Design of a Full Text Search index for a database management system

Osku Salerma

M. Sc. Thesis

Department of Computer Science
UNIVERSITY OF HELSINKI

HELSINGIN YLIOPISTO — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Faculty	Laitos — Department
Faculty of Science	Department of Computer Science

Tekijä — Author
Osku Salerma

Työn nimi — Title

Design of a Full Text Search index for a database management system

Oppiaine — Subject
Computer Science

Työn laji — Level	Aika — Month and year	Sivumäärä — Number of pages
Master of Science thesis	January 2006	71 pages

Tiivistelmä — Abstract

Full Text Search (FTS) is a term used to refer to technologies that allow efficient retrieval of relevant documents matching a given search query. Going through each document in a collection and determining if it matches the search query does not scale to large collection sizes, so more efficient methods are needed.

We start by describing the technologies used in FTS implementations, concentrating specifically on inverted index techniques. Then we conduct a survey of six existing FTS implementations, of which three are embedded in database management systems and three are independent systems. Finally, we present our design for how to add FTS index support to the InnoDB database management system. The main difference compared to existing systems is the addition of a memory buffer that caches changes to the index before flushing them to disk, which gives us such benefits as real-time dynamic updates and less fragmentation in on-disk data structures.

ACM Computing Classification System (CCS):
H.3.3 Information Search and Retrieval

Avainsanat — Keywords
full text search, inverted index, database management system, InnoDB

Säilytyspaikka — Where deposited

Muita tietoja — Additional information

Contents

List of Tables

List of Figures

1 Introduction

It is hard to overstate how much computers have changed the way people approach information. In the days before the Internet and good search engines like Google, looking up information was hard work and people did not bother if they did not have a real need for the information.

Now, many types of information can be found by typing in a few relevant words to an Internet search engine, which then somehow manages to find the most relevant pages from among the billions of pages available. It does all this in under a second.

The above has become such a common part of our daily lives that we are no longer amazed by it. But searching through the full text of millions of documents, and ranking the results such that the most relevant results are returned first, needs specialized search technology. Such technologies are the subject of this thesis.

Before computers, full text search was not possible, so information had to be categorized in various ways so that people could find it. This works well for some information types, but it does not work at all for some.

For example, if you hear a song on the radio without catching the name of the song or the performer, with lyric databases widely available on the Internet, you can usually find the song's name by typing in a few lines of the lyrics that you remember to a search engine. This kind of searching for information by its content, rather than its category or author, is only possible with full text search.

Full Text Search (FTS) is a term used to refer to technologies that allow efficient retrieval of relevant documents matching a given search query. Going through each document in the collection and determining if it matches the search query does not scale to large collection sizes, so more efficient methods are needed [WMB99].

To demonstrate that this is actually true, we used the Unix `grep` command to do

a case-insensitive Boolean search that matched either of the words "computer" or "thesis". The machine used for the tests had a 1.33GHz Athlon CPU and enough memory so that all the documents were cached, i.e. no disk accesses occurred during any of the tests. Table 1 contains the results of the tests.

Collection size (MB)	**Search time (s)**
25	0.4
50	0.8
100	1.7
200	3.3

Table 1: `grep` benchmark

As can be seen, the search time increases linearly with the collection size. In real life the whole collection would not already be in memory, necessitating disk reads, which would slow down the operation even more. But even if the data were to fit in memory, the above speeds are simply too slow for many applications. For example, consider a web site with 200 MB of data: if it gets just one search request every three seconds, it fails to keep up with the demand and users never get their search results back.

That is not very much data these days, either. Wikipedia [wik] has over 2 GB of data just for the English language portion of the site as of this writing. LjSEEK [ljs], a site that provides a search capability for LiveJournal [liv] postings, indexes over 95 million documents with a total size over 100 GB. A sample search for "thesis" on LjSEEK completed in 0.14 seconds and found 44295 matching documents.

Extrapolating the `grep` benchmark results to 100 GB, it would have taken around 27 minutes for the same search. That assumes the data was cached in memory, which is not likely for 100 GB of data. Just reading 100 GB from disk at a rate of 40 MB per second takes 42 minutes.

So clearly we need more efficient search technologies.

The rest of this thesis is organized as follows. Section 2 describes the various index types available for implementing FTS. Section 3 describes the techniques needed to implement a search engine using an inverted index and the remaining unsolved problems.

Section 4 looks at the architectures of existing FTS implementations, in both database management systems such Oracle, MySQL, and Postgres, and in separate implementations such as Lucene and Sphinx.

Section 5 describes our design for how to add FTS index support to the InnoDB database management system. It has some important advantages over existing systems, such as real-time dynamic updates and less fragmentation in on-disk data structures.

Finally, Section 6 summarizes our conclusions.

2 FTS index types

Inverted indexes are in practise the only method used today in FTS implementations [ZMR98], so that is what we concentrate on. We also briefly describe two other index types: suffix trees/arrays and signature files.

2.1 Inverted indexes

Documents are normally stored as lists of words, but inverted indexes invert this by storing for each word the list of documents that the word appears in, hence the name "inverted index".

There are several variations on inverted indexes. At a minimum, you need to store for each word the list of documents that the word appears in. If you want to support phrase and proximity queries (see Sections 3.2.3 and 3.2.4) you need to store word positions for each document, i.e. the positions that the word appears in.

The granularity of a position can range from byte offset to word to paragraph to section, but usually it is stored at word position granularity. You can also store just the word frequency for each document instead of word positions.

Storing the total frequency for each word can be useful in optimizing query execution plans. Some implementations store two inverted lists, one storing just the document lists (and usually the word frequencies) and one storing the full word position lists. Simple queries can then be answered consulting just the much shorter document lists.

Some implementations go even further and store meta-information about each "hit", i.e. word position. They typically use a byte or two for each hit that has bits for things like font size, text type (title, header, anchor (HTML), plain text, etc.) [BP98]. This information can then be used for better ranking of search results as words that have special formatting are usually more important.

Another possible variation is whether the *lexicon* is stored separately or not. The lexicon stores all the tokens indexed for the whole collection. Usually it also stores statistical information for each token like the number of documents it appears in. The lexicon can be helpful in various ways that we refer to later on.

The space used by the inverted index varies somewhere in the range of 5-100% of the total size of the documents indexed. This enormous range exists because inverted index implementations come in so many different variations. Some store word positions, some do not, some do aggressive document preprocessing to cut down the size of the index, some do not, some support dynamic updates (they cause fragmentation and usually one must reserve extra space for future updates), some do not, some use more powerful (and slower) compression methods than others, and so on.

Table 3 contains three examples of inverted indexes for the document collection from

Id	Contents
1	The only way not to think about money is to have a great deal of it.
2	When I was young I thought that money was the most important thing in life; now that I am old I know that it is.
3	A man is usually more careful of his money than he is of his principles.

Table 2: Sample document collection

table 2. No stop words or stemming (see Section 3.1) are used in this example. The indexes are:

List 1 Just the document lists. The format is $(d_1, d_2, \ldots)$, where d_n is the document id number.

List 2 Document lists with word frequencies. The format is $(d_1{:}f_1,\ d_2{:}f_2,\ \ldots)$, where d_n is the document id number and f_n is the word frequency.

List 3 Document lists and word positions with word granularity. The format is $(d_1{:}(w_1,\ w_2,\ \ldots),\ (d_2{:}(w_1,\ w_2,\ \ldots),\ \ldots)$, where d_n is the document id number and w_n are the word positions.

Table 3 is also a good example of how time-consuming manual construction of inverted indexes is. It took over half an hour to create the lists by hand, but that pales when compared to Mary Cowden Clarke, who in 1845 published a concordance (an archaic term for an inverted index) of Shakespeare's works that had taken her 16 years to create [Cla45].

2.1.1 Compression

Storing inverted lists totally uncompressed wastes huge amounts of space. Using the word "is" in table 3 as an example, if we stored the numbers as fixed-width 32-bit integers, list 1 would take 12 bytes, list 2 would take 26 bytes (using a special marker byte to mark ends of word position lists), and list 3 would take 30 bytes.

Word	**List 1**	**List 2**	**List 3**
a	1,3	1:1, 3:1	1:(12), 3:(1)
about	1	1:1	1:(7)
am	2	2:1	2:(19)
careful	3	3:1	3:(6)
deal	1	1:1	1:(14)
great	1	1:1	1:(13)
have	1	1:1	1:(11)
he	3	3:1	3:(11)
his	3	3:2	3:(8,14)
i	2	2:4	2:(2,5,18,21)
important	2	2:1	2:(12)
in	2	2:1	2:(14)
is	1,2,3	1:1, 2:1, 3:2	1:(9), 2:(25), 3:(3,12)
it	1,2	1:1, 2:1	1:(16), 2:(25)
know	2	2:1	2:(22)
life	2	2:1	2:(25)
man	3	3:1	3:(2)
money	1,2,3	1:1, 2:1, 3:1	1:(8), 2:(8), 3:(9)
more	3	3:1	3:(5)
most	2	2:1	2:(11)
not	1	1:1	1:(4)
now	2	2:1	2:(16)
of	1,3	1:1, 3:2	1:(15), 3:(7,13)
old	2	2:1	2:(20)
only	1	1:1	1:(2)
principles	3	3:1	3:(15)
than	3	3:1	3:(10)
that	2	2:2	2:(7,23)
the	1,2	1:1, 2:1	1:(1), 2:(10)
thing	2	2:1	2:(13)
think	1	1:1	1:(6)
thought	2	2:1	2:(6)
to	1	1:2	1:(5,10)
usually	3	3:1	3:(4)
was	2	2:1	2:(9)
way	1	1:1	1:(3)
when	2	2:1	2:(1)
young	2	2:1	2:(4)

Table 3: Inverted list example

There are many ways to store the lists in a more compact form. They can be divided into two categories depending on whether the number of bits they use for coding a single value is always a multiple of 8 or not. The non-byte-aligned methods are slightly more compact, but more complex, harder to handle if dynamic updates are needed, and much slower to encode/decode. In practise, simple byte-aligned methods are the preferred choice in most cases [SWYZ02].

Variable length integers Instead of using 32 bits to store every value, we can use variable length integers that only use as many bytes as needed. There are many variations on these, but a simple and often used variation marks the final byte of the value by setting the high bit (`0x80`) to 1. The lower 7 bits of each byte are concatenated to form the value.

Elias gamma Elias gamma coding [Eli75] consists of the number written in binary, prefixed by N zeros, where N = number of bits in the binary representation − 1.

This is efficient for small values, for example 1 is coded as `1`, 2 as `010`, and 3 as `011`, but inefficient for bigger values, for example 64396 is coded as `00000000000000011111101110001100`, which is 31 bits.

Elias delta Elias delta coding [Eli75] consists of separating the number into the highest power of 2 it contains (2^N) and the remaining N binary digits of the number, encoding N+1 with Elias gamma coding, and appending the remaining N binary digits.

This is slightly more inefficient than Elias gamma coding for very small values, but much more efficient for large numbers. For example, 1 is coded as `1`, 2 as `010|0`, 3 as `010|1`, and 64396 as `000010000|111101110001100`, which is 24 bits. The character `|` in the examples is used to mark the boundary between the two parts of the coded value.

Golomb-Rice Golomb coding [Gol66] differs from Elias codes in that it is a parameterized one. The parameter b changes how the values are coded, and must be chosen according to the distribution of values to be coded, either real or expected. If b is a power of two, the coding is known as Golomb-Rice coding [Ric79], and is the one usually used, since shift operations can then be used instead of divides and multiplies.

The number to be coded is divided into two parts: the result of a division by b, and the remainder. The quotient is stored first, in unary coding, followed by the remainder, in truncated binary encoding.

Using a value of 4 for b, 1 is coded as `101`, 2 as `110`, 3 as `111`, and 4 as `0100`. Coding the number 64396 with $b = 4$ would take over 16100 bits, so using a roughly correct value for b is of critical importance.

Delta coding

We can increase our compression ratios for the lists of numbers significantly if we store them delta coded. This means that instead of storing absolute values, we store the difference to the previous value in the list. Since we can sort the lists before storing them, and there are no duplicate values, the difference between consecutive values is always ≥ 1.

The smaller the values are that we store, the more efficient the compression methods described above are. It takes less space to store (1, 12, 5, 2, 3, 15, 4) than (1, 13, 18, 20, 23, 38, 42).

In table 3, the word “is” has the following for list 3: “1:(9), 2:(25), 3:(3, 12)”. Applying delta coding would produce the following list: “1:(9), 1:(25), 1:(3, 9)”.

```
def memoryInvert(documents):
    index = {}

    for d in documents:
        for t in tokenize(d):
            if t.word not in index:
                index[t.word] = CompressedList()

            index[t.word].add(t)

    return index
```

Figure 1: In-memory inversion

2.1.2 Construction

Constructing an inverted index is easy. Doing it without using obscene amounts of memory, disk space or CPU time is a much harder task. Advances in computer hardware do not help much as the size of the collections being indexed is growing at an even faster rate.

If the collection is small enough, doing the inversion process completely in memory is the fastest and easiest way. The basic mechanism is expressed in Python [pyt] pseudocode in Figure 1.

In-memory inversion is not feasible for large collections so in those cases we have to store temporary results to disk. Since disk seeks are expensive, the best way to do that is to construct in-memory inversions of limited size, store them to disk, and then merge them to produce the final inverted index.

Moffat and Bell describe such a method [MB95]. To avoid using twice the diskspace of the final result they use an in-place multi-way mergesort to merge the temporary blocks. After the sort is complete the index file is still not quite finished, since due to the in-place aspect of the sort the blocks are not in their final order, and need to be permuted to their correct order.

Heinz and Zobel present a modified version of the above algorithm that is slightly

more efficient [HZ03], mainly because it does not require keeping the lexicon in memory permanently during the inversion process and also due to their careful choice of data structures used in the implementation. Keeping the lexicon in memory permanently during the inversion process is a problem for very large collections, because as the size of the lexicon grows, the memory available for storing the position lists decreases.

2.2 Suffix trees/arrays

While inverted indexes are the best method available for indexing large collections of normal text documents, some specialized applications have needs that are not met by inverted indexes. Inverted indexes are word-based and are not suitable for indexing data that has no word boundaries. Such data is commonly encountered in genetic databases, where you have strings millions of bytes long representing DNA/protein sequences.

Suffix trees

The best known indexing technique for such data is the suffix tree [McC76]. A suffix tree considers each position in the text as a *suffix*, i.e., a string extending from that position to the end of the text. Each suffix is unique and can be identified by its starting position.

A tree is constructed of all of the suffixes with the leaf nodes containing pointers to the suffixes and the internal nodes containing single characters. To avoid wasting space, paths where each node has just one child are compressed by removing the middle nodes and storing an indication of the next character position to consider in the root node of the path. This is known as a Patricia tree [Mor68].

This tree can be constructed in $O(n)$ time (n being the size of the text), but the constant is quite big. The resulting tree is also big, about 20 times the size of the

source text.

Suffix trees can be used to find out the answer for many questions about the source text efficiently. For example, finding substrings of length m takes $O(m)$ time, finding the longest common substring takes $O(n)$ time and finding the most frequently occurring substrings of a given minimum length takes $O(n)$ time [Gus97].

Suffix trees in their native form are only applicable for indexing a single document. To index a collection of documents you must combine the documents into one long document, index that, and then filter the search results by consulting an external mapping that keeps track of which document id is associated with each position in the text.

Suffix trees are not used for normal text indexing applications because they are slow to construct, take up huge amounts of space, cannot be updated easily, and are slower than inverted indexes for most query types.

Suffix arrays

Suffix arrays are basically a representation of suffix trees in a more compact form [MM90]. Instead of storing a tree, it stores a linear array of the leaf nodes of the suffix tree in lexicographical order. This can be searched using a binary search, but that has the problem of requiring random disk seeks, which are slow. One possibility is to build an extra index for some of the suffixes on top of this array whose purpose is to fit in memory and reduce disk seeks for searches.

Suffix arrays reduce the space overhead to about four times the size of the source text, but otherwise mostly share the advantages and problems of suffix trees, so they are also not used for normal text indexing applications.

With suffix trees/arrays there is the possibility of not indexing every possible suffix, but indexing only selected points, e.g. the start of each word. This lessens the time

and space overhead they have compared to inverted indexes, but at the same time removes most of the advantages also [BYRN99].

Recent research [CHLS05] has considerably improved the space efficiency of suffix trees/arrays and made updating them possible, but so far the algorithms work only on collections small enough to fit in memory. It is possible that in the future the algorithms will be extended to work with disk-based collections as well, at which point it would be interesting to test whether they are a viable competitor to inverted indexes for normal text indexing.

Note that in the suffix tree community, "full text search" is sometimes used to refer only to techniques capable in principle of reconstructing the source text from the index, which is quite a different meaning from the one used elsewhere, where it is used to refer to any technology capable of efficiently searching large text collections.

2.3 Signature files

When using the signature files indexing method, for each document in the collection a *signature* is generated and stored to disk [FC84]. The signature is generated by first preprocessing the document (see Section 3.1) to get the indexable tokens. Then a hash function that returns a bitvector of length v is applied to each token. These bitvectors are combined by a bitwise OR function, and the result is the document's signature, with length v.

The index is searched by generating a signature for the (combined) search terms using the same method, and then comparing this signature to each document signature by a bitwise AND function. If the result is the same as the search signature, it is a *possible match*. Since the hashing process can produce collisions, all documents found as possible matches must be preprocessed and searched through sequentially to be sure they actually match the query. This is quite slow, especially if there are

large documents in the collection. This is especially bad since the larger a document is, the more likely it is that its signature will have many 1 bits, i.e., it will be a possible match to almost all queries and must be searched through.

The above method requires reading the whole index for all queries. A more efficient way is to transpose the index, which allows reading only the relevant columns in the signature. If the signature length is v, you would then have v files of length $\frac{n}{8}$ bytes where n is the number of documents in the collection. Each file's ith bit is 1 or 0 according to whether the bit is on or off in the signature of document i.

The above is called a *bitsliced* signature file. Using that storage format, answering a query requires reading $m * \frac{n}{8}$ bytes where m is the number of 1 bits in the query signature and n is the number of documents in the collection.

Signature files do not support more complex queries like phrase or proximity queries since they only store information about whether a given search term exists in a document and nothing about its position within a document.

In practise, signature files have roughly the same space overhead as compressed inverted indexes but are slower and support less functionality, so there is no reason to use them.

3 Inverted index techniques

In this section we describe the techniques needed to implement a search engine using an inverted index. At the end of the section we describe some of the remaining unsolved problems.

3.1 Document preprocessing

Documents are normally not indexed as-is, but are preprocessed first. They are converted to tokens in the lexing phase, the tokens are possibly transformed into more

generic ones in the stemming phase, and finally some tokens may be dropped entirely in the stop word removal phase. The following sections describe these operations.

3.1.1 Lexing

Lexing refers to the process of converting a document from a list of characters to a list of *tokens*, each of which is a single alphanumeric word. Usually there is a maximum length for a single token, typically something like 32 characters, to avoid unbounded index size growth in atypical cases.

To generate these tokens from the input character stream, first case-folding is done, i.e. the input is converted to lowercase. Then, each collection of alphanumeric characters separated by non-alphanumeric characters (whitespace, punctuation, etc.) is added to the list tokens. Tokens containing too many numerical characters are usually pruned from the list since they increase the size of the index without offering much in return.

The above only works for alphabetic languages. Ideographic languages (Chinese, Japanese, Korean) do not have words composed of characters and need specialized search technologies, which we will not discuss in this thesis.

3.1.2 Stemming

Stemming means not indexing each word as it appears after lexing, but transforming it to its morphological root (*stem*) and indexing that instead. For example, the words “compute”, “computer”, “computation”, “computers”, “computed” and “computing” might all be indexed as “compute”.

The most common stemming algorithm used for the English language is Porter’s [Por80]. All stemming algorithms are complex, full of exceptions and exceptions to the exceptions, and still do a lot of mistakes, i.e., they fail to unite words that should be united or unite words that should not be united.

They also reduce the accuracy of queries, especially phrase queries. In the old days, stemming was possibly useful since it decreased the size of the index and increased the result set for queries, but today the biggest problems search engines have are too many results returned by queries and ranking the results so that the most relevant ones are shown first, both of which are hindered by stemming.

For this reason, many search engines (Google, for example) do not do stemming at all. This trend will probably increase in the future, as stemming can be emulated quite easily by wildcard queries or by query expansion.

3.1.3 Stop words

Stop words are words like “a”, “the”, “of”, and “to”, which are so common that nearly every document contains them. A stop word list contains the list of words to ignore when indexing the document collection. For normal queries, this usually does not worsen the results, and it saves some space in the index, but in some special cases like searching for “The Who” or “to be or not to be” using stop words can completely disable the ability to find the desired information.

Since stop words are so common the differences between consecutive values in both document number and word position lists for them are smaller than for normal words, and thus the lists compress better. Because of this, the overhead for indexing all words is not as big as one might think.

Like with stemming, modern search engines like Google do not seem to use stop words, since doing so would put them at a competitive disadvantage. A slightly bigger index is a small price to pay for being able to search for any possible combination of words.

3.2 Query types

There are many different ways of searching for information. Here we describe the most prominent ones and how they can be implemented using an inverted index as the base structure.

Sample queries are formatted in **bold type**.

3.2.1 Normal

A normal query is any query that is not explicitly indicated by the user to be a specialized query of one of the types described later in this section. For queries containing only a single term, the desired semantics are clear: match all documents that contain the term.

For multi-word queries, however, the desired semantics are not so clear. Some implementations treat it as an implicit Boolean query (see the next section for details on Boolean queries) by inserting hidden AND operators between each search term. This has the problem that if a user enters many search terms, for example 10, then a document that only contains 9 of them will not be included in the result set even though the probability of it being relevant is high.

For this reason, some implementations choose another strategy: instead of requiring all search terms to appear in a document, they allow some of the terms to be missing, and then rank the results by how many of the search terms were found in each document. This works quite well, since a user can specify as many search terms as he wants without fear of eliminating relevant matches. Of course it is also much more expensive to evaluate than the AND version, which is probably the reason most Internet search engines do not seem to use it.

3.2.2 Boolean

Boolean queries are queries where the search terms are connected to each other using the various operators available in Boolean logic [Boo54], most common ones being AND, OR and NOT. Usually parentheses can be used to group search terms.

A simple example is **madonna AND discography**, and a more complex one is **bruce AND mclaren AND NOT ("formula one" OR "formula 1" OR f1)**.

These are implemented using an inverted index as follows:

NOT A pure NOT is usually not supported in FTS implementations since it can match almost all of the documents. Instead it must be combined with other search terms using the AND operator, and after those are processed and the preliminary result set is available, that set is then further pruned by eliminating all documents from it that contain the NOT term.

This is done by retrieving the document list for the NOT term and removing all document ids in it from the result set.

OR The query $\mathbf{term}_1$ **OR** $\mathbf{term}_2$ **OR** ... $\mathbf{term}_n$ is processed by retrieving the document lists for all of the terms and combining them by a union operation, i.e., a document id is in the final result set if it is found in at least one of the lists.

AND The query $\mathbf{term}_1$ **AND** $\mathbf{term}_2$ **AND** ... $\mathbf{term}_n$ is processed by retrieving the document lists for all of the terms and combining them by an intersection operation, i.e., a document id is in the final result set if it is found in all of the lists.

Unlike the OR operation which potentially expands the result set for each additional term, the AND operation shrinks the result set for each additional term. This allows AND operations to be implemented more efficiently. If

we know or can guess which search term is the least common one, retrieving the document list for that term first saves memory and time since we are not storing in memory longer lists than are needed. The second document list to retrieve should be the one for the second least common term, etc.

If we have a lexicon available, a good strategy is to sort the search terms by each term's document count found in the lexicon, with the term with the smallest document count being first, and then doing the document list retrievals in that order.

As an example, consider the query **cat AND toxoplasmosis** done on a well-known Internet search engine. If we processed **cat** first, we would have to store a temporary list containing 136 million document ids. If we process **toxoplasmosis** first, we only have to store a temporary list containing 2 million document ids. In both cases the temporary list is then pruned to contain only 200,000 document ids when the lists for the terms are combined.

Another way to optimize AND operations is by not constructing any temporary lists. Instead of retrieving the document lists for each term sequentially, they are all retrieved in parallel, and instead of retrieving the whole lists, they are read from the disk in relatively small pieces. These pieces are then processed in parallel from each list and the final result set is constructed.

Which one of the above optimizations is used depends on other implementation decisions in the FTS system. Usually the latter one is faster, however.

3.2.3 Phrase

Phrase queries are used to find documents that contain the given words in the given order. Usually phrase search is indicated by surrounding the sentence fragment in quotes in the query string. They are most useful for finding documents with common words used in a very specific way. For example, if you do not remember

the author of some quotation, searching for it on the Internet as a phrase query will in all likelihood find it for you. An example would be **"there are few sorrows however poignant in which a good income is of no avail"**.

The implementation of phrase queries is an extension of Boolean AND queries, with most of the same optimizations applying, e.g., it is best to start with the least common word. Phrase queries are more expensive though, because in addition to the document lists they also have to keep track of the word positions in each document that could possibly be the start position of the search phrase.

For example, consider the query **"big deal"**. The lexicon is consulted and it is determined that "deal" is the rarer word of the two, so it is retrieved first. It occurs in document 5 at positions 1, 46 and 182, and in document 6 at position 74. We transform these so that the word positions point to the first search term, giving us 5(0, 45, 181) and 6(73). Since position 0 is before the start of the document, we can drop that one as it cannot exist.

Next we retrieve the document lists for the word "big" and prune our result set so it only contains words where "big" occurs in the right place. If "big" occurs in document 5 at positions 33 and 45 and in document 53 at position 943, the final result set is "document 5, word position 45".

Since the above is more expensive than normal searches, there have been efforts to investigate the use of auxiliary indexes for phrase searches. For example, Bahle, Williams and Zobel propose using a "nextword index" [BWZ02, WZB04], which indexes selected two-word sentence fragments. They claim that it achieves significant speedups with only a modest disk space overhead.

3.2.4 Proximity

Proximity queries are of the form **term$_1$ NEAR(n) term$_2$**, and should match documents where term$_1$ occurs within n words of term$_2$. They are useful in many cases, for example when searching for a person's name you never know whether a name is listed as "Osku Salerma" or "Salerma, Osku", so you might use the search **osku NEAR(1) salerma** to find both cases. Queries where n is 1 could also be done as a combination of Boolean and phrase queries (**"osku salerma" OR "salerma osku"**), but for larger n, proximity queries cannot be emulated with other query types. An example of such a query is **apache NEAR(5) "performance tuning"**.

Proximity queries are implemented in the same way as phrase queries, the only difference being that instead of checking for exact relative word positions of the search terms, the positions can differ by a maximum of n.

3.2.5 Wildcard

Wildcard queries are a form of fuzzy, or inexact, matching. There are two main variants:

- Whole-word wildcards, where whole words are left unspecified. For example, searching for **Paris is the * capital of the world** matches documents that contain phrases "Paris is the romance capital of the world", "Paris is the fashion capital of the world", "Paris is the culinary capital of the world", and so on.

  This can be implemented efficiently as a variant of a phrase query with the wildcard word allowed to match any word.

- In-word wildcards, where part of a single word is left unspecified. It can be

the end of a word (**Helsin***), the start of the word (***sinki**), the middle of the word (**Hel*ki**) or some combination of these (***el*nki**).

These can be handled by first expanding the wildcard word to all the words it matches and then running the modified query normally with the search term replaced by **(word$_1$ OR word$_2$ OR ... word$_n$)**. To be able to expand the word, the inverted index needs a lexicon available. If it does not have a lexicon, there is no way to do this query.

If the lexicon is implemented as a tree of some kind, or some other structure that stores the words in sorted order, expanding suffix wildcards (**Helsin***) can be done efficiently by finding all the words in the given range ([Helsin, Helsio[). If the lexicon is implemented as a hash table this cannot be done.

Expansion of non-suffix wildcards is done by a complete traversal of the lexicon, and is potentially quite expensive.

Since in-word wildcard queries need an explicit lexicon and are much more expensive in terms of time – and possibly space – needed than other kinds of queries, many implementations choose not to support them.

3.3 Result ranking

There are certain applications that do not care about the order in which the results of a query are returned, such as when the query is done by a computer and all the matching documents are processed identically. Usually, however, the query is done by a human being who is not interested in all the documents that match the query, but only in the few that *best* do so.

It is for the latter case that ranking the search results is so important. With the size of the collections available today, reasonable queries can match millions of documents. If the search engine is to be of any practical use, it must be able to somehow

sort the results so that the most relevant are displayed first.

Traditionally, the information retrieval field has used a *similarity measure* between the query and a document as the basis for ranking the results. The theory is that the more similar the query and the document are to each other, the better the document is as an answer to the query. Most methods of calculating this measure are fairly similar to each other and use the factors listed below in various ways.

Some of the factors to consider are: the number of documents the query term is found in (f_t), the number of times the term is found in the document ($f_{d,t}$), the total number of documents in the collection (N), the length of the document (W_d) and the length of the query (W_q).

If a document contains a few instances of a rare term, that document is in all probability a better answer to the query than a document with many instances of a common term, so we want to weigh terms by their *inverse document frequency* (IDF, or $\frac{1}{f_t}$). Combining this with the *term frequency* (TF, or $f_{d,t}$) within a document gives us the famous TF $\times$ IDF equation.

The *cosine measure* is the most common similarity measure. It is an implementation of the TF $\times$ IDF equation with many variants existing, with a fairly typical one shown below [WMB99]:

$$cosine(Q, D_d) = \frac{1}{W_d W_q} \sum_{t \in Q \cap D_d} (1 + \log_e f_{d,t}) \cdot \log_e \left(1 + \frac{N}{f_t}\right)$$

The details of how W_d and W_q are calculated are not important in this context. Typically they are not literal byte lengths, or even term counts, but something more abstract like the square root of the unique term count in a document. They are not even necessarily stored at full precision, but perhaps with as few bits as five.

The above similarity measures work reasonably well when the queries are hundreds of words long, which is the case for example in the TREC (Text REtrieval Conference)

competitions [tre], whose results are often used to judge whether a given ranking method is good or not.

Modern search engine users do not use such long queries, however. The average length of a query for web search engines is under three words, and the similarity measures do not work well for such queries [WZSD95].

There are several reasons for this. With short queries, documents with several instances of the rarest query term tend to be ranked first, even if they do not contain any of the other query terms, while users expect documents that contain all of the query terms to be ranked first.

Another reason is that the collections used in official competitions like TREC are from trusted sources and contain reasonable documents of fairly similar lengths, while the collections indexed in the real world contain documents of wildly varying lengths and the documents can contain anything at all. People will spend a lot of time tuning their documents so that they will appear on the first page of search results for popular queries on the major web search engines.

Thus, any naive implementation that tries to maximize the similarity between a query and a document is bound to do badly, as the makers of Google discovered when evaluating existing search engines [BP98]. They tried a search for "Bill Clinton" and got as a top result a page containing just the text "Bill Clinton sucks", which is clearly not the wanted result when the web is full of pages with relevant information about the topic.

Web search engines use much more sophisticated ranking strategies that take into account what documents link to each other, what words those links contain, how popular each site is, and many other factors. The exact strategies are highly guarded trade secrets, just like the query evaluation optimization strategies discussed in the next section.

Some of the more recent published research on better ranking methods have been on things like *cover density ranking* [CCT00], which uses the proximity of the query terms within a document as the basis for ranking, and on *passage ranking* [KZSD99], which divides long documents into shorter passages and then evaluates each of those independently.

3.4 Query evaluation optimization

Much research over the last 20 years has been conducted on optimizing query evaluation. The main things to optimize are the quality of the results returned and the time taken to process the query.

There are surprising gaps in the published research, however. The only query type supported by the best Internet search engines today is a hybrid mode that supports most of the extended query types discussed in Section 3.2 but also ranks the query results.

The query evaluation optimization research literature, however, ignores the existence of this hybrid query type almost completely and discusses just plain *ranked* queries, i.e., queries with no particular syntax which are supposed to return the k most relevant documents as the first k results. This is unfortunate since normal ranked queries are almost useless on huge collections like the Internet, because almost any query besides the most trivial one needs to use extended query methods like disallowing some words (Boolean AND NOT) and matching entire phrases (phrase query) to successfully find the relevant documents from the vast amounts in the collection.

The reason for this lack of material is obvious: the big commercial search engines power multi-billion dollar businesses and have had countless very expensive man-years of effort from highly capable people invested in them. Of course they are

not going to give away the results of all that effort for everyone, including their competitors, to use against them.

Some day the details will leak out or an academic researcher will come up with them on his own, but the bar is continuously being raised, so I would not expect this to happen any time soon.

That said, we now briefly mention some of the research done, but do not discuss the details of any of the work.

Most of the optimization stragies work by doing some kind of *dynamic pruning* during the evaluation process, by which we mean that they either do not read all the inverted lists for the query terms (either skipping a list entirely or not reading it through to the end) or read them all, but do not process all the data in them if it is unnecessary. The strategies can be divided into *safe* and *unsafe* groups, depending on whether or not they produce the exact same results as unoptimized queries. Buckley and Lewit [BL85] were one of the first to describe such a heuristic. Turtle and Flood [TF95] give a good overview of several strategies. Anh and Moffat [AM98] describe yet another pruning method.

Persin et al. [PZSD96] describe a method where they store the inverted lists not in document order as is usually done, but in frequency-order, realizing significant gains in processing time.

There are two basic methods of evaluating queries: term-at-a-time and document-at-a-time. In term-at-a-time systems, each query term's inverted list is read in turn and processed completely before proceeding to the next term. In document-at-a-time systems, each term's inverted lists are processed in parallel.

Kaszkiel and Zobel [KZ98] investigate which of these is more efficient, and end up with a different conclusion than Broder et al.[BCH$^+$03], who claim that document-at-a-time is the more efficient one. To be fair, one is talking about *context-free*

queries, i.e. queries that can be evaluated term by term without keeping extra data around, while the other one is talking about *context-sensitive* queries, e.g. phrase queries, where the relationships between query terms are important. Context-sensitive queries are easier to handle in document-at-a-time systems since all the needed data is available at the same time.

Anh and Moffat also have another paper, this time on *impact transformation* [AM02], which is their term for a method they use to enhance the retrieval effectiveness of short queries on large collections. They also describe a dynamic pruning method based on the same idea.

Anh and Moffat make a third appearance with a paper titled "Simplified similarity scoring using term ranks" [AM05], in which they describe a simpler system for scoring documents than what has traditionally been used.

Strohman et al. describe an optimization to document-at-time query evaluation they call *term bounded max_score* [STC05], which has the interesting property of returning exactly the same results as an unoptimized query evaluation while being 61% faster on their test data.

Carmel et al. describe a *static index pruning* method [CCF+01] that removes entries from the inverted index based on whether or not the removals affect the top k documents returned from queries. Their method completely removes the ability to do more complex searches like Boolean and phrase searches, so it is usable only in special circumstances.

Jónsson et al. tackle an issue left alone in information retrieval research so far, buffer management strategies [JFS98]. They introduce two new methods: 1) a modification to a query evaluation algorithm that takes into account the current buffer contents, and 2) a new buffer-replacement algorithm that incorporates knowledge of the query processing strategy. The applicability of their methods to generic FTS systems is

not especially straightforward, since they use a very simplistic FTS system with only a single query running at one time and other restrictions not found in real systems, but they do have some intriguing ideas.

3.5 Unsolved problems

While many aspects of full text search technologies have matured in the last 10-15 years to the point where there can be said to be established techniques with no significant problems, some important aspects remain unsolved.

3.5.1 Dynamic updates

Dynamic updates are by far the hardest problem in FTS systems. Almost all research in the area considers only static indexes with a footnote thrown in saying "Updates can be handled by rebuilding the whole index". Such an approach is feasible in some cases, but as the collection sizes keep growing and people's expectations about information retrieval keep going up, better approaches are needed.

There has been some research done on dynamic updates [CP90, BCC94, CCB94, TGMS94, CH99], but most of it is dated and of little practical use, which is demonstrated in Section 4 when we look at existing FTS systems and note that none of them feature real-time updates with good performance.

Lester et al. [LZW04] divide the possible ways of implementing updates into three categories: in-place, re-merge and rebuild. In-place refers to updating each term's inverted list in its current location and moving it to a new place if not enough space is available in the current location. Re-merge refers to constructing a separate index from the updated documents, merging that with the original index, and replacing the original index with the result. Rebuild refers to simply indexing the whole collection again.

They implemented all three methods and benchmarked them. Rebuild was found to be prohibitively expensive and only suitable for small collections with little update activity and no need for absolutely up-to-date query results. It also means that if it is not acceptable for the search engine to be unusable while the index is being rebuilt the old index needs to be retained during the construction of the new one, leading to a doubling of the disk space used.

Re-merge was found to be reasonably competitive, but it also suffers from the need to keep the old index available during construction of the new one. It is also probable that re-merge does not scale as well as the in-place method for very large collections as it needs to recreate the whole index even though the portion of it that actually needs updating keeps getting smaller and smaller.

In-place was about the same speed as re-merge, but scaled much better when the batch update size was smaller. In-place is the only method that needs to touch only those portions of the index actually changed, so it is probable that the larger the collection is, the better in-place is compared to the other methods.

All of the methods above rely on the updates being batched, i.e., several documents being added / modified / deleted at once. The overhead for doing the update after every document change is too much, since each document contains hundreds or thousands of separate terms. If we assume the inverted list for a single term can be updated with just one write operation (which is hopelessly optimistic), that a newly added document contains 1000 unique terms and that the disk used has a seek time of 8 ms, then the update using the in-place algorithm would take $1000 * 0.008 = 8$ seconds.

By delaying the update until it can be done for many documents at once we benefit since most of the terms added are found in more than one of the new documents and we can thus write out the changes for several documents in one disk operation,

lowering the per-document cost.

Some of the publications refer to keeping the batched documents in an inverted form in a memory buffer, which is then flushed to disk when the update is actually done. This has the benefit that query evaluation can consult this memory cache in addition to the disk contents, and by doing so, achieve real-time updates. This is an obvious optimization, but it seems that implementing it is difficult since none of the systems studied in Section 4 use it.

3.5.2 Multiple languages

Document collections containing documents in more than one language result in several additional challenges to overcome. All of the document preprocessing settings, i.e. lexing, stemming, and stop word lists, are language-specific. For example, if you try to use a stemming algorithm designed for the English language on a Finnish language document, the results will be catastrophic.

Tagging each document with its language ranges from trivial to impossible. If it is possible, the problems can mostly be solved by having language-specific document preprocessing settings and using the appropriate ones for each processed document.

Query processing is also a problem, since we would need to know what language each query term is in, and normal users are not willing to use queries of the form "eng:word1 fin:word2 spa:word ...".

Even more challenging are cases where a single document contains text in multiple languages. It is hard to know where the language changes mid-document, and even if you do, you now have to cope with arbitrarily many preprocessing settings per document.

4 Existing FTS implementations

In this section we describe the overall architecture of six different FTS implementations. The descriptions concentrate on the index structures and their storage implementations, since that is where most of the differences are found. Document preprocessing is configurable in almost all FTS systems and is thus uninteresting to discuss separately for each system. Query and ranking algorithms are not publicly described for any of the systems, and are thus ignored as well, except for some special mentions.

Special attention is paid to how well each system handles dynamic updates.

4.1 Database management systems

In the following subsections we look at three FTS implementations in databases. While there has been very little academic research on integrating FTS functionality into a database system [Put91, DDS+95, KR96], most big commercial databases and some of the free sofware ones do provide FTS functionality. The architecture, capabilities and scalability of the systems differ widely, however, as will be seen in the following sections.

Search solutions totally outside the database are widely used, but they suffer from several problems:

- Hard to keep data synchronized, and real-time updates are impossible.
- Hard to integrate with other operations, whereas in a database that has FTS something like `"SELECT id FROM products WHERE description LIKE '%angle grinder%' AND price < 50"` is easy and efficient to do, especially if the database can optimize query execution plans according to each condition's selectivity factor [MS01].

4.1.1 Oracle

Oracle [oraa] is one of the largest commercial database management systems in the world and has been developed continuously for over 25 years, so it is no surprise then that it includes well-developed FTS support. The architecture of their implementation, known as Oracle Text, is described by several papers at Oracle's website [orab]. It uses Oracle's Extensibility Framework which allows developers to create their own index types by providing a set of functions to be called at appropriate times (insert, update, delete, commit, etc.).

The index consists of four tables known as **$I**, **$K**, **$N** and **$R**. The actual table names are formed by concatenating "**DR$**", the name of the index, and the suffix (e.g. **$I**).

Quoting from Oracle documentation, the tables are:

"The **$I** table consists of all the tokens that have been indexed, together with a binary representation of the documents they occur in, and their positions within those documents. Each document is represented by an internal document id value.

The **$K** table is an index-organized table (IOT) which maps internal document id values to external row id values. Each row in the table consists of a single document id / row id pair. The IOT allows for rapid retrieval of a document id given the corresponding row id value.

The **$R** table is designed for the opposite lookup from the **$K** table – fetching a row id when you know the document id value.

The **$N** table contains a list of deleted document id values, which is used (and cleaned up) by the index optimization process."

There are other tables used by the implementation (**$PENDING**, **$DELETE**, etc.) but they are only used to efficiently implement transactional updates of the

index's contents. Once a document is permanently inserted to the index, the four tables described above are all that store any information about it.

The **$I** table stores the inverted list for a term in a BLOB column with a maximum length of around 4000 bytes (the limit is chosen so that the column can be stored in-line with the other row data). A single term can have many rows. A popular term in a large collection can have many megabytes of data, so it is not clear how well this 4000 byte limit scales.

The Oracle Text indexes are real-time only for deletions (which are easy to handle by keeping a record of deleted document ids and pruning those from search results); updates and inserts are not immediately reflected. Only when a synchronization operation (SYNC) is run are the new documents available for searching. There is an option to search through the documents listed in the **$PENDING** table on every query, but that is too expensive to be usable in most cases.

The SYNC operation parses the documents whose row ids are listed in the **$PENDING** table and inserts new rows to the **$I**, **$K** and **$R** tables. The frequency with which SYNC is done must be selected by each site to comply with their needs. If new data needs to be searchable soon after insertion, SYNC must be run frequently which leads to fragmentation in the **$I** table as data for one term is found in many short rows, and thus leads to bad query performance.

Another source of non-optimality are the deleted or updated rows whose data still exists in the **$I** table. Oracle provides two operations to optimize the indexes, OPTIMIZE FAST and OPTIMIZE FULL.

OPTIMIZE FAST goes through the **$I** table and for each term combines rows until there is at most one row that has a BLOB column smaller than the maximum size (to some degree of precision, as the BLOB contents must be cut at document boundaries). It does not remove the data for updated / deleted documents from the

BLOBs and it can not be stopped once started.

OPTIMIZE FULL does everything OPTIMIZE FAST does and additionally deletes the data for updated / deleted documents from the BLOBs. It can also be stopped and later restarted at the same point, and thus it can be scheduled to run periodically during low-usage hours.

4.1.2 MySQL

MySQL [mys] has multiple table types, and one of these, MyISAM, has support for an FTS index. The way it is implemented is quite different from other FTS systems. Instead of storing the inverted lists in a tightly packed binary format, it uses a normal two-level B-Tree. The first level contains records of the form (word, count), where count is the number of documents the word is found in. The second level contains records of the form (weight, rowid), where weight is a floating point value signifying the relative importance of the word in the document pointed to by rowid.

The index is updated in real-time. That is almost its sole good point, however. MySQL's MyISAM FTS suffers from at least the following problems:

- It does not store word position information so it can not support phrase searches, proximity searches, some forms of wildcard searches and other forms of complex query types. It also can not use the proximity data for better ranking of the returned documents.

- It has two search modes: normal search and Boolean search. The normal search does not support any Boolean operations and does simple ranking of the search results. The quality of this ranking is not too good, as it often returns top-ranked documents which have none of the search terms in them while failing to return documents which have all the search terms in a single

sentence in the correct order.

The Boolean mode does not rank the returned results in any way so it is less useful than it could be.

- It does not scale. This is widely known, but to get some concrete numbers, it was tested using Wikipedia's [wik] English language pages as the document collection. The collection contains 1,811,554 documents totaling 2.7 GB of text.

  The default settings for the MyISAM FTS were used which include a very aggressive stop word list of over 500 words, so the created FTS index was considerably smaller than it would normally be. The tests were done on a 2.0 GHz Athlon 64 machine with 1 GB of RAM and a 7200 RPM hard drive.

  The FTS index creation took 37 minutes, which can only be described as slow. The index size was 1.2 GB.

  A simple two-word query on this newly created index took 162 seconds. Document deletion took 8.3 seconds and document insertion took 3.1 seconds.

  These are the optimum numbers on a newly created index; apparently, as the index is updated, it fragments and performance goes down, meaning users periodically have to rebuild their indexes from scratch.

  In light of these numbers it is no wonder that Wikipedia had to stop using MyISAM FTS as their search engine; it simply does not scale even to their very moderate collection size.

- It has no per-index configuration. All configuration is done on a global level, which means that if you want to customize some aspect of indexing or searching you can only have one FTS index per database.

It is not our intention to criticize MySQL unfairly. However, MyISAM FTS has

numerous well-known flaws that make it unsuitable for anything but the smallest operations, and not disclosing those flaws would be dishonest.

4.1.3 PostgreSQL

PostgreSQL [pos] has no built-in support for FTS, but it does have third-party extensions providing it. Tsearch2 [tse] is an extension providing the low-level FTS index support, and OpenFTS [ope] builds on top of that to provide a complete FTS solution.

Tsearch2 adds a column of type `tsvector` to each table indexed, which contains an ordered list of terms from the document along with their positions. It can be thought of as saving the result after the document has been preprocessed. Which is understandable, as the second stage consists of an index for this new column type, and this index is implemented as a signature file (see Section 2.3).

Since signature files can only confirm the non-existence of a term in a document, not the existence, after index scanning, each of the the remaining possible matches have to be scanned to see whether they match the query or not.

The good thing about signature files is that updates are easy since you only need to update a limited amount of data. The bad thing is that they do not scale, and this is true for Tsearch2 as well. The upper practical limit for an indexable collection varies depending on the amount of unique terms, average document size, and total number of documents in the collection, but the limit exists and is relatively low.

That is probably the reason why Tsearch2 and OpenFTS have remained relatively little known, even in the PostgreSQL world; there is a limited amount of interest in solutions that do not scale to real-world needs.

4.2 Other

In the following three subsections we look at FTS implementations that are not tied to a specific database management system.

4.2.1 Lucene

Lucene [luc] is a search engine library written in Java. It was originally written by Doug Cutting, who had previously done both research in the FTS field [CP90] and developed a commercial FTS implementation while working at Excite [exc]. Currently Lucene is an Apache Software Foundation [apa] project with multiple active developers.

The Apache Software Foundation has another project, Nutch [nut], that uses Lucene as a building block in a complete web search application. Since we are interested in the search technology in Lucene itself, this distinction does not matter to us.

Lucene uses separate indexes that it calls segments. Each of them is independent and can be searched alone, but to search the whole index, each segment must be searched. New segments are created as new documents are added to the collection (or old ones are updated; Lucene does not support updates per se, one must manually delete the old document id from Lucene and add the new one). There is a setting, `mergeFactor`, that controls how many new documents are kept in memory before a new segment is written to disk. Increasing the setting results in faster indexing speeds, however, the memory buffer is insert-only, it is not used for searches, so you must flush the new segment to disk to get the new documents included in search results.

The `mergeFactor` setting is also used for merging segments. When `mergeFactor` segments of any given size exist, they are merged into a single segment.

Each segment consists of multiple files. The most interesting ones are:

Token dictionary A list of all of the tokens in the segment, with a count of the documents containing each token and pointers to each token's frequency and proximity data.

Token frequency data For each token in the dictionary, lists the document ids of all the documents that contain that token, and the frequency of the token in that document.

Token proximity data For each token in the dictionary, lists the word positions that the token occurs in each document.

Deleted documents A simple bitmap file containing a 1-bit in bit position X if the document with the document id X has been deleted.

This is a standard FTS implementation using file-based inverted lists. They store two lists for each document/token pair, one containing the token frequency in the document and one containing the token positions within the document.

Lucene is aimed mostly at indexing web pages and other static content. This is evident in the fact that all index updates are blocking, i.e. two insertions to the index cannot be run concurrently. Updates must also be done in fairly large batches to achieve reasonable performance.

4.2.2 Sphinx

Sphinx [sph] is a FTS engine written by Andrew Aksyonoff. It is not yet publicly available but can be acquired by contacting the developer directly. Several large sites are already using it, with LjSEEK [ljs], a site that provides a search capability for LiveJournal [liv] postings, being probably the biggest one. They index over 95 million documents with a total size over 100 GB. As an indication of Sphinx's speed, a sample search for "thesis" on LjSEEK completed in 0.14 seconds and found 44295 matching documents.

Sphinx is also quite fast at indexing; the Wikipedia data takes just 10 minutes, compared to 37 minutes for MySQL (see Section 4.1.2).

Sphinx derives its speed from being a good implementation of a classical static-collection FTS engine. The only kind of collection updating it supports is appending new documents. Document updates and deletions require a complete index rebuild.

The append functionality is implemented by the user periodically rebuilding a smaller index that contains only documents with a document id greater than the largest document id in the main index. Query evaluation then consults both indexes and merges the results.

Sphinx's index structure is very simple. There are two files, a small term dictionary and the inverted list file. The term dictionary contains for each term an offset into the inverted list file and some statistics. The inverted list file is simply a list of all of the occurrences for all of the terms, with no empty space left anywhere.

The term dictionary is small enough to be kept cached in the memory and the inverted list for a term is a single contiguous block in the inverted list file. Reading the inverted list for a term thus requires a maximum of one disk seek. This is the main reason behind Sphinx's fast query speed.

It is also the reason why Sphinx does not support any kind of updates, since the inverted list file can not be updated without rebuilding it entirely. But for situations that do not require real-time updates and whose collection is small enough to be re-indexed when necessary, Sphinx is a good choice.

4.2.3 mnoGoSearch

mnoGoSearch [mno] is a GPL-licensed search engine mostly meant for indexing web sites. It originally supported storing the inverted index both in a custom format in normal files and in a database, but now it has just the database storage option. It

supports several databases but it is unclear how well tested some of them are.

mnoGoSearch has three ways of storing the inverted index. Quoting from its documentation, they are:

Single All words are stored in a single table of structure (url_id, word, weight), where url_id is the ID of the document which is referenced by the rec_id field in "url" table. Word has the variable char(32) SQL type. Each appearance of the same word in a document produces a separate record in the table.

Multi Words are located in 256 separate tables using hash function for distribution. Structures of these tables are almost the same with "single" mode, but all word appearances are grouped into a single binary array, instead of producing multiple records.

BLOB Words are located in a single table of structure (word, secno, intag), where intag is a binary array of coordinates. All word appearances for the current section are grouped into a single binary array. This mode is highly optimized for search, indexing is not supported. You should index your data with "multi" mode and then run "indexer -Eblob" to convert "multi" tables into "blob". Note: this mode works only with MySQL.

It is doubtful how well mnoGoSearch scales, especially in the default "single" storage mode, since retrieving the inverted list for a term can require a very large number of disk seeks. Updates are also a problem, as not only do the database rows have to be deleted and re-inserted, but the multiple indexes on the table have to be updated as well.

5 My design

We have previously described generic FTS techniques and surveyed six specific implementations. In this section we present our design for adding FTS index support to the InnoDB [inn] database management system. We do not discuss specific query evaluation algorithms and other implementation details that are easily changed. Instead we concentrate on the high-level architecture and permanent on-disk data structures. If these are designed right, allow all needed operations, and scale as needed, we can be sure that problems can be addressed without re-designing the entire system.

The architecture of our design is very similar to the one used by Oracle, described in Section 4.1.1. It is essential that you have read and understood that section before reading this one.

Our design differs from Oracle in that we have added a memory buffer on top of the disk-based inverted index to cache changes before flushing them to disk. This allows us to avoid the annoying problem in Oracle, i.e., the fact that there is no optimal setting for how often to run SYNC. Run it too frequently and your index gets fragmented; run it too infrequently and your queries return stale data.

The memory buffer is an in-memory inverted index that is updated every time a document is added or updated in the collection. When evaluating queries both the disk-based inverted index and the memory buffer are consulted. This way we achieve real-time FTS index updates and thus remove the disadvantage of doing SYNC infrequently. In fact, we can delay doing SYNC for almost arbitrarily long by increasing the size of the memory buffer without any ill-effects.

The use of a memory buffer is not a new idea. It has been proposed several times in the past [CP90, Put91], but we are not aware of any FTS implementation actually using it.

The rest of this section is organized as follows: Section 5.1 describes the tables used, Section 5.2 describes miscellaneous implementation issues, Section 5.3 goes through the possible operations and describes in moderate detail how they are implemented, Section 5.4 describes the background threads used, and Section 5.5 outlines the open issues left in the design.

Note that you will probably want to refer back to the table descriptions when reading the operation descriptions, as either of them is impossible to understand without understanding the other one as well.

5.1 Table definitions

All the FTS index data resides in normal InnoDB tables. For each table that has an FTS index, a copy of each of the tables described below is created with a name of something like `__innodb_fts_$(TABLE_NAME)_$(FTS_TABLE_NAME)`. As an example, if you added an FTS index to a table named `orders`, the FTS code would then create tables named `__innodb_fts_orders_index`, `__innodb_fts_orders_added`, and so on.

5.1.1 index

```
CREATE TABLE index (
  word VARCHAR(32),
  first_doc_id INTEGER NOT NULL,
  last_doc_id INTEGER NOT NULL,
  doc_count INTEGER NOT NULL,
  ilist BLOB NOT NULL,

  PRIMARY KEY (word, first_doc_id)
);
```

This contains the actual inverted index. `first_doc_id` and `last_doc_id` are the first and last document ids stored in the `ilist`, and `doc_count` is the total number of document ids stored there.

The `ilist` BLOB field contains the document ids and word positions within each document in the following format, described in a slightly modified Extended Backus-Naur Form [Wir77]:

```
data := doc+
doc := doc_id word_positions+
doc_id := vlc_integer
word_positions := word_position+ word_position_end_marker
word_position := vlc_integer
word_position_end_marker := '0x00'
```

`doc_id` and `word_position` are stored as variable length coded integers for space efficiency, and they are also delta-coded in respect to the previous value for the same reason. The compression scheme chosen is byte-oriented because it allows efficient combining of `index` rows during OPTIMIZE, and it is faster and does not use much more space than bit-oriented compression schemes. It is also much simpler.

`vlc_integer` consists of 1-n bytes with the final byte signaled by the high-bit (`0x80`) being 1. The lower 7 bits of each byte are used as the payload and the number is encoded in a most-significant-byte (MSB) first format. Thus 1 byte can encode values up to 127, 2 bytes up to 16383, 3 bytes up to 2097151, 4 bytes up to 268435455, and 5 bytes up to 34359738368. On decoding the decoded value is added to the previous value in the sequence, which is 0 at the start of a sequence.

The reason for using MSB instead of LSB (least-significant-byte first) is because we want to be able to signal the end of a sequence by storing a zero byte. In LSB the encoding for a number can start with a zero byte so we cannot differentiate between an end-of-sequence marker and a normal number, while in MSB, a zero byte can not appear as the first byte of an encoded number.

Table 4 contains examples of the encoding format.

Value	First byte	Second byte	Third byte
0	10000000		
1	10000001		
2	10000010		
...			
127	11111111		
128	00000001	10000000	
129	00000001	10000001	
130	00000001	10000010	
...			
16383	01111111	11111111	
16384	00000001	00000000	10000000
16385	00000001	00000000	10000001

Table 4: `vlc_integer` encoding examples

5.1.2 row_ids

```
CREATE TABLE row_ids (
  doc_id INTEGER PRIMARY KEY,
  row_id BINARY(6) NOT NULL
);
```

This contains the mapping from document ids to row ids (InnoDB's row ids are 6 bytes in length).

5.1.3 doc_ids

```
CREATE TABLE doc_ids (
  row_id BINARY(6) PRIMARY KEY,
  doc_id INTEGER NOT NULL,
);
```

This contains the mapping from row ids to document ids.

5.1.4 added

```
CREATE TABLE added (
  doc_id INTEGER PRIMARY KEY
);
```

This contains the document ids of documents that have been added or updated but whose contents are not in `index` yet.

5.1.5 deleted

```
CREATE TABLE deleted (
  doc_id INTEGER PRIMARY KEY
);
```

This contains the document ids of documents that have been deleted but whose data has not yet been removed from `index`.

5.1.6 deleted_mem_buf

```
CREATE TABLE deleted_mem_buf (
  doc_id INTEGER PRIMARY KEY
);
```

This is similar to `deleted` except that data to this is added at a different time (see Section 5.3.4).

5.1.7 being_deleted

```
CREATE TABLE being_deleted (
  doc_id INTEGER PRIMARY KEY
);
```

This contains the document ids of documents that have been deleted and whose data we are currently in the process of removing from `index`.

5.1.8 being_deleted_mem_buf

```
CREATE TABLE being_deleted_mem_buf (
  doc_id INTEGER PRIMARY KEY
);
```

This is similar to `being_deleted` (see Section 5.3.4).

5.1.9 config

```
CREATE TABLE config (
  key TEXT PRIMARY KEY,
  value TEXT NOT NULL,
);
```

This contains the user-definable configuration values.

5.1.10 stopwords

```
CREATE TABLE stopwords (
  word TEXT PRIMARY KEY,
);
```

This contains the stopword list.

5.1.11 state

```
CREATE TABLE state (
  key TEXT PRIMARY KEY,
  value TEXT NOT NULL,
);
```

This contains the internal state variables.

5.2 Miscellaneous

Unicode [uni] is the only practical choise for new systems that have to support multiple languages and by standardizing on it we will reduce character set related issues to the minimum. Each document and query is converted from its native character set to Unicode before processing. We will use UTF-8 as the on-disk character encoding format since it takes the least amount of space for typical text and either UTF-16 or UTF-32 as the in-memory format. We do not use UTF-8 as the in-memory format because we want to use a fixed-width encoding for simplicity. The choice between UTF-16 or UTF-32 will be determined during the implementation

process as the factors affecting the decision are mostly which is faster on current CPUs and how much extra memory would be used by UTF-32 (in absolute terms; relatively it of course uses twice as much as UTF-16). Unicode characters that can not be represented as a single UTF-16 character, instead needing two UTF-16 surrogate characters, are almost irrelevant since those characters are not used in any language supported by normal FTS techniques.

At least the first version will be designed and aimed for alphabet-based languages only. No effort will be spent on supporting pictorial languages as they require significantly different techniques.

Lexing, case-folding, and stemming will be handled by using user-defined functions to achieve total flexibility. Sane defaults for major Western languages will be made available, probably with some amount of user customization available for those who do not want to write their own functions. Stemming and stopword lists will be disabled by default.

At least the following configuration variables will exist:

- Size of the memory buffer (`mem_buffer_size`).
- Lexing, case-folding and stemming function names.
- Maximum concurrency value, i.e., how many searches to allow at the same time. This is needed to limit the worst case memory usage.
- Frequency (in seconds) at which to checkpoint progress during OPTIMIZE (`optimize_checkpoint_freq`).

At least the following internal state variables will exist:

- Next available document id (`next_doc_id`).

- Last token processed in OPTIMIZE (`last_optimize_word`). It is an empty string if OPTIMIZE has not yet been run or the last run has finished in its entirety.

5.3 Operations

When the document collection is modified, the FTS system is activated when a transaction is ready to commit, with full information about added, deleted and updated row ids for the transaction. This simplifies matters greatly and the inability to query modified data within a single transaction is of no great consequence.

Before activating the FTS system, the change information needs to be normalized. By this we mean that if a transaction does several operations on a single document, only the final state of the document should be considered. The following transformations are thus needed:

- If the document existed before the transaction started and it is modified more than once during the transaction, only the last modification will be signaled.
- If the document existed before the transaction started and it is modified during the transaction, followed by a deletion of the document, only the deletion will be signaled.
- If a new document is added during the transaction (and possibly modified after its initial insertion) but it is deleted before the end of the transaction, nothing will be signaled.
- If a new document is added during the transaction and modified after its initial insertion, only the addition will be signaled.

5.3.1 Add

When a new document is added to the database, the following steps are taken:

- A document id is assigned to it (`id = state.next_doc_id`), and the next available document id is incremented (`state.next_doc_id += 1`). To avoid blocking other threads wanting a new document id until this thread is ready to commit everything, getting a document id and incrementing the stored value are done in a separate transaction that is committed immediately (`SELECT ... FOR UPDATE; UPDATE ...; COMMIT`).
- One new row is added to each of the tables `row_ids`, `doc_ids` and `added`.
- A flag is set to mark that the "Add" thread should be activated at the end of this commit (see Section 5.4.1).

5.3.2 Delete

When a document is deleted from the database, the following steps are taken:

- The document id is retrieved from the `doc_ids` table.
- The row for the document is removed from the tables `row_ids`, `doc_ids` and `added` (it may or may not exist in `added`), and a new row is added to `deleted`. If the row in `added` existed, the memory buffer must be notified about the deletion so it can properly keep track of things (see Section 5.3.4).

5.3.3 Update

Updating of a document is handled by doing a "deletion" of it followed by its "insertion".

5.3.4 SYNC

SYNC writes out the data from the memory buffer as new rows to the `index` table, clears the memory buffer, and deletes the document ids from `added` that have now been committed to disk. This is simple enough in itself, but two factors complicate the implementation: concurrency requirements and deleted documents.

By concurrency requirements we mean that it is not acceptable to acquire an exclusive lock on the memory buffer for the entire duration of the SYNC operation since concurrent queries would be blocked for too long. This problem is solved by noticing that the SYNC operation spends almost all its time reading data from the memory buffer and writing that data to disk, and only at the very end does it want to modify the memory buffer by emptying it completely, and that this operation takes a negligible amount of time.

We can thus allow queries to consult the memory buffer while SYNC is running if we have some way of atomically upgrading the SYNC operation's shared lock to an exclusive lock at the end. The atomicity is needed so that there is no chance of another thread grabbing the lock from the SYNC operation.

The lock type needed for the above is a read/upgrade/write lock, which is similar to a normal read/write lock except it has a third lock type, an upgrade lock, that can be atomically upgraded to a write lock. The lock can be held at any time by one writer, one upgrader, many readers, or one upgrader and many readers. When an upgrader has acquired the lock new readers are still allowed, but when the upgrader has indicated that he wants to upgrade his lock to a write lock, new readers are kept waiting, and the write lock is granted when all of the existing readers have finished. Appendix A contains an implementation of such a lock using POSIX mutexes and condition variables.

The "deleted documents" problem refers to the situation when a document has

been added but its data is only in the memory buffer, not in `index` table yet, and that document is then deleted. If the memory buffer did not keep track of deleted documents, the following scenario would be possible: OPTIMIZE would be run eventually, it would do its thing, and at the end it would remove the deleted document's document id from the `deleted` table. Then SYNC would be run and it would write the contents of the memory buffer to the `index` table, including data for the deleted document. The problem now is that we would have no way of knowing the document id that was deleted and that needs to be cleaned up by a later OPTIMIZE pass since the earlier OPTIMIZE pass deleted that data from the `deleted` table.

We solve this problem by having a list of deleted documents in the memory buffer and two additional tables, `deleted_mem_buf` and `being_deleted_mem_buf` that are exact copies of the `deleted` and `being_deleted` tables. These are then used to coordinate things between document deletion, OPTIMIZE and SYNC so that there is no chance of a document deletion going unnoticed forever. We do not use the standard memory buffer lock for the deleted document list because if we did, deleting a single document when SYNC is running would block until SYNC finishes, which could take a considerable amount of time. By using a separate mutex for it we can make sure that document deletion always succeeds rapidly.

Figure 2 contains a detailed pseudocode description of how SYNC works.

The SYNC operation will be exposed to the users so that if they want to minimize the delay on startup (see Section 5.3.6) they can run SYNC('`table_name`') before shutting down the database.

5.3.5 OPTIMIZE

OPTIMIZE is the same as Oracle's OPTIMIZE FULL, i.e., it goes through the `index` table and for each token combines rows until there is at most one row that

```
def SYNC():
    mem_buf_lock.u_lock()

    exec_sql("BEGIN TRANSACTION")

    # get the list of deleted documents that are either in the memory
    # buffer or were headed there but were deleted before the add thread
    # got to them.
    mem_buf_deleted_mutex.lock()
    deleted = mem_buf_get_deleted_doc_ids()
    mem_buf_deleted_mutex.unlock()

    for token in mem_buf_get_tokens():
        write_token(token)

    mem_buf_lock.u_upgrade()

    exec_sql("DELETE FROM added WHERE doc_id IN (%s)" % mem_buf_get_doc_ids())

    for doc_id in deleted:
        exec_sql("INSERT INTO deleted_mem_buf VALUES (%d)" % doc_id)

    mem_buf_clear()

    mem_buf_deleted_mutex.lock()

    for doc_id in deleted:
        mem_buf_remove_deleted_mark(doc_id)

    mem_buf_deleted_mutex.unlock()

    exec_sql("COMMIT TRANSACTION")

    mem_buf_lock.w_unlock()

def write_token(token):
    node = mem_buf_find(token)

    # we assume here that the maximum length of the hitlist for a token in
    # the memory buffer is not greater than the maximum BLOB length in the
    # database. it would be simple enough to split the hitlist into
    # BLOB-sized chunks here if needed.
    exec_sql("INSERT INTO index VALUES ('%s', %d, %d, %d, '%s')" % (
        token, node.first_doc_id, node.last_doc_id, node.doc_count,
        to_blob(node.hitlist)))
```

Figure 2: SYNC algorithm

has a BLOB column smaller than the maximum size (to some degree of precision, as the BLOB contents must be cut at document boundaries). While doing this it also deletes the data for updated or deleted documents from the BLOB contents. It can be stopped and later restarted at the same point, and thus it can be scheduled to run periodically during low-usage hours.

OPTIMIZE starts by collecting the list of deleted document ids from the `deleted[_mem_buf]` tables and storing them in the `being_deleted[_mem_buf]` tables. This needs to be done because otherwise we would not know which rows to delete from `deleted[_mem_buf]` when OPTIMIZE is finished, since new rows can be added to it while OPTIMIZE is running. Figure 3 contains a detailed pseudocode description of how OPTIMIZE works.

It is possible that we may discover during the implementation process that OPTIMIZE (FULL) is too slow and that we also need to provide the equivalent of Oracle's OPTIMIZE FAST. If so, that is not a problem, as it can be added without any complications.

5.3.6 Crash recovery

If the server crashes for whatever reason, the contents of the memory buffer are lost. Because of this, on server startup the FTS system must re-index all documents in the `added` table to the memory buffer. This can be done in the background and should not take that long for reasonably sized memory buffers. Whether FTS queries block or return possibly stale data during this is to be determined during implementation.

We could write redo logs for the new data but it would cause a lot of extra disk accesses and add code complexity, so it is not worth pessimizing the common case (server does not crash) to optimize the uncommon case (server crashes).

```
def OPTIMIZE():
    if not mutex_try_lock(optimize_mutex):
        return "OPTIMIZE is already running"

    if not state.last_optimize_word:
        exec_sql("INSERT INTO being_deleted SELECT doc_id FROM deleted")
        exec_sql("INSERT INTO being_deleted_mem_buf SELECT doc_id FROM deleted_mem_buf")

    # lists of document ids to delete
    d1 = exec_sql("SELECT doc_id FROM being_deleted")
    d2 = exec_sql("SELECT doc_id FROM being_deleted_mem_buf")
    dBoth = d1.union(d2)

start:
    while True:
        if optimize_stop_signaled():
            break

        exec_sql("BEGIN TRANSACTION")

        rows = exec_sql("SELECT * FROM index WHERE word > %s \
            ORDER BY word, first_doc_id" % state.last_optimize_word)

        lastCheck, combiner = get_current_time(), None

        for r in rows:
            if combiner:
                if combiner.word == r.word:
                    combiner.add_row(r)
                    continue
                else:
                    combiner.write_new_rows_to_db()
                    now = get_current_time()

                    if (now - lastCheck) > config.optimize_checkpoint_freq:
                        state.last_optimize_word = combiner.word
                        lastCheck = now
                        exec_sql("COMMIT TRANSACTION")
                        goto start

            combiner = IndexRowCombiner(r, dBoth)

        if combiner:
            combiner.write_new_rows_to_db()
            state.last_optimize_word = ""

        exec_sql("DELETE FROM deleted WHERE doc_id IN (%s)" % d1)
        exec_sql("DELETE FROM deleted_mem_buf WHERE doc_id IN (%s)" % d2)
        exec_sql("TRUNCATE TABLE being_deleted, being_deleted_mem_buf")
        exec_sql("COMMIT TRANSACTION")
        break

    mutex_unlock(optimize_mutex)
```

Figure 3: OPTIMIZE algorithm

5.3.7 Query

We will not discuss any specific query implementation strategies for reasons given at the beginning of this section. However, to be sure our design is workable we need to present a sample query evaluation algorithm that accesses all the needed data as otherwise there might be undiscovered flaws left in the design. Figure 4 contains a simple query evaluation algorithm that is not meant to be taken as anything else but a demonstration. It is given a list of words and it returns an unsorted list of the document ids that contain all the words.

5.4 Threads

Background threads are used for some operations to avoid wildly varying execution times for users' database operations. If all FTS system operations were done in the same thread as the user's query, then a document insertion might normally take 0.01 seconds as updating the memory buffer is fast, but when the memory buffer needs to be flushed to disk, the operation might take 10 seconds or even longer. To avoid this, we delegate some aspects of finishing the FTS index updates to background threads. This way the user query always finishes in about the same time regardless of how much work we need to do to update the FTS index contents.

Another advantage is that concurrency control becomes easier when we have a single specialized thread doing a complex operation instead of many threads all attempting to do the same thing at the same time.

A disadvantage is that strictly speaking the FTS system is no longer real-time. When a transaction that inserts a new document or modifies an existing one is committed, there is a slight delay before the memory buffer is updated to reflect these changes. In practise the delay is not expected to be significant; an educated guess is well under a second in normal circumstances. For document insertions this

```
def query(tokens):
    exec_sql("BEGIN TRANSACTION")

    results_disk, first_token = set(), True

    for t in tokens:
        rows = exec_sql("SELECT * FROM index WHERE word = '%s'" % t)
        tmp = set()

        for r in rows:
            tmp = tmp.union(unpack_hitlist(r.ilist))

        if len(tmp) == 0:
            results_disk = set()
            break

        if first_token:
            results_disk, first_token = tmp, False
        else:
            results_disk = results_disk.intersection(tmp)

    deleted_disk = exec_sql("SELECT doc_id FROM deleted")
    results_mem, first_token = set(), True
    mem_buf_lock.r_lock()

    for t in tokens:
        node = mem_buf_find(t)

        if not node:
            results_mem = set()
            break

        doc_ids = unpack_hitlist(node.hitlist)

        if first_token:
            results_mem, first_token = doc_ids, False
        else:
            results_mem = results_mem.intersection(doc_ids)

    mem_buf_deleted_mutex.lock()
    deleted_mem = mem_buf_get_deleted_doc_ids()
    mem_buf_deleted_mutex.unlock()

    mem_buf_lock.r_unlock()

    results = results_disk.union(results_mem)
    results = results.difference(deleted_disk.union(deleted_mem))

    return results
```

Figure 4: A sample query algorithm

delay can be ignored as the difference between a document becoming available for searching at time T or time $T + 1$ second is irrelevant. Document updates are a different matter as there the document "flickers" to a non-searchable state for a moment, which is not good for users who rely on finding the correct documents by FTS searches (imagine a Web store that returned "No items found" when a user searched for a product whose description was just updated). A potential solution is to check at the start of each FTS query whether the "Add" thread is running or is scheduled to run, and if so, delay executing the query until the "Add" thread has finished.

5.4.1 Add

The "Add" thread spends most of its time sleeping. It is awakened when a transaction that has added one or more new documents has been committed. As part of that commit, new rows were added to the tables `row_ids`, `doc_ids` and `added`. The "Add" thread goes through the documents listed in `added`, updates the memory buffer for each one, and calls SYNC whenever the size of the memory buffer goes over the user-configured maximum size. It is described in pseudocode in Figure 5.

5.5 Open issues

There are a few open issues, but most of them can only be resolved through implementation experience and measurements on a working system. They are:

- How big the BLOB column in the `index` table should be allowed to be is unknown. Experimentation will be needed to determine if e.g. a 1 MB limit has some non-obvious problems or not.

- The maximum length of a row record in InnoDB is approximately 8 KB. If a row contains columns that do not fit in the row record those columns are

```
def threadAdd():
    docs = exec_sql("SELECT doc_id FROM added");

    for doc_id in docs:
        doc = fetch_document_by_doc_id(doc_id)
        tokens = tokenize_document(doc)
        add_doc_to_memory_buffer(doc_id, tokens)

        if size_of_memory_buffer() > config.mem_buffer_size:
            SYNC()

def add_doc_to_memory_buffer(doc_id, tokens):
    # tokens grouped into per-token lists. key = token text, value = list
    # of word positions.
    hits = {}

    for t in tokens:
        if t.text not in hits:
            hits[t.text] = []

        hits[t.text].append(t.position)

    mem_buf_lock.w_lock()

    # the memory buffer must know all document ids it contains so SYNC
    # can delete the right ones ones from 'added' when it runs.
    mem_buf_add_doc_id(doc_id)

    for t in hits:
        node = mem_buf_find_or_create(t)
        tmp = pack_hitlist(hits[t], doc_id, node.last_doc_id)
        node.add(tmp)

    mem_buf_lock.w_unlock()
```

Figure 5: "Add" thread

stored externally in 16 KB blocks. It may be that InnoDB has to be enhanced to allow larger blocks to get acceptable performance for large BLOBs. It is also possible that allocating 16 KB blocks wastes too much space for smaller BLOBs and that we would need to support smaller block sizes as well.

- The way we guard against SYNC / OPTIMIZE forgetting about deleted documents is clunky and complex. It is quite possible there is a much simpler way of achieving the same end result, but we have not yet been able to find it.

6 Conclusions

In this thesis we have presented a design for adding support for a Full Text Search index for the relational database management system InnoDB. The main difference compared to existing systems is the addition of a memory buffer that caches changes to the index before flushing them to disk, which gives us such benefits as real-time dynamic updates and less fragmentation in on-disk data structures. The performance of the design, if implemented in InnoDB, can not be reliably predicted because it depends on various implementation issues discussed in the previous section.

It is worth noting that this design does not address all possible use cases for full text searches in databases. Oracle, for example, in addition to its normal FTS index support, includes a specialized *catalog* index type that is meant for Web-based e-commerce systems. Such systems differ from traditional document collections in several important ways:

- Each item's name, that is, the text to be indexed, is very short, usually a single sentence fragment.

- The search results are usually ordered by external attributes such as date (for example, the date when an auction closes at eBay).

- The results listing typically contains other external data such as the item's price.

The catalog index type stores all of the information needed to generate search result pages in the index itself, which reduces query times considerably compared to standard FTS indexes.

References

AM98 Vo Ngoc Anh and Alistair Moffat. Compressed inverted files with reduced decoding overheads. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 290–297, New York, NY, USA, 1998. ACM Press.

AM02 Vo Ngoc Anh and Alistair Moffat. Impact transformation: effective and efficient web retrieval. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 3–10, New York, NY, USA, 2002. ACM Press.

AM05 Vo Ngoc Anh and Alistair Moffat. Simplified similarity scoring using term ranks. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 226–233, New York, NY, USA, 2005. ACM Press.

apa Apache Software Foundation. `http://www.apache.org/`.

BCC94 Eric W. Brown, James P. Callan, and Bruce Croft. Fast incremental indexing for full-text information retrieval. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages

192–202, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

BCH+03 Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM '03: Proceedings of the 12th international conference on Information and knowledge management*, pages 426–434, New York, NY, USA, 2003. ACM Press.

BL85 Chris Buckley and Alan F. Lewit. Optimization of inverted vector searches. In *SIGIR '85: Proceedings of the 8th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 97–110, New York, NY, USA, 1985. ACM Press.

Boo54 George Boole. *An Investigation of the Laws of Thought on which are founded the Mathematical Theories of Logic and Probabilities*. Macmillan and Co, London, 1854.

BP98 Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

BWZ02 Dirk Bahle, Hugh E. Williams, and Justin Zobel. Efficient phrase querying with an auxiliary index. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 215–221, New York, NY, USA, 2002. ACM Press.

BYRN99 Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

CCB94 Charles L. A. Clarke, Gordon V. Cormack, and Forbes J. Burkowski. Fast inverted indexes with on-line update. Technical Report CS-94-40, Computer Science Department, University of Waterloo, Canada, 1994.

CCF+01 David Carmel, Doron Cohen, Ronald Fagin, Eitan Farchi, Michael Herscovici, Yoëlle S. Maarek, and Aya Soffer. Static index pruning for information retrieval systems. In *SIGIR '01: Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 43–50, New York, NY, USA, 2001. ACM Press.

CCT00 Charles L. A. Clarke, Gordon V. Cormack, and Elizabeth A. Tudhope. Relevance ranking for one to three term queries. *Information Processing and Management*, 36(2):291–311, 2000.

CH99 Tzi-Cker Chiueh and Lan Huang. Efficient real-time index updates in text retrieval systems. Technical Report TR-66, Experimental Computer Systems Lab, Department of Computer Science, State University of New York, 1999.

CHLS05 Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 13–22, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

Cla45 Mary Cowden Clarke. *The Complete Concordance to Shakespeare, being a Verbal Index to all the Passages in the Dramatic Works of the Poet.* W. Kent and Co., London, 1845.

CP90 Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted

index maintenance. In *SIGIR '90: Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 405–411, New York, NY, USA, 1990. ACM Press.

DDS+95 Samuel DeFazio, Amjad Daoud, Lisa Ann Smith, Jagannathan Srinivasan, Bruce Croft, and Jamie Callan. Integrating IR and RDBMS using cooperative indexing. In *SIGIR '95: Proceedings of the 18th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 84–92, New York, NY, USA, 1995. ACM Press.

Eli75 Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21(2):194–203, March 1975.

exc Excite. `http://www.excite.com/`.

FC84 Chris Faloutsos and Stavros Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, October 1984.

Gol66 Solomon W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT-12(3):399–401, July 1966.

Gus97 Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

HZ03 Steffen Heinz and Justin Zobel. Efficient single-pass index construction for text databases. *Journal of the American Society for Information Science*, 54(8):713–729, 2003.

inn InnoDB. http://www.innodb.com/.

JFS98 Björn T. Jónsson, Michael J. Franklin, and Divesh Srivastava. Interaction of query evaluation and buffer management for information retrieval. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 118–129, New York, NY, USA, 1998. ACM Press.

KR96 Mohan Kamath and Krithi Ramamritham. Efficient transaction support for dynamic information retrieval systems. In *SIGIR '96: Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 147–155, New York, NY, USA, 1996. ACM Press.

KZ98 Marcin Kaszkiel and Justin Zobel. Term-ordered query evaluation versus document-ordered query evaluation for large document databases. In *SIGIR '98: Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 343–344, New York, NY, USA, 1998. ACM Press.

KZSD99 Marcin Kaszkiel, Justin Zobel, and Ron Sacks-Davis. Efficient passage ranking for document databases. *ACM Transactions on Information Systems*, 17(4):406–439, 1999.

liv LiveJournal. http://www.livejournal.com/.

ljs LjSEEK. http://www.ljseek.com/.

luc Lucene. http://lucene.apache.org/.

LZW04 Nicholas Lester, Justin Zobel, and Hugh E. Williams. In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems. In *CRPIT '04: Proceedings of the 27th conference on*

Australasian computer science, pages 15–23, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

MB95 Alistair Moffat and Timothy A. H. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science*, 46(7):537–550, 1995.

McC76 Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, April 1976.

MM90 Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

mno mnoGoSearch. `http://search.mnogo.ru/`.

Mor68 Donald R. Morrison. PATRICIA - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, October 1968.

MS01 Albert Maier and David E. Simmen. DB2 optimization in support of full text search. *IEEE Data Engineering Bulletin*, 24(4):3–6, 2001.

mys MySQL. `http://www.mysql.com/`.

nut Nutch. `http://lucene.apache.org/nutch/`.

ope OpenFTS. `http://openfts.sourceforge.net/`.

oraa Oracle Corporation. `http://www.oracle.com/`.

orab Oracle Text. `http://www.oracle.com/technology/products/text/index.html`.

Por80 Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

pos PostgreSQL. `http://www.postgresql.org/`.

Put91 Steve Putz. Using a relational database for an inverted text index. Technical Report SSL-91-20, Xerox PARC, January 1991.

pyt Python. `http://www.python.org/`.

PZSD96 Michael Persin, Justin Zobel, and Ron Sacks-Davis. Filtered document retrieval with frequency-sorted indexes. *Journal of the American Society of Information Science*, 47(10):749–764, October 1996.

Ric79 Robert F. Rice. Some practical universal noiseless coding techniques. JPL Publication 79–22, Jet Propulsion Laboratory, Pasadena, California, March 1979.

sph Sphinx. `http://shodan.ru/projects/sphinx/`.

STC05 Trevor Strohman, Howard Turtle, and Bruce Croft. Optimization strategies for complex queries. In *SIGIR '05: Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 219–225, New York, NY, USA, 2005. ACM Press.

SWYZ02 Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel. Compression of inverted indexes for fast query evaluation. In *SIGIR '02: Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 222–229, New York, NY, USA, 2002. ACM Press.

TF95 Howard Turtle and James Flood. Query evaluation: Strategies and optimizations. *Information Processing and Management*, 31(6):831–850, 1995.

TGMS94 Anthony Tomasic, Héctor García-Molina, and Kurt Shoens. Incremental updates of inverted lists for text document retrieval. In *SIGMOD '94: Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 289–300, New York, NY, USA, 1994. ACM Press.

tre Text REtrieval Conference (TREC). `http://trec.nist.gov/`.

tse Tsearch2. `http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/`.

uni Unicode. `http://www.unicode.org/`.

wik Wikipedia. `http://www.wikipedia.org/`.

Wir77 Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

WMB99 Ian Witten, Alistar Moffat, and Timothy C. Bell. *Managing Gigabytes*. Academic Press, 1999.

WZB04 Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Transactions on Information Systems*, 22(4):573–594, 2004.

WZSD95 Ross Wilkinson, Justin Zobel, and Ron Sacks-Davis. Similarity measures for short queries. In *Fourth Text Retrieval Conference (TREC-4)*, pages 277–285, 1995.

ZMR98 Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490, 1998.

Appendixes

A Read/upgrade/write lock implementation

Implementing a read/upgrade/write lock of the type discussed in Section 5.3.4 using POSIX mutexes and condition variables is fairly straightforward. Below is an example implementation in Python that favors upgraders over writers and writers over readers when deciding who to grant the lock to.

```
class ruw_lock:
    def __init__(self):
        # mutex protecting access to all variables
        self.mutex = pthread_mutex_init()

        # number of readers waiting to enter, and their condition variable
        self.r_waiting = 0
        self.r_waiting_cv = pthread_cond_init()

        # number of upgraders waiting to enter, and their condition variable
        self.u_waiting = 0
        self.u_waiting_cv = pthread_cond_init()

        # number of writers waiting to enter, and their condition variable
        self.w_waiting = 0
        self.w_waiting_cv = pthread_cond_init()

        # number of readers active
        self.r_active = 0

        # True if an upgrader is active
        self.u_active = False

        # True if a writer is active
        self.w_active = False

        # True if an upgrader is waiting to upgrade its lock to a writer,
        # and its condition variable
        self.u2w_waiting = False
        self.u2w_waiting_cv = pthread_cond_init()
```

```
# acquire a read lock
def r_lock(self):
    pthread_mutex_lock(self.mutex)

    if self.w_active or self.u2w_waiting:
        self.r_waiting += 1

        while self.w_active or self.u2w_waiting:
            pthread_cond_wait(self.mutex, self.r_waiting_cv)

        self.r_waiting -= 1

    self.r_active += 1

    pthread_mutex_unlock(self.mutex)

# release a read lock
def r_unlock(self):
    pthread_mutex_lock(self.mutex)

    self.r_active -= 1

    # wake up (in preference order) one upgrader-waiting-to-be-writer
    # or one writer

    if self.u2w_waiting and (self.r_active == 0):
        pthread_cond_signal(self.u2w_waiting_cv)
    elif (self.w_waiting > 0) and (self.r_active == 0):
        pthread_cond_signal(self.w_waiting_cv)

    pthread_mutex_unlock(self.mutex)

# acquire an upgrade lock
def u_lock(self):
    pthread_mutex_lock(self.mutex)

    if self.u_active or self.w_active:
        self.u_waiting += 1

        while self.u_active or self.w_active:
            pthread_cond_wait(self.mutex, self.u_waiting_cv)

        self.u_waiting -= 1

    self.u_active = True

    pthread_mutex_unlock(self.mutex)
```

```
# upgrade an upgrade lock to a write lock
def u_upgrade(self):
    pthread_mutex_lock(self.mutex)

    if self.r_active > 0:
        self.u2w_waiting = True

        while self.r_active > 0:
            pthread_cond_wait(self.mutex, self.u2w_waiting_cv)

        self.u2w_waiting = False

    self.u_active = False
    self.w_active = True

    pthread_mutex_unlock(self.mutex)

# release an upgrade lock. NOTE: the normal sequence is u_lock,
# u_upgrade, and w_unlock. this is provided just for the rare cases
# when you find out you do not want to upgrade the lock.
def u_unlock(self):
    pthread_mutex_lock(self.mutex)

    self.u_active = False

    # wake up (in preference order) one upgrader or one writer

    if self.u_waiting > 0:
        pthread_cond_signal(self.u_waiting_cv)
    elif (self.w_waiting > 0) and (self.r_active == 0):
        pthread_cond_signal(self.w_waiting_cv)

    pthread_mutex_unlock(self.mutex)

# acquire a write lock
def w_lock(self):
    pthread_mutex_lock(self.mutex)

    if (self.r_active > 0) or self.w_active or self.u_active or \
           (self.u_waiting > 0):
        self.w_waiting += 1

        while (self.r_active > 0) or self.w_active or self.u_active or \
                  (self.u_waiting > 0):
            pthread_cond_wait(self.mutex, self.w_waiting_cv)

        self.w_waiting -= 1

    self.w_active = True

    pthread_mutex_unlock(self.mutex)
```

```
# release a write lock
def w_unlock(self):
    pthread_mutex_lock(self.mutex)

    self.w_active = False

    # whether to wake readers waiting to enter
    wakeReaders = True

    # wake up (in preference order) one upgrader and 0-n readers,
    # one writer, or 0-n readers

    if self.u_waiting > 0:
        pthread_cond_signal(self.u_waiting_cv)
    elif self.w_waiting > 0:
        pthread_cond_signal(self.w_waiting_cv)
        wakeReaders = False

    if wakeReaders and (self.r_waiting > 0):
        pthread_cond_broadcast(self.r_waiting_cv)

    pthread_mutex_unlock(self.mutex)
```