UNIVERSITY of CALIFORNIA
SANTA CRUZ

LARGE SCALE MULTI-TYPE INVERTED LIST INDEXING

A project report submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Joerg Meyer

March 2005

The project report of Joerg Meyer is approved:

Professor Darrell D. E. Long

Professor Scott A. Brandt

Copyright © by

Joerg Meyer

2005

Abstract

Large Scale Multi-Type Inverted List Indexing

by

Joerg Meyer

Full-text indexing using inverted lists has become the preferred method of making very large document collections searchable. Existing research has mostly dealt with the efficient representation of text indexes using a common inverted list format for all terms encountered in the text of the documents. However, in recent years, indexing has gone beyond recording just the occurrences of words in a text document. Additional data mining is being done on the document corpus and document annotations have to be indexed. As a result many of the features to be indexed and associated with documents do not exhibit the same characteristics as text and, therefore, having a common inverted list format is less than optimal. This paper presents a framework to use multiple inverted list formats and shows how this can significantly improve the disk space usage, and as a result, improve query response times by reducing I/O latencies. A priori knowledge about the characteristics of data to be indexed plays an important role in selecting optimal inverted lists formats by providing hints to the index build process. The framework was applied to the full text indexer used in the IBM WebFountain project, which allows the storage of arbitrary data with each occurrence of an index term. Using the framework presented in this paper, the index' disk footprint could be reduced by up to 12%, with some inverted lists exhibiting savings of up to 80%. The additional overhead of the framework did not lead to query performance degradation, with some response times improving by up to 75% for selected queries.

Contents

List of Figures

List of Tables

1 Introduction

Digital document collections (e.g., the World Wide Web, or a corporate intranet) play an important role as sources of information for enterprises as well as for personal use. Text retrieval systems such as search engines aid the user in finding information in such large collections. Almost all of the large scale text retrieval systems make use of compressed inverted list indexes, which are considered the most useful indexing technique for very large collections [22]. Figure 1 shows a simplified inverted list index, which can be understood as a collection of lists of occurrences for the set of unique index terms (i.e., words on a page). Occurrences are also referred as postings. The remainder of this paper will use the term *posting*.

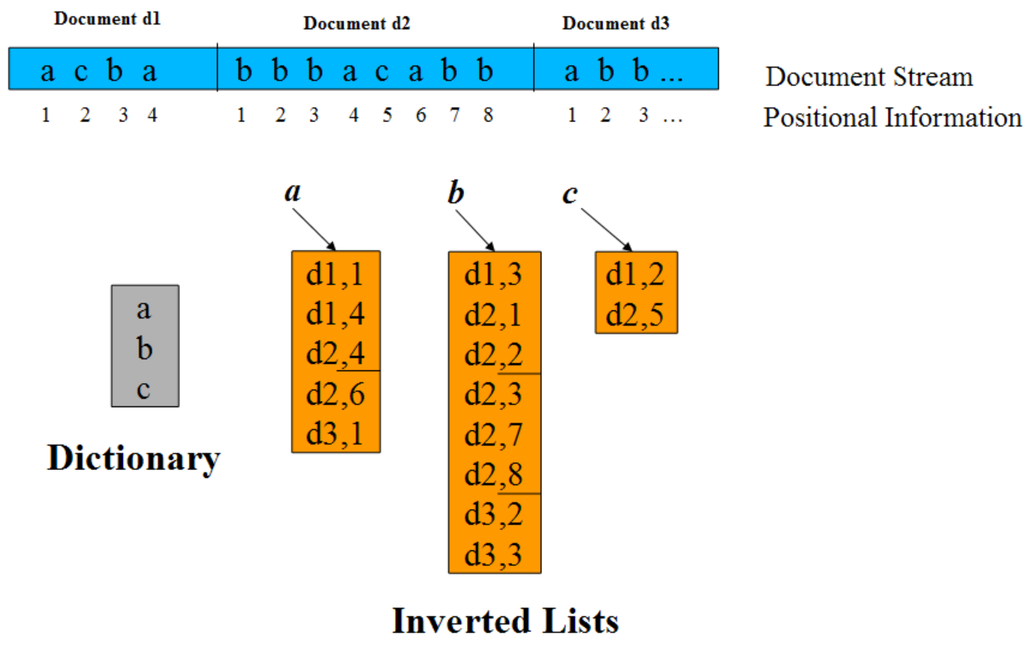

Figure 1.1: Example of an inverted list index.

A lot of research has been done in the area of compressing inverted lists for terms in the text and many compression techniques have been developed, tailored for various inverted lists formats. For example, in the simplest inverted list format a posting would just be a document identifier indicating that the document with the given id contains the term. Obviously, such lists can only be used to answer queries about containment of one or more terms in a page, but they cannot answer queries such as whether two terms appear right next to each other on a page. For such problems, extensions to inverted lists are used, so that a posting can contain information about the document and the position within that particular document. Over time though, as collections grew in size, and simple keyword matching returned too many results, more information was necessary to present the most relevant results. For example, on top of positional information on a page, Google[1] uses posting attributes such as whether the term was capitalized, part of a title, part of an URL, or an anchor[2] [5]. Such information can be used to give certain postings a higher weight.

Other features of a document, such as the language it was authored in, or the domain it belongs to, are indexed to give the users more control on what part of the document collection to search through. For such document meta-data, an inverted list format that contains positional information is overkill. If not all of the features of an inverted list format are necessary for such terms, the use of escape sequences[3] in inverted lists allows for distinguishing between different kinds of postings. However, escape sequences still require extra space and have generally negative effects on query performance as every hit needs to be checked for escape sequences in order to decode it properly based on the type of posting.

IBM's WebFountain[4] project uses a full text indexer and query engine to provide a text retrieval system to a wide variety of applications, such as *Google*-style regular search or classifica-

[1]http://www.google.com
[2]An anchor is the text inside the <A>tag in HTML pages.
[3]A common technique for escape sequences is to use a few bits in one or more data items to indicate to the reader of the data what to read next.
[4]http://www.almaden.ibm.com/webfountain

tion of sets of pages [12]. An important feature of WebFountain's text retrieval system is the ability to do advanced queries like the following:

> *Find me all pages that contain the terms **Joerg** and **Meyer** and that were crawled during the last week.*

Another potential query could be:

> *Find me all pages that contain the terms **Joerg** and **Meyer** and pages with URLS that match the regular expression ***.ibm.com/people/***.'*

Such support is achieved by not only indexing text but also augmentations produced by so-called document miners. Examples for such miners are language detection, geo-spatial location detection, name spotting, classification, to name only a few. For many of these the produced information does not refer to a particular position on a page but rather the whole page. Additionally, many of these terms require the storage of arbitrary length data with each posting, similar to storing text attributes as used in Google. For that purpose, WebFountain started out with a very generic indexer that provided one inverted list index format in which each posting consisted of a location (i.e.,document id and position in document) and data, which in turn consisted of a length and a sequence of bytes. The benefits of such an approach are the ease of implementation and code maintenance and that it can support virtually all needs for storing various pieces of information within a posting. However, with growing index sizes, the one-format-fits-all philosophy results in sub-optimal representations of postings lists. It was also observed that, with very few exceptions, all postings for an index term are of the same nature, meaning that either all of the postings had a variable length datafield or they had a fixed length datafield. Additionally, as part of the evolution of the project, new requirements arose that went beyond handling textual input data.

To address the problem of increased secondary storage for the index, as well as decreased query performance, the WebFountain indexer was modified to take as input not only the name of the data source to be indexed but also of what type the input data is and what its needs for an inverted list are. To avoid having to deal with escape sequences, the modified indexer was restricted to one inverted list format per unique term, which allows the efficient selection of an inverted list reader depending the type of the token. The modified indexer uses significantly less space by avoiding the compression and storage of unnecessary information. Furthermore, it represents an easily extensible framework in which new inverted list formats can be added or existing ones be replaced.

The remainder of this paper provides a more detailed illustration why the existing solution is less than optimal in Section 2. Section 3 describes the overall system design chosen to enhance the existing indexer to make use of token types and provides details about the implementation. Preliminary results and a quantitative analysis of improvements is given in Section 4. The paper concludes with a a brief overview of related work in the area of compressing inverted lists in Section 5 and closing remarks as well as an outlook on future direction in Section 6.

2 Motivation: One-Fits-All Formats are Sub-Optimal

2.1 Existing Index Build Process

The WebFountain indexer was designed for indexing large document corpora. Indexing large corpora of millions of documents generally means that the processing cannot be fully done in memory, even on machines with large amounts of RAM. Therefore, the index build process needs to interact with secondary storage (e.g., hard disk or attached storage). The overall process of indexing is the recording of all postings of an index term. At the end of this process, each unique term in the document corpus is represented by an inverted list[5]

[5]Postings in inverted lists are sometimes as postings and inverted lists as as postings lists.

For example, as of February 2005, individual indexers in the WebFountain production environment process between 10 and 20 million pages, with a number of individual index terms between 150 to 250 million. Holding all terms in-memory during processing would virtually exhaust all available memory, leaving no room for the actual lists of postings. Therefore, the index build process is a multi-step process, consisting of the following phases[6]:

1. Build an in-memory index, which is fundamentally the same as an index stored on disk with the distinction of covering fewer documents.

2. Temporarily storing the in-memory index on disk.

3. Merge all temporarily stored in-memory indexes into the final index.

As shown in Figure 2.1, the first two steps are repeated until all documents are processed, followed by the third and last phase. The process is similar to the hierarchical merging of partial indexes as described by Baeza-Yates et al. [1]. Merging is necessary, because in WebFountain, often times the order in which documents are indexed does not reflect the order in which inverted lists should be stored [14].

The inverted list format within a WebFountain index is a very generic format, with each posting in an inverted list consisting of a *location* and a *datafield*, as shown in Figure 2.2. The datafield consists of a length l and the sequence of l bytes. A location in memory is fixed-size number, i.e. 32 or 64-bits wide. On disk, the locations are encoded using d-gaps, which basically means that locations are represented relative to their previous location [22]. Such deltas are generally relatively small and can be encoded using very few bits. For ease of implementation, the WebFountain

[6]The mentioned phases are used in a static index build process, which processes all documents first and then produces a usable index. Incremental indexers differ from this method by incrementally making more documents searchable. Their drawback is that processing the same number of documents generally takes longer.

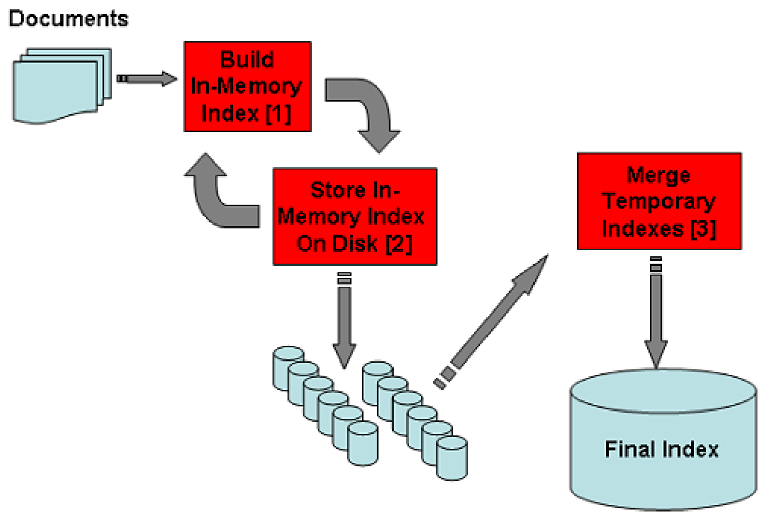
Documents
Build In-Memory Index [1]
Store In-Memory Index On Disk [2]
Merge Temporary Indexes [3]
Final Index

Figure 2.1: Steps in an index build process.

indexer uses a byte-level encoding, i.e. it always uses a multiple of 8 bits to encode a delta [14]. The WebFountain indexer compression scheme falls into the category of non-parameterized models. Parameterized models like global and local Bernoulli models take into consideration density of postings on a per corpus or document level [22].

In the WebFountain system, documents are stored in an XML[7]-like representation as shown in Figure2.3.

All of the keys under the `DOC` node can be indexed treating it as textual data which maps nicely to the generic inverted list format. An indexer is simply given a list of XML element names to look for within each `DOC` element, with minimal input on how the data was to be interpreted.

[7]http://www.w3c.org/XML/

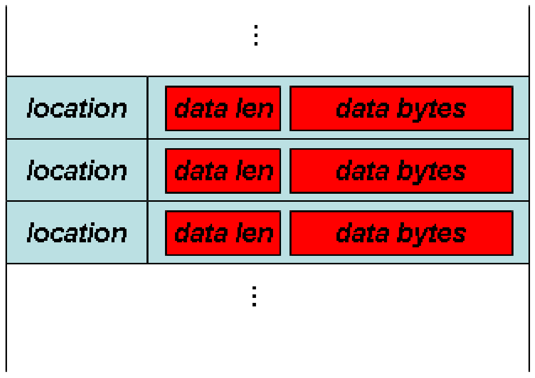

Figure 2.2: Section of a WebFountain index inverted list for a single term.

```
<DOC>
      <ID>00006AF98...</ID>
      <CONTENT> ... </CONTENT>
      <URL>http://www.almaden.ibm.com</URL>
      ...
<DOC>
```

Figure 2.3: Sample WebFountain document.

Clearly, automatically determining how the data should be interpreted, e.g. as a number or a sequence of characters, can often be ambiguous.

2.2 Problems

The indexer's inverted list is not the most efficient representation but it has proved appropriate in the beginning when indexes contained up to 10 million pages and the amount of additional data to be indexed remained fairly small. However, as the WebFountain system ingested more documents, index sizes grew beyond a point, where inefficiently compressed inverted lists cause higher than tolerable I/O latencies. Furthermore, as the project continued to grow, so did the number of data miners that annotate documents. As mentioned in the introduction, there are many annotations now that require data to be stored with an posting. For example, a *date miner* may detect sequences of text on a page that represent a date and a time and insert a *date term* at the position in the document, with the date represented as a 32-bit number. Even though all postings of this term have a datafield of length 4 (in bytes), the indexer stores the length for every posting in the inverted list. Other miners, such as the *language miner* determine attributes of the entire document. In most cases, positional information within the document or datafields are not necessary. Yet, with the existing format, some escape sequence needs to be used to indicate that a datafield is actually not present. Making the escape sequence part of the location cause problems with the delta encoding, therefore, extra space is *wasted* for each posting. Escape sequences have other additional drawbacks because they require to continually check a condition, which slows down the decoding of the inverted list postings [22]. Generally, all postings for any of the terms to be indexed require the same format.

As pointed out earlier, every indexable item of a WebFountain document is treated as text and can be mapped to the generic format. However, this is somewhat equivalent to the idea of storing a 32-bit number in a database using BLOBs[8].

Based on the observation that the majority of terms need only one posting format but not all terms need the same, the task is now to extend the WebFountain indexer to allow for multiple inverted list formats. One of the main objectives is to reduce the on-disk footprint of the entire index or selected inverted lists as that will have a positive impact on query performance. Another important goal is to provide an extensible framework in which new inverted list formats can quickly be tried without requiring major rewrites of large portions of the existing indexer code base. Last

[8]BLOB = Binary Large Object.

but not least, trying to reuse as much code as possible of the existing indexing process in terms of steps 2 and 3, was a desired goal of the entire process.

The fundamental idea is to assign a token[9] type to each index term and propagate this through the index build process. The best token type for an item to be indexed is derived from hints about the input data. Each token type has an unambiguous format, so that no escape sequences will be necessary in the decoding of inverted lists for terms. Having the most-optimal inverted list format for a term, allows us to reduce the on-disk footprint of the entire index. The hints about input to be indexed are provided by the user through an index build specification, that is extensible. The next section provides a detailed description of the new input format, how hints are converted into a token type and how the secondary goals of reusing existing code as much as possible, as well as providing an extensible framework, were achieved.

3 Design and Implementation

The previous section highlighted some of the problems associated with the existing inverted list format used in the WebFountain indexer. This section reiterates some of the design goals and present the solution implemented.

3.1 Design Goals and Constraints

The primary goal in designing a more flexible indexer supporting a variety of inverted list formats was to reduce disk storage use, which in turn would improve query performance by reducing I/O latencies. Aside from that goal, other goals and constraints played significant roles:

[9]In WebFountain index terms are referred to as tokens.

- Create a solution that can easily be extended later on to facilitate research and development of optimized inverted lists for special purpose index terms.
- Avoid a complete rewrite of all the indexer code, which contains the code to perform the index build process as well as code to do the query processing.

The overall structure of the indexer code can be separated into four separate parts, the document consumer, the in-memory indexer, the merger, and the query engine. The document consumer portion of the code is responsible for ingesting the documents, split them into the indexable items and insert the index terms into the indexer. The indexer portion of the code handles the construction of the in-memory indexes, the output of partial indexes, while the merger processes all partial indexers into a final index. The query engine uses the final index to provide search engine functionality. In trying to avoid a major rewrite, the goal was to leave the query engine and in-memory indexer virtually untouched and only modify the reader module in the document consumer as well as the writer module in the merger. This was achieved by introducing token types, create a class hierarchy for posting serializers and a new module to ingest an index build specification.

3.2 Modifications to the Index Build Process

The existing WebFountain in-memory indexer had one basic API to record an posting of a term:

```
bool insert( Token* indexTerm, Token* data, Location loc );
```

The class `Token` is basically a wrapper class around a sequence of bytes and a length field. This is done for reasons of allowing the indexing of arbitrary byte sequences and not rely on null terminated strings. It has one method that serializes the actual data of an instance using:

```
void toBytes( ... );
```

The in-memory indexer keeps track of all encountered terms in a set of documents in its internal in-memory data structures. When the pre-allocated data structures are full, a partial index write to disk is initiated. Index terms are written using the above mentioned `toBytes()` method. The concepts of polymorphism and inheritance enabled the in-memory indexer to now index terms that also carried a type.

Token Types

As mentioned in the previous sections, only one posting format is generally necessary per unique index term. Therefore, a new type called `TokenType` was introduced, which was derived from the base class `Token`. That meant, that existing APIs did not need to be changed in the in-memory indexer. Even the writer modules that write out partial indexes remained unchanged, as the respective token object provides an API for serializing itself into a memory buffer. Adding a member indicating the token type and overloading the `toBytes()` method on the `TypedToken` class was all the coding work necessary to make the in-memory indexer support token types.

Intermediate Data Format

The decision to leave the in-memory indexer unchanged meant that any term that was indexed used the existing very generic format of location and datafield. This concept of having an intermediate data format allowed us to use the in-memory indexer as-is. Therefore, we now needed for each token type a method to turn input data into the intermediate format, as well as a mechanism to turn the generic intermediate format into an posting of a specialized inverted list format (see Figure3.1). For the proper conversion of input data into the intermediate format, most changes were necessary in the document consumer. For the problem of turning the intermediate format into an

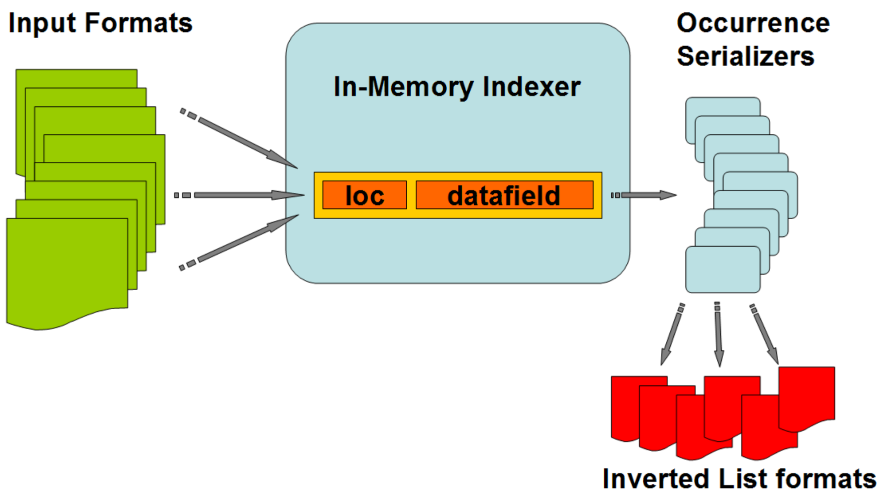

Figure 3.1: Intermediate posting format.

inverted list format, the concept of a `PostingSerializer` was added, one per unique token type.

Document Consumer Changes

Historically, all input to the WebFountain indexer was considered to be textual data. For that reason, the APIs were basically oriented towards the support of indexing textual data in various forms, such as arrays of index terms (pre-tokenized[10] content), strings, or the raw text of a document. The basic choices for interpreting input elements of a WebFountain document, were to index the contents of such an element, or to index the content in the datafield, or to index the content as

[10]Tokenization is the process of splitting a document into individual terms.

an individual key. For example, indexing the elements `LANGUAGE` and `CRAWLDATE` out of the following document

```
<DOC>
  <ID>000064...</ID>
  <LANGUAGE>En</LANGUAGE>
  <CRAWLDATE>124578965</CRAWLDATE>
  <CONTENT> ... </CONTENT>
  ...
</DOC>
```

could be done in one of two ways. For the element *LANGUAGE*, a good choice is to index the name of the element and its value as one term *Language_En*. This implied a datafield length of 0. Another option is to use the element name as the term and put the value into the datafield of a posting. The latter alternative is a good choice for indexing the *CRAWLDATE* element as it allows searching for ranges of dates in large indexes. Aside from generic keys like the aforementioned language and crawl date example, WebFountain documents provide pre-tokenized content. Basically, aside from the above mentioned options of associating an input element with a way to index the information, there were no other ways of specifying other characteristics of the input data. This limited but very simple interface led to a very simple specification of index inputs. In order to take advantage of a more sophisticated indexer supporting various inverted list formats, a mechanism to provide an indexer with more hints about the inputs was needed, such as whether the input requires positional information within the document, whether any datafields are used, or whether datafields have fixed lengths.

Th chosen solution is a fairly simple language based on XML that allows the creation of index build specifications. A simple example of such a specification would look as follows:

An `indexBuildItem` in the context of WebFountain refers to an element under the `DOC` element in a WebFountain document. In addition to specifying the name of the element using

```
<indexBuildSpecification>
  <indexBuildItem>
     <typeName value=''CONTENT''/>
     <format value=''SLV''/>
     <indexRule>
        <style name=''Positional''/>
        <style name=''textattribute''/>
     </indexRule>
  </indexBuildItem>
  <indexBuildItem>
     <typeName value=''CRAWL_DATE''/>
     <format value=''NUMBER''/>
     <indexRule>
        <style name=''data32''/>
     </indexRule>
  </indexBuildItem>
  ...
</indexBuildSpecification>
```

Figure 3.2: Sample WebFountain index build specification.

the `typeName` element, the `indexBuildItem` element specifies the input format of the item, as well as a rule (`indexRule`) on how to index the inputs. An `indexRule` can have multiple styles. For example, in the sample index build specification shown in Figure 3.2, the element `CONTENT` is described with a format of `SLV` and an index rule using the styles `Positional` and `textattribute`. This means that the format of the content of the element named *CONTENT* in a WebFountain document is of the format *SLV*, which is a special compact binary format that can be understood as vector of index terms with associated positions on the page. In this case the indexer itself does not need to do any tokenization. The `Positional` style indicates that every item in this vector needs to be indexed with positional information. The `textattribute` indicates that each item in the vector also contains a 1 byte fixed-length datafield. Therefore, the needs for an inverted list for this type of input is to store a location with in-document position followed by a 1 byte datafield. The other element illustrated in the sample index build specification in Figure 3.2 is an example for which the input element can be interpreted as a number and no positional information is necessary. Furthermore, the datafield to be used for the posting is a 32-bit number. Therefore, an posting in an inverted list suitable for this kind of input needs to consist of a location without positional information and a fixed-length datafield. As one can see, in both cases the previously used length for the datafield becomes unnecessary to store in the inverted list. All that needs to be done is to use an appropriate token type to select an object that knows exactly how to read the inverted list for a particular type. These objects are called *posting serializers*.

3.3 Merge Process Changes

The changes to the merge process were relatively simple. In the previously existing merge process, one writer object (`PostingsWriter`)was used to write all inverted lists to disk. For each unique terms, all postings were written using the the

```
writePosting( LOC* loc, Token* dataField )
```

interface. The simplified sequence of code to write the postings for an index term from a partial index to an inverted list is shown in Figure 3.3. For clarity the code that handles all partial indexes was omitted.

```
for each unique term do:
   add term to dictionary
   for all postings of the current term do:
     writer.writePosting( posting.loc, posting.data );
   end
end
```

Figure 3.3: Pseudo code for index build merge process.

The only places in the merger code base that needed to be changed was the basic algorithm as shown in Figure 3.3 and the implementation of `writePosting` method. The changes to the merge process consisted of using the token type of the current term to select the appropriate posting serializer and pass this serializer to the postings writer object. The modifications in the `writePosting` method consisted of using the serializer object to write the posting to an output buffer or device. Figure 3.4 shows the modified merge process.

```
for each unique term do:
    add term to dictionary
    select posting serializer based on token term
    writer.setSerializer( serializer )
    for all postings of the current term do:
      writer.writePosting( posting.loc, posting.data );
    end
end
```

Figure 3.4: Pseudo code for modified index build merge process.

For the selection of the serializer based on a term's token type, we chose the concept of a *PostingSerializerFactory* based on the factory design pattern described by Gamma et al. [10]. Selecting a serializer through the factory is done through a single API like

```
PostingSerializer* getSerializer( TokenType tt );
```

and allows us in the future to easily add new token types and their serializer implementations by simply updating the Factory implementation. The basic algorithm to build the index (in-memory indexer and merger code bases) can remain unchanged.

3.4 Posting Serializers and Query Engine Changes

The concept of an abstract *posting serializer* allows us to keep the basic algorithms for building an index unchanged when adding or trying new inverted list formats. In the case of Web-Fountain an posting serializer implements a very basic API, shown in Figure 3.5. The serializers

```
class PostingSerializer {
   public:
      int serialize( Loc* loc, Token* data, char* buffer );
      int deserialize( char* buffer, Loc& loc, Token& data );
};
```

Figure 3.5: Abstract Posting Serializer API.

are used for writing term postings out to a buffer (`serialize` method), as well as reading from a buffer back into memory (`deserialize` method). Both methods return the number of bytes written to or read from an i/o buffer (`buffer` parameter). The `deserialize` method also returns the location and a reference to the datafield in the parameters. For clarity, helper methods to determine whether there is room in the i/o buffer have been omitted. Using this basic API, the changes to the PostingsWriter's `writePosting` method were then limited to replacing the serialization of the individual pieces of an posting (i.e., location and datafield) inside the `writePosting` method with a call to the serializer's `serialize` method. Section 4 illustrates how this basic framework was used to achieve index builds using multiple inverted list formats.

Currently, WebFountain has a wide variety of index terms in the system. Table 3.1 reiterates samples of different inverted list needs for different kinds of index terms[11]. An *Entity* refers

[11]The term *Stop-words* in table 3.1 refers to very frequently occurring terms. For example in the English language, terms like *the*, *and* are considered stop-words.

Token Type	Location	Datafield	Extra	Example
Text	positional	1-byte fixed	none	Words on a page.
Text	positional	none	none	punctuation, Stop-words
Meta-Data	doc only	none	none	Language, Mime-Type, etc.
	doc only	4-byte fixed	none	Dates, Ranks, Sizes, etc.
	doc only	variable length	none	URL, Title
Entities	positional	var. length	span	Person, Geo-Location

Table 3.1: Sample of inverted list needs for different index terms.

to a sequence of words within the document and requires additional information in each posting. The additional information is called a span, which denotes the number of tokens in the sequence of tokens within the document the entity refers to.

For each of the rows in the table an individual serializer had to be written. Overall, the implementation effort for creating the set of individual serializers consumed a relatively small portion of time, because a lot of the serializers share common behaviors. So far, we have implemented 8 different serializers for being able to write up-to 14 different inverted list formats. Figures 3.6 and 3.7 illustrates the class hierarchy of the various serializer classes. The class hierarchy shown in Figure 3.6 is used to write postings for terms that require positional information onto the page. The *64* in the names of the serializers refers to the fact that the WebFountain indexer uses 64-bit numbers in-memory to represent the two *d*-gaps (i.e., one for document numbers and one for intra-document positions). On the other hand, the class hierarchy shown in Figure 3.7 is used to read and write postings for index terms, that do not require positional information. Serializer subclasses with *DataX* as part of their name in either hierarchy are used to write postings that have fixed length datafields.

Similar to the changes in the merger code base, minimal changes to the query engine were necessary to be able to read multiple inverted list formats. The existing query engine used

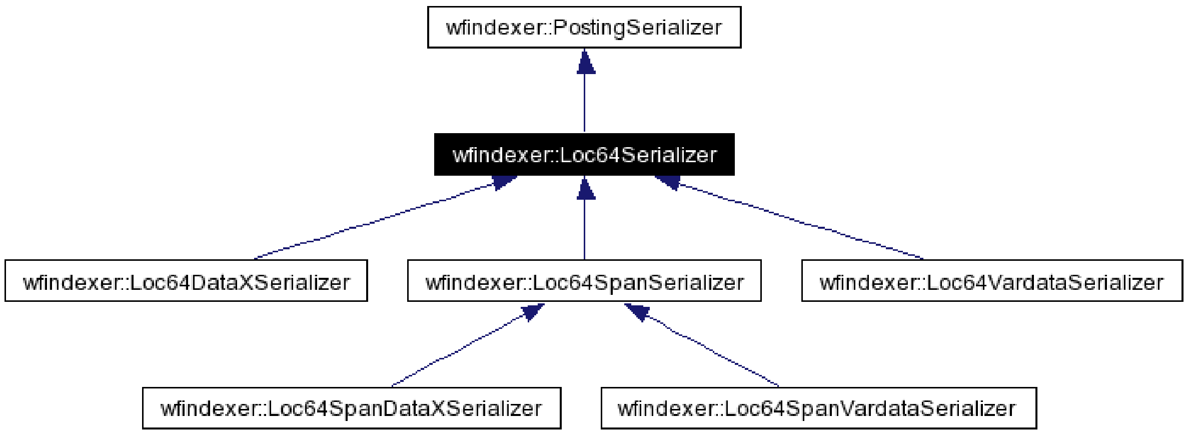
wfindexer::PostingSerializer
wfindexer::Loc64Serializer
wfindexer::Loc64DataXSerializer
wfindexer::Loc64SpanSerializer
wfindexer::Loc64VardataSerializer
wfindexer::Loc64SpanDataXSerializer
wfindexer::Loc64SpanVardataSerializer

Figure 3.6: Posting Serializer class hierarchy for terms requiring positional information.

one class to access inverted lists on disk. The implementation of that class was changed such that the deserialization of an posting out of an inverted list was now done calling the `deserialize` method of the appropriate serializer object. The selection of the appropriate serializer object was achieved by using the same factory. The token type necessary for selection of the correct serializer was read from the dictionary entry. The dictionary of the WebFountain index contains the set of unique terms. With each of the terms, additional information is stored in a dictionary entry, such as the pointer to the inverted list, the number of entries in the list and, as part of the changes of the work described in this paper, the token type of the index term. When performing a query, the query terms are checked against the dictionary. If they exist, objects for accessing inverted lists are

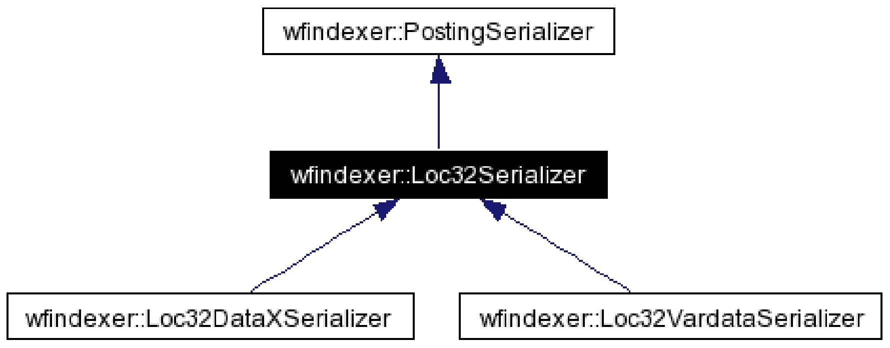
wfindexer::PostingSerializer
wfindexer::Loc32Serializer
wfindexer::Loc32DataXSerializer
wfindexer::Loc32VardataSerializer

Figure 3.7: Posting Serializer class hierarchy for terms without positional information.

created. The creation of these object now includes the selection of the posting serializer based on the token type read out of the dictionary entry.

3.5 Implementation Details

The WebFountain indexer and all of the aforementioned changes were implemented using standard C++. The compiler used was gcc, version 3.0.3. For parsing the XML index build specification, the Xerces[12] package, version 2.6.0 was used. The identification of token types was done through an `enum`. By implementing the serializers as share-nothing objects, only one instance of a particular serializer was necessary for each unique token type. Accessing the appropriate serializer is done through a table lookup using the token type to index into an array of `PostingSerializer` objects. This incurs virtually no overhead during the build process.

[12] http://xml.apache.org/xerces-c/

4 Applying the Framework and Experimental Results

The previous section laid out the changes necessary to allow for the use of multiple inverted list formats inside the WebFountain indexer. The primary goal of this undertaking was to optimize inverted lists for various kinds of index terms. In order to determine the usefulness of the approach discussed in this paper, an index build of the existing WebFountain indexer with an was copared with an index build using the new multi-inverted list format indexer. The results and findings of this experiment are presented in this section. Results referring to the existing WebFountain indexer are labelled *current*, results referring to the new approach discussed in this paper are labelled *new*.

4.1 Test Data

A random selection of 2 million Web documents from the WebFountain cluster served as the test data set. That data was indexed with both versions to compare their performance in terms of disk usage, index build times, and query performance. Unless otherwise noted, all results refer to tests using the indexes covering all 2 million documents. Both index builds indexed the same data as stored in the WebFountain cluster. The set of 2 million documents contains approximately 41.5 million unique index terms. Approximately 1.2 million of these pages were detected to be English documents, the remaining documents covering languages such as German(114K), French(73K), Chinese(65K), and Arabic (1K).

4.2 Reduction in Index Size

The first test was to determine how the new framework can reduce the final index size on disk. For this first test, the mechanism was simply for not storing positional information for

	File Size$_{current}$	File Size$_{new}$	Improvement [%]
Dictionary	1.69GB	1.61GB	95%
List Descriptors	0.71GB	0.58GB	82%
Inverted Lists	5.5GB	4.82GB	87%
Total	7.9GB	7GB	88%

Table 4.1: File Sizes for index builds of 2 million documents.

all index terms that fall into the category of meta-data. Such terms include dates (e.g., the date a page was collected, authored or last modified), MIME types, and languages to name a few. For the index terms that referred to words within a document, the new framework does not offer any improvement because the serialization of postings of a word in a document still requires the writing of two d-gaps and a 1-byte text attribute. This is identical to the currently used indexer because an escape sequence was used that allowed to have 1 byte datafields without storing a length. The benefit of the new format is that checking for this escape sequence now becomes obsolete. The other major feature that was used for this initial test was to declare some inputs as numbers, to take advantage of fixed-length datafields without having to write out the length of the datafield. With these minimal changes, the index footprint on disk could be reduced by approximately 12% for the entire index and slightly more than 13% for the inverted lists alone. Table 4.1 shows the file sizes for both index builds. The column labelled *Improvement* indicates the percentage of space used by the new indexer in comparison to the currently used indexer. Translating this to the WebFountain production environment which has indexes covering approximately 15 million pages, the savings for this test would translate to savings of roughly 7GB for a final index.

Index Build times could be improved slightly by having to write less data in the merge phase. Improvements were not expected in the document consumer and in-memory indexer portions of the code because the basic algorithm remained unchanged.

4.3 Reduction for Individual Index Terms

More pronounced than the savings in overall index size are the savings for some of the index terms used in WebFountain indexes. As previously mentioned, the WebFountain data miners augment documents by adding information about them. For example, the crawler[13] adds information on when the page was crawled. Classification miners add tags to the document, on whether the page falls into certain categories or not. Another miner produces various hashes of parts of the content of the document, similar to the idea of producing shingles as described by Broder [6]. These shingles are treated as regular terms in a WebFountain index. For the aforementioned terms, a significant saving could be achieved by only having to encode the fact whether it occurred on the page and not where on the page. Some of the terms shown in Table 4.2 are examples of taht kind. As before, the column labelled *Improvement* shows the percentage of space used by the new inverted list format compared to the currently used format. The sizes of the inverted lists are given in the number of 4KB blocks. For terms like `Language_En`, more savings could be achieved, by not having to write a datafield at all. For terms like `CrawlDate`, it is advantageous that all postings had a datafield with length 4. The term `[[docid]]` is used to store the actual document identifiers in an inverted list. Document identifiers in WebFountain are 16-byte MD-5 hashes of the URLs of the documents. In this case, the inverted list size could be reduced by using a format that store document d-gaps only and uses fixed-length 16-byte datafields. Some of the more significant savings stem from the previous approach of treating everything as text without any knowledge about the nature of the data. The term `[[paragraph]]` describes postings of paragraph boundaries. This allows the support of queries that require terms to appear within a sentence or paragraph. For those terms, positional information was still used but the 1-byte text attribute was eliminated.

[13]The crawler is the component in the WebFountain system that collects Web documents.

Term Name	Inverted List Size$_{current}$	Inverted List Size$_{new}$	Improvement [%]
Language_En	1983	332	16%
CrawlDate	9167	2704	29%
[[docid]]	20815	9166	44%
[[paragraph]]	43026	22548	52%

Table 4.2: Selected inverted list sizes for a 2 million document index.

4.4 Stop-Words – Very Frequently Occurring Terms

In the current WebFountain indexing environment and results illustrated in the previous sections, all index terms that appear within a document use an inverted list format of positional information as well as a fixed-length 1-byte datafield. This datafield is mainly used to store postings attributes, such whether the posting is part of a title, anchor, heading or other distinguishing document features. The assumption in the past was that all terms are equal in terms of indexing them. However, not all terms are equal when it comes to the number of occurrences or their importance during query time.

One such group of index terms that is generally not considered as useful as other terms is the set of stop-words [1]. Examples of stop-words in the English language are articles, prepositions and conjunctions. Table 4.3 shows the 10 most frequently occurring terms and their number of postings in the 2 million test index. The general idea is that a stop-word used by itself as a query term is a poor choice to search through a large document collection. However, they are still useful and even necessary, if the text retrieval system supports phrase searches. One possible optimization for indexing such stop-words is to eliminate storing the text attributes for such stop-words, which would save 1 byte per posting. Applying this approach by using a separate token type for stop-words, approximately 50MB alone could be saved for the 10 terms shown in Table4.3. For the 230 most frequently occurring terms in the 2 million documents appearing at least 250000 times, this

Term Name	Term Frequency
the	11469882
to	6357260
and	6281822
of	5857122
a	5504061
in	4054749
for	3332138
is	2350735
de	2290000
you	2251109

Table 4.3: Top 10 list of stop-words in terms of frequency in a 2 million document index.

would save approximately 170MB, which represents approximately 2.5% of the entire space needed for storing the inverted lists for such terms.

Aside from needing to have a PostingSerializer implementation that only writes positional information, the document consumer code base needed to be modified to detect stop-words. Possible implementations to check index terms for belonging to the set of stop-words is the use of a Hash table or a burst trie [18, 24].

4.5 Query Processing Performance

With the reduction of the sizes of inverted lists on disk come other benefits besides saving space. On of them is the reduction of I/O latency. One of the fundamental processes in query processing is the enumeration of the postings in an inverted list. Some tests were run enumerating over all postings for certain terms in the inverted lists.

For most index terms that correspond to text, the enumeration times were virtually identical to the enumeration times of the existing indexer. For example, all 11 million occurrences of the term *the* could be enumerated in about 8 seconds (i.e., deserializing approx. $1375\frac{postings}{ms}$. The elimination of checking for escape sequences has not had a significant effect on improving performance, on the other hand introducing a class hierarchy to do the serialization has not hampered performance due to using virtual functions. Proper inlining of the serialization methods and careful optimizations can further improve the new indexer enumeration times over the existing version.

For the terms listed in Table4.2, enumeration time savings between 10 and 15% could be observed. The enumeration tests were done repeatedly on a warm file system cache, and the reported savings are averages over 10 runs.

As mentioned in the introduction, WebFountain supports advanced queries like the following:

> *Find me all pages that contain the terms* ***Joerg*** *and* ***Meyer*** *and that were crawled during the last week.*

Another potential query could be:

> *Find me all pages that contain the terms* ***Joerg*** *and* ***Meyer*** *and pages with URLS that match the regular expression* ****.ibm.com/people/**** *.'*

The datafield for postings plays an important role to support these kinds of queries. In particular, limiting a query to the recently ingested pages is a very frequently used operation in WebFountain. For example, looking for documents that contain the index term *joerg*, the indexes returned 175 hits in virtually the same amount of time. Adding the additional restriction to only return pages that were crawled within a very small time window, the new multi-inverted list indexer more than twice as fast. This is mostly due to the elimination of expensive string conversion operations but also due to the reduction in space on disk. The term *joerg* is a relatively infrequent term in the index. Using a date range query as described with query terms that occur more frequently will cause more range checks. Savings in date range queries of up to 80% in runtime could be observed. The application of an inverted list format for terms with very long datafields, such as URLs or entities, is currently under evaluation.

Overall, most queries run within the same time frame or slightly faster using the multi-type inverted list index. Using a class hierarchy and virtual functions limits the ability to inline frequently used code sequences. Lea showed that this can cause significant performance problems [13]. Known techniques to improve performance are the use of inlining by eliminating virtual functions where possible and make use of templates instead. In our case, the primary goal was to create a clean framework that can be used to create and try new inverted list formats quickly.

5 Related Work

Formats for inverted lists have been discussed in the research literature at great length. Most compression schemes are based on the idea of sorting the inverted lists in ascending order of document numbers and positions within the documents and then replace the full information of a posting with a d-gap representation relative to the previous posting. Witten *et al.* reviewed the basic techniques and state that for the majority of practical purposes, the most suitable index compression technique is the local Bernoulli method using Golomb coding [22]. There has to be a trade-off between the computational compression and de-compression overhead and the amount of I/O necessary to load an inverted list. In order to avoid decompressing all previous postings to get to any posting in the inverted list, auxiliary information is added to the list, fundamentally dividing the list into blocks [20]. The WebFountain indexer uses the blocking technique and a compression scheme that falls into the category of non-parameterized models. The compression scheme is a simple byte-aligned variable length integer encoding scheme which has been shown to be more efficient during query time although it is not as space-efficient as Golomb codes [16]. The work presented in this paper provides an easy to use framework for evaluating the various approaches in terms of compression ratio and query evaluation times.

Alternatives to using inverted lists are bit vectors, signature files and block addressing inverted indexes. Zobel et al. and Witten et al. showed that inverted lists are the most appropriate for large collections [25, 22]. The WebFountain index is designed as an inverted list index, but the framework described in this paper could be used to have a bit-vector token type. While a bit-vector can be viewed as a form of an inverted list with at most one posting per document, simulating signature files with this framework is not easily feasible because they represent a fundamentally different approach than inverted lists.

Most research has dealt with the compression of inverted lists for index terms that represent single words within a document. Williams et al. present an approach to make phrase queries more efficient by combining next-word indexes with indexing phrases [21]. Commonly available features in search engines also include the use of ranges and site names. However, generally the additional index terms are treated as texts and so are their inverted lists. The multi-type inverted list index presented here allows for the integration of an optimal inverted list format. IBM recently released the *Unstructured Information Management Architecture SDK*, which allows the analysis and annotation of unstructured information (e.g., web documents) and provides a framework to build an index of the annotation and analysis results [15].

The index build process described in this paper assumes a single node index build with currently up to 16 million pages per machine. Clearly, web scale indexes need more than a few machines. A variety of approaches to indexing large corpora of documents on a cluster of machines have been proposed. Common techniques include *term partitioning* and *document partitioning* schemes, which have one node in the cluster either store the complete inverted lists for a set of terms (term partitioning) or the inverted lists of all unique terms in a set of documents. Sornil proposes a parallel inverted list-index which use a hybrid-partitioning scheme [17]. Google uses 10000+ machines to distribute a search engine [2], but uses replication through the Google File system [11]. Either partitioning scheme can be used with our proposed framework because it is applied on the level of the inverted lists and not at the level of document or term distribution.

As a reference implementation the *static index build* process was used. Static indexing schemes can generally only be updated by re-indexing the entire set of documents. *Incremental* or *dynamic indexing* schemes allow for updates without having to revisit all the pages. Instead they add, extend or modify inverted lists into an existing index. One of the main concerns with incremental indexing schemes are disk space fragmentation and their ability to process large volumes of updates quickly. Various approaches have been proposed to address these issues. Cutting and Pedersen show how the Zipfian[14] distribution of term postings can lead to space and time optimizations unique to the incremental indexing tasks [8]. Tomasic et al. use a similar approach by dynamically separating short and long inverted lists and optimized the retrieval and update for each kind of list [19]. Brown et al. proposed an approach that uses a persistent object store and its data management facilities to provide efficient incremental updates [7]. The framework presented in this paper is not specific to static indexing and can therefore be used to support multi-type inverted lists can be used in the scope of incremental indexing.

In a way, the approach presented here borrows some mechanisms from the relational database community. Creating appropriate PostingSerializer objects for custom postings is roughly

[14]George Zipf observed in 1949 that the frequency of an item tends to be inversely proportional to its rank [23]. This observation is widely used in information retrieval system by assigning weights to terms by taking the inverse of the number of documents a term is in.

the same as creating a table and an inverted list posting represents a row in such a table. The reason for not using relational databases for large scale text retrieval systems vary, among them inadequate performance due to DBMS overhead, and lack of requirement for ACID transactions, yet some of the basic principles do apply [4]. The work presented in this paper gives us the ability to have appropriate formats for different data types.

Aside from managing inverted lists in a text retrieval system, the query processing and the relevance of the results have been a topic of intensive research. Most users do not care how a search engine works they expect relevant results as a response to a query [3]. With growing document collection sizes, some searches literally turn into *finding the needle in the haystack*. The two main issues to deal with are *recall* and *precision*. A search that returns virtually every document for a given query has a high recall, while a search that returns only relevant pages is said to have a high precision [22]. Numerous approaches have been proposed to strike a good balance between recall and precision, such as using document link-structure information [5], automatic query refinement and relevance feedback to name only a few. The topic of search quality is not subject of this paper, the interested reader may refer to Croft's *Advances in Information Retrieval* [9].

6 Conclusion and Future Work

The WebFountain project faced the problem of having to index a lot of inputs that were not index terms in the traditional sense and growing corpus sizes. The existing indexer was based on a generic inverted list format that was extremely powerful but not extraordinarily efficient for the growing list of requirements. A better solution was needed, to decrease the amount of storage space necessary and to more efficiently handle different forms of input.

Our primary goals in developing a multi-type inverted list indexer were the reduction of the amount of disk space used by an inverted list index, and the creation of a framework that enables the research and development of efficient indexing formats and techniques. Based on the IBM WebFountain full-text indexer which used a generic inverted list format, a framework was created that allows

- for the exact specification of inputs to the indexing process and their types,
- and provides simple to use APIs to create new inverted list formats, without having to modify the basic indexing algorithm.

Most of the development effort went into the abstraction of the index build process by keeping a generic intermediate format and the development of the index build specification. The ability to assign *token types* to index terms is the key to support multiple inverted list formats. Each unique token type has its own format. As of March 2005, 8 different PostingSerializer classes supporting 14 different formats have been implemented.

Taking advantage of the different nature of inputs (i.e., index terms requiring positional information, meta-data not requiring positional information, terms requiring datafields, etc.) led the reduction in the disk usage of an index by 12%. Section 3 also provided examples on how to save additional disk space by treating different kinds of terms differently (e.g., stop-words). Selected inverted lists could be reduced by up to 80% due to choosing an optimized posting representation, that eliminated intra-document d-gaps and made use of fixed length datafields.

The entire framework was implemented in C++ making extensive use of inheritance and polymorphism so that the overall algorithm had to undergo only minimal changes. This way of implementing it negates some performance optimization by the compiler, such as inlining. However, overall query response times remained on the same level or even improved slightly for queries with terms that already were very efficient. This indicates that the performance penalty of using virtual functions is negligible. Queries using terms that underwent significant space savings saw improvements of up to 75%, mainly due to the elimination of costly type conversions and the elimination of obsolete information in the inverted list. The overall performance in terms of index build times and query response times is very encouraging as very little work has been put into optimizing the new framework.

While the reduction in disk space was one of the primary goals, it was not a goal because of limited availability of disk space but rather the discrepancy between processor and memory speeds and disk performance. Reducing the on disk footprint has positive effects on query response times because having to read less data reduces cache pollution and I/O latencies. Furthermore, a reduction of disk space helps with the transfer of indexes to other nodes for replication or load balancing reasons.

The framework was applied in the static index build setting and proved useful as an extension of an existing indexer with minimal changes to its core algorithm, namely the in-memory indexing. Keeping a generic intermediate posting format avoids modifications to the core algorithms. This enables the quick implementation and evaluation of new inverted list formats and test their impact on index build performance and query response times. Furthermore, it allows for easy *mixing and matching* of different indexing techniques, such as bit-vectors with full positional inverted lists.

Future plans for the use of this framework is the integration into WebFountain's incremental indexer. In an incremental index, inverted lists are constantly updated. Having multiple inverted list formats at your disposal allows for potentially switching formats during index update phases.

This may be done based on observations of the existing list and its growth, as well as the availability of new optimized postings lists formats. Additionally, alternatives to our current d-gap compression schemes can be tried and evaluated, based on existing research as reviewed in section 5 and observations made as part of handling billions of documents in the WebFountain system. Some of the index terms in WebFountain require large datafields, which are currently stored on disk as-is, without any compression. For example, to allow for searches using regular expression on relatively short text strings[15]. Being able to use a tailored inverted list format in which datafields can be individually compressed is a compelling use of the framework presented in this paper.

[15] URLs are considered short text strings compared to Web documents.

References

[1] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison Wesley, New York, NY, 1999.

[2] Luiz A. Barroso, Jeffrey Dean, and Urs Hoelzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[3] Michael W. Berry and Murray Browne. *Understanding Search Engines*. Society for Industrial and Applied Mathematics, Philadelphia, 1999.

[4] Eric A. Brewer. Combining systems and databases: A search engine retrospective. In *Readings in Database Systems, 3rd Edition*, University of California, Berkeley, 2004.

[5] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *WWW7: Proceedings of the seventh international conference on World Wide Web 7*, pages 107–117. Elsevier Science Publishers B. V., 1998.

[6] Andrei Z. Broder. On the resemblance and containment of documents. In *SEQS: Sequences '91*, 1998.

[7] E.W. Brown, J.P. Callan, and W.B. Croft. Fast incremental indexing for full-text information retrieval. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB)*, pages 192 – 202, Santiago, Chille, September 1994.

[8] Doug Cutting and Jan Pedersen. Optimizations for dynamic inverted index maintenance. In *Proceedings of the 13th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 405–411, 1990.

[9] Bruce W. Croft (Editor). *Advances in Information Retrieval*. Kluwer Academic Publishers, Norwell, Massachusetts, 2000.

[10] Ralph Johnson John Vlissides Erich Gamma, Richard Helm. *Design Patterns; 1st edition (January 15, 1995)*. Addison-Wesley Professional, Reading, MA, 1995.

[11] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*

'03: Proceedings of the nineteenth ACM symposium on Operating systems principles, pages 29–43. ACM Press, 2003.

[12] D. Gruhl, L. Chavet, D. Gibson, J. Meyer, P. Pattanayak, A. Tomkins, and J. Zien. How to build a webfountain: An architecture for very large-scale text analytics. *IBM Syst. J.*, 43(1):64–77, 2004.

[13] Doug Lea. Customization in c++. In *C++ Conference*, pages 301–314, 1990.

[14] J. Zien S. Rajagopalan M. Fontoura, E. Shekita and A. Neumann. High performance index build algorithms for intranet search engines. In *Proceedings of the 30th VLDB Conference*, Toronto, Canada, 2004.

[15] IBM Research. Unstructured information management architecture sdk, 2004.

[16] 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. ACM Press, 2002.

[17] Ohm Sornil. *Parallel Inverted Index for Large-Scale, Dynamic Digital Libraries*. Ph.D. thesis, Virginia Polytechnic Institute and State Univeristy, 2001. Department of Computer Science.

[18] Justin Zobel Steffen Heinz and Hugh E. Williams. Burst tries: a fast, efficient data structure for string keys. *ACM Trans. Inf. Syst.*, 20(2):192–223, 2002.

[19] Anthony Tomasic, H&#233;ctor Garc&#237;a-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. ACM Press, 1994.

[20] Anh Ngoc Vo 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. ACM Press, 1998.

[21] Hugh E. Williams, Justin Zobel, and Dirk Bahle. Fast phrase querying with combined indexes. *ACM Trans. Inf. Syst.*, 22(4):573–594, 2004.

[22] Ian Witten, Alistair Moffat, and Timoty Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1999.

[23] George Zipf. *Human Behavior and the Principle of Least Effort*. Addison Wesley, Reading, MA, 1949.

[24] Justin Zobel, Steffen Heinz, and Hugh E. Williams. In-memory hash tables for accumulating text vocabularies. *Inf. Process. Lett.*, 80(6):271–277, 2001.

[25] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Trans. Database Syst.*, 23(4):453–490, 1998.