Using Bitmap Index for Joint Queries on Structured and Text Data*

Kurt Stockinger‡ John Cieslewicz§ Kesheng Wu‡ Doron Rotem‡ Arie Shoshani‡

Abstract

The database and the information retrieval communities have been working on separate sets of techniques for querying structured data and text data, but there is a growing need to handle these types of data together. In this paper, we present a strategy to efficiently answer joint queries on both types of data. By using an efficient compression algorithm, our compressed bitmap indexes, called FastBit, are compact even when they contain millions of bitmaps. Therefore FastBit can be applied effectively on hundreds of thousands of terms over millions of documents. Bitmap indexes are designed to take advantage of data that only grows but does not change over time (append-only data), and thus are just as effective with append-only text archives. In a performance comparison against a commonly used database system with a full-text index, MySQL, we demonstrate that our indexes answer queries 50 times faster on average. Furthermore, we demonstrate that integrating FastBit with a open source database system, called MonetDB, yields similar performance gains. Since the integrated MonetDB/FastBit system provides the full SQL functionality, the overhead of supporting SQL is not the main reason for the observed performance differences. Therefore, using FastBit in other database systems can offer similar performance advantages.

1 Introduction

The records in data warehouses are usually extracted from other database systems and therefore contain only what is known as structured data [10, 8, 29]. In these cases, most vendors are reusing existing database techniques to perform analysis tasks. However, data warehouses and database systems are starting to include a large amount of text documents, and the existing database techniques are inadequate for processing efficiently joint queries over structured data and text data.

Data warehouses typically contain records that are not modified once added to the collection [8, 9, 14]. This is very similar to most text collections, but different from transactional data, where the records are frequently modified. For this reason, techniques developed for data warehouses are likely also useful for queries on text, provided that they can be used effectively over thousands or even millions of terms.

Bitmap indexes are designed to take advantage of data that only grows but does not change over time (append-only data). Recent work on compressed bitmap indexing has shown that they can be applied to attributes (columns) with high cardinality; i.e., attributes that have a large number of possible distinct values. In particular, our compressed bitmap indexes, called FastBit, are compact and perform extremely well even when the index contains millions of bitmaps [40, 41]. It was therefore natural to investigate whether such indexes could be applied to searches over append-only text data containing hundreds of thousands of terms over millions of documents. If successful, this would enable efficient joint queries over structured and text data.

In this paper, we extend FastBit for searches over text data. We show that our proposed approach can significantly speed up joint queries on structured data and text data. An additional advantage is that we achieve this high performance gains by using the same indexing technique for both structured data and text data. We demonstrate that this can be done with minimal modification to the existing FastBit code. A number of database systems already implement various bitmap indexes, and the same modification can be made there too.

Originally, database management systems (DBMS) only handle structured data as tables, rows and columns, where each column value must be an atomic data type. Recently, many DBMSs have removed this limitation and allowed more complex data, such as date and time. Some of them even allow text. This enables text data to be stored

*This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098. Part of the funding was provided by a US Department of Homeland Security Fellowship administered by Oak Ridge Institute for Science and Education. We also thank the MonetDB Team at CWI, Netherlands for their great support of the integration effort.

†This work was supported by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Part of the funding was provided by a US Department of Homeland Security Fellowship administered by Oak Ridge Institute for Science and Education. We also thank the MonetDB Team at CWI, Netherlands for their great support of the integration effort.

‡Lawrence Berkeley National Laboratory, University of California, Berkeley, CA.

§Computer Science Department, University of Columbia, New York, NY.

together with the structured data. However, to support efficient searching operations on text, additional indexing data structures are introduced, e.g., the inverted index [4, 25, 49]. Among the popular database systems, MySQL is reputed to have an efficient implementation of an inverted index. Therefore, we chose to compare FastBit against MySQL to evaluate the merit of our approach. This comparison showed a large performance difference, with FastBit being 50 times faster in answering an average joint query on structured and text data.

To better understand whether the performance gain was due to difference in the indexing data methods or the system overhead, we integrated FastBit into an open-source database system called MonetDB [5, 20]. The integrated MonetDB/FastBit system has full SQL support and is much closer to MySQL in overall functionality than FastBit alone. We found that the integrated MonetDB/FastBit system has similar performance as FastBit alone. This confirms the performance advantage of FastBit over the full-text index in MySQL and indicates that our bitmap index is a valid approach for handling joint queries on structured data and text data.

The paper is organized as follows. In Section 2, we review related work on indexing data structures for structured data and text data. We also discuss the advantages of compressed bitmap indexes for querying both structured data and text data. In Section 3, we briefly describe the test dataset, referred to as the Enron dataset, that has been used in a number of studies on social networks. This dataset is particularly attractive since it contains a natural mixture of structured data and text data. In Section 4, we describe our framework for indexing text data with our bitmap index implementation, called FastBit. The challenges of integrating FastBit into MonetDB are briefly described in Section 5. An experimental evaluation of the combined MonetDB/FastBit system is presented in Section 6, with MySQL as the reference point. We summarize the findings of our studies in Section 7 and point out open research topics on using bitmap indexes for text searches.

2 Related Work

2.1 Indexing techniques for structured data

In the database community, a general strategy to reduce the time to answer a query is to devise an auxiliary data structure, or an index, for the task. Earlier database systems were more commonly used for transaction type applications, such as banking. For this type of applications, indexing methods such as B^+-tree and hash-based indexes are particularly efficient [10, 22]. One notable characteristic of data in these applications is that they change frequently and therefore their associated indexes must also be updated quickly.

As more data are accumulated over time, the need to analyze large historical data sets gained more attention. A typical analysis on such data warehouses is known as On-Line Analytical Processing (OLAP). For these operations, bitmap indexes are particularly efficient since they take advantage of the stable nature of the data (i.e., permitting efficient append operations, but not updates) [21, 46, 44, 41]. OLAP queries typically return a relative large number of selected values (also known as hits). In these cases, a bitmap index answers the queries much faster than a B^+-tree, but it takes longer to modify a bitmap index to update an existing record. However, for most data warehouses, existing records are not updated, and the only change to a data warehouse is the addition of a large number of new records. Appending new records to a bitmap index usually takes less time than updating a B^+-tree because the time to append to bitmap indexes is a linear function of the number of new records while the time to update a B^+-tree is always a superlinear function due to sorting involved. For these reasons, bitmap indexes are well-suited for data warehousing applications.

In Fig. 1, we show a small example of a bitmap index for an integer column **A** that takes its value from 0, 1, 2, and 3. In this case, we say that the *column cardinality* of **A** is 4. The basic bitmap index consists of four bitmaps, b_1, b_2, b_3, and b_4. Each bitmap corresponds to one of the four possible values of **A** and contains as many bits (0 or 1) as the number of rows in the table. In the basic bitmap index, a bit is set to 1 if the value of **A** in the given row equals the value associated with the bitmap.

Let N denote the number rows in a table and C denote the column cardinality. It is easy to see that a basic bitmap index contains CN bits in the bitmaps for the given column. As the column cardinality increases, the basic bitmap index requires correspondingly more storage space. In the worst case where each value is distinct, $C = N$, the total number of bits is N^2. There are a number of different strategies to reduce this maximum index size; we organize them into three orthogonal strategies: binning, encoding, and compression.

Binning: Instead of recording each individual value in a bitmap, the strategy of binning is to associate multiple values with a single bitmap [17, 30, 45]. For example, to index a floating-point valued column **B** with a domain between 0 and 1, we may divide the domain into 10 equi-width bins: $[0,0.1)$, $[0.1,0.2)$, $\ldots$, $[0.9,1]$. In this case, only 10 bitmaps are used. Binning can control the number of bitmaps used. However, the index is no longer able to resolve all queries accurately. For example, when answering a query involving the query condition "$\mathbf{B} < 0.25$," the information we get from the above binned index is that the entries in bins $[0,0.1)$ and $[0.1,0.2)$ definitely satisfy the query condition, but the records in bin $[0.2,0.3)$ have to be examined to determine whether they actually satisfy the condition. We call the records in bin $[0.2,0.3)$ candidates. The process of examining these candidates, called *candidate check*, can be expensive [24, 34]. For this reason, all commercial bitmap indexes do not use binning[21, 35].

Encoding: We can view the output from binning as a set of bin numbers. The encoding procedure translates

RID	A	bitmap index			
		=0	=1	=2	=3
1	0	1	0	0	0
2	1	0	1	0	0
3	3	0	0	0	1
4	2	0	0	1	0
5	3	0	0	0	1
6	3	0	0	0	1
7	1	0	1	0	0
8	3	0	0	0	1
		b_1	b_2	b_3	b_4

Figure 1: A example bitmap index, where RID is the record ID and **A** is an integer column with values in the range of 0 to 3.

these bin numbers into bitmaps. The basic bitmap index [21] uses an encoding called *equality encoding*, where each bitmap is associated with one bin number and a bit is set to 1 if the value falls into the bin, 0 otherwise. Other common encoding strategies include range encoding and interval encoding [6, 7]. In range encoding and interval encoding, each bitmap corresponds to a number of bins that are ORed together. They are designed to answer one-sided and two-sided range queries efficiently. The three basic encoding schemes can also be composed into multi-level and multi-component encodings [6, 31, 42]. One well-known example of a multi-component encoding is the binary encoding scheme [39, 23], where the jth bitmap of the index represents the value 2^j. This encoding produces the fewest number of bitmaps, however, to answer most queries, all bitmaps in the index are accessed. In contrast, other encodings may access a few bitmaps to answer a query, for example, an interval encoded index only need to access two bitmaps to answer any query.

Compression: Each bitmap generated from the above steps can be compressed to reduce the storage requirement. Any compression technique may be used, however, in order to reduce the query response time, specialized bitmap compression methods are preferred. Bitmap compression is an active research area [32, 41]. One of the best-known bitmap compression methods is the Byte-aligned Bitmap Code by Antoshenkov [3, 15]. A more efficient bitmap compression method is the Word-Aligned Hybrid (WAH) code [44, 41]. In multiple timing measurements, WAH was shown to be about 10 times faster then BBC [43]. As for the index size, the basic bitmap index compressed with WAH is shown to use at most $O(N)$ words, where N is the number of records in the dataset [44, 41]. In most applications, a WAH compressed basic bitmap index is smaller than a typical B-tree implementation.

Given a range condition that can be answered with a WAH compressed index, the total response time is proportional to the number of hits [41]. This is optimal in terms of computational complexity. In addition, compressed bitmap indexes are in practice superior to other indexing methods because the result from one index can be easily combined with that of another through bitwise logical operations. Therefore, bitmap indexes can efficiently answer queries involving multiple columns, a type of query we call the multi-dimensional query.

2.2 Database Systems for Full-Text Searching

Traditionally, DBMS treat text attributes as strings that have to be treated as a whole, or as opaque objects that can not be queried. However, users frequently need to identify text containing certain keywords. Supporting such keyword based text retrieval in database systems is an important research topic. The research literature discusses incorporating text retrieval capabilities into different types of database systems such as relational database systems [16], object oriented databases [47], and XML databases [1]. Our approach follows the general theme of combining relational databases with text searching capabilities [38, 18]. In this context, it is crucial to address the modeling issues, query languages and appropriate index structures for such database systems [27, 26]. A recent prototype of such a system, called QUIQ [16]. The engine of this system, called QQE, consists of a DBMS that holds all the base data and an external index server that maintains the unified index. Inserts and updates are made directly to the DBMS. The index server monitors these updates to keep its indexes current. It can also be updated in bulk-load mode. Another recent paper describes a benchmark called TEXTURE which examines the efficiency of database queries that combine relational predicates and text searching [12]. Several commercial database systems were evaluated by this benchmark.

Another approach for combining text retrieval and DBMS functionality is to use the external function capability of object oriented databases. Combining the structured-text retrieval system (TextMachine) with an object-oriented database system (OpenODB) has also been explored before [47].

As XML document is able to represent a mix of structured and text information, a third approach that is recently gaining some popularity is to combine text retrieval with XML databases. For example, there are proposals to extend the XQuery language with complex full-text searching capabilities [1].

Supporting text in databases requires appropriate index structures. One type of index proposed for text searching is called Signature files [13]. The space overhead of this index (10%-20%) is lower than that of inverted files, but any search always accesses the whole index sequentially. This index uses a hash function that maps words in the text to bit masks consisting of B bits called signatures. The text is then divided into blocks of b words each. The bit mask for each block is obtained by ORing the signatures of all the words in the block. A search for a query word is conducted by comparing its signature to the bit mask of each block. In case that at least one bit of the query signature is not present in the bit mask of a block, the word cannot be present in this block. Otherwise, the block is called a *candidate block* as the word may be present in it. All candidate blocks must be examined to verify that they indeed contain the query word.

Another common structure for indexing text files, found in commercial database systems and text search engines, is the inverted file index [49]. This data structure consists of a vocabulary of all the terms and an inverted list structure [36]. For each term t the structure contains the identifiers (or ordinal numbers) of all the documents containing t as well as the frequency of t in each document. Such a structure can also be supplemented with a table that maps ordinal document numbers to disk locations. As inverted files are known to require significant additional space (up to 80% of the original data) [50], we next review the compression issue.

2.3 Compressing Inverted Files

The inverted indexes commonly used for text searching are usually compressed [19, 37, 48]. The primary use of the compressed data in an inverted index is to reconstruct the document identifiers. Reducing the amount of time required to read the compressed data into memory is key to reducing the query response time. For this reason, the compression methods used to be measured exclusively by their compressed sizes. However, recently there has been some emphasis on compute efficiency as well [2, 37]. In particular, Anh and Moffat have proposed a Word-Aligned Binary Compression for text indexing, which they call slide [2], even though their primary design goal was to reduce the compressed sizes rather than improving the search speed. Making the decompression (i.e., reconstruction of the document identifiers) more CPU friendly is only a secondary goal. They achieve this by packing many code words that require the same number of bits into a machine word. Because all these code words require the same number of bits, they save space by only representing their sizes once. In contrast, WAH imposes restrictions on lengths of the bit patterns that can be compressed so that the bitwise logical operations can be performed on compressed words directly [44, 41]. In particular, a WAH code word is always a machine word. These properties yield efficient computations.

Because of their differences, it is usually not efficient to use a bitmap compression method to compress document identifiers or a compression method for the inverted index to compress bitmaps. What we propose to do in this paper is to turn a term-document matrix (a version of the inverted index) into a bitmap index, then compress the bitmap index. This approach allows us to make the maximum use of the efficient bitmap compression method WAH and reuse the bitmap indexing software for keyword searches.

The approach we take in this paper is to use compressed bitmaps to represent inverted files, an approach that is efficient for relatively small number of distinct terms. However, recently WAH-compressed indexes have been shown to be very efficient for high cardinality numerical data [41]. The strength of this particular work is to demonstrate that the compressed bitmap approach is efficient for text data with a large number of distinct terms. Using a compressed bitmap index, the results of processing a query condition is a compressed bitmap. The bitmaps representing the answers to different query conditions can be efficiently combined to form the final answer to the user query through logical operations. Since WAH compressed bitmaps can participate in logical operations efficiently without decompression, we anticipate this to be a great advantage of our approach.

For performance comparison, we choose to use MySQL for two reasons. The first reason is that MySQL implements an inverted index for full-text searches and its full-text search capability is well-regarded by the user community. The second reason is that MySQL is widely available, and has one of the most commonly used inverted indexes [49].

3 Case Study: The Enron Data Set

The Enron dataset, a large set of email messages, is used by various researchers in the areas of textual and social network analysis. The dataset was made public by the US Federal Energy Regulatory Commission during the criminal investigation into Enron's collapse in 2002. This dataset is particularly attractive for studies of index data structures since it contains numerical, categorical, and text data. Our case study is based on the data prepared by Shetty and Adibi [28] and contains 252,759 email messages stored in four MySQL tables, namely `EmployeeList`, `Message`, `RecipientInfo` and `Reference Info`.

In early performance experiments comparing FastBit with MySQL, we showed that FastBit greatly outperformed MySQL for queries over structured data, including all *numerical* and *categorical* values [33]. One of the key findings of these experiments was that materializing some tables to avoid expensive join operations can significantly reduce query response time. We plan to use the same dataset for our study of querying combined structured data and text

Table 1: Schema of database table Headers.

Column Name	Explanation
mid	Message ID
senderFirstName	First name of the sender (only Enron employees)
senderLastName	Last name of the sender (only Enron employees)
senderEmail	Email address of the sender
recipientFirstName	First name of the recipient (only Enron employees)
recipientLastName	Last name of the recipient (only Enron employees)
recipientEmail	Email address of the recipient
day	Day email was sent
time	Time email was sent
rtype	Receiver type: "TO," "CC," and "BCC"

Table 2: Schema of database table Messages.

Column Name	Explanation
mid	Message ID
subject	Subject of email message
body	Body of email message
folder	Name of folder used in email client

data. Following the above observation, we combined six tables into two tables called `Headers` and `Messages` (see Tables 1 and 2), where the first contains primarily structured data and the second contains primarily text data.

The table `Headers` contains both numerical and categorical values, whereas `Messages` contains only text data. Table `Headers` is a materialization of the parts from the three original tables `EmployeeList`, `Messages` and `RecipientInfo`. Table `Messages` contains a subset of the columns of the original table `Messages`. The advantage of this schema design is to use bitmap indexes for query processing for both tables. The column `mid` in `Messages` is a foreign key to the column of the same name in `Headers` and is used to join message text columns with the corresponding numerical and categorical data.

4 Extending Bitmap Indexes to Support Full Text Search

FastBit compressed bitmap index technology was originally designed to speed up queries on numerical data. In this section we describe how to extend bitmap indexes to support keyword queries over text data. Indexing text usually requires the following two steps: text parsing and term extraction and index generation.

In our framework we use Lucene [11] for text parsing and term extraction. The output of Lucene is a *term-document list* which is an inverted index that contains all identified terms across all documents and a set of document identifiers (IDs) of each document containing the term. Once the term-document list is obtained, we convert the term-document list into a bitmap index consisting of a dictionary of terms and a set of compressed bitmaps. Note that the use of a dictionary is not essential to the approach, but it is a convenient way to reuse the existing software for indexing integer data.

As illustrated in Figure 2, assuming a database table called `Messages` containing four columns `mid`, `body`, `subject` and `folder`, we proceed to build our bitmap index as follows. We first extract each text value, say from column `body`, into a file named after the `mid` column. Note that the Message IDs, `mid`, are the same as used in the MySQL version of the Enron e-mail message dataset [28]. In Figure 2, these files are indicated by "body1," "body2," etc.. Next, we pass this set of files to Lucene to identify terms in the files. The output from Lucene is a list of terms, and for each term a list of files containing the term. Since the file names are the `mid`s, we effectively produce a list of `mid` values for each term. For instance, the term "Berkeley" appears in the messages with the IDs 1, 3, 5 and 8. Similarly, the term "Columbia" appears in the messages with the IDs 5, 7, 8 and 9. These lists are the core content of a typical inverted index. Additional content typically include term frequencies [4, 25]. Since other inverted indexes contains more information than the term-document list, they take more space than the term-document list without compression.

The next step is to convert the term-document list into a bitmap index. However, before we can index the identified terms with bitmaps, we need to introduce an auxiliary data structure, called a *dictionary*, that provides a mapping between the terms and the bitmaps. In our example, "Berkeley" is represented by the numerical value 1, "Columbia"

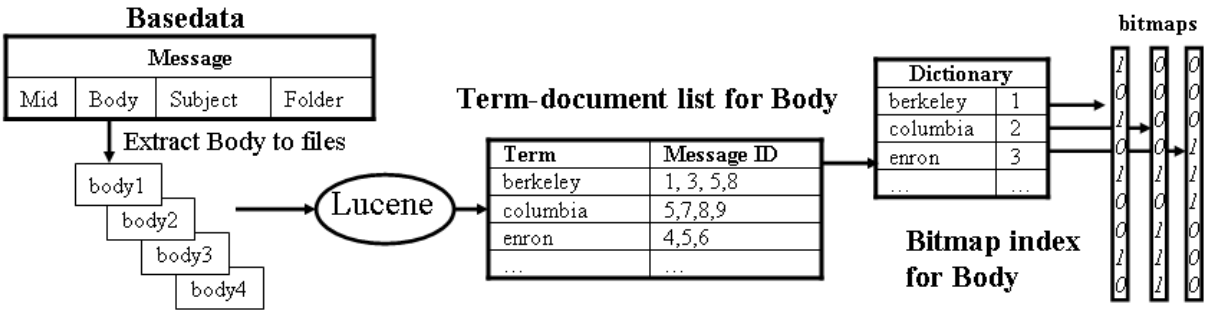


Figure 2: Framework for Indexing Text with FastBit. This illustration use the column "Body" as the example.

by the value 2 and "Enron" by the value 3 (see "dictionary" in Figure 2). Next, the message IDs originally stored in the term-document lists can be encoded with bitmaps. For instance, the bitmap representing "Berkeley" contains the bit string 101010010 to indicate that "Berkeley" is contained in the messages 1, 3, 5 and 8. Similarly, the bitmap representing "Columbia" contains the bit string 000010111 to indicate that "Columbia" is contained in the message 5, 7, 8, and 9. In other words, a bit is set to 1 if the respective term is contained in a message, otherwise the bit is set to 0[1].

Using compressed bitmap indexes for storing term-document lists supports keyword searches efficiently. For instance, finding all emails where `body` contains the terms "Berkeley" and "Columbia" requires reading two bitmaps and combining them with a logical AND operation. As showed in the past, such basic bitmap operations are very efficient [43].

5 Integrating FastBit into MonetDB

We decided to integrate FastBit into a relational database system for two reasons. First, as part of a relational database management system, FastBit would benefit from the system's ability to undertake tasks beyond indexing and querying, such as performing joins between tables and enforcing consistency in the records. Second, by adding FastBit to a relational database system, relational data can benefit from FastBit's high performance indexes. With the addition of text searching to FastBit, adding FastBit to a relational system also provides a high performance tool for keyword searches. The database system we chosen is MonetDB, an open source database system developed by CWI [20]. In this section we begin by describing MonetDB and our reasons for choosing it, then briefly describe our integration of FastBit with MonetDB.

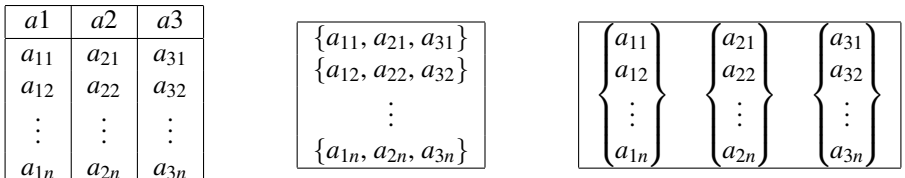
5.1 Why MonetDB?

MonetDB is our target relational database system for FastBit integration because of its data layout. Unlike most databases, such as MySQL or Oracle, that use horizontal or row-based storage, MonetDB uses vertical partitioning also known as a decomposed storage model (DSM) [5]. See Figure 3 for an illustration of the storage techniques. In a database with row-based storage, entire records are stored contiguously, thus making access to entire records efficient, but wasting I/O and memory bandwidth when only a small subset of columns is required [5, 29, 35]. For instance, in Figure 3b the records are read right to left, top to bottom during a scan even if the query is interested in only column $a2$ in each record. That is, the entire record is loaded even though only a small part of it is needed. With a DSM, single columns are stored contiguously (Figure 3c) resulting in efficient I/O for queries that involves only a subset of the columns. MonetDB's data layout is analogous to FastBit indexing, where each column is indexed and stored separately.

The MonetDB SQL Server is a two-layer system [20]. On the bottom is the MonetDB kernel that manages the actual data. At this layer, the data is not stored as a complete relational table, but is decomposed into separate Binary Association Tables (BAT)–one for each column. Each entry in the BAT is a two-field record containing an object identifier (OID) and a column data value. All column values for the same relational tuple have the same OID even though they are stored in separate BATs. Interaction with these BATs is accomplished via the Monet Interpreter Language (MIL), that can be extended with new commands which we make use of as outlined next.

The SQL module sits atop the MonetDB kernel and provides an SQL interface for client applications. Though the relational tables are actually decomposed into many BATs, the SQL module allows users to interact with the data in the normal relational manner. The SQL module is responsible for transaction and session management as well as

[1] In general, the document identifiers may not be directly used as row numbers for setting the bits in the bitmaps. We may actually need an additional step of mapping the document identifiers to row numbers. This additional level of operational detail is skipped for clarity.



(a) Logical View. Each row has three columns

(b) Horizontal Partitioning. Store values from the same row contiguously

(c) Vertical Partitioning Store values from the same column contiguously

Figure 3: An illustration of horizontal and vertical Partitioning.

transforming SQL queries into MIL code to be executed by the MonetDB kernel. With the help of the MonetDB developers at CWI, we decided it would be best to integrate FastBit into the SQL module rather than the underlying MonetDB kernel. In the following sections we give an overview of the changes required to integrate FastBit into MonetDB/SQL. Note that all changes described occurred within the SQL module; the MonetDB kernel was left unchanged. The MonetDB kernel was version 4.12 and the SQL module was version 2.12. Both are available from the MonetDB website at `http://monetdb.cwi.nl/`.

5.2 Integrating MonetDB and FastBit

Integrating FastBit into MonetDB's SQL module (MonetDB/SQL) required four tasks: (1) addition of the `FASTBIT` keyword to MonetDB's SQL parser, (2) functionality to allow MonetDB to send data to a FastBit library for index construction, (3) rules to recognize subqueries that are FastBit eligible during query optimization, and (4) integration of FastBit and MonetDB execution so that a unified query result is produced by MonetDB. We next briefly describe each of these tasks.

Overall, users continue to interact with MonetDB/SQL as the front end. To allow MonetDB to invoke FastBit index creation functions, we need to do two things: first to inform the MonetDB system that it needs to invoke FastBit indexing creation functions, and then to prepare the necessary base data for index creation. We modified the parser in the SQL module to recognize the keyword `FASTBIT` in the index creation command. This keyword informs MonetDB to invoke FastBit for index creation.

In order for FastBit to create an index, it needs to know the raw data of the column to be indexed. This set of data is written to a specific directory under the data directory for MonetDB. MonetDB server then invokes the appropriate functions to create FastBit indexes. The metadata held by the MonetDB server is modified to reflect the existence of FastBit indexes and it has to update the FastBit indexes when the data is modified.

Since FastBit can only perform a subset of queries that MonetDB supports, recognizing which part of the query can take full advantage of FastBit indexes is critical. We achieve this by examining and modifying the query plan generated by the MonetDB SQL parser. All equality conditions and range conditions on variables that have FastBit indexes are recognized as suitable for FastBit processing, and they are combined together into a special node in the query plan. This special node along with its parameters are later passed to FastBit. To perform a keyword search, we overload the operator '=,' for example, the MySQL expression "MATCH(body) AGAINST('Berkeley')" would be expressed as "body = 'Berkeley'."

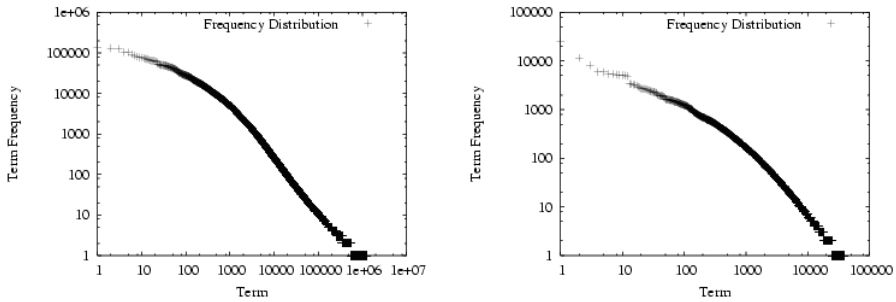
The special node in the query execution plan is translated into a command in Monet Interpreter Language called `fastbit_execute`. The MonetDB execution engine recognizes this command and composes the appropriate query string for FastBit. Following a typical evaluation command, the command `fastbit_execute` also produces a list of OIDs. This allows other operations in MonetDB to proceed as usual.

6 Experiments

This section contains a discussion of our experimental results and demonstrates that adding compressed bitmap indexes to a relational database system enables high performance, integrated querying of structured data and text data.

The first two parts of this section (6.1 and 6.2) present some statistics about the term distribution in our text data as well as the size and cardinality of the bitmap indexes constructed for all columns in the Enron Data Set. The remainder of the section presents the timing results. The experiments compare MySQL, a popular open-source database management system supporting text searching, a stand-alone FastBit client, and MonetDB integrated with FastBit. The experiments are broken down into three groups: (1) queries over structured data, (2) queries over text data, and (3) queries over both structured data and text data.

Each query in the following experiments was issued both as a *count query* and as an *output query*. A *count query* returns only the number of hits, that is, the SQL `SELECT` clause starts with `SELECT COUNT(*) FROM`. An *output*



(a) Messages "body" **(b)** Messages "subject"

Figure 4: Term frequency distribution in the message "body" and "subject."

query retrieves data values associated with the tuples in the result set. Note that all performance graphs are shown with a log-log scale.

All experiments were conducted on a server with dual 2.8 GHz Pentium 4 processors, 2 GB of main memory, and an IDE RAID storage system capable of sustaining 60 MB/sec for reads and writes. Before we executed each set of 1000 queries, we unmounted and remounted the file system containing the data and the indexes as well as restarted the database servers in order to ensure cold cache behavior.

6.1 Data Statistics

Figure 4 shows the term frequency distributions in the "body" and the "subject" of the Enron emails. The terms were extracted with Lucene. Both distributions match Zipf's law as commonly observed in many phenomena in nature. The total number of distinct terms in the message body is more than 1.2 million. The total number of distinct terms in the message subject is about 40,000.

6.2 Size of Bitmap Indexes

Table 3 shows the size of raw data compared with the size of the compressed bitmap indexes for each column of table `Headers`. The column cardinalities of each columns are also given. Note that we have chosen not to index column `mid` because it is unique for every message and is therefore more efficient to directly work with the raw data to answer typical queries involving it. The sizes of the compressed bitmap indexes are much smaller than the raw data. For instance, the index size for column `recipientEmail` is about 20% or raw data even though this column has a very high column cardinality, about 70,000. For lower cardinality columns, such as `senderEmail`, the bitmap index sizes are only about 2∼3% of the raw data.

Table 4 shows the size of the compressed bitmap indexes for the table `Messages`, i.e., the table that stores the text data. In addition to the size of the raw data, we also provide the size of the uncompressed term-document list. We see that the space required for table `Messages` is dominated by the column `body`. Since it contains more than 1.2 million distinct terms, its index is also the largest. In this case, the size of the compressed bitmap index is about half the size of the term-document list, which in turn is about half the size of the raw data. On average, we use less than 100 bytes per term indexed. Overall, the compressed bitmap indexes are smaller than the term-document lists, which are the minimal information in typical inverted indexes.

From earlier analyses of the WAH compressed bitmap indexes [44, 41], we know that the upper bound of the total size of bitmaps is a linear function of the number of rows in the dataset. For typical high-cardinality data, the total size of bitmaps may be twice the size of the base data. The relative sizes shown in Tables 3 and 4 indicates that the actual index sizes are well within the predicted upper bounds. Note that the index sizes reported in Tables 3 and 4 include all information associated with the compressed bitmaps such as the dictionary for text data.

6.3 Query performance on structured data

In this first set of timing measurements, we present the time required to answer queries on structured data from table `Headers`. We tested queries of one- and two-dimensions, with and without retrieving data values.

Figure 5 shows the timing results of running 1000 one- and two-dimensional queries. In the query expressions, "`:S`" and "`:D`" denote the values that vary in the 1000 queries. In the one-dimensional queries, our queries use top 1000 senders as the value of "`:S`." To answer these count queries FastBit is clearly faster than MySQL. On average, FastBit is about a factor of 36 faster than MySQL as shown in Table 5.

For each count query, we also execute an output query with the same set of query conditions. The timing results for answering these queries are shown in Figure 5(c) and (d). On these queries, FastBit takes much longer to retrieve

Table 3: Size of the raw data compared with the size of compressed bitmap indexes for each column of the table `Headers`. For the categorical values also the dictionary sizes are also given.

Column	Card.	Data [MB]	Dict. [MB]	Bitmap Index [MB]	Bitmap Index [% Data]
mid	252,759	8.26			
senderFirstName	112	7.21	0.0007	0.14	1.9
senderLastName	148	7.70	0.0001	0.14	1.8
senderEmail	17,568	48.00	0.4336	1.41	2.9
recipientFirstName	112	7.20	0.0007	0.14	1.9
recipientLastName	148	7.52	0.0001	1.40	18.6
recipientEmail	68,214	47.78	1.5454	13.69	28.6
day	1,323	8.26		0.74	9.0
time	46,229	8.26		3.24	39.2
rtype	3	6.45	0.0001	0.49	7.6

Table 4: Size of the raw data and the term-document list (td-list) compared with the size of compressed bitmap indexes including the dictionary for each column of table `Messages`.

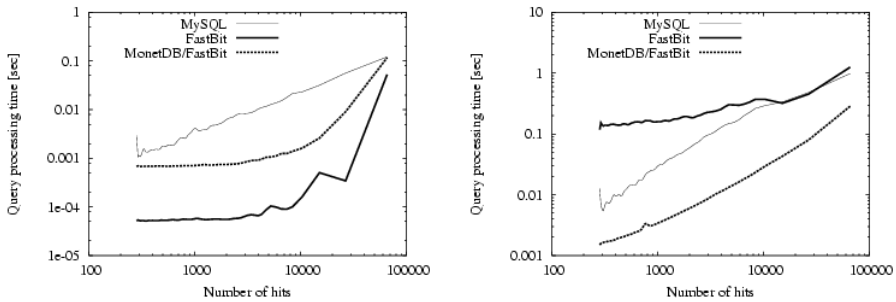
Column	Card.	Data [MB]	td-list [MB]	Dict. [MB]	Bitmap Index Size [MB]	[% Data]	[% td-list]
mid	252,759	1.01					
subject	38,915	7.56	8.20	0.31	5.23	69.2	63.8
body	1,247,922	445.27	245.57	16.92	121.72	27.3	49.6
folder	3,380	20.98	8.09	0.04	0.14	0.7	1.7

the selected values than the other two. This is because FastBit reconstruct the string values from the content of the dictionary and the bitmap representing the hits. In contrast, the combined system uses FastBit to retrieve the object identifiers and uses MonetDB to retrieve the values. Clearly, this is a better option. Overall, using MonetDB/FastBit is about eight times faster than using MySQL, see Table 5.

When a WAH compressed bitmap index is used to answer a query, analyses show that the worst-case query response time is bounded by a linear function of the number of hits [44, 41]. This worst case can be achieved with uniform random data. Since the actual index sizes shown in Table 3 are much smaller than predicted worst-case sizes and the average query response time is proportional to the index size, the query response time should be proportionally less than in the worst case. This expectation is for one-dimensional queries shown in Figure 5(a). The answers to multi-dimensional queries are composed from answers to multiple one-dimensional queries. One average, the total query response time for a k-dimensional query is k times that of a one-dimensional query.

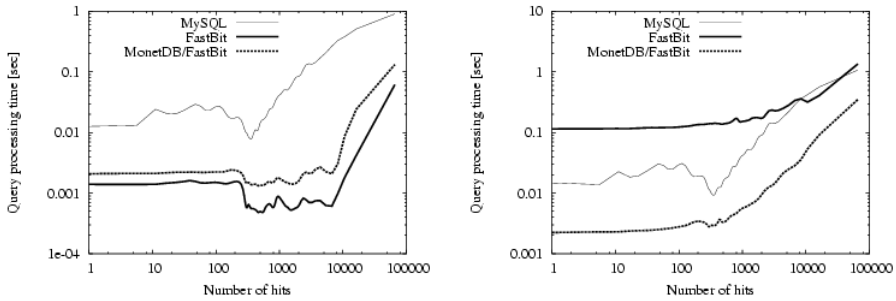
Table 5: Total time in seconds for running 1,000 queries against table `Headers`. This is a summary of the results presented in Figure 5.

	MySQL	FastBit	MonetDB/FastBit
Fig. 5a	5.17	0.17	1.53
Fig. 5b	51.56	1.29	2.73
Total time	56.72	1.46	7.45
Speedup		36.4	13.3
Fig. 5c	47.66	181.64	6.35
Fig. 5d	53.17	95.93	5.86
Total time	100.93	277.57	12.21
Speedup		0.36	8.27



(a) 1D Count Query:
SELECT count(*)
FROM Header
WHERE senderEmail = :S

(c) 1D Output Query:
SELECT recipientEmail
FROM Header
WHERE senderEmail = :S



(b) 2D Count Query:
SELECT count(*)
FROM Header
WHERE senderEmail = :S
AND day $<$:D

(d) 2D Output Query:
SELECT recipientEmail, day
FROM Header
WHERE senderEmail = :S
AND day $<$:D

Figure 5: Count and output queries on the table `Headers`. A summary of the performance measurements is given in Table 5.

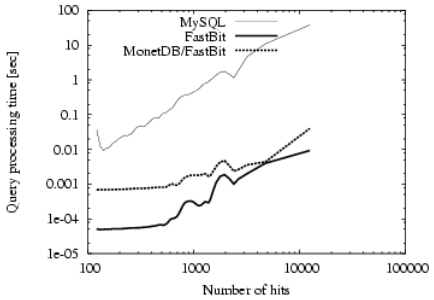
Table 6: Total time in seconds for running 1,000 count queries against the table Messages. This table is a summary of the results presented in Figure 6.

	MySQL	FastBit	MonetDB/FastBit
Fig. 6a	324.79	0.58	1.56
Fig. 6b	311.11	0.58	1.45
Fig. 6c	532.12	13.34	13.32
Fig. 6d	518.70	18.06	15.71
Total time	1706.72	32.56	32.04
Speedup		52.42	53.26

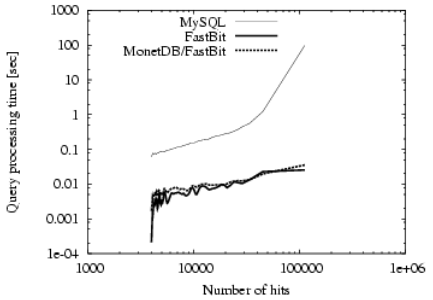
6.4 Query Performance for Text Searching

Next, we study the keyword searching capability of FastBit and compare it with that of MySQL. Figure 6 shows both the timing results and the queries used. As before, our actual queries replace the variables with top 1000 frequent terms from each of the text columns. Presumably, the most frequent terms are also "interesting" for text analysis since these terms are more discussed among people in Enron emails and might thus have a higher semantic meaning.

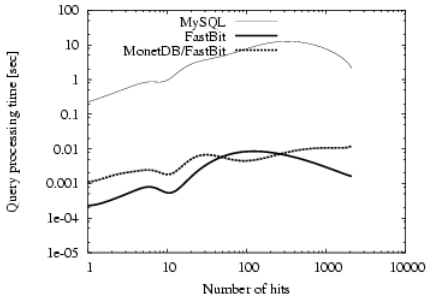
Figure 6 shows the response times for 1000 queries over the subject and the body of the email messages from the Enron data set. Table 6 shows a summary of timing information. For those queries containing one keyword, using FastBit as a stand-alone system, is more than 500 times faster than using MySQL. Since these queries need to pass a relatively larger number of OIDs from FastBit to MonetDB, the combined MonetDB/FastBit takes longer time using FastBit alone. Nevertheless, the combined system still show very impressive speedup over MySQL, about 50.



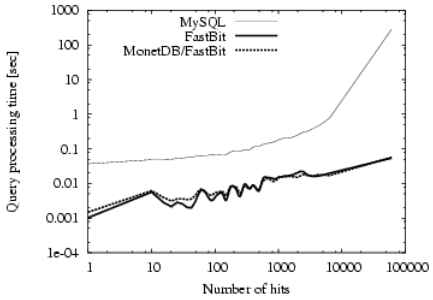
(a) 1D Subject Query:
SELECT count(*) FROM Messages
WHERE MATCH(subject) AGAINST (:S1)



(c) 1D Body Query:
SELECT count(*) FROM Messages
WHERE MATCH(body) AGAINST (:B1)



(b) 2D Subject Query:
SELECT count(*) FROM Messages
WHERE MATCH(subject) AGAINST (:S1)
AND MATCH(subject) AGAINST (:S2)



(d) 2D Body Query:
SELECT count(*) FROM Messages
WHERE MATCH(body) AGAINST (:B1)
AND MATCH(body) AGAINST (:B2)

Figure 6: Count queries on the "subject" and "body" columns of the table `Messages`. A summary of the performance measurements is given in Table 6.

6.5 Query Performance for both Numerical and Text Data

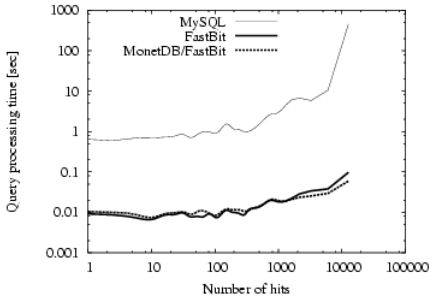
Our last set of experiments is the most challenging because it requires a join operation over the tables `Headers` and `Messages`. Since FastBit currently does not support join operations, we implemented a simple sort-merge join algorithm outside of FastBit. In particular, a join query over two tables consists of four FastBit queries. The first query evaluates the query condition on the table `Headers`. The second query evaluates the query condition on the table `Messages`. Next, the lists of resulting message IDs (mids) of both queries are sorted and intersected to find the common ones. The list of common mids is then sent back as two queries in the form of "`mid IN (12, 35, 89, ...)`." Finally, the desired columns are retrieved from the two tables. The count queries can skip the last step since they do not retrieve any values. Retrieving values through FastBit this way is likely to be slow because FastBit is not efficient at retrieving string values, and parsing the long query expression involving thousands of `mid`s is also time consuming.

In Figures 7 and 8, we plot the query response time against the number of hits for the combined queries on both structured data and text data. In these tests, the combined MonetDB/FastBit uses FastBit to perform the filtering one each table and then perform the sort-merge join on `mid`. Since our external join algorithm essentially does the same thing, we see that FastBit and MonetDB/FastBit take about the same amount of time. Both of them are significantly faster than MySQL. Table 7 shows the total time to answer 1000 queries. Overall, we see that FastBit and MonetDB/FastBit are about 52 and 67 times faster than MySQL in answering these count queries.

Figure 8 shows the query response time of the output queries. Because retrieving string values using FastBit is slow, the overall speed of FastBit versus MySQL decreases to 16, but MonetDB/FastBit combined system remains about 64 times faster than MySQL.

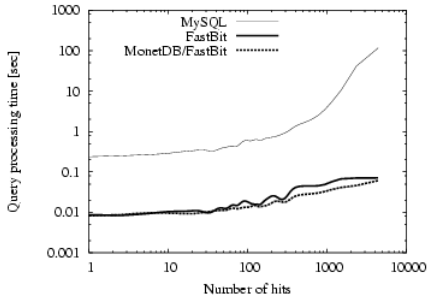
7 Conclusions and Future Work

We propose a way of using compressed bitmaps to represent the commonly used term-document matrix to support keyword searches on text data. By using a compute-efficient compression technique, we are able to not only keep the indexes compact but also answer keyword queries very efficiently. In our detailed experimental study we show that



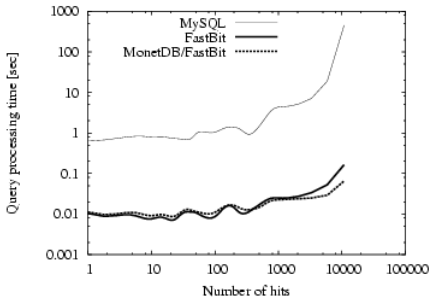
(a)

```
SELECT count(*)
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND senderEmail = :S
  AND m.mid = h.mid
```



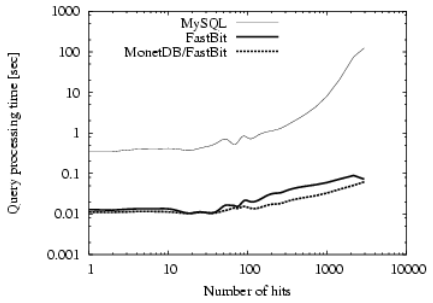
(b)

```
SELECT count(*)
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND recipientEmail = :S
  AND m.mid = h.mid
```



(c)

```
SELECT count(*)
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND senderEmail = :S
  AND day < :D AND m.mid = h.mid
```



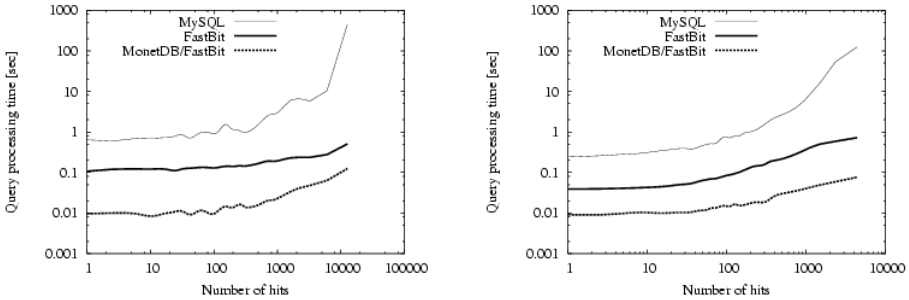
(d)

```
SELECT count(*)
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND recipientEmail = :S
  AND day < :D AND m.mid = h.mid
```

Figure 7: Integrated numerical and text count queries. These queries are shown in the format used by MySQL. A summary of the performance measurements is given in Table 7.

Table 7: Total time in seconds for running 1,000 join queries against tables Headers and Messages. This table is a summary of the results in Figures 7 and 8.

	MySQL	FastBit	MonetDB/FastBit
Fig. 7a	1302.28	16.54	16.03
Fig. 7b	866.98	23.37	17.45
Fig. 7c	975.18	24.81	18.10
Fig. 7d	556.80	6.40	2.98
Total time	3701.24	71.12	54.56
Speedup		52.04	67.84
Fig. 8a	1303.24	82.55	18.24
Fig. 8b	970.31	78.64	18.51
Fig. 8c	977.99	53.00	19.17
Fig. 8d	557.27	21.24	3.35
Total time	3808.81	235.43	59.27
Speedup		16.18	64.26

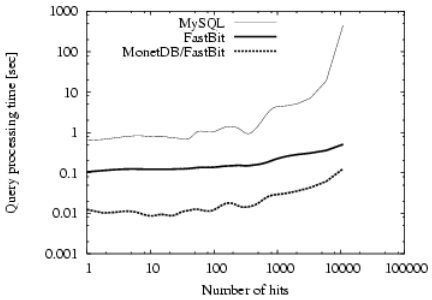


(a)

```
SELECT recipientEmail, day, subject
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND senderEmail = :S
  AND m.mid = h.mid
```

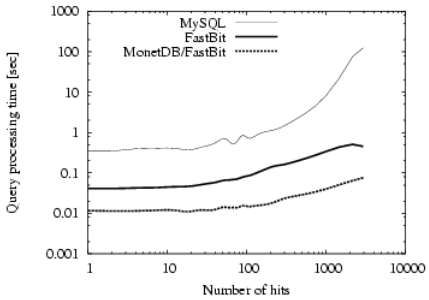

(b)

```
SELECT senderEmail, day, subject
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND recipientEmail = :S
  AND m.mid = h.mid
```



(c)

```
SELECT recipientEmail, day, subject
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND senderEmail = :S
  AND day < :D
  AND m.mid = h.mid
```



(d)

```
SELECT senderEmail, day, subject
FROM Messages m, Headers h
WHERE MATCH(body) AGAINST (:B1)
  AND recipientEmail = :S
  AND day < :D
  AND m.mid = h.mid
```

Figure 8: Integrated numerical and text output queries. These queries are shown in the format used by MySQL. A summary of the performance measurements is given in Table 7.

our bitmap index technology called FastBit answers count queries over text data about 50 times faster than MySQL.

To provide the full functionality of SQL including text search, we integrated our text index technology with an open-source database management system called MonetDB. This integration introduces text-search capability to MonetDB. Our performance experiments demonstrate that the integrated system significantly reduces the time needed to answer joint queries over structured data and text data. Compared with MySQL, the integrated system is 60 times faster on average at retrieving text values subject to multi-dimensional query conditions. This demonstrates that the demand of providing full SQL support does not necessarily diminish performance. It further validates our compressed bitmap approach as an efficient way of accelerating joint queries on structured data and text data.

The work presented in this paper only supports Boolean queries over the text data, i.e., without ranking the results. Future versions of bitmap indexes may include ranking information, as well as proximity of terms in text documents.

References

[1] AmerYahia, S., Botev, C., Shanmugasundaram, J.: TeXQuery: A FullText Search Extension to XQuery. In: WWW2004. New York, New York, USA (2004)

[2] Anh, V.N., Moffat, A.: Improved Word-Aligned Binary Compression for Text Indexing. IEEE Transactions on Knowledge and Data Engineering **18**(6), 857–861 (2006)

[3] Antoshenkov, G.: Byte-aligned Bitmap Compression. Tech. rep., Oracle Corp. (1994). U.S. Patent number 5,363,098

[4] Baeza-Yates, R.A., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)

[5] Boncz, P.A., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: Memory access. In: The VLDB Journal, pp. 54–65 (1999)

[6] Chan, C.Y., Ioannidis, Y.E.: Bitmap Index Design and Evaluation. In: SIGMOD. ACM Press., Seattle, Washington, USA (1998)

[7] Chan, C.Y., Ioannidis, Y.E.: An Efficient Bitmap Encoding Scheme for Selection Queries. In: SIGMOD. ACM Press., Philadelphia, Pennsylvania, USA (1999)

[8] Chaudhuri, S., Dayal, U.: An Overview of Data Warehousing and OLAP Technology. ACM SIGMOD Record **26**(1), 65–74 (1997)

[9] Chaudhuri, S., Dayal, U., Ganti, V.: Database technology for decision support systems. Computer **34**(12), 48–55 (2001)

[10] Comer, D.: The ubiquitous B-tree. Computing Surveys **11**(2), 121–137 (1979)

[11] Doug Cutting, e.a.: Apache lucene. Http://lucene.apache.org

[12] Ercegovac, V., DeWitt, D.J., Ramakrishnan, R.: The texture benchmark: Measuring performance of text queries on a relational dbms. In: VLDB, pp. 313–324 (2005)

[13] Faloutsos, C., Christodoulakis, S.: Signature files: an access method for documents and its analytical performance evaluation. ACM Trans. Inf. Syst. **2**(4), 267–288 (1984)

[14] Inmon, W., Hackathorn, R.: Using the data warehouse. Wiley-QED Publishing, Somerset, NJ, USA (1994)

[15] Johnson, T.: Performance Measurements of Compressed Bitmap Indices. In: International Conference on Very Large Data Bases. Morgan Kaufmann., Edinburgh, Scotland (1999)

[16] Kabra, N., Ramakrishnan, R., Ercegovac, V.: The QUIQ Engine: A Hybrid IR-DB System. In: ICDE, pp. 741–743. IEEE (2003)

[17] Koudas, N.: Space efficient bitmap indexing. In: International Conference on Information and Knowledge Management. ACM Press., McLean, Virginia, USA (2000)

[18] L.V. Saxton, V.R.: Design of an integrated information retrieval/database management system. IEEE Transactions on Knowledge and Data Engineering **2**, 210–219 (1999)

[19] Moffat, A., Zobel, J.: Self-indexing inverted files for fast text retrieval. ACM Transcations on Information Systems **14**(4), 349–379 (1996)

[20] MonetDB: Query Processing at Light-Speed. Http://monetdb.cwi.nl

[21] O'Neil, P.: Model 204 Architecture and Performance. In: 2nd International Workshop in High Performance Transaction Systems. Springer-Verlag, Asilomar, California, USA (1987)

[22] O'Neil, P., O'Neil, E.: Database: pronciples, programming, and performance, 2nd edn. Morgan Kaugmann (2000)

[23] O'Neil, P., Quass, D.: Improved Query Performance with Variant Indexes. In: Proceedings ACM SIGMOD International Conference on Management of Data. ACM Press, Tucson, Arizona, USA (1997)

[24] Rotem, D., Stockinger, K., Wu, K.: Optimizing candidate check costs for bitmap indices. In: CIKM (2005)

[25] Salton, G.: Automatic text processing: the transformation, analysis, and retrieval of information by computer. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)

[26] Schek, H.J.: Nested Transactions in a combined IRS-DBMS Architecture. In: International ACM SIGIR Conference on Research and Development in Information Retrieval. ., Cambridge, England (1984)

[27] Schek, H.J., Pistor, P.: Data Structures for an Integrated Data Base Management and Information Retrieval System. In: International Conference on Very Large Data Bases. Morgan Kaufmann., Mexico City, Mexico (1982)

[28] Shetty, J., Adibi, J.: The Enron Email Dataset, Database Schema and Brief Statistical Report. Tech. rep., Information Sciences Institute, Marina del Rey, California (2006). URL http://www.isi.edu/ãdibi/Enron/Enron_Dataset_Report.pdf

[29] Shoshani, A.: OLAP and statistical databases: similarities and differences. In: Principles Of Database Systems (PODS), pp. 185–196 (1997)

[30] Shoshani, A., Bernardo, L.M., Nordberg, H., Rotem, D., Sim, A.: Multidimensional indexing and query coordination for tertiary storage management. In: 11th International Conference on Scientific and Statistical Database Management, Proceedings, Cleveland, Ohio, USA, 28-30 July, 1999, pp. 214–225. IEEE Computer Society (1999)

[31] Sinha, R.R., Winslett, M.: Multi-resolution bitmap indexes for scientific data. ACM Trans. Database Syst. **32**(3), 16 (2007)

[32] Stabno, M., Wrembel, R.: Rlh: bitmap compression technique based on run-length and huffman encoding. In: DOLAP '07: Proceedings of the ACM tenth international workshop on Data warehousing and OLAP, pp. 41–48. ACM, New York, NY, USA (2007). DOI http://doi.acm.org/10.1145/1317331.1317339

[33] Stockinger, K., Rotem, D., Shoshani, A., Wu, K.: Bitmap Indexing Outperforms MySQL Queries by Several Orders of Magnitude. Tech. Rep. LBNL-59437, Berkeley Lab, Berkeley, California (2006)

[34] Stockinger, K., Wu, K., Shoshani, A.: Evaluation Strategies for Bitmap Indices with Binning. In: DEXA. Springer-Verlag., Zaragoza, Spain (2004)

[35] Sybaseiq. Http://www.sybase.com/products/information management/sybaseiq

[36] Tomasic, A., Garcia-Molina, H., Shoens, K.A.: Incremental Updates of Inverted Lists for Text Document Retrieval. In: SIGMOD Conference. Minneapolis, Minnesota, USA (1994)

[37] Trotman, A.: Compressing inverted files. Information Retrieval **6**, 5–19 (2003)

[38] de Vries, A., Wilschut, A.: On the Integration of IR and Databases. In: IFIP 2.6 DS-8 Conference. Rotorua, New Zealand (1999)

[39] Wong, H.K.T., Liu, H.F., Olken, F., Rotem, D., Wong, L.: Bit transposed files. In: Proceedings of VLDB 85, Stockholm, pp. 448–457 (1985)

[40] Wu, K.: FastBit: an efficient indexing technology for accelerating data-intensive science, *J. Phys.: Conf. Ser.*, vol. 16, pp. 556–560. Institute of Physics (2005). Software available at http://sdm.lbl.gov/fastbit/

[41] Wu, K., Otoo, E., Shoshani, A.: An efficient compression scheme for bitmap indices. ACM Transactions on Database Systems **31**, 1–38 (2006)

[42] Wu, K., Otoo, E.J., Shoshani, A.: Compressed bitmap indices for efficient query processing. Tech. Rep. LBNL-47807, LBL, Berkeley, CA (2001)

[43] Wu, K., Otoo, E.J., Shoshani, A.: Compressing bitmap indexes for faster search operations. In: SSDBM, pp. 99–108 (2002)

[44] Wu, K., Otoo, E.J., Shoshani, A.: On the performance of bitmap indices for high cardinality attributes. In: M.A. Nascimento, M.T. Özsu, D. Kossmann, R.J. Miller, J.A. Blakeley, K.B. Schiefer (eds.) Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004, pp. 24–35. Morgan Kaufmann (2004)

[45] Wu, K.L., Yu, P.: Range-based bitmap indexing for high cardinality attributes with skew. Tech. Rep. RC 20449, IBM Watson Research Division, Yorktown Heights, New York (1996)

[46] Wu, M.C., Buchmann, A.P.: Encoded bitmap indexing for data warehouses. In: Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA, pp. 220–230. IEEE Computer Society (1998)

[47] Yan, T.W., Annevelink, J.: Integrating a Structured-Text Retrieval System with an Object-Oriented Database System. In: International Conference on Very Large Databases, pp. 740–749. Santiago, Chile (1994)

[48] Ziviani, N., de Moura, E.S., Navarro, G., Baeza-Yates, R.: Compression: a key for next-generation text retrieval systems. IEEE Computer **33**, 37–44 (2000)

[49] Zobel, J., Moffat, A.: Inverted files for text search engines. ACM Computing Serveys **38**(2) (2006)

[50] Zobel, J., Moffat, A.: Inverted Files for Text Searching. ACM Computing Surveys **38**(3) (2006)