**myjournal manuscript No.**
(will be inserted by the editor)

# Using Bitmap Indexing Technology for Combined Numerical and Text Queries

**Kurt Stockinger[1], John Cieslewicz[2], Kesheng Wu[1], Doron Rotem[1], Arie Shoshani[1]**

[1] Computational Research Division
Lawrence Berkeley National Laboratory
University of California

[2] Department of Computer Science
Columbia University

Received: date / Revised version: date

**Abstract** In this paper, we describe a strategy of using compressed bitmap indices to speed up queries on both numerical data and text documents. By using an efficient compression algorithm, these compressed bitmap indices are compact even for indices with millions of distinct terms. Moreover, bitmap indices can be used very efficiently to answer Boolean queries over text documents involving multiple query terms. Existing inverted indices for text searches are usually inefficient for corpora with a very large number of terms as well as for queries involving a large number of hits. We demonstrate that our compressed bitmap index technology overcomes both of those short-comings. In a performance comparison against a commonly used database system, our indices answer queries 30 times faster on average.

To provide full SQL support, we integrated our indexing software, called FastBit, with MonetDB. The integrated system MonetDB/FastBit provides not only efficient searches on a single table as FastBit does, but also answers join queries efficiently. Furthermore, MonetDB/FastBit also provides a very efficient retrieval mechanism of result records.

## 1 Introduction

Bitmap indexing is a technology that was considered appropriate for small cardinality attributes only, usually below 100 distinct values, such as the 50 states in the US, or 100 age values for people. Consequently, they were not used for numeric values with a large cardinality, such as floating point temperature values, or for a large number of terms.

The main difficulty with bitmap indices is that usually a bitmap is required for each value, and if left uncompressed their sizes and the corresponding search time grows with the number of bitmaps. For example, consider having 1,000,000 people, and an attribute containing their yearly income, where it can be in the range of 0-100,000. Assume that the value of income is represented in a 32 bit integer, then the total size of that attribute would be 32,000,000 bits. If this was represented as 100,000 bitmaps where each is 1,000,000 bits long, the total volume of the index would be $10^{11}$ bits. For this reason compressed bitmap indices are used. Typically, the size of the compressed bitmaps is a fraction of the original data, as compared to 2-3 times the size of the data for a conventional index, such as a B-tree. However, there is an additional cost of processing the search over the compressed bitmap indices, because of the need to decompress them before a search can be performed efficiently.

Recent work [33], has introduced a specialized compression technique for bitmap indices, called Word-Aligned Hybrid (WAH), that is compute friendly, in that logical operations can be performed on the compressed bitmaps directly. Over the past few years, we have successfully demonstrated that the index which uses WAH encoding, referred to as FastBit, significantly improves the performance of multi-attribute queries on numerical data with high-cardinality attributes. In this paper we take the idea of bitmap indices a step further and show that compressed bitmap indices can also be used for *Boolean queries* over text to find all documents that match a certain search criterion.

FastBit is an efficient compressed bitmap index technology that provides a subset of the SQL functionality. In particular FastBit works extremely well on at-

tributes (columns) of the same relational table, evaluating a multi-attribute query by applying logical operation on the bitmap search results from each attribute. However, FastBit is not designed to perform join operations. In order to provide a complete SQL interface we integrated FastBit with the in-memory database system *MonetDB* being developed at CWI, Netherlands. We chose MonetDB since it is based on a column-wise storage model that is well-suited for *partial range queries* where only the attributes involved in the query are brought into memory for search. Since bitmap indices are designed to index one attribute at-a-time, MonetDB is a natural choice for the use of bitmap indices.

We present a detailed performance study to evaluate the efficiency of bitmap indices for both Boolean queries over text and traditional database queries over numerical values. Our performance experiments are based on the Enron email data set consisting of emails sent and received by Enron employees. This data set is particularly attractive for studies on index data structures since it contains numerical, categorical and text data. Typical database systems would use (1) B-trees for querying numerical and categorical values and (2) inverted files for text searches. Our performance study demonstrates that bitmap indices are very efficient for both.

The methodology used for the performance studies is based on separating text data (subject and body of the email messages) from the rest of the data (sender email address, recipient email address, day, time, etc.) This allowed us to perform separate queries on the *text data* and the more "conventional" data (*numerical data*). We then ran combined queries that require a join over the two tables. We executed a large number of queries (varying the hit ratio) using only FastBit, then combining results from the two tables by applying an external join (essentially a sort-merge algorithm), and finally running the same queries on the MonetDB/FastBit system. The results will be discussed in detail, but in general, we observed that the application of FastBit to both numerical and text data to perform combined queries has proven to be very efficient.

The paper is organized as follows. In Section 2 we revise the related work on index data structures for querying numerical and categorical values. In particular we focus our attention on bitmap indices. Next, we discuss databases systems that support full-text searching and motivate the advantages of compressed bitmap indices for querying both numerical and text data. In Section 3 we introduce the Enron data set which is large collection of email messages that was made public recently. This data set is particularly attractive for research on index data structures since it contains numerical, categorical 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 the open-source database management system MonetDB are described in Section 5. Next we perform a detailed experimental evaluation of querying both numerical and text data with FastBit and compare the performance with the integrated system MonetDB/FastBit as well as with MySQL (see Section 6). We summarize the findings of our studies in Section 7 and point out open research topics with respect to using bitmap indices for text searching.

## 2 Related Work

### *2.1 Querying Numerical and Categorical Values*

In the database community, a general strategy to reduce the time needed 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 application, indexing methods such as $B^+$-tree and hash-based indices were found to be particularly efficient [9, 20]. One notable characteristic of data in these applications is that they change constantly and therefore their associated indices must also be updated quickly.

As more data are accumulated over time, the need to analyze large historical data sets is gaining attention. The storage system for this type of relatively stable data is generally known as data warehouse [7,8,13]. The type of operations performed on data warehouses are often called On-Line Analytical Processing (OLAP). For these operations, bitmap indices are particularly efficient [18, 36,35]. In comparison with a typical $B^+$-tree, a bitmap index usually answers queries with a large number of hits faster than a $B^+$-tree, but it takes longer to update a bitmap index after an insertion of a new record or a modification of an existing record. However, for most data warehouses, the typical update operation is to append a large number of new records. In this case, appending new records to a bitmap index may even take less time than updating a $B^+$-tree because the time to append to bitmap indices is a linear function of the number of new records while the time to update a $B^+$-tree is always a superlinear function. For this reason, bitmap indices are well-suited for OLAP type of applications.

As an illustration, let us consider a bitmap index for an integer attribute **A** that takes its value from 0, 1, 2, and 3. In this case, we say that the attribute 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 attribute cardinality. It is easy to see that a basic bitmap index contains $CN$ bits in its bitmaps. As the attribute cardinality increases, the basic bitmap

| 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 |
| | | $b_1$ | $b_2$ | $b_3$ | $b_4$ |

**Fig. 1** A sample of an equality-encoded bitmap index, where RID is the record ID and **A** is an integer attribute with values in the range of 0 to 3.

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 groups.

*Binning* Instead of recording each individual value in a bitmap, the strategy of binning seeks to associate multiple values with a single bitmap [32,24]. For example, to index a floating-point valued attribute **B** with a domain between 0 and 1, we may divide the domain into a number of bins, say 10. In this case, a bitmap will be associated with each bin, such as $[0, 0.1)$, $[0.1, 0.2)$, ..., $[0.9, 1]$.

Binning clearly can control the number of bitmaps used. However, the index is no longer able to resolve all queries. For example, when answering a query involving the condition "$\mathbf{B} < 0.05$", the most helpful information we can get from the above binned index is that there are a number of records with **B** in the bin of $[0, 0.1)$. We call these records the candidate hits of the query (or simply candidates). These candidates have to be examined to determine exactly which ones are actually hits [26]. Checking the candidates often dominates the total query response time. A bitmap Index without binning can be thought of as binning with one bin per value.

*Encoding* We can view the output from binning as a set of bin numbers. The encoding step is to translate these bin numbers into bits in bitmaps. The basic bitmap uses an encoding, referred to as *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 [5,6]. In both range encoding and interval encoding, each bitmap corresponds to a range of bins. They are designed to answer range queries with one or two of the bitmaps. These encoding schemes can also be composed into multi-level and multi-component encodings [5,34]. One well-known example of a multi-component encoding is the binary encoding scheme [31, 19], where the $i$th binary digit of the bin number is concatenated together to form the $i$th bitmap of the index. This encoding produces the fewest number of bitmaps, however, to answer most queries, all of the bitmaps in the index are accessed. In contrast, other encoding schemes, such as the interval encoding, usually access only some of the bitmaps when answering a query.

*Compression* Compression can be applied on the bitmaps generated from the above binning and encoding steps to reduce the storage requirement. Any text compression technique may be used here. However, in order to reduce the query response time, specialized bitmap compression methods are preferred. One of the best-known bitmap compression methods is the Byte-aligned Bitmap Code by Antoshenkov [3].

Another efficient bitmap compression method is the Word-Aligned Hybrid (WAH) code [35,33]. In a number of timing measurements, it was shown to be about 10 times faster then BBC on a variety of datasets. 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. In the worst case, the proportionality constant is 4. This worst case index size is comparable with the typical size of a $B^+$-tree used in some popular DBMS.

Given a range condition involving an index compressed with WAH, the total response time is proportional to the number of hits. This is optimal in terms of computational complexity. In addition, compressed bitmap indices 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, thus performing multiple-attribute queries efficiently.

### *2.2 Database Systems for Full-Text Searching*

Supporting text retrieval in database systems has become an important research topic as many applications require queries that combine both text and other database attributes. The research literature discusses incorporating text retrieval capabilities into different types of database systems such as relational database systems [14] , object oriented databases [37] and XML databases [1] . Proposals for architecture for systems that combine relational databases with text searching capabilities are reported in [10] and [15]. Other papers deal with modeling issues, query languages and appropriate index structures [22,21] for such database systems. A more recent prototype of such a system, called QUIQ, is described in [14]. 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 indices current. It can also be updated in bulk-load mode. Another recent paper, [12], describes a benchmark called TEXTURE which examines the efficiency of database queries

that combine relational predicates and text searching. Several commercial database systems were evaluated by this benchmark.

Another approach for combining text retrieval and DBMS functionality is to use object oriented databases using the external function capability of the database system. A prototype that uses this approach is reported in [37] which combines a structured-text retrieval system (TextMachine) with an object-oriented database system (OpenODB).

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 in [1], it is proposed to extend the XQuery language with complex full-text searching capabilities.

Supporting text in databases requires appropriate index structures. One type of index proposed for text searching is called Signature files. The space overhead of this index is lower than inverted files (10%-20%) but the search is always sequential over the whole index. 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. This data structure consists of a vocabulary of all the terms and an inverted list structure [29]. 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), recent work [2] deals with methods of compressing inverted files. The authors also show that their method is quite efficient in decompressing the inverted files. A comprehensive survey of inverted files for text searching can be found in [39].

### *2.3 Compressing Inverted Files with Bitmap Indices*

The inverted indices commonly used for text searching are usually compressed as well [16,30,38]. One obvious difference between a bitmap compression method and a compression method for inverted indices is that they are designed to represent different types of data. A bitmap compression scheme represents bitmaps (0s and 1s), while a compression scheme for an inverted index generally compresses differences between successive document identifiers or term frequencies. The primary use of the compressed data in an inverted index is to reconstruct the document identifiers. For this reason, the compression methods designed for the inverted index used to be measured only by their compressed sizes. However, recently there has been some emphasis on compute efficiency as well [2,30]. In particular, Anh and Moffat have proposed a Word-Aligned Binary Compression for text indexing, which they call slide [2]. We note that 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. In particular, a WAH code word is always a machine word.

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.

In this paper, we apply compressed bitmap indices to inverted files. Such an approach was not considered viable in the past for indexing over a large number of terms, since bitmap indices were considered efficient only for searching attributes with low cardinality. However, the WAH-compressed indices have been shown to be very efficient even in the case of high cardinality attributes, and therefore are a good candidate for supporting search within text attributes in databases. We show in this paper that such indices are indeed very efficient when used for inverted file structures. A great advantage of using this method for text data is that it allows the WAH-based bitmap indices to combine efficiently queries for both numeric and text data. This is accomplished by simply applying logical operations on the resulting bitmaps of both types of attributes after the individual attributes are searched. Extensive performance experiments shown in this paper confirm this methodology.

**Table 1** Schema of database table EnronUniversal.

| Column Name | Explanation |
|---|---|
| mid | Message ID |
| senderFirstName | Only for Enron employees |
| senderLastName | Only for Enron employees |
| senderEmail | Email of any sender |
| recipientFirstName | Only for Enron employees |
| recipientLastName | Only for Enron employees |
| recipientEmail | Email of any recipient |
| day | Day email was sent |
| time | Time email was sent |
| rtype | Information about how message was sent: "TO", "CC", "BCC" |

**Table 2** Schema of database table EnronMessage.

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

## 3 Case Study: The Enron Data Set

The Enron data set, a large set of email messages, is used by various researchers in the areas of textual and social network analysis. The data set was made public by the Federal Energy Regulatory Commission during the criminal investigation into Enron's collapse in 2002. This data set is particularly attractive for studies on index data structures since it contains numerical, categorical and text data. Our case study is based on the data prepared by Shetty and Adibi [23] and contains 252,759 email messages stored in four MySQL tables, namely `EmployeList`, `Message`, `RecipientInfo` and `Reference Info`.

In early performance experiments comparing FastBit with MySQL, we showed that FastBit significantly outperforms MySQL for queries over *numerical* and *categorical* values [27]. One of the key findings of these experiments was to materialize parts of the tables in order to avoid expensive join operations during query processing.

In this paper we go one step further and also evaluate the performance of bitmap indices for *Boolean queries* over text. In particular, we search the *subject* and the *body* of the email messages for certain terms. In order to allow queries over both numerical and text data, we chose a different database schema design than originally proposed by Shetty and Adibi [23]. Rather than using 4 tables, our database schema only uses two tables called `EnronUniversal` and `EnronMessage` (see Tables 1 and 2).

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

Let us briefly point out one particular point about the content of table `EnronUniversal`. The first and last names of both the sender and the recipients are only available for some Enron employees. However, the full email addresses of the senders and recipients are available for all messages. This artifact might be due to a problem in the original collection of the email messages.

## 4 Extending Bitmap Indices to Support Full Text Search

In the past we successfully used FastBit's compressed bitmap index technology to efficiently query large sets of numerical data [27]. In this section we will describe how to extend bitmap indices to support Boolean queries over text data.

Indexing text usually requires the following two steps:

- Text parsing and term extraction
- Index generation

In our framework we use Lucence [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 idenfiers (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 the terms and a set of compressed bitmaps.

We will demonstrate our indexing framework with a simple example illustrated in Figure 2. Let us assume we have a database table called `EnronMessage` that contains the four columns `mid`, `body`, `subject` and `folder`. The column `mid` contains the Message ID used in the MySQL version of the Enron e-mail message dataset [23]. It has integer values. The columns `body`, `subject` and `folder` contain text values. In order to use Lucene to identify the terms contained in the messages, we store each message body in a separate file with the `mid` as the file name. In Figure 2 separation is indicated by "body1", "body2", etc. Lucene is used to parse each file and extract the terms from the documents. The output from Lucene is a list of terms, and for each term a list of files containing the term (the inverted list). Since the file names are the

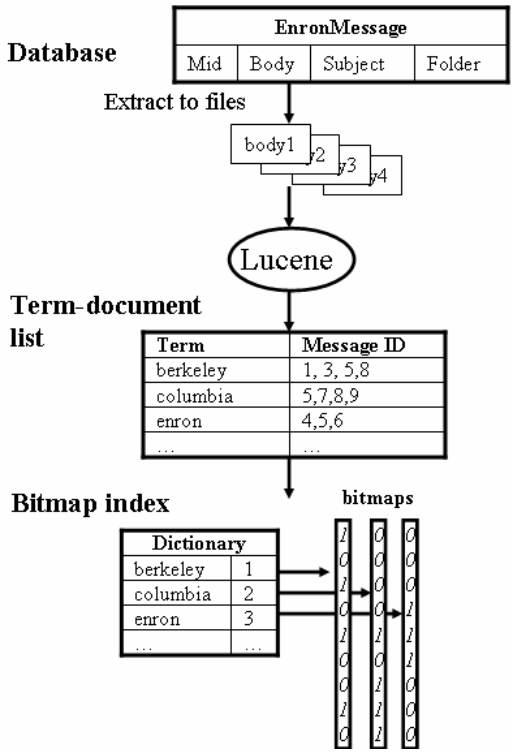



**Fig. 2** Framework for Indexing Text with FastBit.

mids, 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.

The next step is to build the bitmap index. However, before we can index the identified terms with bitmaps, we need to introduce an auxilary 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" 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. Similarily, 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 indices for storing term-document lists efficiently supports Boolean queries over terms. 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 we showed in the past, these basic bitmap operations are very efficient.

[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.

## 5 Integrating FastBit into MonetDB

We decided to integrate FastBit into a relational database 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 data. Second, by adding FastBit to a relational database system, relational data can benefit from FastBit's high performance when queries or subqueries are of a form that can use a FastBit index. With the addition of text searching to FastBit, adding FastBit to a relational system also provides a high performance tool for integrating Boolean queries over text with more traditional database queries.

The database system we targeted is MonetDB, an open source, in-memory database developed by CWI [17]. In this section we will begin by describing MonetDB and our reasons for choosing it. We will then present the details of our integration of FastBit into 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). See Figure 3 for a comparison 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 attributes is required [4, 25, 28]. 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 attribute $a2$ in each record. That is, the entire record is loaded even though only a small part of it is needed. With a DSM, single attributes are stored contiguously (Figure 3c) resulting in efficient I/O that involves only the required attributes. MonetDB's data layout is therefore analogous to FastBit indexing, where each attribute is indexed and stored separately.

The MonetDB SQL Server is a two-layer system [17]. 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 attribute. Each entry in the BAT is a two-field record containing an object identifier (OID) and an attribute data

| $a1$ | $a2$ | $a3$ |
|---|---|---|
| $a1_1$ | $a2_1$ | $a3_1$ |
| $a1_2$ | $a2_2$ | $a3_2$ |
| $\vdots$ | $\vdots$ | $\vdots$ |
| $a1_n$ | $a2_n$ | $a3_n$ |

**(a)** Logical Table
Each record (a row) has three attributes

$\{a1_1, a2_1, a3_1\}$
$\{a1_2, a2_2, a3_2\}$
$\vdots$
$\{a1_n, a2_n, a3_n\}$

**(b)** Horizontal Partitioning
Attributes from the same record
are stored contiguously

$\left\{\begin{matrix} a1_1 \\ a1_2 \\ \vdots \\ a1_n \end{matrix}\right\}$ $\left\{\begin{matrix} a2_1 \\ a2_2 \\ \vdots \\ a2_n \end{matrix}\right\}$ $\left\{\begin{matrix} a3_1 \\ a3_2 \\ \vdots \\ a3_n \end{matrix}\right\}$

**(c)** Vertical Partitioning
Same attribute from different records
are stored contiguously

**Fig. 3** Horizontal vs. Vertical Partitioning

value. All attribute values for the same relational tuple will have the same OID even though they are stored in separate BATs. Interaction with these BATs is accomplished via the Monet Interpreter Language (MIL), which can be extended with new commands as we describe in Section 5.2.4.

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 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 will 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 Nuts and Bolts*

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. In the following subsections we will describe each of these tasks.

MySQL:
```
SELECT COUNT(*)
FROM EnronMessage m, EnronUniversalFB u
WHERE MATCH(body) AGAINST('HAVE')
AND day < 20010101 AND m.mid = u.mid
```

MonetDB:
```
SELECT COUNT(*)
FROM EnronMessage m, EnronUniversalFB u
WHERE body = 'have'
AND dday < 20010101 AND m.mid = u.mid
```

**Fig. 4** The same query written for MySQL and MonetDB. In the case of MonetDB, we will assume that a FastBit index exists on the text attribute. If not, this query will test for the condition where attribute `body` equals 'have', rather than testing for the presence of the term 'have' in the `body`.

*5.2.1 SQL* To enable FastBit index use within MonetDB, the keyword "`FASTBIT`" was added to the parser. FastBit indices may be created using the standard "`CREATE INDEX`" syntax: "`CREATE FASTBIT INDEX index_name ON table_name (`$\mathtt{a}_1, \mathtt{a}_2, \ldots$`)`", where the $\mathtt{a}_k$ are the attributes within the table on which an index should be created. The order of the attributes does not matter, unlike with other index types.

MonetDB does not have a text searching capability such as MySQL's `MATCH (...) AGAINST (...)` built into its SQL parser. To overcome this problem, we chose to override the "`=`" operator in the `WHERE` clause. When "`=`" is used with a column that is a text attribute, i.e., a character blob but not a fixed or variable length string, we interpret that as a Boolean query for the presence of a term in that attribute. To search for multiple terms, several equality expressions may be combined using the normal SQL logical operators such as "`AND`" and "`OR`". Figure 4 shows an example of the syntax we used for doing full text searching within MonetDB with FastBit.

*5.2.2 Building Indices* Building FastBit indices within MonetDB begins with the parsing and execution of the "`CREATE FASTBIT INDEX`" command described in the previous section. First, the types of attributes that the user specified for the index are checked because FastBit supports only a subset of the data types found in MonetDB. Currently, FastBit can index integers, floating point numbers, character strings (variable and fixed length), and large text objects. The SQL Date type and user-defined data types are examples of attribute types that FastBit

cannot yet index. If all of the attribute types are acceptable, the data for each attribute is read from MonetDB's storage and passed to the FastBit library, which builds a compressed bitmap index over the attribute. The OIDs for the indexed records are also sent to FastBit, but they are not indexed. Instead they are stored and are used during query execution (see Section 5.2.4) to translate a FastBit result into a form that is usable by MonetDB/SQL.

As data is added or modified in the the underlying indexed relations, FastBit indexes must be updated or rebuilt. We trap for these events within MonetDB/SQL, but handle them in only a naïve way. Improving the update, delete, and append performance is future work.

*5.2.3 Recognizing FastBit Eligible Queries* Recognizing queries that can benefit from FastBit is a more challenging piece of the implementation. FastBit cannot support as rich a set of operations, such as joins, that a full-fledged database system like MonetDB can. Therefore, FastBit cannot answer an entire query by itself, but it can dramatically speed important parts of query processing. The key, then, is to determine what parts of the query, if any, FastBit can execute.

When MonetDB receives a query from a client it performs the same steps as most database management systems: the query string is parsed and the parse tree is used to generate a query plan, which can then be optimized and executed. In MonetDB an additional final step transforms the query plan into MIL code that is then sent to the MonetDB kernel for execution.

We use the tree representing the query plan (see Figure 5) to identify subqueries that FastBit can execute. In the query plan tree, subqueries are represented by subtrees, so our task was actually to identify appropriate subtrees. Instead of using the final tree, we found it easier to perform our FastBit subquery identification on an intermediate version of the tree–one that contains all of the necessary relational operations but that has not yet been made into a binary tree. This was convenient because a `WHERE` clause involving multiple attributes can be found in a single node at this stage. Identifying a subquery answerable by FastBit requires finding a subtree with the following properties:

1. The subtree must involve only one table. FastBit does not perform operations between tables such as joins.
2. The subtree must involve only attributes on which a FastBit index has been created.
3. The operations involving those attributes must be operations that FastBit can perform. For instance, SQL supports similarity matching, i.e., the `LIKE` comparison, but FastBit does not.

We use rules about the structure of the tree, including node types, to locate subtrees that satisfy the above requirements. When such a subtree is found, we replace it with a new node of type FastBit that contains the information necessary to execute a FastBit query equivalent to the subtree.

During the MIL code generation phase, the query plan tree is traversed and appropriate code is generated for the nodes encountered. We have extended MIL with a `fastbit_execute` command which is essentially a wrapper for a call to the FastBit library with some extra code to translate the FastBit result into a form that can be used by MonetDB. When a FastBit node is encountered during code generation, the FastBit command is created and supplied with the name of the index and attributes to use in the query as well as the location of the constants to be used in any comparisons.

*5.2.4 Integrating FastBit and MonetDB Execution* FastBit execution within MonetDB/SQL query execution begins when a `fastbit_execute` MIL command is encountered. Figure 6 shows a `fastbit_execute` MIL code snippet.

A call to `fastbit_execute`, as shown in Figure 6, provides the FastBit library with the following information. The first parameter is a data structure that contains system wide information and is a standard component of most commands. The second parameter is the name of the index, in this case "eu_fb". The third parameter contains the query string for FastBit to process; "%s" is a placeholder for the constants specified as `var A0` and `var A1` in this example. The fourth parameter tells FastBit how many attributes are in the query, and variables containing constants for comparison with those attributes follow in the next positions. This command can handle queries with any number of attributes.

The FastBit execution wrapper constructs a FastBit query from the information provided in the `fastbit_execute` command. This query returns an array that contains the OIDs of tuples that satisfy the query. These OIDS that were stored alongside the FastBit indices (Section 5.2.2) are the key to integrating the FastBit result into the larger MonetDB/SQL query plan. The original query plan would have returned a BAT containing the OIDs satisfying the subquery represented by the tree, so in order to integrate the FastBit result into MonetDB/SQL, the `fastbit_execute` command merely has to return a similarly constructed BAT. We do this by allocating a temporary BAT. We then store the OIDs into the BAT and return it as the result of the command's execution. For example, in Figure 6, when the `fastbit_execute` command completes, `var s7` will be a BAT containing the OIDs of the tuples which satisfied the query sent to FastBit.

## 6 Experiments

This section contains a discussion of our experimental results and demonstrates that adding compressed bitmap indices to a relational database system enables high

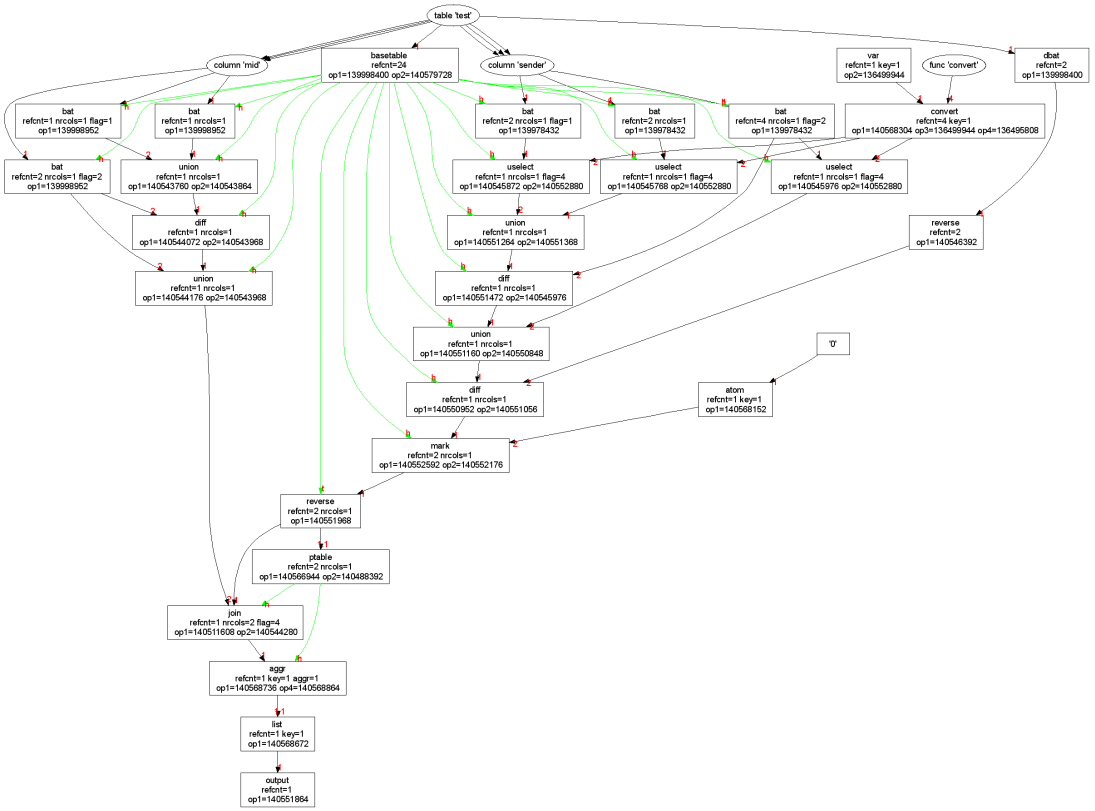



**Fig. 5** A query plan generated by MonetDB for answering a query over one attribute. If there is a FastBit index on the attributes, then the right half of this plan can be replaced by one FastBit operation.

```
var A0 := "john.smith@enron.com";
var A1 := int("20010101");
var s8 := A0;
var s9 := A1;
var s7 := fastbit_execute(myc, "eu_fb",
       "%s = senderemail AND dday < %s", 2, s8, s9);
       :
       <Other MIL Commands>
```

**Fig. 6** An MIL code snippet showing FastBit execution within MonetDB via the `fastbit_execute` command. The use of FastBit does not require any modification to the query execution following the FastBit command. That is, the result of the FastBit command is of the same form as the result that would have been generated by the subplan that it replaced.

performance, integrated querying of numerical 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 indices constructed for all attributes in the Enron Data Set. The remainder of the section presents the experimental 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 main areas: (1) queries over only numerical and categorical data, i.e., traditional relational data, (2) queries over only text data, and (3) queries across both numerical and text data involving a join between the two data sources.

Each query in the following experiments was issued both as a *count query* and as an *output query*, terminology we will use throughout this section. A *count query* returns only the number of hits, that is, the SQL `SELECT` clause contains `SELECT COUNT(*)`. An *output query* retrieves data values associated with the tuples in the result set. These two types of queries cover important classes of queries, namely those that generate statistics about data and those that retrieve sets of values matching the query conditions. Also 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 indices as well as restarted the database servers in order to ensure cold cache behavior.

**Table 3** Size of the raw data compared with the size of compressed bitmap indices for each column of the table `EnronUniversal`. For the categorical values also the size of the dictionary is given.

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

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

| Column | Card. | Data [MB] | td-list [MB] | Dict. [MB] | Bitmap [MB] |
|---|---|---|---|---|---|
| mid | 252,759 | 1.01 | | | 8.09 |
| subject | 38,915 | 7.56 | 8.20 | 0.31 | 5.23 |
| body | 1,247,922 | 445.27 | 245.57 | 16.92 | 121.72 |
| folder | 3,380 | 20.98 | 8.09 | 0.04 | 0.14 |

### *6.1 Data Statistics*

The Figures 7a and 7b show the term frequency distributions of the "body" and the "subject" of the Enron emails. The terms are extracted with Lucene. Note that 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 Indices*

Table 3 shows the size of raw data compared with the size of the compressed bitmap indices for each column of the table `EnronUniversal`. The table also shows the attribute cardinalities of the respective columns to better demonstrate the potential size of the indices. Apart from the index size for the column `mid`, the sizes of the compressed bitmap indices are much smaller than the raw data. Consider, for instance, the size of the bitmap index for the column `recipientEmail` which has an attribute cardinality of nearly 70,000. Even for such a high-cardinality attribute, the size of the compressed bitmap is only about 29% of the raw data. For the column `senderEmail`, which has a lower attribute cardinality, the size of the bitmap index is only about 3% of the raw data.

Table 4 shows the size of the compressed bitmap indices for the table `EnronMessage`, 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. Let us consider the index size for the email body, which contains more than 1.2 million distinct terms. 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. This result clearly shows that the size of compressed bitmap indices is reasonably small even for indexing term-document lists with very high cardinalities.

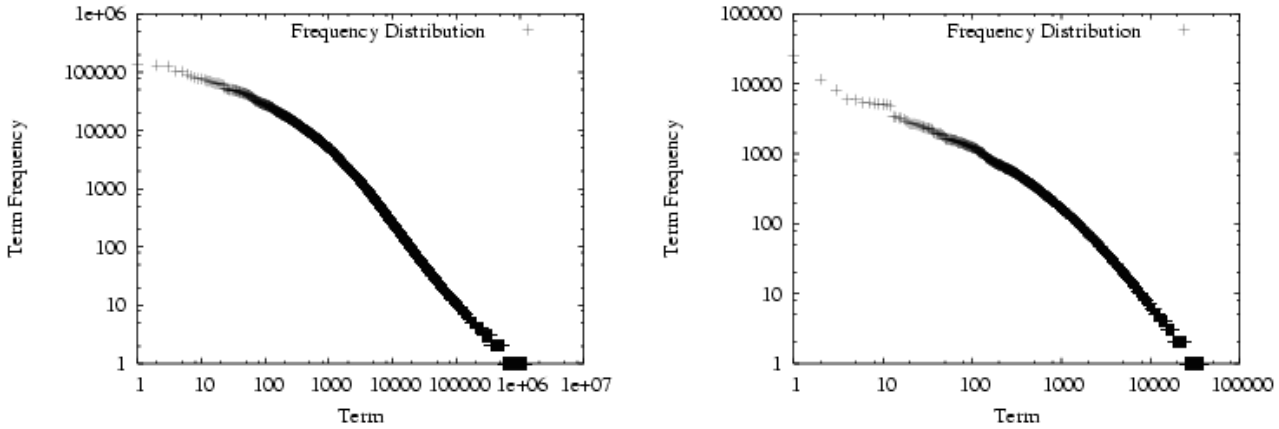
### *6.3 Query Performance for Numerical and Categorical Values*

In this section we evaluate the performance of FastBit for querying numerical and categorical values of the table `EnronUniversal`. We performed one, two, and three dimensional queries over the table. In this section we will begin with a brief comparison of MonetDB with FastBit against MonetDB without FastBit. Then we will compare the performance of MySQL, MonetDB with FastBit, and the FastBit stand-alone client for both the count and output queries.

*6.3.1 MonetDB With and Without FastBit* Figure 8 shows the performance of MonetDB with and without FastBit indices for one, two, and three-dimensional count queries. In many, but not all cases, MonetDB with FastBit outperforms MonetDB without FastBit. A key to understanding these performance graphs is to recall from Section 5.2.4 that integrating the FastBit query result into MonetDB requires copying the OIDs returned by FastBit into a BAT that MonetDB can understand. As the total number of hits increases, more copying is required. In Figures 8a and 8b the effect of this copying is evident as the number of hits becomes very large.

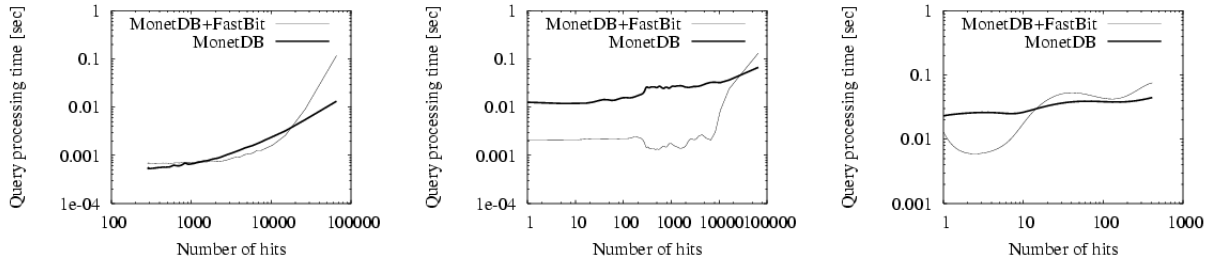
Figure 8b, a two-dimensional query, demonstrates FastBit's ability to improve query performance. Whereas MonetDB performs selection over two BATs and joins the results, FastBit can directly combine the indices of both attributes to answer the query. The three- dimensional query shown in Figure 8c shows MonetDB with FastBit to be better or about as good as MonetDB without FastBit.

*6.3.2 Count Queries* After evaluating the performance of integrating FastBit with MonetDB, we will now compare the performance of MySQL, FastBit and




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

**Fig. 7** Term frequency distribution in the message "body" and "subject" of the Enron data set.




**(a)** 1D Query **(b)** 2D Query **(c)** 3D Query

**Fig. 8** Performance of MonetDB with FastBit vs. MonetDB without FastBit on count queries of one, two, and three dimensions. Note that for the three dimensional queries (c), no query produced more than 1000 hits so the scale of this graph differs from the others.

MonetDB/FastBit for queries over the table `EnronUniversal`. The first set of experiments shows one, two, and three dimensional count queries (see Figures 9a to c). As already previously stated, for each experiment we executed 1000 queries with different equality conditions.

Consider, for instance, the query:

```
SELECT count(*)
FROM EnronUniversal
WHERE senderEmail' = :S
```

In this case, "`:S`" means that this value varies for all the 1000 queries. In particular, we have chosen the top 1000 senders and recipients of emails. The results clearly show that for these types of multi-dimensional count queries, FastBit significantly outperforms MySQL. On average, FastBit is about a factor of 30 faster than MySQL (for details see the summary Table 5).

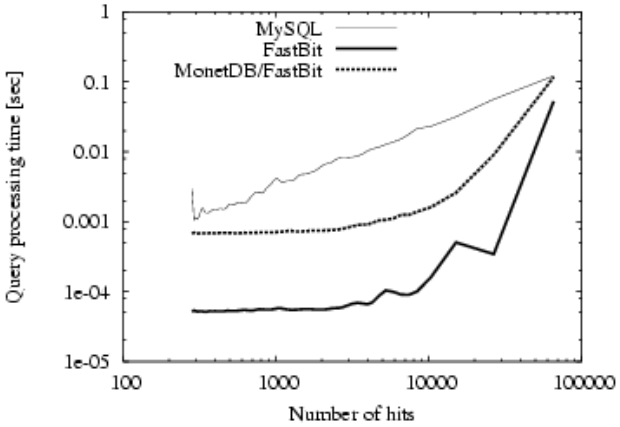
*6.3.3 Output Queries* In the next set of experiments we measured the performance of output queries (see Figures 9d to e). These experiments clearly show that MonetDB/FastBit is the overall winner. We also note that FastBit by itself performs worse than the integrated system MonetDB/FastBit. The reason for this behavior of FastBit is relying on its bitmap indices for the retrieval of string values. This approach is not efficient because the index size is larger than the size (number of bytes) of the selection. In our tests, the number of hits in join queries is not more than 100,000, but the index sizes are on the order of megabytes. The approach used by MonetDB, which directly accesses the raw data is more efficient.

In summary, these results underline the performance advantage of integrating FastBit with MonetDB.

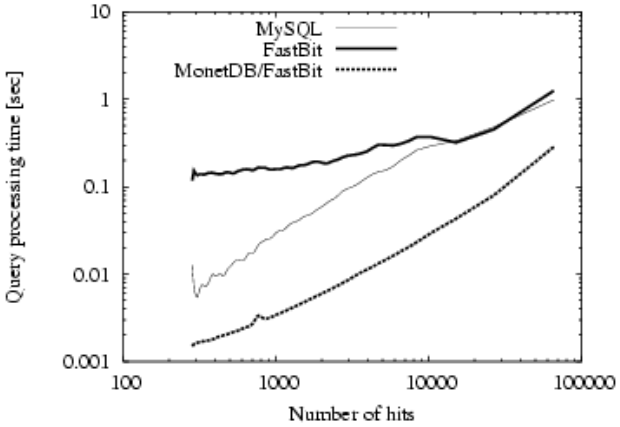
### *6.4 Query Performance for Text Searching*

In this section we evaluate the performance of FastBit's Boolean text search capability against MySQL's full-text index based on inverted files. All experiments measure the response time for retrieving the document IDs (emails) that match a certain search criterion. Note that FastBit does not perform any scoring operation for ranking the query results.

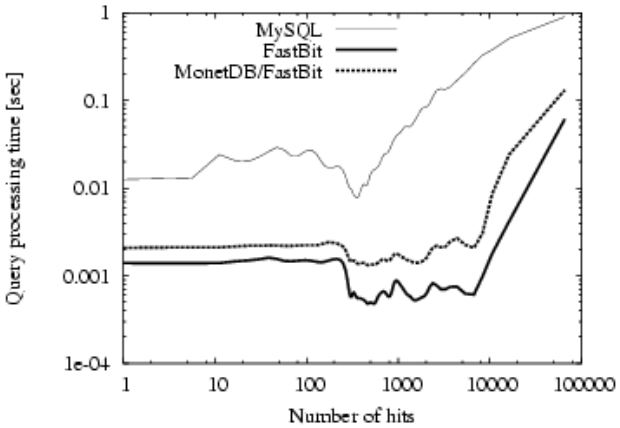
For the query conditions we chose the top 1000 most frequent terms. Using the top 1000 most frequent terms is also "interesting" from a text analysis point of view



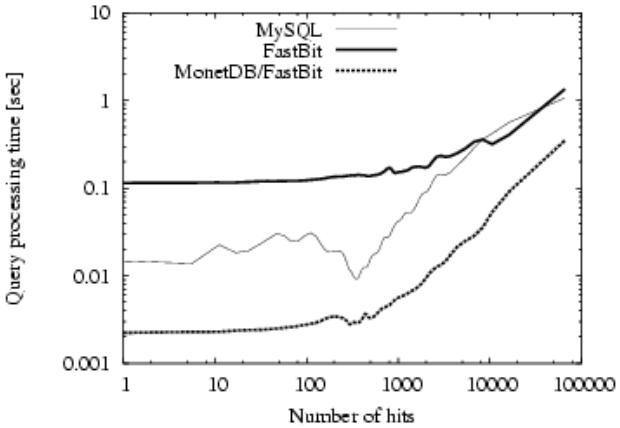

**(a)** 1D Count Query:
SELECT count(*)
FROM EnronUniversal
WHERE senderEmail = :S



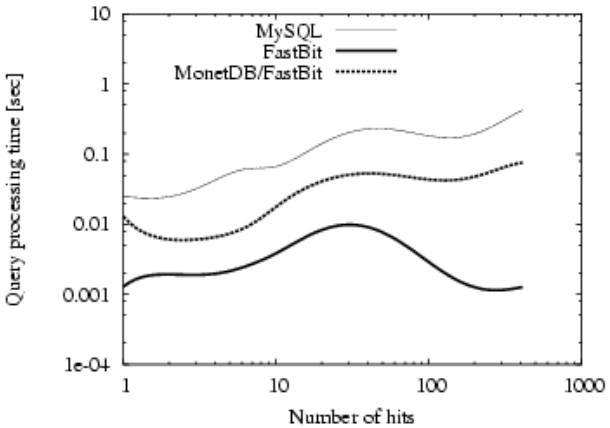

**(d)** 1D Output Query:
SELECT recipientEmail
FROM EnronUniversal
WHERE senderEmail = :S



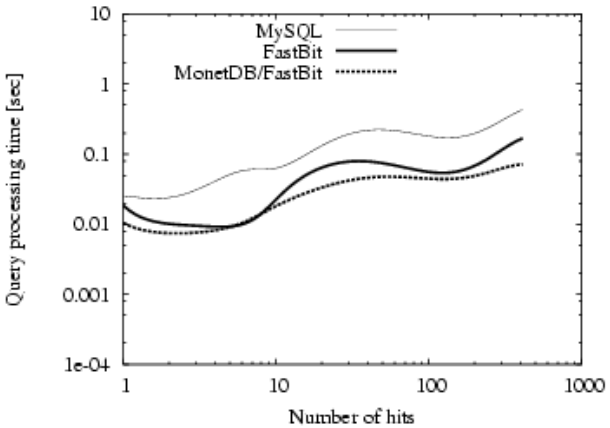

**(b)** 2D Count Query:
SELECT count(*)
FROM EnronUniversal
WHERE senderEmail = :S AND day < :D




**(e)** 2D Output Query:
SELECT recipientEmail, day
FROM EnronUniversal
WHERE senderEmail = :S AND day < :D

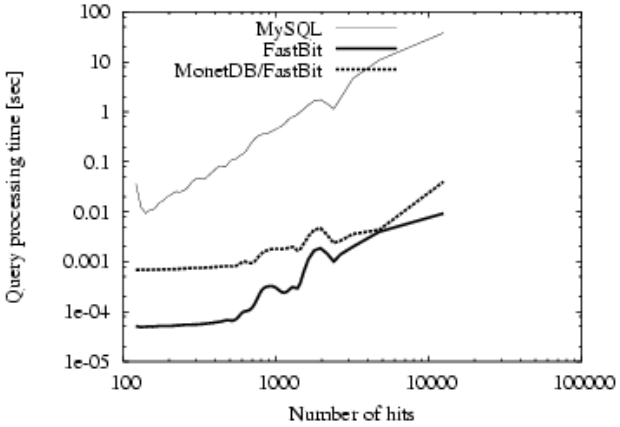



**(c)** 3D Count Query:
SELECT count(*)
FROM EnronUniversal
WHERE senderEmail = :S AND day < :D
AND recipientEmail = :R

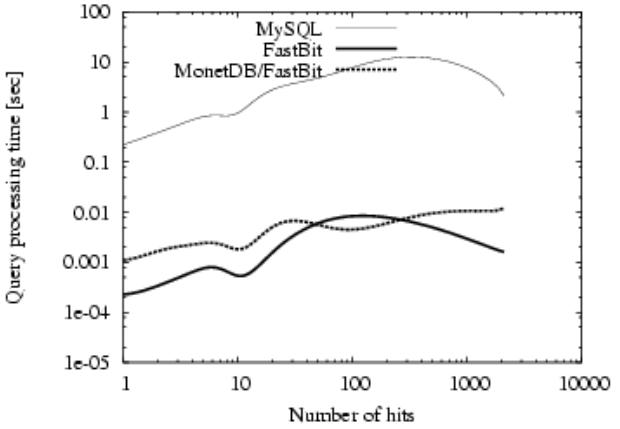



**(f)** 3D Output Query:
SELECT day, time
FROM EnronUniversal
WHERE senderEmail = :S AND day < :D
AND recipientEmail = :R

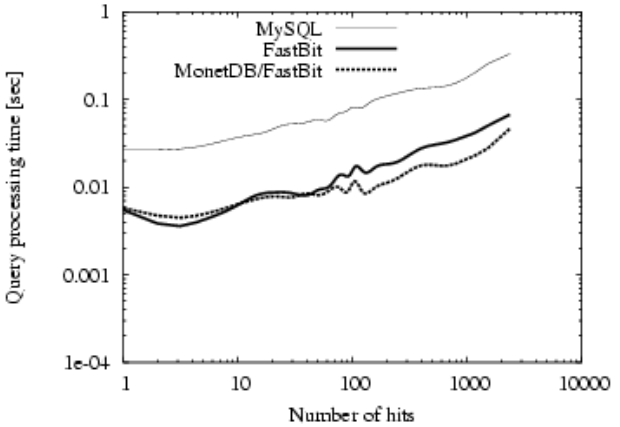
**Fig. 9** Count and output queries on the table `EnronUniversal`. A summary of the performance measurements is given in Table 5.



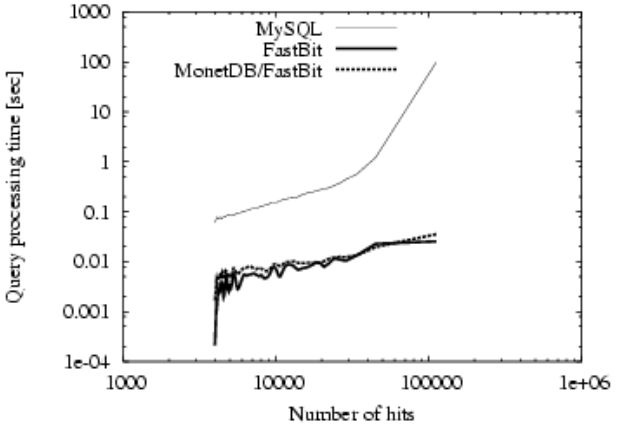

**(a)** 1D Subject Query:
SELECT count(*)
FROM EnronMessage
WHERE MATCH(subject) AGAINST (:S1)



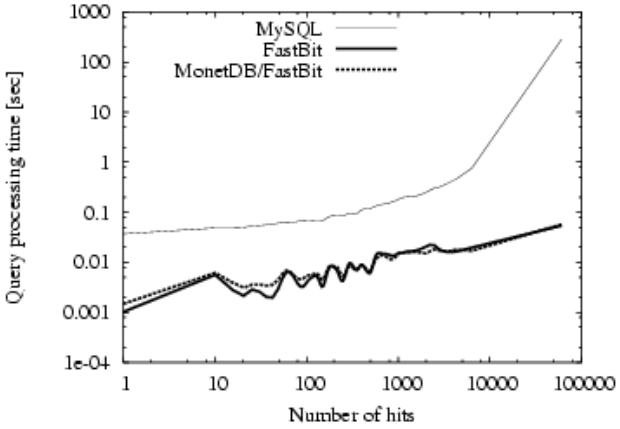

**(b)** 2D Subject Query:
SELECT count(*)
FROM EnronMessage
WHERE MATCH(subject) AGAINST (:S1)
AND MATCH(subject) AGAINST (:S2)



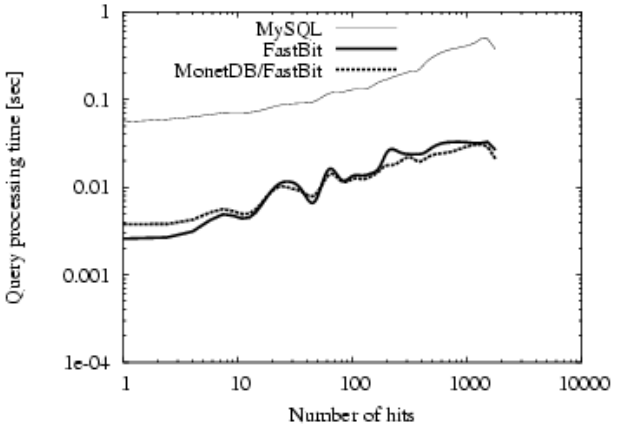

**(c)** 2D Subject and Body Query:
SELECT count(*)
FROM EnronMessage
WHERE MATCH(subject) AGAINST (:S1)
AND MATCH(body) AGAINST (:B1)




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




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




**(f)** 3D Body Query:
SELECT count(*)
FROM EnronMessage
WHERE MATCH(body) AGAINST (:B1)
AND MATCH(body) AGAINST(:B2)
AND MATCH(body) AGAINST (:B3)

**Fig. 10** Count queries on the subject and body attributes of the table `EnronMessage`. A summary of the performance measurements is given in Table 6.

since these terms are more discussed among people and might thus have a higher semantic meaning.

Figure 10 shows the response times for 1000 Boolean text queries over the subject and the body of the email messages from the Enron data set. Note that 10a searches for one term contained in the message subject while the query depicted in Figure 10b searches for two terms. Similarly, Figures 10d, 10e, and 10f show one, two, and three term queries for the message body, respectively.

FastBit is about a factor of 500 faster than MySQL for Boolean queries over the message subject (for details on the total query executation times see Table 6). For Boolean queries over the message body, the performance improvement of FastBit over MySQL is on average about a factor of 30 (see rows 4 to 6 in Table 6).

The graph in Figure shows the performs of queries over body and subject. This query is particularly interesting since it searches for terms in two different columns. The results show that for queries with a lower number of hits, FastBit is the winner. As the number of hits increases, the integrated system MonetDB/FastBit shows the best performance.

### *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 `Enron` Universal and `EnronMessage`. Note that FastBit by itself currently does not support join operations so 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 `EnronUniversal`. Analogously, the second query evaluates the query condition on the table `EnronMessage`. 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. In addition to the slow retrieval time for FastBit, this ad hoc join procedure is slow because the need to parse the long query string involving the mids.

For count queries (see Figure 11) FastBit is again the most performant strategy. We also note that the performance of the integrated system MonetDB/FastBit is only slightly worse.

As expected from our previous results, for output queries, the integrated system MonetDB/FastBit performs the best (see Figure 12).

### *6.6 Summary of Performance Results*

A summary of all the results is presented in Tables 5, 6 and 7. These tables show the total time for running 1000 queries against various tables as well as the speedup factors of FastBit and MonetDB/FastBit with respect to MySQL.

**Table 5** Total time in seconds for running 1,000 queries against the table EnronUniversal. The first set of times is for count queries. The second set is for select queries. This table is a summary of the results presented in Figure 9.

| | MySQL | FastBit | MonetDB/FastBit |
|---|---|---|---|
| Fig. 9a | 5.17 | 0.17 | 1.53 |
| Fig. 9b | 51.56 | 1.29 | 2.73 |
| Fig. 9c | 30.84 | 1.57 | 3.19 |
| Total time | 87.57 | 3.03 | 7.45 |
| Speedup | | 28.9 | 11.75 |
| Fig. 9d | 47.66 | 181.64 | 6.35 |
| Fig. 9e | 53.17 | 95.93 | 5.86 |
| Fig. 9f | 30.90 | 2.84 | 3.30 |
| Total time | 131.73 | 280.41 | 15.51 |
| Speedup | | 0.47 | 8.49 |

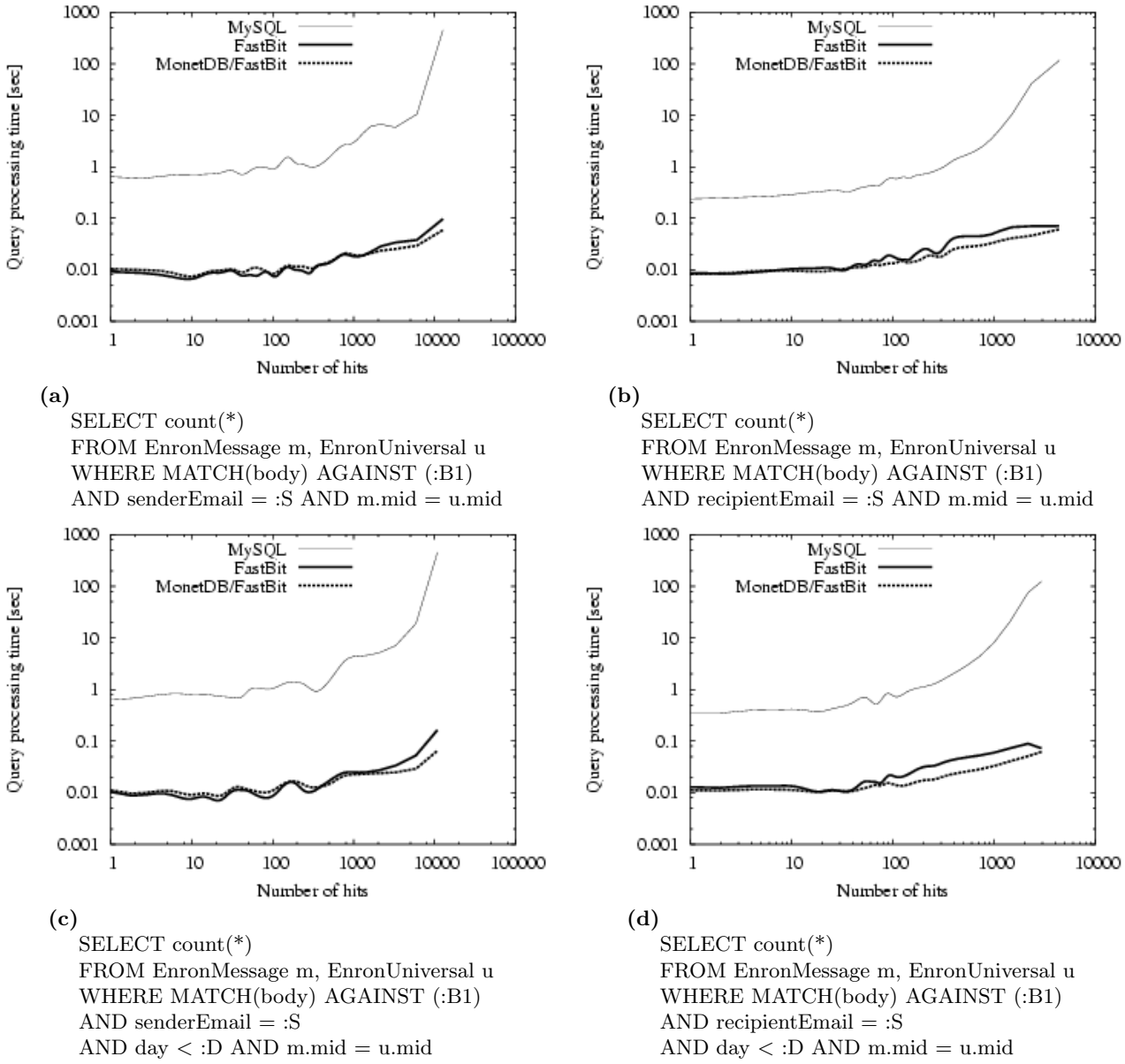
**Table 6** Total time in seconds for running 1,000 count queries against the table EnronMessage. This table is a summary of the results presented in Figure 10.

| | MySQL | FastBit | MonetDB/FastBit |
|---|---|---|---|
| Fig. 10a | 324.79 | 0.58 | 1.56 |
| Fig. 10b | 311.11 | 0.58 | 1.45 |
| Fig. 10c | 41.08 | 15.68 | 12.68 |
| Fig. 10d | 532.12 | 13.34 | 13.32 |
| Fig. 10e | 518.70 | 18.06 | 15.71 |
| Fig. 10f | 515.64 | 20.66 | 17.51 |
| Total time | 2243.44 | 68.90 | 62.23 |
| Speedup | | 32.56 | 36.05 |

## 7 Conclusions

We have investigated a way of using compressed bitmap indices to represent the commonly used term-document matrix to support Boolean queries on text data. By using a compute-efficient compression technique, we not only are able to keep the indices compact but also make it possible to answer Boolean queries very efficiently. In our detailed experimental study we show that our bitmap index technology called FastBit answers Boolean count queries over text data about 30 times faster than MySQL.

To further extend the functionality of FastBit, we have also integrated our technology with the open-source database management system called MonetDB. The advantage of this integration is that MonetDB can leverage an efficient text search capability for Boolean queries. Analogously, FastBit's benefits from the integration is full support of the SQL interface that was previously not available in FastBit. Our performance experiments




**(a)**

```
SELECT count(*)
FROM EnronMessage m, EnronUniversal u
WHERE MATCH(body) AGAINST (:B1)
AND senderEmail = :S AND m.mid = u.mid
```

**(b)**

```
SELECT count(*)
FROM EnronMessage m, EnronUniversal u
WHERE MATCH(body) AGAINST (:B1)
AND recipientEmail = :S AND m.mid = u.mid
```

**(c)**

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

**(d)**

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

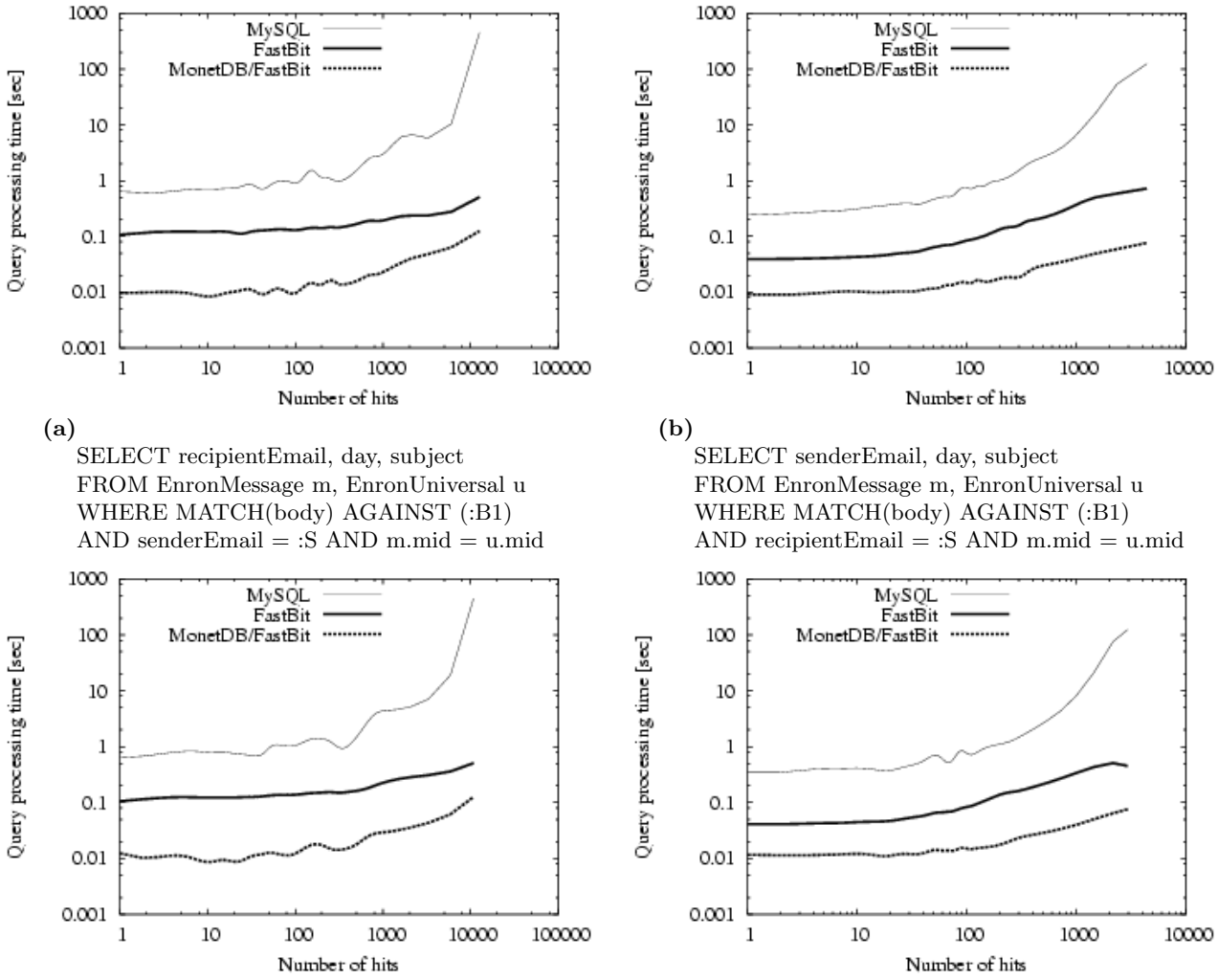
**Fig. 11** Integrated numerical and text count queries. These queries are shown in the format used by MySQL. A summary of the performance measurements is given in Table 7.

demonstrate that the integrated system significantly reduces the time required to answer join queries over both numerical and text data. It addition, the combined system shows significant performance improvements for retrieving string values subject to a multi-dimensional query condition.

The work presented in this paper extends the FastBit bitmap index technology to efficiently query both numerical and text data. Our current research focused on Boolean queries on text. In the future we will investigate the feasibility of using compressed bitmap indices for other types of text searches such as rank or similarity queries. Another avenue for further research is to compare the compression techniques used for bitmap indices with the compression techniques used for inverted indices.

## Acknowledgment

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 U.S. Department of Homeland Security Fellowship administered by Oak Ridge Institute for Science and Education. We would like to thank the MonetDB Team at CWI, Netherlands for their great support and technical




**(a)**

```
SELECT recipientEmail, day, subject
FROM EnronMessage m, EnronUniversal u
WHERE MATCH(body) AGAINST (:B1)
AND senderEmail = :S AND m.mid = u.mid
```

**(b)**

```
SELECT senderEmail, day, subject
FROM EnronMessage m, EnronUniversal u
WHERE MATCH(body) AGAINST (:B1)
AND recipientEmail = :S AND m.mid = u.mid
```

**(c)**

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

**(d)**

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

**Fig. 12** 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.

assistance during the integration of FastBit into MonetDB.

## References

1. Sihem AmerYahia, Chavdar Botev, and Jayavel Shanmugasundaram. TeXQuery: A FullText Search Extension to XQuery. In *WWW2004*, New York, New York, USA, May 2004.
2. Vo Ngoc Anh and Alistair Moffat. Improved Word-Aligned Binary Compression for Text Indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, 2006.
3. G. Antoshenkov. Byte-aligned Bitmap Compression. Technical report, Oracle Corp., 1994. U.S. Patent number 5,363,098.
4. Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *The VLDB Journal*, pages 54–65, 1999.
5. C.-Y. Chan and Y. E. Ioannidis. Bitmap Index Design and Evaluation. In *SIGMOD*, Seattle, Washington, USA, June 1998. ACM Press.
6. C. Y. Chan and Y. E. Ioannidis. An Efficient Bitmap Encoding Scheme for Selection Queries. In *SIGMOD*, Philadelphia, Pennsylvania, USA, June 1999. ACM Press.
7. S. Chaudhuri and U. Dayal. An Overview of Data wharehousing and OLAP Technology. *ACM SIGMOD Record*,

**Table 7** Total time in seconds for running 1,000 join queries against the tables EnronUniversal and EnronMessage. The first set of times is for count queries. The second set is for select queries. This table is a summary of the results presented in Figures 11 and 12.

| | MySQL | FastBit | MonetDB/FastBit |
|---|---|---|---|
| Fig. 11a | 1302.28 | 16.54 | 16.03 |
| Fig. 11b | 866.98 | 23.37 | 17.45 |
| Fig. 11c | 1312.75 | 18.15 | 16.93 |
| Fig. 11d | 975.18 | 24.81 | 18.10 |
| Fig. 11e | 556.80 | 6.40 | 2.98 |
| Fig. 11f | 436.64 | 8.18 | 4.10 |
| Total time | 5450.63 | 97.45 | 75.59 |
| Speedup | | 55.93 | 72.11 |
| Fig. 12a | 1303.24 | 82.55 | 18.24 |
| Fig. 12b | 970.31 | 78.64 | 18.51 |
| Fig. 12c | 1313.37 | 54.62 | 17.96 |
| Fig. 12d | 977.99 | 53.00 | 19.17 |
| Fig. 12e | 557.27 | 21.24 | 3.35 |
| Fig. 12f | 463.98 | 21.45 | 4.44 |
| Total time | 5586.16 | 311.5 | 81.67 |
| Speedup | | 17.93 | 68.4 |

26(1):65–74, March 1997.
8. Surajit Chaudhuri, Umeshwar Dayal, and Venkatesh Ganti. Database technology for decision support systems. *Computer*, 34(12):48–55, December 2001.
9. Douglas Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, 1979.
10. A. de Vries and A. Wilschut. On the Integration of IR and Databases. In *IFIP 2.6 DS-8 Conference*, Rotorua, New Zealand, January 1999.
11. et al. Doug Cutting. Apache lucene. http://lucene.apache.org.
12. Vuk Ercegovac, David J. DeWitt, and Raghu Ramakrishnan. The texture benchmark: Measuring performance of text queries on a relational dbms. In *International Conference on Very Large Data Bases*, pages 313–324, 2005.
13. W.H. Inmon and R.D. Hackathorn. *Using the data warehouse*. Wiley-QED Publishing, Somerset, NJ, USA, 1994.
14. N. Kabra, R. Ramakrishnan, and V. Ercegovac. The QUIQ Engine: A Hybrid IR-DB System. Technical Report TR-1449, University of Wisconsin, October 2002.
15. V.V. Raghavan L.V. Saxton. Design of an integrated information retrieval/database management system. *IEEE Transactions on Knowledge and Data Engineering*, 2:210–219, June 1999.
16. Alistair Moffat and Justin Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transcations on Information Systems*, 14(4):349–379, 1996.
17. MonetDB: Query Processing at Light-Speed. http://monetdb.cwi.nl.
18. P. O'Neil. Model 204 Architecture and Performance. In *2nd International Workshop in High Performance Transaction Systems*, Asilomar, California, USA, September 1987. Springer-Verlag.
19. P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proceedings ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, USA, May 1997. ACM Press.
20. Patrick O'Neil and Elizabeth O'Neil. *Database: pronciples, programming, and performance*. Morgan Kaugmann, 2nd edition, 2000.
21. Hans-Joerg Schek. Nested Transactions in a combined IRS-DBMS Architecture. In *International ACM SIGIR Conference on Research and Development in Information Retrieval*, Cambridge, England, July 1984. .
22. Hans-Jrg Schek and Peter Pistor. Data Structures for an Integrated Data Base Management and Information Retrieval System. In *International Conference on Very Large Data Bases*, Mexico City, Mexico, September 1982. Morgan Kaufmann.
23. J. Shetty and J. Adibi. The Enron Email Dataset, Database Schema and Brief Statistical Report. Technical report, Information Sciences Institute, Marina del Rey, California, 2006.
24. A. Shoshani, L. M. Bernardo, H. Nordberg, D. Rotem, and A. Sim. 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*, pages 214–225. IEEE Computer Society, 1999.
25. Arie Shoshani. OLAP and statistical databases: similarities and differences. In *Principles Of Database Systems (PODS)*, pages 185–196, 1997.
26. K. Stockinger, K. Wu, and A. Shoshani. Evaluation Strategies for Bitmap Indices with Binning. In *International Conference on Database and Expert Systems Applications (DEXA)*, Zaragoza, Spain, September 2004. Springer-Verlag.
27. Kurt Stockinger, Doron Rotem, Arie Shoshani, and Keshend Wu. Bitmap Indexing Outperforms MySQL Queries by Several Orders of Magnitude. Technical Report LBNL-59437, Berkeley Lab, Berkeley, California, 2006.
28. Sybaseiq. http://www.sybase.com/products/information management/sybaseiq.
29. A. Tomasic, H. Garcia-Molina, and K. A. Shoens. Incremental Updates of Inverted Lists for Text Document Retrieval. In *SIGMOD Conference*, Minneapolis, Minnesota, USA, May 1994.
30. Andrew Trotman. Compressing inverted files. *Information Retrieval*, 6:5–19, 2003.
31. H. K. T. Wong, H.-F. Liu, F. Olken, D. Rotem, and L. Wong. Bit transposed files. In *Proceedings of VLDB 85, Stockholm*, pages 448–457, 1985.
32. K.-L. Wu and P. Yu. Range-based bitmap indexing for high cardinality attributes with skew. Technical Report RC 20449, IBM Watson Research Division, Yorktown Heights, New York, May 1996.
33. Kesheng Wu, Ekow Otoo, and Arie Shoshani. An efficient compression scheme for bitmap indices. *ACM Transactions on Database Systems*, 31:1–38, 2006.
34. Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Compressed bitmap indices for efficient query processing. Technical Report LBNL-47807, LBL, Berkeley, CA, 2001.
35. Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. On the performance of bitmap indices for high cardinality at-

tributes. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 24–35. Morgan Kaufmann, 2004.

36. M.-C. Wu and A. P. Buchmann. Encoded bitmap indexing for data warehouses. In *Fourteenth International Conference on Data Engineering, February 23-27, 1998, Orlando, Florida, USA*, pages 220–230. IEEE Computer Society, 1998.
37. Tak W. Yan and Jurgen Annevelink. Integrating a Structured-Text Retrieval System with an Object-Oriented Database System. In *International Conference on Very Large Databases*, pages 740–749, Santiago, Chile, 1994.
38. Nivio Ziviani, Edleno Silva de Moura, Gonzalo Navarro, and Ricardo Baeza-Yates. Compression: a key for next-generation text retrieval systems. *IEEE Computer*, 33:37–44, 2000.
39. Justin Zobel and Alistair Moffat. Inverted Files for Text Searching. *ACM Computing Surveys*, 38(3), July 2006.