure, of the application has quite some influence on the results. Overall the CST-Tree is the better choice due to the better cache behavior, however in some cases the array has an advantage.

When the overall number of entries in all the data structures is still low the array has a slight advantage over the CST-Tree at certain stages. During this period it is likely that many of the array entries are still in cache, albeit not very efficiently, but the overhead of traversing the trees will cause the

CST-Tree to be slower.

For example given we have an array with 1200 values we start performing a binary search. First we retrieve the center value at position 600 in the array. Next we subsequently retrieve the value at position 300 or 900 depending on whether the value is smaller or larger than 600. This binary search will result in a, for the most part, filled cache where each cacheline will contain one of the values that was checked. As long as the cache is not full these values will be kept in cache effectively making the next binary search in this array very fast.

When the threshold is exceeded that these values will no longer fit in cache the CST-Trees cache conscious design will have the upper hand. This can be clearly seen in the benchmarks in figures 4.13 and 4.14.

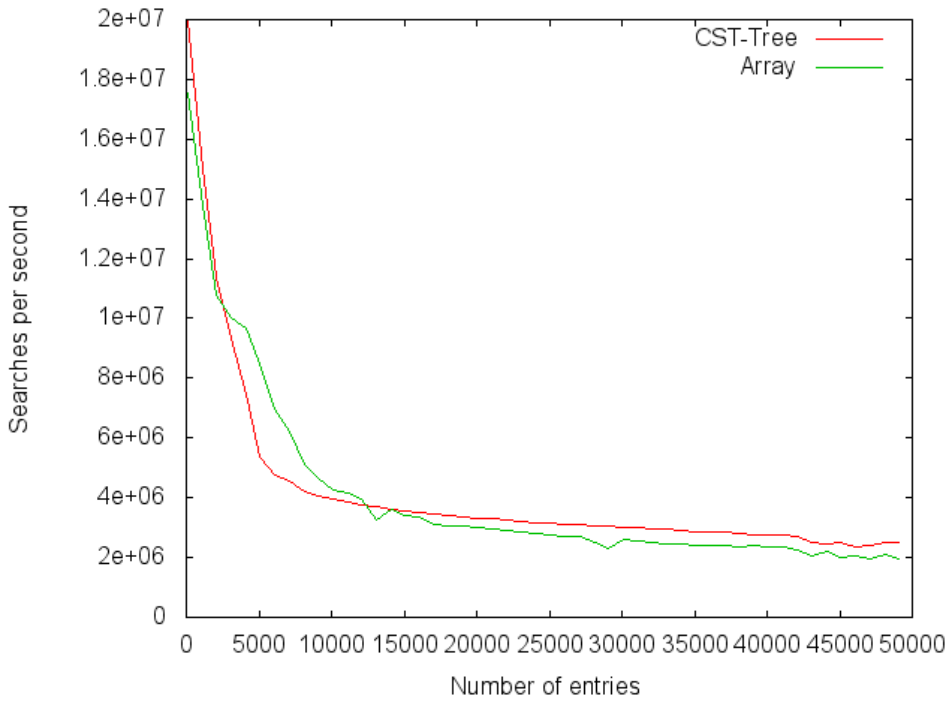

Figure 4.13: Number of searches per second with 300 data structures in memory and 1 to 50.000 entries in each data structure

During the benchmark with 300 data structures (4.13) the CST-Tree starts to do more queries per second at about 13.000 entries resulting in 3.900.000 total entries. During the benchmark with 1000 data structures (4.14) this turning point happens at 3700 entries per data structure. In total this is 3.700.000 entries giving us roughly the same amount of entries that will fit in cache as the benchmark with 300 data structures.

With the benchmarks in figures 4.13 and 4.14 the data structures con-

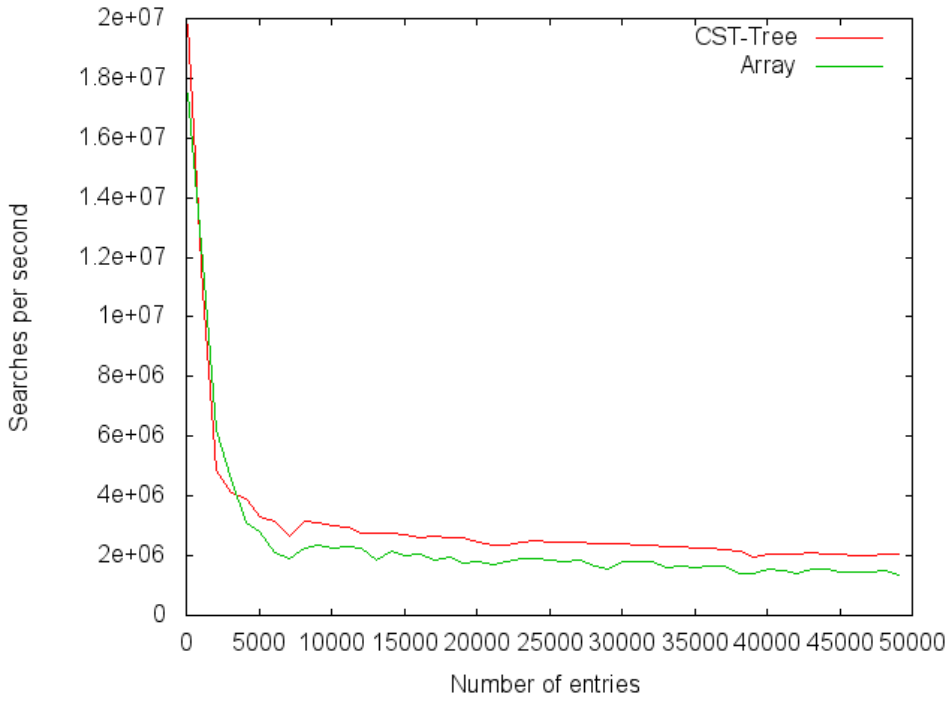

Figure 4.14: Number of searches per second with 1000 data structures in memory and 1 to 50.000 entries in each data structure

tained almost every value, therefore the "misses" are quite rare. As soon as we run the benchmarks again with a data structures that contain the same number of values, yet the values themselves are a subset of a substantially larger set we get slightly different, but negligible, results.

4.6 Union and intersection

Once all the posting lists for the different query words have been retrieved they have to be combined depending on the kind of query that is performed. The most common operation will be the intersection of posting lists as an `AND` query will be more interesting than a `OR` query which requires a union.

When multiple terms are entered the user will expect to see records that contains all query terms. This means the default operation will be an intersection. When an `OR` query needs to be performed the user will have to explicitly use this keyword in the query.

Combining these posting lists can be done using different strategies. We define two different strategies:

1. Merging two lists by comparing each value

2. For each value in a list search in the other list

Whichever strategy to use depends on a number of factors. The number of items in the data structure plays an important role in this decision, but also the type of the data structure.

Traversing an array one by one is a lot faster than traversing a CST-Tree. Not only is there more work involved with traversing the CST-Tree because positions have to be calculated, but it also leads to more random access in memory. This can cause cache misses, which in turn could lead to cachelines that will have to be retrieved from memory perhaps even multiple times. A small advantage is that the CST-Tree is wide and not very deep. This means we do not have as many cache misses compared to for example a binary tree as less parent nodes have to be kept in cache. Obviously larger cachelines will result in less nodes which also helps minimize cache misses.

An array on the other hand will have hardly any cache misses as everything is read in cache sequentially and the processor cache optimization algorithms will likely prefetch the next needed cacheline in parallel with the first.

4.6.1 Intersection

When intersecting the posting lists for a single query the maximum number of document IDs that can be returned is the length of the smallest posting list. We therefore try to get the result list as small as possible as quickly as possible. So starting with the smallest list we ensure that the intermediate result is never larger than the smallest list. Note that this might not always be the optimal solution.

The next step is to choose either of the two strategies to intersect the two lists. When intersecting the posting lists an intermediary posting list has to be kept with the result of the query. This intermediate result is always an array because of the fast insertion and inspection of the values.

Search in list

The most obvious choice for searching for the values of the first or intermediate posting list is when there is a large difference in size between the two lists. By searching for the first lists values in the second list we avoid having to load the entire second list in memory, and only a small part as we want to inspect the least amount of memory as possible. A threshold has to be determined when this lists differ enough in size and what the minimum and maximum size can be.

Merging

In cases where both lists are of equal length or where the querying of each value in the second list will likely result in more memory to be inspected the merge strategy will be picked.

Whether the most efficient strategy was picked depends a huge deal on the corpus and the query performed. There are cases where the search in array strategy can still be faster than merging. For example given the following two posting lists:

List A: `[1, 2....500, 1000]`

List B: `[500, 600...999]`

In this case both posting lists are of a similar length with only one record ID that is present in both posting lists. Here we have to retrieve both complete posting lists from memory to cache memory in order to merge.

When using the search in list strategy the same "route" down the CST-Tree will be followed over and over. When these couple nodes are kept in cache this could save many cycles waiting for a new cacheline to be retrieved from memory. However the processor cache algorithms could (positively) influence merge speed as well. Once the pattern is detected that cachelines are retrieved sequentially from memory the processor could start prefetching relevant cachelines before they are needed.

4.6.2 Union

Performing a union operation on two posting lists is similar to intersection, however the result set could be as long as the sum of the length all posting lists. To avoid having to copy as much data we still, like with the intersection, try to keep the list as small as possible till the end. We therefore start with merging the the smaller lists first.

For example if we have 2 posting lists of only a single value and one list of 100 values of all unique values as follows:

$$\begin{aligned} \text{L1} &= [1] \\ \text{L2} &= [2] \\ \text{L3} &= [3..103] \end{aligned}$$

$$I1 = L1 \cup L2 \tag{4.1}$$

$$R1 = I \cup L3 \tag{4.2}$$

$$I2 = L3 \cup L2 \tag{4.3}$$

$$R2 = I \cup L1 \tag{4.4}$$

Using the smallest list first strategy, we need to copy in total 104 values. For the intermediate result set $I1$ (4.1) we will copy 2 values, and for the

result set $R1$ (4.2) another 102 values will be copied. When starting with the biggest list first we will copy $100 + 1$ for the intermediate result set $I2$ (4.3) and $101 + 1$ for $R2$ (4.4). We therefore copy 203 values.

Currently we will always traverse the data structure when performing a union operation as each value needs to be added the result set. Therefore traversing is the preferred strategy.

4.6.3 Merge speed and traversal

Inspecting each document ID from a CST-Tree or array in order is an operation that is performed often when intersecting or performing a union of two or more lists. Inspecting every value of an array can be done very quickly as described in 4.6.1 due to prefetching. That prefetching will help performance is very clearly visible in graph 4.15. The number of values read will increase as more values can be prefetched from memory into cache in a single transfer.

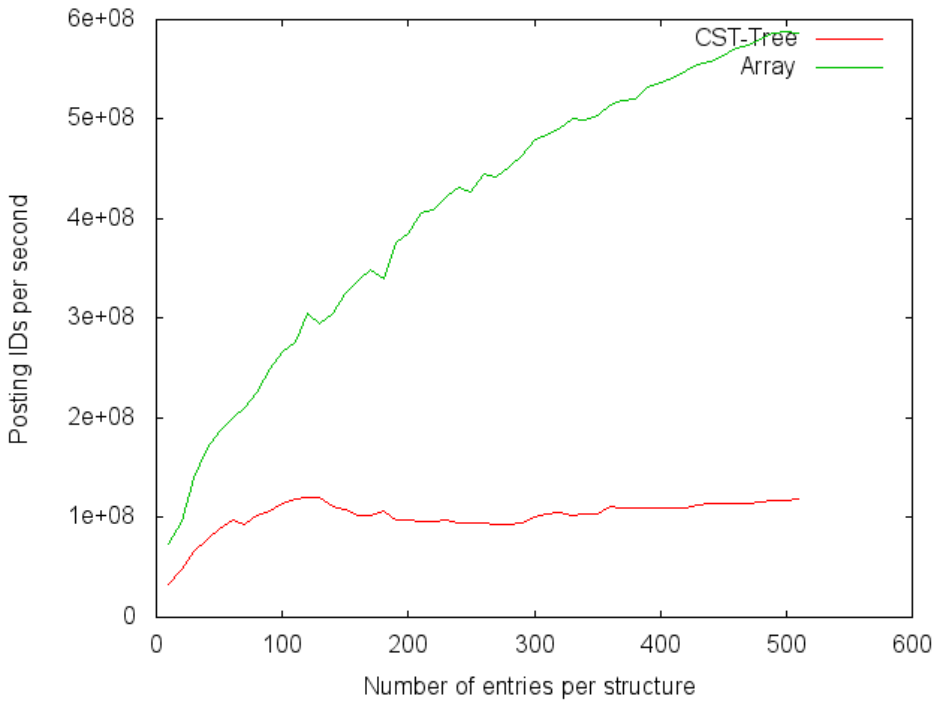

Figure 4.15: Number of IDs per second traversing 1000 data structures in memory and 10 to 500 entries in each data structure

In our experiments we found that inspecting each value in order in an array is about 4.5 to 8 times faster than traversing a CST-Tree depending on the size of the data structures. This is a big difference as the number of values we can inspect per second is very large. For the array we recorded

speeds of about 700 million values per second compared to 155 million values per second with a CST-Tree, as can be seen in the graph in figure 4.16

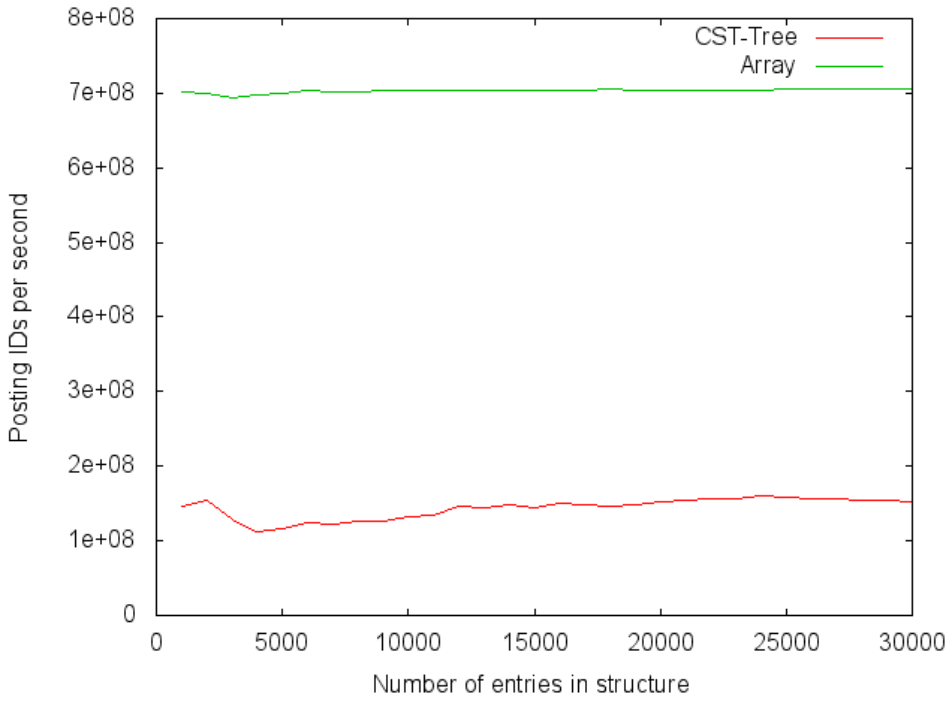

Figure 4.16: Number of IDs per second traversing 1000 data structures in memory and 1.000 to 30.000 entries in each data structure

This is such a substantial amount of document IDs that we can inspect per second that is often the preferred method to use when intersecting and merging posting lists.

4.7 Delta index

We can conclude that, because getting each value from the data structure in order is the predominant operation, the number of values that can be retrieved and intersected or unified per second plays an important part in the performance of the tool.

The immutable array outperforms the the tree structures by up to 8 times. The array would be the preferred data structure if it where not for the extremely poor update behavior. In this case the CST-Tree is the preferred data structure.

To get optimal solution we use both data structures. The array will be the immutable main data structure storing the bulk of the data while we keep the CST-Tree as a delta index to record all changes. Using a delta index adds another layer of complexity, however the gains in overall performance outweigh the added complexity.

In the following subsections we will discuss how different actions are performed.

4.7.1 Searching

Since the search operation is performed for a single value we can simply search both the array structure and the CST-Tree and return whether or not the value is found in either of the lists. If the value is found in the first structure the second structure can be ignored (listing 4.4).

Listing 4.4: Searching the delta array

```
searchDelta(value) {
  return inArray(value) || inCST-Tree(value)
}
```

4.7.2 Values in order

Retrieving each value in order from both structures simultaneously will require a union of both lists. Both structures are traversed using an iterator. The extra layer on top of the two structures adds a third iterator structure which internally keeps track of the iterators for both the array and the CST-Tree.

Once the next value is required the values from both structures are inspected and the smallest is returned (listing 4.5).

Listing 4.5: Delta index iterator implementation

```
nextValue(iterator) {
 smallestValue = 0;
```

```
 if (iterator.cst-treeValue != 0 &&
   (iterator.arrayValue != 0 ||
   iterator.cst-treeValue < iterator.arrayValue))
 {
   smallestValue = iterator.cst-treeValue;
   iterator.cst-treeValue = iterator.cst-tree.nextValue();
 }
 else if (iterator.arrayValue != 0 &&
   (iterator.cst-treeValue != 0 ||
   iterator.arrayValue < iterator.cst-treeValue))
 {
   smallestValue = iterator.arrayValue;
   iterator.arrayValue = iterator.array.nextValue();
 }

 return smallestValue;
}
```

4.7.3 Insertions and deletions

Every new ID is added to the delta structure, a CST-Tree. Once the delta index reaches a predetermined size the main data structure is rebuild taking the changes inside the delta structure into account. The delta index is subsequently reset to empty. One exception is made. When the value is greater than the largest document ID in the array structure the value is appended to the array. This allows for easy addition of incremental document IDs and helps with bulk loading.

Additionally this allows us to use the optimal structure, the array, when there are only incremental inserts. Random insert IDs will be inserted into the CST-Tree and will therefore only have a small impact on the overall performance of the index.

Deletion depends on which structure the ID resides in. When the ID is stored in the delta CST-Tree it is simply removed from the tree, and the tree is rebalanced if necessary. When the ID resides in the array structure tombstones are used to identify removed document IDs.

4.7.4 Rebuild strategy

There are a number of strategies possibles for rebuilding the structure and incorporating the values of the delta index inside the static index.

1. Block everything while rebuilding the postings list

2. Lock only the postings list being rebuild and continue other queries (threaded)

3. Rebuild the postings list in another thread and replace the list when done (threaded)

4.7.5 Blocking rebuild

The first strategy, being the least complex of the three, is very likely going to slow down requests. Even queries that do not require the postings list that is being rebuild will block.

The other two will require threads that ensure non-blocking, or partly blocking behavior.

4.7.6 Lock postings list

The second strategy requires a locking mechanism that locks the index that is currently being rebuild. By locking just the single structure all queries that do not require that particular structure will be able to continue. Rebuilding a postings list for an n-gram that is not used often will therefore not interfere as long as the postings list that is being rebuild is not necessary for any operation. However rebuilding a postings list that is used often will have the same result as the first blocking rebuild strategy.

This strategy can, in the worst case scenario, perform worse than the blocking rebuild due to the added complexity and locking overhead.

4.7.7 Rebuilding in background

The third strategy tries to minimize the time the single structure that is being rebuild is locked. By rebuilding the structure in the background and only locking the old structure when it is replaced with the new, rebuilt, structure the locking time is kept to a minimum.

When a new document ID has to be inserted into a postings list a small check is performed to see whether or not the structure is currently being rebuild. If this is not the case a flag is set to ensure that new inserts are added to the new structures CST-Tree. When it is not being rebuild and the threshold is exceeded the rebuild flag is set and a background thread is given the task to rebuild the index. Inserts into the new CST-Tree while rebuilding cannot exceed the threshold.

Because new IDs are inserted into the new structure while the old structure is still being used there is a possibility that during the rebuilding of the structure new IDs are added to the new structure and therefore not yet present in the result set.

It depends on the application if this is acceptable or not. For web applications where speed is more important than correctness this would be an

acceptable strategy. For example with Mininova a missing search result for a number of milliseconds is acceptable. Having the user wait for an entire second for the search result can result in a huge decrease of users [6].

Benchmark

From the three strategies described above we implemented the first strategy of a blocking rebuild. Even with this strategy the structure is considerably faster than just a CST-Tree due to the speed gains from traversal. In our benchmarks we kept a search (consisting of an intersection and a traversal) and insert ratio of 98 to 2. This is consistent with the insert to search ratio on Mininova.

An insert of a new record will result in a lot of document ID inserts. Given 8 million searches per day, it would result in 160,000 inserts.

Given the database of Mininova we determined each new record on average consists of 8.025306 words, the words themselves are on average 4.79 characters long. For a word of 4.79 characters we will need to insert our document ID into 9.10 different posting lists. This results in 73.035 new IDs to be inserted into our index for each new record. Taking the 160,000 inserts that would be equivalent to 2190 new records, which is consistent with Mininova and therefore would be a realistic ratio for our benchmark.

The two benchmarks conducted once with a 1 percent insert rate (figure 4.17) and once a 2 percent insert rate (figure 4.18) show that the increased insert rate only has a minimal effect on the overall performance of the application. The array structure was not included in these benchmarks due to the poor insert performance.

Deletion

Deletions can pose a problem while rebuilding the index. There are two scenarios possible with the threaded strategies.

1. document ID resides in the new CST-Tree.

   The to be removed ID resides in the new CST-Tree as it was added while rebuilding the index. In this case the ID can simply be removed from the CST-Tree without side effects.

2. document ID resides in the array structure.

   The ID is present in the array. The array cannot be altered while it is being rebuild, therefore a list has to be kept for IDs that are to be removed from the array structure after rebuilding from the old structure is complete and before it replaces the old index.

Deletions on the other hand can cause false positives as document IDs are still present in the old structure. As all document IDs will have to be

Figure 4.17: Benchmark of 1% insertions and 99% intersection and traversal of posting lists

retrieved from the main database the deleted IDs will simply not return a record therefore not resulting in any false positives.

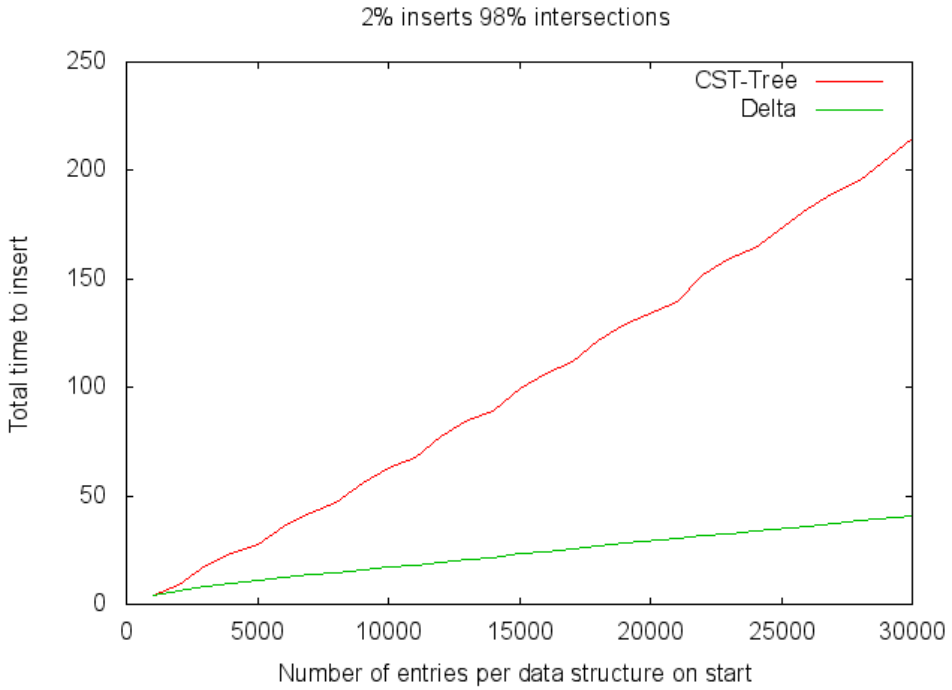

Figure 4.18: Benchmark of 2% insertions and 98% intersection and traversal of posting lists

Chapter 5

Conclusions

We can conclude that creating a string searching database in main memory is no easy task. A lot of different aspects influence the design and performance of the tool. Currently available tools such as Sphinx and Lucene are not designed for small indexes completely stored in memory. By storing the entire index in main memory, the main memory becomes the bottleneck. Therefore it is important for the structure to have good caching behavior.

Additionally it is important that the structure is also able to do updates within an acceptable timeframe, keeping the potential blocking of an updating to a minimum.

5.1 Data structures

For the new in memory string searching tool we researched a number of data structures. These data structures will be used within the index to store the posting lists. We give a small overview for each of the data structures below.

5.1.1 Bitmap index

While the bit array is extremely fast when performing union and intersection operations it requires a lot of space, especially for sparse indexes. Every new record in the database can potentially adds a new bit to each posting list. As we index every possible substring of a record this would result in huge amounts of memory being used.

Compressing the bit array can decrease memory usage, however also introduces a big performance hit compressing and decompressing part or the entire posting list.

5.1.2 B+-Tree

The B+-Tree has relatively good caching behavior compared to other "traditional" data structures. By tuning the node size to fit in a single cacheline

we can get quite good results for most operations.

Searching and inserting is relatively slow compared to some of the other data structures researched.

5.1.3 CST-Tree

Being optimized for cache behavior this structure yields the best results when searching and performing modify operations. The CST-Tree is designed in such a way that makes optimal use of the a single cacheline. By storing the tree itself in array encoded binary trees the same size as a cacheline the tree can navigated downwards several levels before a new cacheline has to be retrieved from memory to cache.

A disadvantage of the CST-Tree is that large blocks of memory have to be allocated in advance to minimize pointer operations. This takes up a lot of (potentially unused) space. By sacrificing some of the cache sensitivity for flexibility we get a more memory efficient data structure that better fits the needs of a string searching database where millions of CST-Trees will reside in memory.

Traversing on the other hand is not one of the strengths of the CST-Tree, although still acceptable in most cases.

CST-Tree adaption

The CST-Tree is quite new to field of data structures and has not been widely adapted as of yet. We were unable to locate any other instance where the CST-Tree was used in a full text search tool or keyword matching tool, making our tool the first of its kind.

5.1.4 Array

The array structure is extremely fast when traversing, however random insert operations require a massive effort. In the worst case scenario the entire old tree has to be moved if the value has to be inserted on the first position of the postings list.

5.1.5 Delta index

Of all the different data structures we tested the array structure has the best performance traversing postings list, while the CST-Tree performed the best with edit operations and search operations. In order to get an optimal result we created a delta structure that uses both these data structures. The array is used as a mostly static data structure to store the bulk of the data, while the CST-Tree is used to track changes.

By combining these data structures we benefit from the strengths of both structures while only paying a relatively small price for the overhead of maintaining the array and CST-Tree.

5.2 Caching

Influencing caching behavior is extremely difficult. Cache optimizations depend on the processor and compiler, and well intended code optimizations and processor hints can easily have a negative effect on caching behavior and performance.

This especially holds when not developing for a single processor and compiler, but developing a program that can be deployed on different architectures and compiled using different compilers.

However this does not mean that one should completely ignore caching behavior while programming. There are several guidelines to keep in mind:

- Use data as much as possible in an as narrow block of code as possible.

  Data still in cache is an order of magnitude faster than retrieving it from memory. Therefore use data as much as possible in an as small as possible code block to avoid the data in cache to be replaced with other data and retrieved from memory a second time.

- Try to keep cachelines in mind when designing and implementing data structures.

  Try to fit your data structure in a single cacheline. If your data exceeds cacheline size by only a single byte two cachelines are needed to store your data and both have to be retrieved instead of one. This can potentially half the amount of data you are processing that can fit in cache.

- Try to access data sequentially as much as possible.

  By storing and retrieving data in memory sequentially the processor can detect a pattern and prefetch large amounts of data before needed, thereby saving processor idle time waiting for data to be retrieved from memory.

5.3 Search tool

By using the delta index in our specialized search tool we created a tool that vastly outperforms the applications we inspected in chapter 2, in some cases up to an order of magnitude.

The tool itself has been specifically designed for our use case: a small database, containing only small records, but with a large volume of searches

per second. It therefore works extremely well for our use case, however as it was designed with these constraints in mind the tool will be less suitable for large databases and large individual records.

5.3.1 Full text search

When storing large documents memory usage will grow quickly with each new record. Our test corpus with 100 byte records resulted, on average, in insertions in 73 postings lists for single new record. Adding large documents in our tool as a single record would require large amounts of memory, and is therefore less suitable.

5.4 Future work

Future work is mainly related to multithreading. By using all cores of a single processor we should be able to increase performance considerably.

Other possible improvements that can be made are with a caching layer. By storing intermediate results that can be reused it is possible a lot of time can be saved. For example by analyzing old searches and keeping a list of keywords that are often merged.

Bibliography

[1] William B. Cavnar, William B. Cavnar, and John M." Trenkle. N-gram-based text categorization. *IN PROC. OF SDAIR-94, 3RD ANNUAL SYMPOSIUM ON DOCUMENT ANALYSIS AND INFORMATION RETRIEVAL*, pages 161–175, 1994.

[2] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[3] Kamel Aouiche Daniel Lemire, Owen Kaser. lemurbitmapindex. http://code.google.com/p/lemurbitmapindex/.

[4] Ulrich Drepper and Ulrich" Drepper. What every programmer should know about memory. page 114, 2007.

[5] GCC. Gcc builtin functions manual. http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html.

[6] Aberdeen Group. The performance of web applications: Customers are won or lost in one second. 2008.

[7] Marios Hadjieleftheriou, Amit Chandel, Nick Koudas, and Divesh Srivastava. Fast indexes and algorithms for set similarity selection queries. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 267–276, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Marios Hadjieleftheriou, Nick Koudas, and Divesh Srivastava. Incremental maintenance of length normalized indexes for approximate string matching. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 429–440, New York, NY, USA, 2009. ACM.

[9] Stefan Hinz. Mysql internal algorithms. http://forge.mysql.com/wiki/MySQL_Internals_Algorithms.

[10] Ig hoon Lee, Junho Shim, Sang goo Lee, and Jonghoon Chun. Cst-trees: Cache sensitive t-trees, 2007.

[11] Jeffrey Jestes, Feifei Li, Zhepeng Yan, and Ke Yi. Probabilistic string similarity joins. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 327–338, New York, NY, USA, 2010. ACM.

[12] Kyungwha Kim, Junho Shim, and Ig hoon Lee. Cache conscious trees: How do they perform on contemporary commodity microprocessors?, 2007.

[13] Tobin J. Lehman and Michael J. Carey. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 294–303, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[14] Chen Li, Bin Wang, and Xiaochun Yang. Vgram: improving performance of approximate queries on string collections using variable-length grams. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 303–314. VLDB Endowment, 2007.

[15] Sun Li-mei, Song Bao-yan, Yu Ya-xin, Li Fang-fang, and Yu Ge. Cache-conscious index mechanism for main-memory databases. *Wuhan University Journal of Natural Sciences*, 11(1):309–312, 2006.

[16] Lucene. Lucene search engine. http://www.lucene.apache.org.

[17] S. Manegold. *Understanding, Modeling, And Improving Main-Memory Database Performance*. PhD thesis, Universiteit van Amsterdam (UvA), December 2002.

[18] Simon J. Puglisi, W. F. Smyth, and Andrew Turpin. Inverted files versus suffix arrays for locating patterns in primary memory, 2006.

[19] Jun Rao and Kenneth A. Ross. Making b+- trees cache conscious in main memory. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 475–486, New York, NY, USA, 2000. ACM.

[20] Berthier Ribeiro-Neto Ricardo Baeza-Yates. *Modern Information Retrieval*. Addison Wesley, 1999.

[21] Valgrind. Cachegrind. http://valgrind.org/info/tools.html.

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