TERM MATCHING AND BIT-SLICED INDEX ARITHMETIC

A Dissertation Presented

by

Denis Rinfret

Submitted to the Oce of Graduate Studies, University of Massachusetts Boston, in partial fulllment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

June 2002

Computer Science Program

TERM MATCHING AND BIT-SLICED INDEX ARITHMETIC

A Dissertation Presented

by

Denis Rinfret

Approved as to style and content by:

Patrick O'Neil, Professor
Chairperson of Committee

Elizabeth O'Neil, Professor
Member

Dan A. Simovici, Professor
Member

Betty J. Salzberg, Professor
Member

Dan A. Simovici, Program Director
Computer Science Program

Peter Fejer, Chairperson
Computer Science Department

ABSTRACT

TERM MATCHING AND BIT-SLICED INDEX ARITHMETIC

June 2002

Denis Rinfret, B.S., Université du Québec à Trois-Rivières
Ph.D., University of Massachusetts Boston

Directed by Professor Patrick O'Neil

The bit-sliced index (BSI) was originally dened in [OQ97]. The current paper introduces the concept of BSI arithmetic. For any two BSI's X and Y on a table T, we show how to eciently generate new BSI's S, U, V, W, and Z such that $S = X + Y$, $U = X \quad Y$, $V = X \quad Y$, $W = c \quad X$ (where c is a constant integer), $Z = MIN(X, Y)$; this means that if a row r in T has a value x represented in BSI X and a value y in BSI Y, the value for r in BSI S will be $x + y$, the value in U will be $x \quad y$, the value in V will be $x \quad y$, the value in W will be $c \quad x$, and the value in Z will be $MIN(x, y)$. Since a bitmap representing a set of rows is the simplest bit-sliced index, BSI arithmetic is the most straightforward way to determine multisets of rows (with duplicates) resulting from the SQL clauses `UNION ALL` (addition), `EXCEPT ALL` (subtraction), and `INTERSECT ALL` (minimum) (see [OO00a, IBM] for denitions of these clauses). Another contribution of the current work is to generalize BSI range restrictions from [OQ97] to a new non-Boolean form: to determine the top k BSI-valued rows, for any meaningful value k between one and the total number of rows in T. Together with bit-sliced addition, this permits us to solve a common basic problem of text retrieval: given an object-relational table T of rows representing documents, with a collection type column K representing keyword terms, we demonstrate an ecient algorithm to nd k documents that share the largest number of terms with some query list Q of terms. This problem is called “simple term matching”. A more complicated problem, called “weighted term matching”, uses document and term weights, and the cosine similarity function to compute similarities between documents and queries. A great deal of published work on such problems exists in the Information Retrieval (IR) eld. The algorithm we introduce, which we call *Bit-Sliced Term Matching*, or BSTM, uses an approach favorably comparable in performance to the most e-cient known IR algorithm, a major improvement on current DBMS text searching algorithms, with the advantage that it uses only indexing we propose for native database operations.

A ma mere . . .

Table of Contents

List of Figures

CHAPTER 1

Introduction

To begin with, I will review the concept of *Term Matching (TM)* and *Ranked Queries*, as dened in *Information Retrieval (IR)*. I will describe basic and more advanced algorithms from the IR eld solving the TM problem, and discuss some drawbacks of those methods. The motivation of this thesis comes from these drawbacks, and I will propose new algorithms using data structures already available natively in up-to-date Object Relational Database Management Systems (ORDBMS). This work is based on a paper written with Patrick O'Neil and Betty O'Neil, published in SIGMOD 2001 [ROO01]. It did not include work on Weighted TM, except a short section on how it could be done. This dissertation focuses on weighted TM, since it is a generalization of simple TM, and it is more useful than simple TM in practice. This work includes also a chapter on future work, which could lead to other interesting research projects.

A *Ranked Query* is a query in which we want to not only get a set of rows back from a DBMS based on some criteria, but we also want to get a rank for each of the elements returned by the query. Commonly, the rank is a percentage value. For each element returned, we get a number between 1 and 100, the higher the value, the more meaningful the element is. The elements are returned in order by their rank, most signicant rst. Ranked queries are often more interesting than non-ranked queries, especially when dealing with user initiated queries and large return sets. For example, when making a search on the World Wide Web, getting a few hundreds or thousands (or more) unranked links can be quite time consuming, but if the links are sorted by their signicance, looking only at the rst few links may be enough to nd what we need. Ranked queries are obviously worthwhile.

Term Matching (TM) algorithms are a way to allow ranked queries to be executed on a textual database . If the documents in the database are very large, it can be useful to break the long documents into pages or *passages*, since executing a ranked query on small documents can be faster (less terms per document to process). It can also help a user by pointing him to a particular passage of a document, instead of presenting him a document of possibly hundreds of pages,

The techniques introduced in this work can be extended to non-textual databases, but we will look at that later.

and telling him that a passage of the document is meaningful, without knowing exactly where that passage is in the document. See [KZS99] for more on passage ranking. In the remainder of this work, I will refer to a document as a piece of text, either a full document, or a piece (a passage) of a larger document.

For every document in the database, *terms* are obtained by parsing the documents. The terms are usually words or expressions extracted from the documents, where some preprocessing can be done before the indexes are built on the database. Here is a list of possible term preprocessing steps:

1. document format ltering (remove HTML tags like `<H1>` and `</P>` and remove image anchors)
2. stop-words removal ("a", "and", "of", "the", and "to" can be removed from the indexes since they are very common and do not provide much information on the documents)
3. variant endings removal (with Lovin's stemming algorithm) ("enumerate", "enumeration", and "enumerated" all indexed as "enumerat")
4. synonyms replacement (index "gumbo" as "okra" in a cooking recipes database)
5. dierent spellings and misspellings replacement (index "colour" as "color", or "retreival" as "retrieval")
6. expression detection (index "computer science" as an expression-term, not only as the two word-terms "computer" and "science")
7. language and alphabet translation (consider "Qubec" and "Quebec" as only one term)

Preprocessing terms is a complicated subject in itself, and the algorithms discussed in this work apply to terms after the preprocessing stage.

Figure 1.1 provides an example of a ranked query and a corresponding result set. The query asks to nd the best recipes (the recipes with the highest ranks, i.e. the recipes with the highest similarity with the given set of query terms) having lentil, onion, tomato, garlic and cumin in it, and returns the recipes in order by their ranking. *RANK* is a ranking function, and RK is alias for the ranks column. The *Lentil and Vegetable Soup* recipe has a higher rank than the *Marinara Tomato Sauce* because the former would have all ve ingredients in it, while the later has only two (tomato and garlic) ingredients out of ve in it. The former is "closer" to the query than the later.

```
SELECT TOP 10 RANK(*) RK, RID, TITLE FROM RECIPES
WHERE {“lentil”, “onion”, “tomato”, “garlic”, “cumin”} IN INGREDIENTS
GROUP BY RID
ORDER BY RK DESC
```

RESULTS

RK	RID	TITLE
100	230	Lentil and Vegetable Soup
92	87	Dhal
80	22	Spicy Tomato Chutney
73	121	Lentil Salad
⋮	⋮	⋮
32	345	Marinara Tomato Sauce
⋮	⋮	⋮

Figure 1.1: Ranked Query Example.

The book *Managing Gigabytes* [WMB99a] oers good coverage of the term matching and other Information Retrieval problems.

1.1 IR Term-Matching Algorithm

Information Retrieval Term Matching (IRTM) algorithms make use of *inverted les* to nd top ranked documents. There are dierent variations and renements on the denition of inverted les, but let’s start with a simple one, given by Perry and Willet in [PW83].

Denition 1.1 (Simple Inverted File) *An inverted le to a document collection consists of a set of lists, called inverted lists, each of which contains pointers to the document records which have been indexed by one particular term in the set of terms used for the characterization of the documents in the collection. (From the introduction of [PW83]).*

In the above denition, *“the set of terms used for the characterization of the documents in the collection”* is the set of terms after the preprocessing steps have been applied to the document terms in the collection. For each such term, a list of pointers to document records (i.e. a list of row IDs or RIDs), is put in

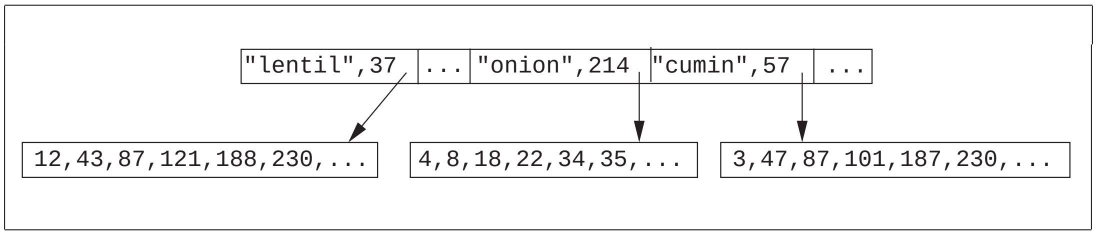

Figure 1.2: Example of Inverted Lists in a Hash Table.

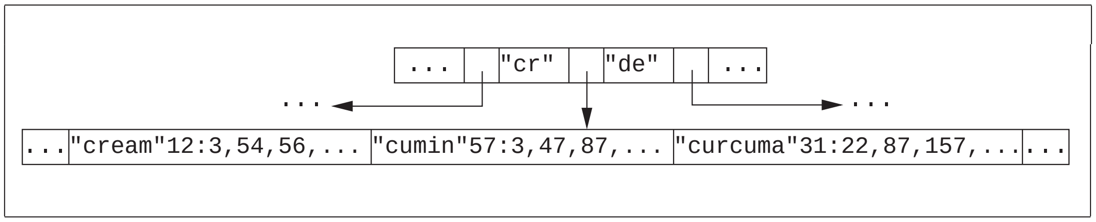

Figure 1.3: Example of Inverted Lists in a B-tree Index.

the inverted le, often called a vocabulary or a lexicon, to be retrieved easily. Hash tables with terms as keys, and disk pointers to inverted lists as values can be used as vocabularies. Figure 1.2 shows an example of that. In most IRTM implementations, the shortest inverted lists are processed rst, and the longest inverted lists last. B-tree indexes can also be used instead of hash table, also with terms as keys, and inverted lists as values. See Figure 1.3 for an example of a part of the bottom two levels of such a B-tree.

Usually, the number of terms in the list is stored just before the list, or just before the list pointer. It allows for a selection of terms by list length, i.e. by within-database term frequency. Terms with low frequencies are processed rst because they are more meaningful and discriminating. This selection process is similar to the one used when computing the conjunction of more than lists of documents. It is better to keep the list of candidates small. [KZS99] has a good discussion of this subject.

In my work, I use a *Bitmap Index*, which is a B-tree index with terms as keys, and bitmaps as values. See Chapter 2 for more on bitmap indexes. Both index types contain the same information, but algorithms built on such bitmap indexes can make use of more parallelism to compute ranked queries. These algorithms, along with the algorithms and data structures used for weighted term matching, are part of the new concept called *Bit-Sliced Index Arithmetic*, which is the subject of this work.

Algorithm 1.1 STM. Given a query Q consisting of a set of terms, nd the top k best matching documents. There is one accumulator A_d for every document d in the database. Every A_d value is initialized with 0.

```
for each term $t \in Q$ do
    nd  $I$ in the vocabulary, the inverted list for $t$
    for each document $d \in I$ do
        $A_d = A_d + 1$
nd  $k$ largest $A_d$ values (using a heap sort)
return top $k$ $(d, A_d)$ pairs
```

Figure 1.4: Simple Term Matching Algorithm

1.1.1 Simple Term Matching Algorithms

The rst kind of term matching, called *Simple Term Matching*, was described in [PW83] as "algorithm D". I will also refer to it as the *Simple Term Matching (STM)* algorithm. It is simple because the similarity measure used to compute the ranking of documents is simple: for every term in the query, if a document contains the term, then add 1 to the accumulator for that document. So, in other words, the ranking of a document is equal to the number of terms in common between the set of query terms and the set of document terms. Using the cooking recipes example again, the ranking of a recipe would be the number of ingredients from the query ingredients it contains. The "Lentil and Vegetable Soup" would be the most signicant recipe since it has all ve ingredients given in the query. Its ranking computed with the simple similarity measure would be 5 and it would be the maximal score, so some scaling is necessary to obtain rankings between 0 and 100. The STM algorithm is given in Figure 1.4.

1.1.2 Weighted Term Matching

The retrieval eectiveness, or the measurement of ranking performance, of the simple similarity measure described above is not as good as other more complex measures. The recall (the proportion of relevant documents that have been retrieved) and precision (the proportion of retrieved documents that are relevant)[MZ96] are not very high. The simple similarity measure provides a quick and easy way to get a ranking on documents, but with more complicated measures signicant improvement on both recall precision can be obtained.

Algorithm 1.2 WTM. Given a query Q consisting of a set of (term, weight) pairs, nd the top k best matching documents. There is one accumulator A_d for every document d in the database. Every A_d value is initialized with 0. sim is some similarity measure.

```
for each (term, weight) pair $(t, w_t) \in Q$ do
    nd $I$ in the vocabulary, the inverted list for $t$
    for each (document, weight) pair $(d, w_d) \in I$ do
        $A_d = A_d + sim((t, w_t), (d, w_d))$
nd $k$ largest $A_d$ values (using a heap sort)
return top $k$ $(d, A_d)$ pairs
```

Figure 1.5: Weighted Term Matching Algorithm

To increase the retrieval eectiveness, *Weighted Term Matching (WTM)* algorithms are used. Instead of treating every term in every document as equal as in simple TM, weights are assigned to every (term, document) pair. In the inverted lists, (document, weight) pairs appear instead of document numbers only. Then some similarity measure is applied, instead of simply adding 1 to an accumulator, to compute the ranking of every document, similarly to the STM algorithm described earlier. Figure 1.5 shows the WTM algorithm.

Alternative similarity measures were studied in a paper written by Zobel and Moat [ZM98]. Unfortunately, they were not able to pick a winner out of all the similarity measures. They conducted an exhaustive investigation of all standard similarity measures by considering dierent combining functions (inner product, cosine measure, Jacquard formulation, ...), dierent ways to assign term weights, document-term weights, relative term frequencies and document and query lengths. Although there were no clear winner, the *Cosine Similarity Measure* is believed to be a good overall measure, and many researchers right now are using it in their work (e.g. [KZS99, MZ96] to name only two). I will use the formulation and notation given in [KZS99], which I include in Figure 1.6, throughout this work. Note that the expression $W_{d,t}$ is not used in the denition, but it is going to be useful later on in Chapter 4.

$$Cosine(d,q) = \frac{\sum_{t \in q \wedge d} (w_{q,t} \quad w_{d,t})}{W_d}$$

where

q is a query (a set of terms)

d is a document (a set of terms)

$W_d = \sqrt{\sum_{t \in d} w_{d,t}^2}$

$w_{d,t} = log_e(f_{d,t} + 1)$

$w_{q,t} = log_e(f_{q,t} + 1) \quad log_e(\frac{N}{f_t} + 1)$

$f_{x,t}$ is the number of occurrences or *frequency* of term t in x

N is the number of documents

f_t is the number of distinct documents containing t

the expression $log_e(\frac{N}{f_t} + 1)$ is the “inverse document frequency”, a representation of the rareness of t in the collection.

$W_{d,t} = \frac{w_{d,t}}{W_d}$

Figure 1.6: Cosine Similarity Measure

1.2 Some Problems with IRTM Algorithms

A very important problem with the IR term matching algorithms given above in Figures 1.4 and 1.5 is scalability. When there is a large number of documents to match a query against, the accumulator array can get very large since it needs one accumulator per document. Four bytes per accumulator are used [WMB99b], so when the number of documents gets large, say one million documents, then 4 Mbytes of RAM are necessary to hold those accumulators. When running in a multi-user environment, if many queries are done at the same time, the RAM could ll up quickly since dierent queries cannot share their accumulators.

The need to limit the number of accumulators is obvious. [KZS99] has a good solution to that problem. Instead of using a static data structure (the accumulator array), as in Algorithm 1.2, they create accumulators dynamically (in a hash table). In the rst phase, they process inverted lists starting by the shortest lists, i.e. by rarest (probably most signicant) terms rst. In this phase, new accumulators are created freely. When they get to more common terms, they stop adding new accumulators, but they keep updating the already existing accumulators. The reasoning behind this process is that if a document doesn't contain any of the rarest, most signicant terms, it cannot itself be a signicant document since the common terms' weights cannot add up high enough to make a document without rare terms signicant. They continue to update the existing accumulators because the more common terms can help discriminate which documents containing rare terms are more signicant. This is a heuristic, but in their experimental results, they found that limiting the number of accumulators to about 5% of the total number of documents had no impact on retrieval eectiveness. In [MZ96], this strategy is called the *continue* strategy. [MZ96] also has a *quit* strategy, which stops processing inverted lists immediately when the maximum number of accumulators has been reached, without continuing to update the existing accumulators. It is shown that the retrieval eectiveness of the continue strategy is better than the quit strategy.

One problem with using hash tables is that the document ID has to be stored with the accumulator, while in an array, the document ID is the index into the array. Each accumulator entry takes 8 bytes (4 bytes for the accumulator proper and 4 bytes for the document ID) instead of just 4 bytes. 5% of 1 million documents gives 50,000 accumulators, and supposing the hash table ll factor is about 63%(= $1 \quad e^{\ 1}$, to keep the collision chain small [OO00b]), almost 80,000 hash table entries are necessary at 8 bytes each, therefore, about 618 Kbytes per query are needed for the hash table. As we will see later in Section 4.3, using a bit-sliced index (BSI) to store the accumulators and BSI arithmetic to compute the accumulator values will not require much more space than the IRTM with hashed accumulators.

Another important problem is the inverted le processing costs, as written in the abstract of [MZ96]:

> Query processing costs in large text databases are dominated be the need to retrieve and scan the inverted list of each query term. Retrieval time for inverted lists can be greatly reduced by the use of compression, but this adds to the CPU time required.

An inverted list is processed document by document, so when it gets long, especially when a term is common and the number of documents is large, retrieval and processing costs can get large. Many I/Os per inverted list may be needed, and processing documents one-by-one can be costly in terms of CPU time. In [MZ96], *Self-Indexing Inverted Files* are dened. They compress inverted lists to reduce disk space usage. The problem compressed inverted lists is the need to decompress the whole list to access a piece of data near the end of the list since decompression can start only at the beginning of the list. By building self-indexing inverted les, data can be accessed anywhere in a list without decompressing the whole list. The details of this process are beyond the scope of this work; refer to the paper for more.

One problem with those self-indexing inverted les is that it is not clear how inserts and updates can be done on such les, but this problem is probably not common since indexes in information retrieval are usually assumed to be read-only, like in a Data Warehouse. Another problem is the need to process each inverted list document by document. Parallelism techniques are not as easily applied here as in the BSI arithmetic algorithms, as we will see later. BSI arithmetic algorithms uses the SIMD (Single-Instruction, Many-Data) trick to gain parallelism. We could decide to have one dierent thread running for each inverted list involved in the query, but this is dicult to do because threads processing rare terms could create new accumulators, while the others could not, and it is not known in advance exactly which terms are part of the rare terms and which are not. Also, many threads accessing the accumulators at the same time would create many race conditions, so a lot of locking and unlocking would need to be done. I do not think the use of parallelism could be useful here. As we will see later, two dierent forms of parallelism can be used with BSI arithmetic algorithms.

1.3 Multisets

Another problem solved by BSI arithmetic is how to compute multisets of rows in a database. This problem is strongly related to the simple term matching problem stated above. Their implementation, except for the query processing step, is the

Q 1.1
```
SELECT COUNT(*) CT, CAR_ID FROM
  (SELECT CAR_ID FROM CARS WHERE COLOR = 'RED' UNION ALL
   SELECT CAR_ID FROM CARS WHERE MAKE = 'FERRARI' UNION ALL
   SELECT CAR_ID FROM CARS WHERE YEAR >= '1996' UNION ALL
   SELECT CAR_ID FROM CARS WHERE LOCATION = 'CANADA')
AS RANKED_CARS
GROUP BY CAR_ID;
```

Q 1.2
```
SELECT TOP 10 COUNT(*) CT, CAR_ID FROM
  (SELECT CAR_ID FROM CARS WHERE COLOR = 'RED' UNION ALL
   SELECT CAR_ID FROM CARS WHERE MAKE = 'FERRARI' UNION ALL
   SELECT CAR_ID FROM CARS WHERE YEAR >= '1996' UNION ALL
   SELECT CAR_ID FROM CARS WHERE LOCATION = 'CANADA')
AS TOP10_CARS
GROUP BY CAR_ID
ORDER BY CT DESC;
```

Figure 1.7: Multisets Example Queries

same. It is not possible in standard SQL-99 to specify queries dened with a `TOP` modier, so I will use DB2 [IBM] notation to provide examples in this section.

Consider Query 1.1 in Figure 1.7. It asks to retrieve some `CARS` data based on four subqueries. `CAR_ID` is a unique ID. The four sets will be `UNION`ed `ALL`, i.e. the four sets will be unioned, keeping duplicates in the multisets. For example, if there is a 1998 Ferrari red car located outside Canada in the `CARS` table, then its corresponding `CAR_ID` will appear 3 times in the multiset, while a 1997 blue Chevy car located outside Canada will have its corresponding `CAR_ID` appear just once in the multiset. The query nishes by doing a `GROUP BY CAR_ID`, and by computing the count of each group. What we end up with is a table in the style of Figure 1.8. One column has `CAR_ID` values, and the other counts, i.e. the `CAR_ID` multiplicities in the multiset created by the `UNION ALL` of the subqueries.

If Query 1.1 is modied to obtain Query 1.2, the result set is reordered to get the highest multiplicities rst, and then the top ten rows are selected. The idea is to get the ten most signicant rows, ten rows that best match the query. A row with a high multiplicity is assumed to be more relevant since it is "closer" to the query. A "1998 Ferrari red car located outside Canada" is closer to the query than a "1997 blue Chevy car located outside Canada", thus more interesting. This is just the Simple Term Matching approach, a natural result in SQL with `UNION ALL` queries.

RANKED_CARS

CT	CAR_ID
1	5
3	24
2	49
1	73
1	98
4	111
1	132
2	155
1	161
⋮	⋮

TOP10_CARS

CT	CAR_ID
4	111
4	666
3	24
3	179
3	221
3	387
3	453
3	687
3	721
2	49

Figure 1.8: Results of Queries 1.1 and 1.2

Examples of queries where multiplicities are subtracted, using `EXCEPT ALL`, and the minimum multiplicity of two multisets is determined, using `INTERSECT ALL`, can also be constructed. Note that in the case of `EXCEPT ALL` and `INTERSECT ALL`, any negative numbers in the result multiset must be replaced with zeroes, since rows do not appear with negative multiplicities in SQL. Refer to [OO00c, IBM] for more on these predicates.

1.4 What Bit-Sliced Index Arithmetic Brings

Bit-sliced index arithmetic brings a radically dierent way of building multisets of rows and implementing term matching algorithms, using native data structures and indexes of up-to-date *Object-Relational Database Systems (ORDBMS)*. It has been shown in [OQ97], *Improved query performance with variant indexes*, that bitmap and bit-sliced indexes are useful in many dierent ways other than term matching. Boolean conditions evaluation, range searches, and aggregate functions computation all benet from the use of bitmap and bit-sliced indexes.

Ranked queries can use BSI arithmetic to get better scalability. On the tests performed comparing the *Bit-Sliced Index Term Matching (BSTM)* and *Information Retrieval Term Matching (IRTM)* approaches (see Chapter 5), when varying the number of documents to search on and xing the number of terms per query, IRTM is constantly between about 2 to 2.5 times slower than BSTM for 5, 10 and 20 terms per query, and between 1.3 and 1.7 times slower than BSTM for 30, 40 and 50 terms. Users of search engines very rarely ask for more than 10 terms in one search[†]. Also, parallelization of the algorithms is easier with BSI arithmetic than with IRTM. The dierence between the two approaches will increase when concurrent programming on multi-CPU computers or on clusters of computers will be used to implement the algorithms.

Chapter 2 reviews some fundamental concepts for those not very familiar with bitmaps and bit-sliced indexes and discusses their implementation. Chapter 3 is an introduction to BSI arithmetic, covering basic arithmetic operations on BSIs. New Term Matching algorithms are covered in Chapter 4, with experimental results comparing IRTM with the new algorithms in Chapter 5. Chapter 6 discusses how BSI arithmetic can be extended to other applications. It includes some ideas about how a search engine could allow users to make Term Matching queries. Multi-column queries, arithmetic queries, and preference queries using BSI arithmetic are discussed. Nearest-Neighbor searches could also benet from BSI arithmetic.

[†][MZ96] *"Query of perhaps 3-10 terms are the norm for general-purpose retrieval systems".*

CHAPTER 2

Fundamental Concepts

In this chapter, bitmap and bit-sliced indexes are presented. Bitmap indexes were rst introduced in [ON87], and bit-sliced indexes in [OQ97]. Variations on these denitions were studied in [CI98, CI99, WB98, Wu99].

2.1 Bitmap Indexes

Denition 2.1 (Bitmap Index) *[ON87, OQ97] To create a bitmap index, all N rows of the underlying table T must be assigned ordinal numbers:* $1, 2, \ldots, N$, *called* Ordinal row-positions*, or simply* Ordinal positions*. Then for any index value x_i of an index X on T, a list of rows in T that have the value x_i can be represented by an Ordinal-position-list such as:* $4, 7, 11, 15, 17, \ldots$*, or equivalently by a verbatim bitmap,* `00010010001000101`.... *Note that sparse verbatim bitmaps (having a small number of 1's relative to 0's) will be compressed, to save disk and memory space.*

Ordinal row-positions $1, \ldots, N$ can be assigned to table pages in xed size blocks of size J, 1 through J on the rst page, $J + 1$ through $2J$ on the second page, etc., where J is the maximum number of rows of T that will t on a page (i.e., the maximum occurs for the shortest rows). This makes it possible to determine the zero-based page number pn for a row with Ordinal position n by the formula $pn = (N \quad 1)/J$. A known page number can then be accessed very quickly when long extents of the table are mapped to contiguous physical disk addresses. Since variable-length rows might lead to fewer rows on a page, some pages might have no rows for the larger Ordinal numbers assigned; for this reason, an *Existence Bitmap (EBM)* is maintained for the table, containing 1 bits in Ordinal positions where rows exist, and 0 bits otherwise. The EBM can also be useful if rows are deleted, making it possible to defer index updates.

It is a common misunderstanding that every row-list in a bitmap index must be carried in verbatim bitmap form. In reality, some form of compression is always used for sparse bitmaps (although verbatim bitmaps are preferred down to a relatively sparse ratio of 1's to 0's such as 1/50, because many operations on verba-

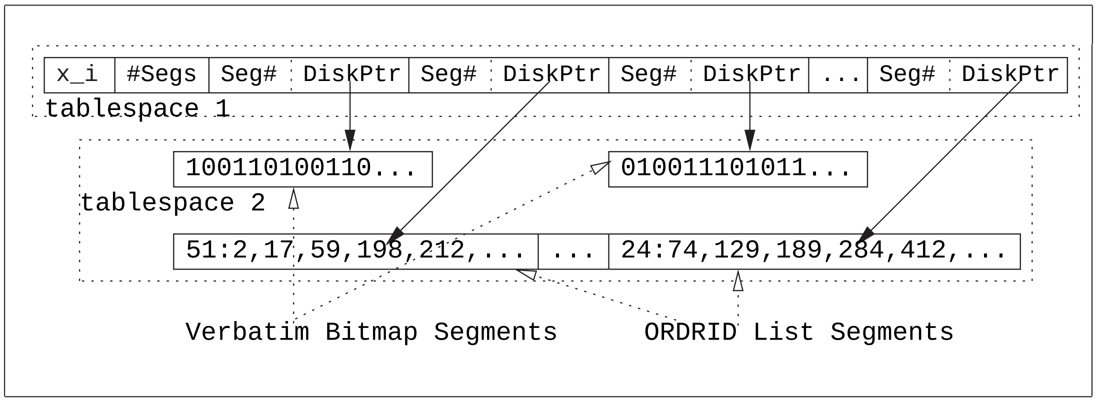

Figure 2.1: Bitmap Index Leaf Entry

tim bitmaps are more CPU ecient than on compressed forms). In the prototype system we implemented, called the RIDBIT project, bitmap compression simply involves converting sparse bitmap pages into ordered lists of *segment-relative ordinal positions* called ORDRIDs (dened below). We rst describe segmentation, which was introduced in [ON87, OQ97] and is used in the RIDBIT project.

2.1.1 Segments and Segment Relative Addressing

We break the rows of table T into equal-size blocks so that the bitmap fragment for the set of rows in each block will t on a single disk page. These blocks of rows are called *segments*, following the MODEL 204 nomenclature of [ON87]. The RIDBIT project uses 4KByte disk pages, so segments contain $S = 8 \quad 4000 = 32,000$ rows. (We use $S = 32,000$ as a rough estimate; the true number is larger, but not quite $2^{15} = 32,768$, because we leave space on the bitmap page for a count of 1-bits to tell us when compression is needed, and for the disk page overhead.)

A B-tree index entry for an index value x_i has the format shown in Figure 2.1. The entry in Figure 2.1 can grow to the length available on the B-tree leaf page where it resides, and another entry with the same index value x_i can follow on a successive leaf page if more segments make it necessary. Each Seginfo block (the (Seg#, DiskPtr) pairs) represents a segment in the bitmap anchor, Seg# being the segment number of the segment it represents, and DiskPtr the disk pointer to the ORDRID-list or bitmap. See the next section for a description of an ORDRID-list. The Seginfo blocks for an index entry are held in order by Seg#, and if a segment contains no row for x_i, then the Seginfo block for that segment will be missing in Figure 2.1. (This fact can be used at an early execution point to exclude segments from consideration that have no Seginfo block in one of the index entries.)

2.1.2 ORDRIDs and ORDRID-lists

Since the S bits of a segment bitmap must t on a 4 KByte page, $S < 2^{15}$, and a segment-relative ORDRID will t in two bytes (in what follows we will refer to a segment-Relative ORDRID simply as an ORDRID). This short length provides a signicant advantage in disk space and I/O speed during a range search. An ORDRID value k in segment m can be translated into a Table-Relative Ordinal position t by the formula $t = m \quad S+k$. An ORDRID-list for a segment of an index entry (pointed to by DiskPtr in Figure 2.1) contains ORDRIDs in ordered sequence. ORDRID lists are also stored in order on disk, and usually many ORDRID-lists will t on a page. If the dividing line between sparse bitmap and ORDRID-list occurs at a bit density 1/50, then the longest ORDRID-list will take up at most 16/50 of a disk page, and contiguous lists can be stored in a disk-resident B-tree with at least three entries per leaf page. ORDRID-lists use a separate continuum of pages (not intermixed with Index B-tree pages or Bitmaps) for fastest disk access, and are ordered by index-value and segment number, that is: x_i || Seg#. The DiskPtr used to address ORDRID-lists has the same format used in row addressing, consisting of (Disk Page#, Slot#), where Slot# addresses an oset directory entry that locates the ORDRID-list on the page.

To avoid having segments that keep getting converted back and forth between the verbatim bitmap and ORDRID-list forms, possibly because many updates are performed and the density keeps oscillating around 1/50, two ratios can be used: if the density drops below 1/64, convert the verbatim bitmap to an ORDRID-list, and if the density increases above 1/50, convert the ORDRID-list to a verbatim bitmap. This technique provides an hybrid density zone where the segment could be in any form.

Note that when we refer to a Bitmap index, this is a generic name meaning that Bitmaps are a possible form of representation, and does not mean that every row representation for every index value x_i is a Bitmap: it may be a Bitmap or an ORDRID-list, or a segment-by-segment combination of the two forms, whichever is most appropriate based on the density of rows for that value in the given segment. Similarly, when we speak of a Bitmap in a Bitmap index, an ORDRID-list might be the actual representation; we will dierentiate between bitmap and ORDRID-list when the dierence is important to our discussion. As a side note, the name *RIDBIT* comes from the compression of *ORDRID-Bitmap*.

2.1.3 Operations on Bitmaps

Pseudo-code for logical operations AND, OR, NOT, and COUNT on bitmaps were provided in [OQ97], so we limit ourselves here to short descriptions. Given two

`i`	`i`$_2$	`counts[i]`
0	`00000000`	0
1	`00000001`	1
2	`00000010`	1
3	`00000011`	2
4	`00000100`	1
5	`00000101`	2
6	`00000110`	2
7	`00000111`	3
8	`00001000`	1
9	`00001001`	2

Figure 2.2: First 10 Entries of the `counts` Array

verbatim bitmaps B_1 and B_2, we can create the bitmap $B_3 = B_1\ AND\ \ B_2$ by treating memory-resident segment fragments of these bitmaps as arrays of `long ints` in C, and looping through the fragments, setting $B_3^C[i] = B_1^C[i]\ \&\ B_2^C[i]$, where $B_n^C[i]$ refer to the ith `long int` in the C array representation of B_n. The logic can stream through successive segment fragments from disk (for B_1 and B_2) and to disk (B_3), until the operation is complete. The bitmap $B_3 = B_1\ OR\ \ B_2$ is computed in the same way, and $B_3 = NOT\ B_1$ is computed by setting $B_3^C[i] = \neg B_1^C[i]\ \&\ EBM^C[i]$ in the loop. Note that the eciency of bitmap operations arises from a type of parallelism in Boolean operations in CPU registers, specically SIMD (Single-Instruction- Multiple-Data), where many bits (32, or 64) are dealt with in a single AND, OR, or NOT operation occurring in the simplest possible loop. To nd the number of rows represented in a bitmap B_1, $COUNT(B_1)$, another SIMD trick is used: the bitmap fragment to be counted is overlaid with a `short int` array, and then the loop through the fragment uses the `short ints` as indexes into an auxiliary array, called `counts`, containing $2^8 = 256$ 1-byte integers, where `counts[i]` is equal to the number of bits on in the binary representation of `i`, aggregating these into a `count` variable. Figure 2.2 shows the rst few entries in the `counts` array. This technique saves a good amount of CPU resources, at the cost of using a little more memory space. The `counts` array is only 256 bytes long, worthwhile compared to the approach of counting the bits one-by-one. It is worth noting that 256 bytes will t in cache too on modern processors, so many RAM look ups can be avoided.

We perform logical operations AND and OR on two segment ORDRID-lists B_1 and B_2 by looping through the two lists in order to perform a merge-intersect or merge-union into an ORDRID-list B_3; in the case of OR, the resulting ORDRID-

ID	C	C_2	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
1	5	00000101	0	0	0	0	0	1	0	1
2	0	00000000	0	0	0	0	0	0	0	0
3	127	01111111	0	1	1	1	1	1	1	1
4	23	00010111	0	0	0	1	0	1	1	1
5	200	11001000	1	1	0	0	1	0	0	0
6	9	00001001	0	0	0	0	1	0	0	1
7	64	01000000	0	1	0	0	0	0	0	0
8	39	00100111	0	0	1	0	0	1	1	1

Figure 2.3: BSI Example

list might grow large enough to require conversion to a verbatim bitmap, an easy case to recognize, and easily done by initializing a zero Bitmap for this segment and turning on bits found in the union. The NOT operation on a segment ORDRID-list B_1 is performed by copying the EBM segment and turning o bits in the list corresponding to ORDRIDs found in B_1. To perform AND and OR with a verbatim bitmap B_1 in one index segment and an ORDRID-list B_2 in another, the ORDRID-list is assumed to have fewer elements and eciently drives the loop to access individual bits in the bitmap and perform the Boolean test, in the case of AND, lling in a new ORDRID-list B_3, and in the case of OR, initializing the verbatim bitmap B_3 to B_1 and turning on bits from the ORDRID-list B_2.

2.2 Bit-Sliced Indexes

Denition 2.2 (Bit-Sliced Indexes) *A* Bit-Sliced Index (BSI) *B is an ordered list of bitmaps, B^s, $B^{s\ \ 1}$, ..., B^1, B^0, and is used to represent the values (normally non-negative integers) of some column C of a table T (although the column C might be calculated values associated with rows of T, and have no physical existence in the table). The bitmaps B^i, $0 \quad i \quad s$ are called* bit-slices*, and their bit-values are dened this way:*

$$C[j] = \sum_{i=0}^{s} B^i[j] \quad 2^i$$

where $C[j]$ is the C value for the row with ordinal position j in T. In other words, $B^i[j] = 1$ if and only if bit i in the binary representation of $C[j]$ is on.

See Figure 2.3 for an example of a BSI. In the gure, the column C_2 is the binary representation of column C. Each bit-slice of a BSI is like a vertical partition of a column. Range searches can be executed very eciently using bit-sliced indexes. Some aggregate functions, like the sum, average, median, and n-tile can also be executed eciently on BSIs. Refer to [OQ97] for a complete analysis of aggregate functions and range searches. These algorithms make use of the bitmap Boolean operations, described above, to operate on the BSI slices.

A BSI can also be dened on a non-integer column, but with a few restrictions. A BSI could be dened on an AMOUNT column, where AMOUNT would be a dollar amount. The values stored in the BSI would actually be the number of cents, not the number of dollars. When the number of digits after the decimal point needs to be large in the column the BSI is dened on, it may be impractical to use bit-sliced indexes since the number of bit-slices will be large. But in IR, a 6-bit approximation of documents weights is known to be sucient to get almost unchanged retrieval eectiveness [MZ96], so BSIs used in this work for term matching will have a small number of slices.

It is possible to store negative numbers in a BSI, but then the BSI has to be dened as a 2's complement BSI. A BSI can be an `unsigned BSI (UBSI)`, containing `unsigned int`s, or a `signed BSI (SBSI)`, containing `signed int`s. When not specied, BSI (without the U or S) will mean unsigned BSI, since signed BSIs are rare compared to unsigned BSIs. It is important to remember that a BSI bit-slice is like a vertical partition of a column in its binary representation.

The main memory representation of a BSI is quite simple since a BSI is just an ordered list of bitmaps. A BSI is implemented using a vector of bitmap pointers. On disk, it is a little trickier. In the B-tree leaves, a BSI with only one slice is represented the same way a bitmap is represented, with only a small dierence: a count (equal to 1 in this case) of the number of slices is kept before the bitmap anchor data (see Figure 2.4). The problem when dealing with more slices is the BSI entry size. Having a bitmap anchor for every slice in a BSI may end up taking a large part of a leaf node, thus reducing the number of entries per leaf node signicantly and probably augmenting the B-tree depth. If a BSI has more than one slice, then the leaf entry, called a BSI anchor, consists of the usual key, the number of slices, and a list of slice pointers, i.e. a pointer to a disk block where to nd the bitmap anchor for that slice (see Figure 2.5). Figure 2.6 shows a partial view of a B-tree index with BSI anchors as entries. Note that dierent index levels are placed in dierent tablespaces on disk. This allows better data separation, and these tablespaces could actually be located on dierent disks. Better performance could be achieved by keeping more than one disk busy during index accesses.

New BSI operations are introduced chapters 3, *Bit-Sliced Index Arithmetic* and 4, *Term Matching*. It is important to remember that the algorithms introduced in the following chapters are using the Boolean operations presented in this chapter,

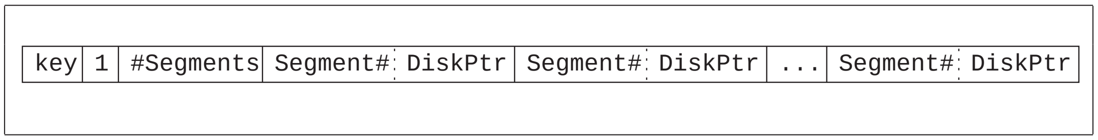

Figure 2.4: 1-slice BSI B-tree Leaf Entry

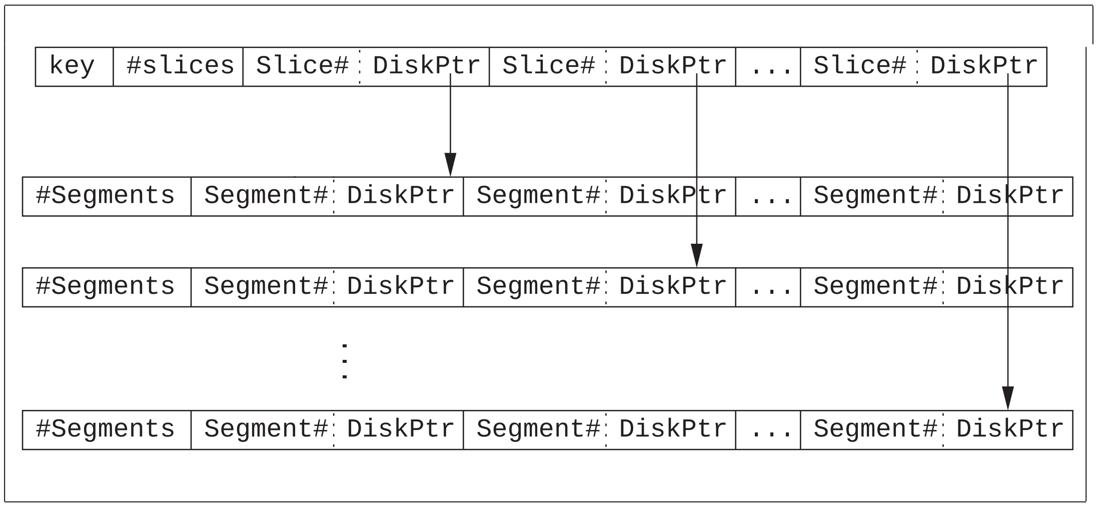

Figure 2.5: BSI B-tree Leaf Entry

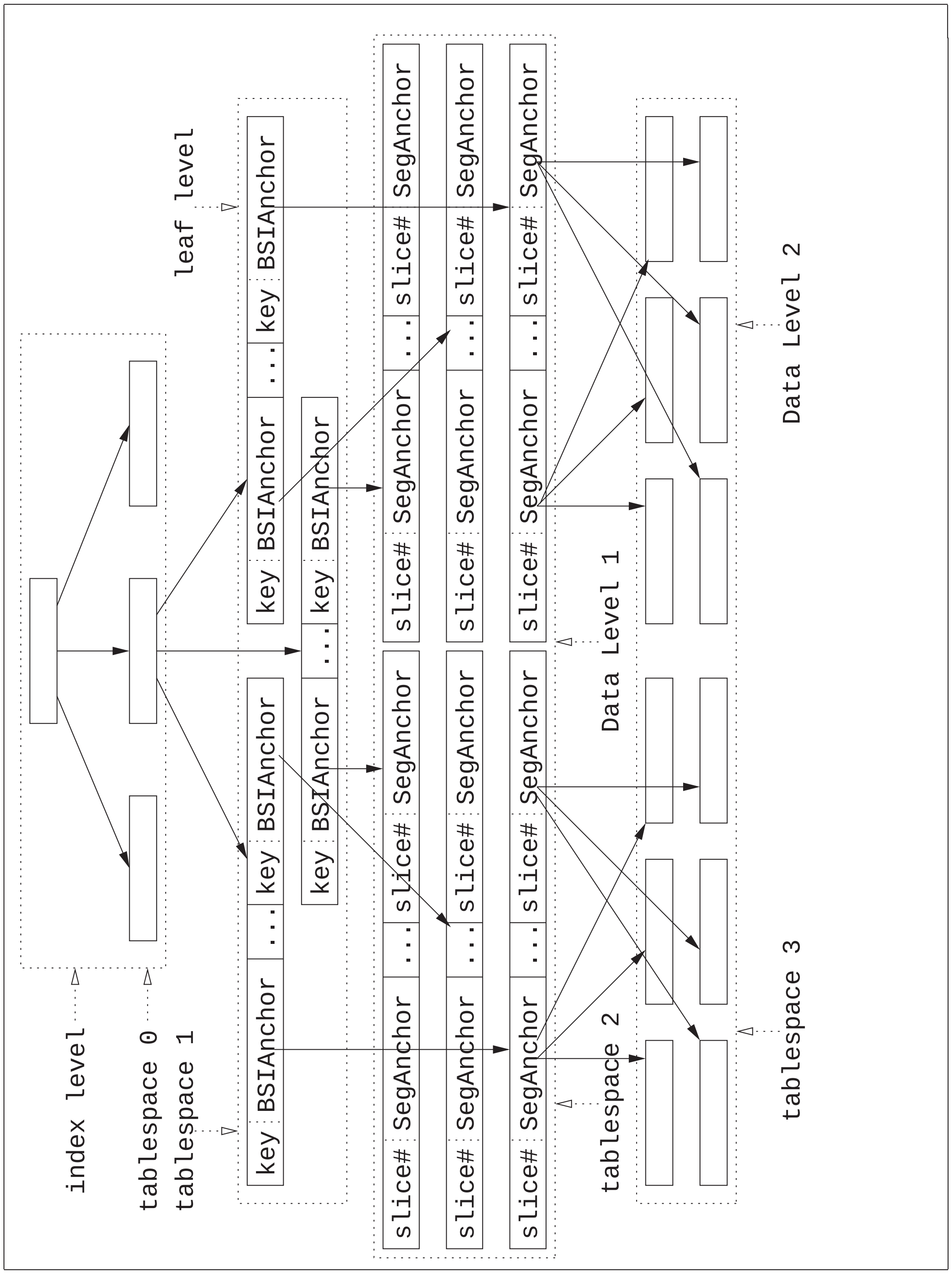

Figure 2.6: Partial BSI B-tree Representation

and that the parallelism obtained from the SIMD operations play an important role in the eciency of those algorithms.

CHAPTER 3

Bit-Sliced Index Arithmetic

This chapter introduces how to perform arithmetic on bit-sliced indexes, using SIMD operations on each of the bit-slices. Since a bitmap can be thought of as a bit-sliced index with only one bit-slice, these operations work correctly on bitmaps too. BSI arithmetic can be used to answer queries involving multiset operations like `UNION ALL`, `EXCEPT ALL` and `INTERSECT ALL`, and, along with the algorithms dened in Chapter 4, term-matching queries.

3.1 Addition

Denition 3.1 (BSI Addition) *Let S_1, S_2, ..., S_M be a series of BSIs to be added together. Then for every r, $1 \quad r \quad N$,*

$$SUM[r] = \sum_{i=1}^{M} S_i[r].$$

Consider Figure 3.1, where B_1, B_2 and B_3 are three bitmaps, representing some subquery found sets. The aim is to build a multiset BSI from these three bitmaps. This is achieved by *adding* the three bitmaps together, represented by SUM. $B_1[1] = B_2[1] = B_3[1] = 0$, row 1 does not appear in any of the bitmaps, and $SUM[1]$ must be 0. $B_1[2] = B_3[2] = 0$ and $B_2[2] = 1$, row 2 appears in one bitmap, and $SUM[2]$ must be 1. $B_1[4] = 0$ and $B_2[4] = B_3[4] = 1$, row 4 appears in two bitmaps, and $SUM[4]$ must be 2. Similar argument for row 5, but $SUM[5]$ must be 3. The expression *adding bitmaps together* is justied since $SUM[r] = B_1[r] + B_2[r] + B_3[r]$. The problem is that SUM cannot be represented as a bitmap; it can, however, be represented as a bit-sliced index! The last two columns of the gure show SUM represented as a BSI with two bit-slices, SUM^0 and SUM^1.

Now how do we obtain this SUM BSI? The algorithm used is practically the same as the standard binary addition algorithm, but applied to bitmaps. First, adding any two bitmaps together is easy. Figure 3.2 shows an example. The SUM

B_1	B_2	B_3	SUM	SUM^1	SUM^0
0	0	0	0	0	0
0	1	0	1	0	1
1	0	0	1	0	1
0	1	1	2	1	0
1	1	1	3	1	1
0	0	0	0	0	0
1	1	0	2	1	0
0	0	1	1	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 3.1: Addition of Bitmaps Example

B_1	B_2	SUM	SUM^1	SUM^0
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
0	1	1	0	1
1	1	2	1	0
0	0	0	0	0
1	1	2	1	0
0	0	1	0	1
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Figure 3.2: Addition of Two Bitmaps

Algorithm 3.1 BSI Addition. Given two BSIs, $A = A^P A^{P\ \ 1} ... A^1 A^0$ and $B = B^Q B^{Q\ \ 1} ... B^1 B^0$, we construct a new sum BSI, $S = A + B$, using the following pseudo-code. We must allow the highest-order slice of S to be $S^{MAX(P,Q)+1}$, so that a carry from the highest bit-slice in A or B will have a place.

```
$S^0 = A^0\ XOR\ B^0$           // bit on in $S^0$ i exactly one bit on in   $A^0$ or $B^0$
$C = A^0\ AND\ B^0$             // C is "Carry" bit-slice
                                // bit on i both bits on in   $A^0$ and $B^0$
for ($i = 1$; $i$     $MIN(P, Q)$; $i + +$) { // While there are further bit-slices
                                            // in both $A$ and $B$
    $S^i = A^i\ XOR\ B^i\ XOR\ C$           // one bit on (or three bits on)
                                            // gives bit on in $S^i$
    $C = (A^i\ AND\ B^i)\ OR\ (A^i\ AND\ C)\ OR\ (C\ AND\ B^i)$
                                            // 2 or 3 bits on gives bit on in $C$
}
if ($P > Q$)                                // if $A$ has more bit-slices than $B$
    for ($i = Q + 1$; $i$     $P$; $i + +$) {   // continue loop until last bit-slice
        $S^i = (A^i\ XOR\ C)$               // one bit on gives bit on in $S^i$
                                            // note that $C$ might be zero!
        $C = (A^i\ AND\ C)$                 // two bits on gives bit on in $C$
                                            // zero if prior $C$ was zero!
    }
else    // $Q$     $P$ and $B$ has at least as many bit-slices as $A$
    for ($i = P + 1$; $i$     $Q$; $i + +$) {   // continue loop until last bit-slice
        $S^i = (B^i\ XOR\ C)$               // one bit on gives bit on in $S^i$
                                            // note that C might be zero!
        $C = (B^i\ AND\ C)$                 // two bits on gives bit on in $C$
                                            // zero if prior $C$ was zero!
    }
if ($C$ is non-zero)          // if still non-zero Carry after $A$ and $B$ end
    $S^{MAX(P,Q)+1} = C$      // Put Carry into nal bit-slice of   $S$
```

Figure 3.3: Addition of BSIs

BSI needs two bit-slices since it needs to represent numbers 0, 1, and 2. To obtain SUM^0, compute B_1 XOR B_2, and to obtain SUM^1, compute B_1 AND B_2, using the SIMD Boolean operations dened in the previous chapter. To generalize from adding bitmaps to adding BSIs, we rst observe that a bitmap can be seen as a BSI with only one bit-slice. Second, we need to interpret the idea of "carrying" in the standard binary addition to the SIMD situation of Boolean bitmap operations. Figure 3.3 contains the *addition of BSIs* algorithm.

A *Carry bit-slice* C can arise in the algorithm of Figure 3.3 whenever two or three bit-slices are added to form S_i, and a non-zero (or non-empty) C must then be added into the next bit-slice S_{i+1}. Note that if C is zero (has no bits on), Boolean operations give the expected results, but a ag to show empty C can short-circuit the operation. Once the bit-slices in either A or B run out, calculations of C are likely to result in zero soon after, and C will never become non-zero again.

3.2 Subtraction

To be able to compute the subtraction of two BSIs, we need to be able to represent negative numbers in a BSI. The solution is to use two's complement numbers. A BSI containing numbers in two's complement form will be called a *signed BSI (SBSI)*, and conversely, an *unsigned BSI (UBSI)* contains only positive numbers and cannot contain negative numbers. The simplest way to compute the subtraction $D = A \quad B$, where A and B are two BSIs (signed or not), is to think of it this way: $D = A + (\quad B)$. The rst step is to negate B, then add B to A using the bit-sliced index addition presented above.

Suppose we need to do the subtraction $D = A \quad B$, where A and B are UBSIs, $A[1] = 5$ and $B[1] = 7$, so for row 1, we need to do $5 \quad 7 = 5 + (\quad 7)$. Suppose A and B have three bit-slices each. $5 = \texttt{101}_2$ and $7 = \texttt{111}_2$. The rst step is to add a sign bit to 5 and 7 to obtain $5 = \texttt{0101}$ and $7 = \texttt{0111}_2$. The high-order bit in a two's complement number is always the sign bit; 0 for a positive number, and 1 for a negative number. Next, negate 7's bits, obtaining `1000`, and add 1 to obtain $7 = \texttt{1001}_2$. To perform the subtraction, we will need another bit-slice, so we sign-extend the two numbers to get $5 = \texttt{00101}$ and $7 = \texttt{11001}_2$. Sign-extending a two's complement number always duplicates the sign bit into a new high-order bit. The next step is to add $\texttt{00101}_2$ and $\texttt{11001}_2$ together, which gives the number $D[1] = \texttt{11110}_2 = \quad 2$. The only place where it is important to know if a bit-sliced index is signed or not in the above example, is when we have to negate a BSI. If the bit-sliced index is unsigned, then we have to add a sign bit-slice to the BSI (which will consist of a zero bitmap). If the BSI is signed, then the sign bit-slice exists already, so don't add a bit-slice to the BSI. Figure 3.4 shows a few more examples,

r	A	A_2	B	B_2	$(\ B)_2$	$(A \ \ B)_2$	$A \ \ B$
1	5	00101	7	00111	11001	11110	-2
2	5	00101	-7	11001	00111	01100	12
3	-5	11011	7	00111	11001	10100	-12
4	-5	11011	-7	11001	00111	00010	2
5	6	00110	3	00011	11101	00011	3
6	6	00110	-3	11101	00011	01001	9

Figure 3.4: BSI Subtraction Examples

Algorithm 3.2 BSI Negation. Given a BSI $B = B^Q B^{Q \ 1} ... B^1 B^0$, we construct a new BSI D, such that $D = \ \ B$, using the following pseudo-code. We must allow the highest-order slice of D to be D^{Q+1} if B is unsigned, and D^Q if B is signed.

```
$max = Q$
if ($B$ is unsigned) {              // add the necessary sign bit-slice
    add a new high-order sign bit-slice (= zero bit-slice)
    $max = max + 1$                 // $max$ = the sign bit-slice number
}
for ($i = 0$; $i \ \ max$; $i$++)   // for every slice of $B$
    $D^i = NOT(B^i)$                // $1^s$ complement of $B$
$D = D +'' all \ \ 1 \ bitmap''$    // $2^s$ complement of $B$
```

Figure 3.5: BSI Negation Algorithm

where both A and B are signed BSIs. The bit-sliced index negation algorithm is shown in Figure 3.5 and the bit-sliced index subtraction algorithm in Figure 3.6.

3.3 Shifting and Multiplication by a Constant

Denition 3.2 (Bit-Sliced Index Left (Right) Shifting) *Let A be a BSI (signed or not), i an integer, and $D = A << i$ ($D = A >> i$), then for every $1 \ \ r \ \ N$, $D[r] = A[r] << i = A[r] \ \ 2^i$ ($D[r] = A[r] >> i = A[r] \ \ 2^{\ i}$).*

Algorithm 3.3 BSI Subtraction. Given two BSIs, $A = A^P A^{P\ \ 1} ... A^1 A^0$ and $B = B^Q B^{Q\ \ 1} ... B^1 B^0$, we construct a new BSI D, such that $D = A \quad B$, using the following pseudo-code. Let $max = MAX(P, Q)$. If both A and B are signed, or if the BSI with more bit-slices is signed, then D must have space for $max + 1$ bit-slices. If both A and B are unsigned or if the BSI with more bit-slices is unsigned, then D must have space for $max + 2$ bit-slices. If A and B have the same number of bit-slices and one of them is unsigned, then D must have space for $max + 2$ bit-slices.

```
if ($A$ is unsigned) {               // sign extend $A$ if necessary
    add a new high-order sign bit-slice (= zero bit-slice) to $A$
    $P = P + 1$
}
if ($B$ is unsigned) {               // sign extend $B$ if necessary
    add a new high-order sign bit-slice (= zero bit-slice) to $B$
    $Q = Q + 1$
}
$C = \quad B$                        // $C$ is the opposite of $B$
// make $A$ and $C$ have the same number of slices
if ($P < Q$) {                       // $A$ is shorter than $B$
    sign extend $A$ to have $Q + 1$ bit-slices
    sign extend $C$ to have $Q + 1$ bit-slices
}
else {                               // $A$ is at least as long as $B$
    sign extend $A$ to have $P + 1$ bit-slices
    sign extend $C$ to have $P + 1$ bit-slices
}
$D = A + C$                          // really do $D = A + (\quad B)$
```

Figure 3.6: BSI Subtraction

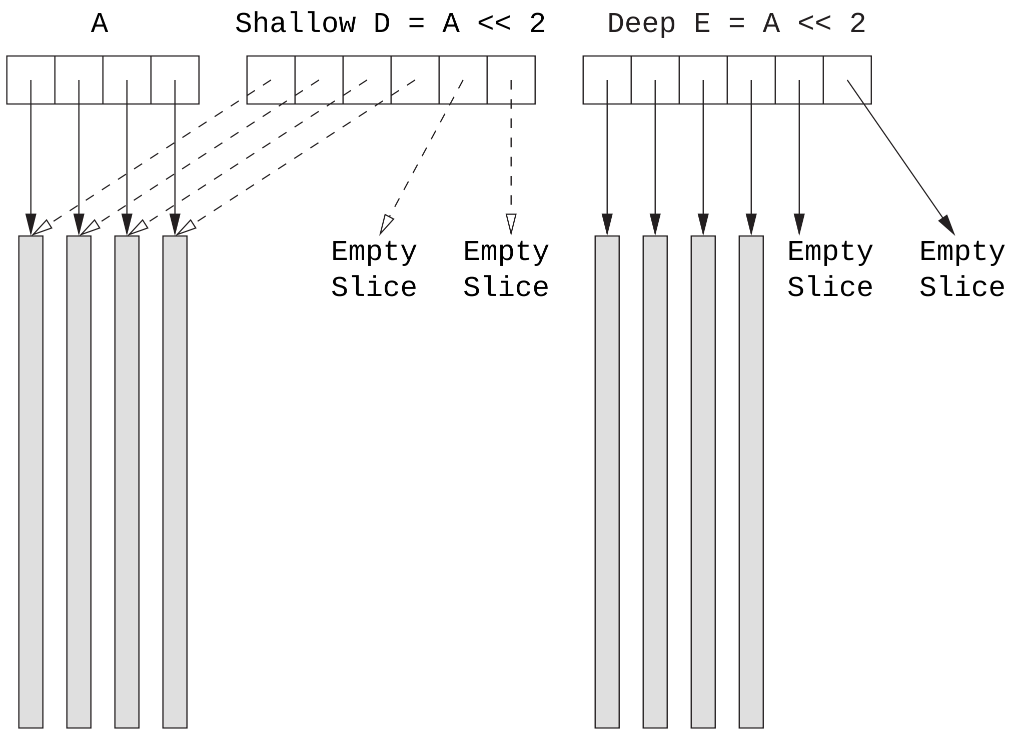

Figure 3.7: Shallow vs. Deep Shifting

In this section bit-sliced index shifting is considered. BSI Left Shifting is used, along with BSI Addition, to implement the multiplication of a BSI by a constant. This is simpler than the multiplication of Section 3.4 which is multiplication of a BSI by another BSI, because multiplication of a BSI A by a constant c multiplies every row in A by the same value c. One could materialize an all-c BSI C, i.e. a BSI where $C[r] = c$ for every row r and then do $A \quad C$, but this is too much work. Instead, for every bit c_i on in $c = c_n ... c_1 c_0$, $0 \quad i \quad n$, we left shift A by i positions, and add the results. Shifting can be fast because only the bit-slice pointers need to be shifted to the left in the bit-slice pointer array. We use *shallow* shifting, instead of *deep* shifting, where bit-slices are copied into another pointer array (see Figure 3.7). Refer to Figure 3.8 and 3.9 for the BSI shifting and BSI multiplication by a constant algorithms.

When BSI multiplication by a constant is executed, a lot of time spent copying slices is saved with shallow shifting, and a lot of buer space is saved at the same time since the same bit-slices get reused possibly many times during the successive additions. The performance of the BSI multiplication by a constant depends on the number of bits on in the constant c. When c is a power of two, then the BSI

Algorithm 3.4 BSI Left Shifting. Given a BSI $A = A^P A^{P \quad 1} ... A^1 A^0$ and an integer $i \quad 0$, we construct a new BSI D such that $D = A <<_{type} i$ using the following pseudo-code. *type* is equal to either *shallow* or *deep*. D will end up with $P + i$ bit-slices. Shifting works if A and D are the same BSI, i.e. A and D point to the same BSI in memory.

```
if ($type == shallow$)
    for ($s = P$; $s \quad 0$; $s$–)
            $D^{s+i} = A^s$      // shallow copy slice $s$ of $A$ into slice $s + i$ of $D$
else
    for ($s = P$; $s \quad 0$; $s$–)
            $D^{s+i} = deepcopy(A^s)$   // deep copy slice $s$ of $A$ into slice $s + i$ of $D$
for ($s = 0$; $s < i$; $s$++)
    $D^s = \emptyset$                  // make sure the $i$ lowest-order slices of $D$ are empty
```

Algorithm 3.5 BSI Right Shifting. Given a BSI $A = A^P A^{P \quad 1} ... A^1 A^0$ and an integer $i \quad 0$, we construct a new BSI D such that $D = A >>_{type} i$. *type* is equal to either *shallow* or *deep*. D will end up with $P \quad i + 1$ bit-slices if $i \quad P$, and 0 bit-slices if $i \quad P + 1$.

```
if ($type == shallow$)
    for ($s = 0$; $s \quad P \quad i$; $s$++)
            $D^s = A^{s+i}$      // shallow copy slice $s + i$ of $A$ into slice $s$ of $D$
else
    for ($s = 0$; $s \quad P \quad i$; $s$++)
            $D^s = deepcopy(A^{s+i})$   // deep copy slice $s + i$ of $A$ into slice $s$ of $D$
for ($s = P$; $s > P \quad i$; $s$–)
      $D^s = \emptyset$                // make sure the $i$ highest-order slices of $D$ are empty
```

Figure 3.8: BSI Shifting

Algorithm 3.6 BSI Multiplication by a Constant. Given a BSI $A = A^PA^{P\ \ 1}...A^1A^0$ and a positive integer $c = c_nc_{n\ \ 1}...c_1c_0$, we construct a new BSI D, such that $D = c \quad A$, using the following pseudo-code. Let j be the smallest bit position for which $c_j = 1$. If such j does not exist, then $c = 0$ and return an empty D immediately.

```
$D = A <<_{deep}\ j$
for every bit $c_i$ on in $c$, $i \neq j$
    $D = D + (A <<_{shallow}\ i)$
```

Figure 3.9: BSI Multiplication by a Constant

multiplication by a constant performs very well since it only needs to make one deep left shift. When c has two bits on in it, then one deep left shift, one shallow left shift and one addition need to be done. If there are three bits on in C, then one deep shift, two shallow shifts and two additions are needed. As we will see later in Chapter 4, the BSTM algorithm makes great use of the multiplication by a constant algorithm, and usually the weights (the constants) used do not need more than 6-bit precision. Therefore, we can expect (on average) to have three bits on in the constants, and we will need on average only two BSI Additions. It is more economical than using the general BSI Multiplication algorithm. If query weights are limited to powers of 2, then the economy is more important.

3.4 Multiplication

Bit-sliced index multiplication is implemented using an adaptation of Booth's algorithm [DM92, Wan]. First consider the multiplication $d = a \quad b$, where a, b, d are integers with 2's complement binary representations $a = a_p \dots a_1a_0$, $b = b_q \dots b_1b_0$ and $d = d_{p+q+2} \dots d_1d_0$. If b is positive, then we could do the multiplication by successive additions, i.e. starting with d initialized to 0, for every bit $b_i = 1, 0 \quad i \quad q$, do $d = d + b_i \quad 2^i \quad a$. But if b is negative, we need something else.

The value of a number in a 2's complement form is:

$$Val(b) \quad = \quad b_q \quad 2^q + \sum_{i=0}^{q\ \ 1} b_i \quad 2^i \tag{3.1}$$

$$= b_0 \quad 2^0 + b_1 \quad 2^1 + \quad + b_{q \ 1} \quad 2^{q \ 1} \quad b_q \quad 2^q \tag{3.2}$$

Let $b_{\ 1} = 0$, we can rewrite the equation as:

$$\begin{aligned} Val(b) &= b_{\ 1} \quad 2^0 + (\quad b_0 \quad 2^0 + b_0 \quad 2^1) + (\quad b_1 \quad 2^1 + b_1 \quad 2^2) \\ &\quad + \quad + (\quad b_{q \ 1} \quad 2^{q \ 1} + b_{q \ 1} \quad 2^q) \quad b_q \quad 2^q \end{aligned} \tag{3.3}$$

since $\ b_i \quad 2^i + b_i \quad 2^{i+1} = b_i \quad 2^i(\ 1 + 2) = b_i \quad 2^i$ for all $0 \quad i \quad q \quad 1$. Factoring $2^0, 2^1, \ldots, 2^q$, we get

$$\begin{aligned} Val(b) &= (b_{\ 1} \quad b_0) \quad 2^0 + (b_0 \quad b_1) \quad 2^1 + (b_1 \quad b_2) \quad 2^2 \\ &\quad + \quad + (b_{q \ 1} \quad b_q) \quad 2^q \end{aligned} \tag{3.4}$$

$$= \sum_{i=0}^{q} (b_{i \ 1} \quad b_i) \quad 2^i \tag{3.5}$$

Multiplying a by $Val(b)$, we get

$$a \quad Val(b) = \sum_{i=0}^{q} (b_{i \ 1} \quad b_i) \quad a \quad 2^i \tag{3.6}$$

To perform the multiplication, we need to look at every bit $b_i, 0 \quad i \quad q$, and do successive additions or subtractions, depending on the value of b_i and $b_{i \ 1}$. If $b_i = b_{i \ 1}$, then $b_{i \ 1} \quad b_i = 0$ and nothing needs to be added to d for this bit b_i. If $b_i = 1$ and $b_{i \ 1} = 0$, then $b_{i \ 1} \quad b_i = \quad 1$ and we subtract $a \quad 2^i$ from d. If $b_i = 0$ and $b_{i \ 1} = 1$, then $b_{i \ 1} \quad b_i = 1$ and we add $a \quad 2^i$ to d. Figure 3.10 gives the algorithm. An example is given in Figure 3.11.

To go from Algorithm 3.7 to the BSI multiplication algorithm, we need to translate the conditions comparing the b_i and $b_{i \ 1}$ values into SIMD Boolean operations on bit-slices. We want to do the multiplication $D = A \quad B$, where A, B, and D are bit-sliced indexes. Remember that we need to perform multiplications in parallel using those SIMD operations, so at any given step, we may need to add the value $A[r_1] \quad 2^i$ to $D[r_1]$ if $B^i[r_1] = 0$ and $B^{i \ 1}[r_1] = 1$, to subtract the value $A[r_2] \quad 2^i$ from $D[r_2]$ if $B^i[r_2] = 1$ and $B^{i \ 1}[r_2] = 0$, and don't change $D[r_3]$ if $B^i[r_3] = B^{i \ 1}[r_3]$. Figure 3.12 shows the algorithm.

One big problem with the current implementation of BSI multiplication comes with the need to negate the BSI A as in Figure 3.12 and to negate many bit-slices. The BSIs used in the term matching algorithms covered later are sparse, i.e. their bit-slices are sparse. When negating a sparse bitmap, we get very dense bitmaps. If a bitmap segment is empty, it does not need to exist, but when negated, a full segment needs to be created and inserted in the bitmap anchor. For very sparse

Actually, a copy of the corresponding EBM segment can be used instead.

Algorithm 3.7 Fast Multiplication (Booth's Algorithm). Given two integers, $a = a^p a^{p\quad 1}...a^1 a^0$ and $b = b^q b^{q\quad 1}...b^1 b^0$, we compute a new integer d, such that $d = a \quad b$, using the following pseudo-code. Let $b_{\quad 1} = 0$.

```
$d = 0$
$c = \quad a$        // keep  $a$ in $c$ to avoid computing it many times
for ($i = 0$; $i \quad q$; $i$++) {                    // for every bit in $b$
    if ($b_i = 1$ and $b_{i\quad 1} = 0$)
        $d = d + c \quad 2^i$                          // subtract $a \quad 2^i$
    else
        if ($b_i = 0$ and $b_{i\quad 1} = 1$
            $d = d + a \quad 2^i$                      // add $a \quad 2^i$
}
```

Figure 3.10: Fast Multiplication (Booth's Algorithm)

Compute $d = 5 \quad\quad 7$ with Booth's Algorithm.
$a = 5$, $b = \quad 7 =$ `1001`$_2$, $b_{\quad 1} = 0$, $d = 0$

i	b_i	$b_{i\quad 1}$	action	d
0	1	0	subtract 5 $\quad 2^0$	0-5=-5
1	0	1	add 5 $\quad 2^1$	-5+10=5
2	0	0	nothing	5
3	1	0	subtract 5 $\quad 2^3$	5-40=-35

Figure 3.11: Fast Multiplication Example

Algorithm 3.8 BSI Multiplication. Given two BSIs, $A = A^P A^{P\ \ 1} ... A^1 A^0$ and $B = B^Q B^{Q\ \ 1} ... B^1 B^0$, we construct a new BSI D, such that $D = A \quad B$, using the following pseudo-code. D needs $P + Q + 3$ slices, all empty to start with. T is a temporary BSI.

```
$C = \quad A$      // keep  $A$ in $C$ to avoid computing it many times
for ($i = 0$; $i \quad Q$; $i$++) {          // for every bit-slice of $B$
    $T = T^P ... T^1 T^0 = 0$               // $T$ is a temporary BSI initialized to 0
    for ($j = 0$; $j \quad P$; $j$++)        // for every bit-slice of $A$
        // $T$ holds the values to be added to $D$ in the last step
        // if $B^1[r] = 1$ and $B^0[r] = 0$, then $T[r] = C[r] = \quad A[r]$
        // if $B^1[r] = 0$ and $B^0[r] = 1$, then $T[r] = A[r]$
        // if $B^0[r] = B^1[r]$, then $T[r] = 0$
        $T^j =$  $(B^1\ AND\ NOT(B^0)\ AND\ C^j)$
               $OR\ (NOT(B^1)\ AND\ B^0\ AND\ A^j)$
    $D = D + T$                             // update $D$
}
```

Figure 3.12: BSI Multiplication

segments, very dense segments are obtained. Therefore, more memory is needed and Boolean operations will be slowed down. If we had opted for a dual-sparse RID-list encoding (the 0 bits' position are encoded instead of the 1 bits' position) when the bitmap density is very high, it would be much less of a problem.

For example, take bitmap B_0 = `001000100000`; its RID-list encoding is $R_0 = (3, 7)$. Let B_1 be the negation of B_0, so B_1 = `110111011111`; the RID-list encoding cannot be applied to it since it is too dense, but its dual encoding can, and we get $R_1 = (3, 7)$, the same as R_0! The only thing we need to do to negate a RID-list is to change a ag telling if the list is encoded the usual way or by its dual. The RIDBIT project does not currently support varying encoding schemes, and adding it would involve modifying the bitmap anchor structure, and making important changes to the Boolean functions to take into account if a RID-list is sparse or dual-sparse, and call new functions operating on dual-sparse RID-lists. This involves a good amount of work, and since it is not critical to the term matching performance results given here, it is left as future work. This subject is strongly related to bitmap compression since RID-lists are a form of compression, and it would make sense to add dual-sparse RID-list encoding at the same time varying bitmap compression schemes are added. See [Joh99] for more on bitmap compression.

3.5 Minimum

Denition 3.3 (Bit-Sliced Index Minimum) *Let A and B be BSIs (signed or not), and $D = min(A, B)$, then for every $1 \quad r \quad N$, $D[r] = min(A[r], B[r])$.*

As seen previously, multisets built with UNION ALL predicates can be implemented with BSI addition, and multisets built with EXCEPT ALL predicates with BSI subtraction. Now, let's look at how INTERSECT ALL predicates can be implemented with BSI minimum. Figure 3.13 shows the algorithm.

Algorithm 3.9 works in this way: suppose A has more bit-slices than B, then for every bit-slice that A has in excess, starting with the most signicant bit-slice, if a row has its bit on in the bit-slice, then its A-value is larger than the corresponding B-value, so turn on the corresponding bit in K_B and K. For every other bit-slice (still going from high to low order bit-slice), nd out which rows have dierent bit-values in the current bit-slice of A and B. Of those rows not already in K, the rows with a 1 in the current bit-slice of A have an A-value larger than their B-value, so turn on their corresponding bit in K_B and K, while the rows with a 1 in the current bit-slice of B must have their corresponding bit in K_A and K turned

Algorithm 3.9 BSI Minimum. Given two BSIs, $A = A^S A^{S\ \ 1} ... A^1 A^0$ and $B = B^P B^{P\ \ 1} ... B^1 B^0$, we construct a new BSI M, such that $M = min(A, B)$. The following pseudo-code deals only with non-negative values. We assume in the loop below that $S \quad P$ (if not we reverse A and B). The highest-order slice of M will be $M^{min(S,P)}$, since the minimum of the two numbers $A[r]$ and $B[r]$ for some row r cannot have more binary digits than $min(S, P)$. K is the bitmap of rows for which we know the minimum value, K_A is the bitmap of rows for which A has lesser value, and K_B is the bitmap of rows for which B has lesser value.

```
$K = K_A = K_B = \emptyset$
for ($i = S$; $i > P$; $i$      )          // recall that $S \quad P$; loop is empty if $S = P$
    $K_B = K_B\ OR\ \ A^i$                  // min must be in $B$ since values not this large
$K = K_B$                                   // all rows for which min is determined so far
for ($i = P$; $i > 0$; $i$      ) {         // loop down to zero
    // rows diering for the 1st time in  $A^i$ and $B^i$
    $X = (A^i\ XOR\ \ B^i)\ AND\ \ NOT(K)$
    // if $A^i$ has 1-bit, new min must be in $B$
    $K_B = K_B\ OR\ \ (A^i\ AND\ \ X)$
    // else $B^i$ has 1-bit and new min must be in $A$
    $K_A = K_A\ OR\ \ (B^i\ AND\ \ X)$
    $K = K\ OR\ \ X$                        // new min rows found in this pass
}                                           // any rows not still in $K$ are equal in $A$ and $B$
$K_B = K_B\ OR\ \ (EBM\ AND\ \ NOT(K))$        // choose rows in $B$ as min
for ($i = 0$; $i <= P$; $i + +$) {          // loop to set BSI $M$ using known $K_A$ and $K_B$
    $M^i = A^i\ AND\ \ K_A$                 // $A^i$ values for rows with bits in $K_A$
    // $B^i$ values for rows with (disjoint) bits in $K_B$
    $M^i = M^i\ OR\ \ (B^i\ AND\ \ K_B)$
}
```

Figure 3.13: BSI Minimum

on. For the rows with equal A and B-values, turn on their corresponding in K_B[†]. If a row has its minimum value in A (if K_A is set for the row), then copy its A-bits in M. Otherwise, copy its B-bits in M.

To handle both positive and negative numbers, we would sign extend A or B with any needed high-order bit-slices, and start by dierentiating negative and positive values in the highest bit-slice. Then we would use Algorithm 3.9 to nd $min(A, B)$ for the bitmap set of non-negative values, and analogous pseudo-code to nd $max(A, B)$ for the bitmap set of negative numbers.

The BSI maximum algorithm works in the same way, except that the K_A and K_B are the bitmap of rows having their maximum value in A and B, respectively. So instead of turning a bit on in K_B when the bigger value is in A, turn a bit on in K_A and vice-versa.

[†]Could choose K_A instead to get the same results.

CHAPTER 4

Term Matching

In this chapter, *Bit-Sliced Index Term Matching (BSTM)* algorithms will be covered. A discussion of how document weights are indexed will be the subject of the rst section, then the new BSTM algorithms will be introduced, followed by a discussion of their implementation. Experimental results will be presented in the following chapter.

4.1 Weights Indexing

Let's recall the cosine similarity measure of Figure 1.6:

$$C(d,q) = \frac{\sum_{t \in q \wedge d}(w_{q,t} \quad w_{d,t})}{W_d}.$$

Only the coecient $w_{q,t}$ is query-specic. The other coecients, $w_{d,t}$ and W_d, are independent of the query. $W_d = \sqrt{\sum_{t \in d} w_{d,t}^2}$ is the length of document d, and $w_{d,t} = log_e(f_{d,t} + 1)$ is the weight of term t in document d. So these values do not change when the query changes. Instead of indexing the document-term frequencies $f_{d,t}$, the ratios $W_{d,t} = \frac{w_{d,t}}{W_d}$ are indexed. At index creation time, for every document, the document-term frequencies $f_{d,t}$ are computed, then the document-term weights and the document length are computed, and the $W_{d,t}$ values are calculated. Finally, for every term, a bit-sliced index containing the $W_{d,t}$ values is created. Figure 4.1 shows the algorithm.

Note that the sort in Algorithm 4.1 is a disk-based sort, since normally there would be too many (term, document, weight) triples to t in buer. The main cost of this algorithm is probably the sort, unless the preprocessing steps get very sophisticated and time consuming. To build the indexes from the sorted triples, we loop through the triples, and build a dierent BSI for every term. When we encounter a triple for the rst time in the sorted order, we nish the previous term's BSI, start a new BSI and insert the document weight. If it is not the rst time we encounter the term, we simply insert the document weight in the current BSI. But the weights as computed in Algorithm 4.1 cannot be indexed directly into a BSI since they are real numbers, not integers. As explained in [MZ96], a

Algorithm 4.1 Weight Indexes Construction The algorithm gets the documents from some source, which could be a list of le names or directory names containing les and reads the documents before preprocessing them with the steps given in Chapter 1. Or the source could be a random document generator (useful for benchmark purposes), in which case the preprocessing steps may not be necessary.

```
while there are more documents to process
    get next document $d$
    apply the preprocessing steps to $d$ if necessary
    $W_d = 0$
    for every term in $d$
        compute $f_{d,t}$
        $w_{d,t} = log_e(f_{d,t} + 1)$
        $W_d = W_d + w_{d,t}^2$
    $W_d = \sqrt{W_d}$
    for every term in $d$
        $W_{d,t}$ = six_bit_approximation($\frac{w_{d,t}}{W_d}$)
        insert the ($t$, $d$, $W_{d,t}$) triples in a sort object
sort the triples with $t$ as rst key and $d$ as second key
go through every sorted triples in order
    for every dierent term, build a BSI from the ( $d$,$W_{d,t}$) values
```

Figure 4.1: Weight Indexes Construction

Algorithm 4.2 BSTM. Given a query Q consisting of a set of (term, weight) pairs, nd the top k best matching documents. C is the cosine BSI, recording the cosine value of every document d in the database relative to Q. C starts empty.

```
for each (term, weight) pair $(t, w_t) \in Q$ do
    nd $B_t$, the BSI for term $t$
    $C = C + w_t \; B_t$        // multiply $B_t$ by the constant $w_t$
nd $k$ largest $C$ values (using Algorithm 4.3)
return top $k$ $(d, C[d])$ pairs
```

Figure 4.2: Bit-Sliced Weighted Term Matching Algorithm

6-bit approximation of documents weights is known to be sucient to get almost unchanged retrieval eectiveness. We follow this advice scaling the weights into 6-bit quantities is appropriate. The 6-bit approximation helps BSTM to perform well since it puts a limit on the number of slices to process during term ranking computation. Fewer carries need to be computed.

4.2 Bit-Sliced Weighted Term Matching

Starting from a query $Q = \{(t_1, w_1), (t_2, w_2), ..., (t_q, w_q)\}$ with q terms and their corresponding weights, the goal is to nd the top k documents with highest weight matching of the query terms. The algorithm is given in Figure 4.2, using a notation similar to the WTM algorithm of Figure 1.5, an adaptation of the STM algorithm of Figure 1.4, taken from [PW83]. At rst glance, the two algorithms do pretty much the same thing, but the dierence lies within the details. Here, the cosine similarity measure is used instead of a generic one. As discussed in Section 4.1, some precalculation is needed to get good performance out of the bit-sliced approach, and the algorithm of Figure 4.1 is specic to the cosine measure. Similar algorithms can easily be written for other measures.

Algorithm 4.2 can be broken into three main steps:

1. cosine BSI build (the for loop)

2. top k documents search

3. top k documents extraction.

Step 1 makes use of BSI multiplication by a constant and BSI addition. Step 2 is implemented with an algorithm similar to the range search algorithm presented in [OQ97], and its description is the subject of the next subsection. A found set bitmap is returned by step 2, representing the set of documents with the top k cosine values. For each such document, we determine its corresponding cosine value in $C[d]$, then sort all those $(d, C[d])$ pairs (highest $C[d]$ rst), and return them to complete step 3.

4.2.1 Top K Documents

To nd the top k values in a BSI, we use an algorithm similar to a range search, which was covered in [OQ97]. Figure 4.3 shows the algorithm. It achieves its goal in a rather subtle way, so before presenting a proof, I will show an example of how it works.

Let's rst consider the table of Figure 4.4. First note that S would be the cosine BSI as computed in the rst part of Algorithm 4.3. Actually, to make the example simpler, I used the number of terms in common between the query and the document as values in S . The actual cosine values would be dierent, but it would be more complicated to write a clear example. The documents in the table are in decreasing order of their S-values. This would not normally happen, but it also makes the example easier to follow.

The goal is to get a found set bitmap, where a document is in the found set if and only if it is a top 4 document, i.e. if its S-value is one of the four largest S-values. In the example table, what we want to get is the bitmap $F =$ `1111000`. Remember that in practice, there could be a few hundred thousands to a few million documents, maybe even more, and that the top k documents could be anywhere, not just at the beginning. And we want to use the SIMD boolean operations described in Chapter 2. The idea is to go from high order bit-slices down to low order bit-slices, and as we go along, record which rows have the largest values. We stop when we have found k large values.

We start with two bitmaps, G and E. G stands for "greater than", and E stands for "equal to". What we nd as we go through the algorithm is the cut-o value m, i.e. the minimum S-value for a document to be in the top k documents. Documents with an S-value smaller than m will not be in F. Documents with an S-value equal to m may or may not be in F because we need exactly k documents and we need to break ties arbitrarily. There are less than k documents with an S-value greater than m, but when we add documents with an S-value equal to m,

That is, I really use simple term matching here, as described in Section 1.1.1; Algorithm 4.3 does not depend on where S comes from.

Algorithm 4.3 Top K Documents. Given a BSI $S = S^P S^P \ \ 1 ... S^1 S^0$, nd the k rows with the largest S-values. F is the found set bitmap.

```
if ($k > COUNT(EBM)$ or $k < 0$)  // test if parameter $k$ is valid
    Error ("$k$ is invalid")
$G = \emptyset$
$E = EBM$
for ($i = P$; $i$     0; $i$       ) {
      $X = G\ OR\ (E\ AND\ S^i)$  // $X$ is trial set
      if (($n = COUNT(X)) > k$) // if $n = COUNT(X)$ has more than $k$ rows
          $E = E\ AND\ S^i$          // $E$ in next pass contains only rows $r$ with
                                     // bit $i$ on in $S[r]$
      else
          if ($n < k$) {             // if $n = COUNT(X)$ has less than $k$ rows
              $G = X$                // $G$ in next pass gets all rows in $X$
              $E = E\ AND\ (NOT\ S^i)$    // $E$ in next pass contains no rows $r$
                                     // with bit $i$ on in $S[r]$
          }
          else {                     // $n = k$; might never happen
              $E = E\ AND\ S^i$      // all rows $r$ with bit $i$ on in $S[r]$ will be in $E$
              break
          }
}           // we know at this point that $COUNT(G)$     $k$
$F = G\ OR\ E$                       // might be too many rows in $F$; check below
if (($n = (COUNT(F)$     $k) > 0$)// if $n$ too many rows in $F$
      turn o   $n$ bits from $E$ in $F$ // throw out some ties to
                                     // return exactly $k$ rows
```

Figure 4.3: Top K Documents Algorithm

Query: nd top 4 documents for {beef, chicken, lamb, pork}

ID	TERMS	S	S^2	S^1	S^0
1	{beef, chicken, duck, sh, lamb, pork }	4	1	0	0
2	{beef, chicken, lamb, pork}	4	1	0	0
3	{chicken, duck, sh, lamb, pork }	3	0	1	1
4	{beef, lamb, pork}	3	0	1	1
5	{chicken, lamb}	2	0	1	0
6	{sh, lamb }	1	0	0	1
7	{sh }	0	0	0	0

Figure 4.4: Top K Documents Example

ID	S^2	S^1	S^0	G	E	X	G	E	X	G	E	X	G	E	F
1	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1
2	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1
3	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1
4	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1
5	0	1	0	0	1	0	0	1	1	0	1	1	0	0	0
6	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0

Figure 4.5: Top K Documents Algorithm Example

we may have too many, so we need to drop some of those documents. G contains the documents (having an S-value larger than m) we know for sure are in F, and E contains the documents with an S-value equal to the part of m we know up to that point (it will become clearer as we go along). At the start, G is empty, E is equal to the EBM, and $m = m_2m_1m_0$, each m_i being a bit in the binary representation of m. We don't know any of these bits yet.

In the rst turn of the loop with $i = 2$, we compute the trial set $X = G\ OR\ (E\ AND\ S^2)$ (see Figure 4.5). X gets the documents that are already in G (currently none), plus the documents of E having a 1-bit in the highest order bit-slice of S. So X gets a set of documents with large S-values. Then we count the number of bits on in X (i.e. the number of documents in X), which is equal to 2 here. We need more documents since $2 < k = 4$, so we do $m_2 = 0$, $G = X$, and $E = E\ AND\ (NOT\ S^2)$. There are not enough documents d with $S^2[d] = 1$, therefore there must be rows in F with $S^2[d] = 0$ and m_2 must be 0. G is the set of documents with S-values greater than m, and doing $G = X$ respects that condition. We know at this point that documents 1 and 2 will be in F. To update E, the set of documents with an S-value equal to m, we have to remove the documents we just added to G since we know now that these documents do not have an S-value equal to m. The documents left in E all have an S-value equal to the part of m we know up to now, i.e. their highest order bit is equal to 0.

For the second turn of the loop with $i = 1$, we compute a new trial set X. X gets all the documents of G, plus the documents of E with a bit on in the bit-slice S^1. So we add the documents with the next largest values we can nd. We try to see if documents 3, 4, and 5 are part of the found set. $COUNT(X) = 5 > k = 4$, and we have too many documents. We need to remove some of these documents. We leave G as it is, we set m_1 to 1 since a document d with $S^2[d] = S^1[d] = 0$ will be under the cut-o value, and we update E this way: $E = E\ AND\ S^1$ for the same reason: we leave out documents with small S-values.

The next time in the loop with $i = 0$, the trial set X gets again the documents in G, plus the documents in E having their bit set to 1 in S^0. After counting the number of documents in X, we nd it is equal to 4, which is what we need. We set $E = E\ AND\ S^0$, $m_0 = 1$, and we are done with the loop. Next we set $F = G\ OR\ E$, and since we do not need to break out ties, the algorithm is nished and the top 4 documents are documents 1, 2, 3, 4.

To see alternative endings of the algorithm, suppose that document 5 has an S-value of 3, the same value as documents 3 and 4 (see Figure 4.6). We would not get 4 documents in the last trial set X, but 5. We would be done looping anyway since there are no more bit-slices to process in S, but in the last condition, where we check how many documents are in excess, we would nd that one document

ID	S^2	S^1	S^0	G	E	X	G	E	X	G	E	X	G	E	F
1	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1
2	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1
3	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1
4	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1
5	0	1	1	0	1	0	0	1	1	0	1	1	0	1	1
6	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0

Figure 4.6: Top K Documents Algorithm Example, Variation 1

ID	S^2	S^1	S^0	G	E	X	G	E	X	G	E	X	G	E	F
1	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1
2	1	0	0	0	1	1	1	0	1	1	0	1	1	0	1
3	0	1	1	0	1	0	0	1	1	0	1	1	1	0	1
4	0	1	0	0	1	0	0	1	1	0	1	0	0	1	1
5	0	1	0	0	1	0	0	1	1	0	1	0	0	1	1
6	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0
7	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0

Figure 4.7: Top K Documents Algorithm Example, Variation 2

needs to be dropped from the found set, and we could arbitrarily choose to drop either document 3, 4, or 5. In the RIDBIT project, we choose the rst document from E, in this case document 3.

Suppose now that both document 4 and 5 have a value of 2 (as shown in Figure 4.7). After the last trial set X is computed, we nd out we don't have enough documents in it, so we add document 3 to G with $G = X$, then we set E to contain only documents 4 and 5. Then we end up in about the situation as the previous variation, except that document 3 is in G, not in E. To get exactly 4 documents in F, we would drop document 4.

Sometimes, it may not be possible to have k documents in the found set. Modify the example by setting the S-value of documents 4, 5, and 6 to 0, and we will end up with only three documents in F. There is nothing wrong with that since it does not make sense to include in F a document with nothing in common with the query. k is really an upper bound on the number of documents to be put in the found set.

The cut-o value m used in the explanation above and in the proof following in the next section does not appear in Figure 4.3 because it not necessary to keep track of its value. It is useful in the proof and in the explanation of the algorithm, but no decision is taken based on its value. It is easier to justify the choice of the letters G and E in the algorithm. If we knew the value of m before hand, then the algorithm would be much simpler since all we would need to do is a range search on S, and maybe to drop some documents out of the found set to arrive at exactly k documents. But we never know m in advance, making the algorithm more complex.

4.2.2 Proof of Algorithm 4.3

We wish to nd F, the bitmap of rows with the k largest S-values in a table T. Denote by m the minimum S-value of any row that lies in F, and assume m has binary representation: $m_P m_{P\ \ 1}...m_1 m_0$, This implies that m is *equal* to the k^{th} largest S-value $S[r]$ of all rows r in T (with possible ties, m might also be the $k + 1^{st}$ largest, etc.). We do not know m in advance, but we determine successive bits m_i of the binary representation as we progress through passes of the loop in Algorithm 4.3 with successively smaller values i.

Variables used in Algorithm 4.3 that exist from one loop pass to the next are the bitmaps G and E; the bitmap X and positive integer n are only temporary, used to hold results within a loop pass for eciency, and could be dropped from the code. We wish to demonstrate the dening properties of G (G contains rows r with $S[r]$ <u>G</u>reater than m) and E (E contains rows r with $S[r]$ <u>E</u>qual to m in a specic initial sequence of bits), so we provide an induction hypothesis specifying contents of G_i and E_i, which we dene as the values of G and E on entry to pass i. We then prove that the induction hypothesis remains true from pass i to successor pass $i \quad 1$, and conclude from this the nal contents of F.

Induction Hypothesis. Assume for an arbitrary row r in T that the binary representation of $S[r]$ is $r_P r_{P\ \ 1}...r_1 r_0$. Our induction hypothesis denes E_i and G_i as follows.

1. A row r in T will be in E_i if and only if $S[r]$ does not dier in its early bit representation $r_P r_{P\ \ 1}...r_{i+1}$ from $m_P m_{P\ \ 1}...m_{i+1}$.

2. A row r in T will be in G_i if and only if the early bit representation $r_P r_{P\ \ 1}...r_{i+1}$ is greater than $m_P m_{P\ \ 1}...m_{i+1}$; this is equivalent to saying that for some bit position j in the range $i + 1 \quad j \quad P$, bit r_j is on with bit m_j o, and bits $r_P r_{P\ \ 1}...r_{j+1}$ are all equal to bits $m_P m_{P\ \ 1}...m_{j+1}$.

We now perform induction. The initial test of Algorithm 4.3 guarantees that

$k \quad COUNT(EBM)$, and since m is the k^{th} largest S-value of any row in T, it guarantees that such a row r with $S[r] = m$ exists. We enter the rst pass of the loop with $i = P$; G_i is initialized to the empty set and obeys the induction hypothesis, since $i + 1 > P$ and thus there is no value j with $i + 1 \quad j \quad P$ to use in the dening property 2 above, so no rows are in G_i; E_i is initialized to EBM and obeys the induction hypothesis, since there are no bits above position $i = P$ that can dier from bits in m, as required in dening property 1.

Now assume the induction hypothesis holds at the beginning of the loop pass for value i: E_i consists of all the rows r in EBM that have early binary representation $r_P r_{P \quad 1} ... r_{i+1}$ equal to $m_P m_{P \quad 1} ... m_{i+1}$. Clearly the row r with $S[r] = m$ must lie in E_i. G_i consists of all the rows r' in EBM where there is some bit position j in the range $i+1 \quad j \quad P$ such that bit r'_j is on with bit m_j o, and bits $r'_P r'_{P \quad 1} ... r'_{j+1}$ are all equal to bits $m_P m_{P \quad 1} ... m_{j+1}$. ($G_i$ can contain no rows until a zero bit shows up in $m_P m_{P \quad 1} ... m_{i+1}$.) Begin by noting that every row in G_i (if there are any) has S-value larger than all the rows in E_i, since each of the rows r' in G_i have an early 1-bit r'_j matched by a 0-bit r_j in all the rows r of E (i.e., with $r_j = m_j$), and all bits prior to j in r' matching bits in r (i.e., the same as in m). Furthermore, since this stated characterization (for some j) holds for any $r' \quad r$ pair with $S[r'] > S[r]$, and since G_i contains all rows r' that obey this characterization, G_i must contain all the rows with S-values larger than *all* rows in E_i.

At the beginning of the loop in Algorithm 4.3, we set $X = G\ OR\ (E\ AND\ S^i)$ which we will rewrite as $X_i = G_i\ OR\ (E_i\ AND\ S^i)$. Now we claim that rows in X_i have the largest S-values of any rows in T. To demonstrate this, consider the following. We know that G_i contains all rows in T with S-values larger than any S-values in E_i. Furthermore, rows r in $(E_i\ AND\ S^i)$ have larger S-values than any of the other rows in E_i, that is rows r' in $(E_i\ AND\ NOT(S^i))$, since they have identical bit positions up to $r_{i \quad 1}$ and bit r_i on where bit r'_i is o. Finally, any row r not in G_i or in E_i, since its S-value representation $r_P r_{P \quad 1} ... r_{i+1}$ cannot be greater than or equal to $m_P m_{P \quad 1} ... m_{i+1}$, must have some bit position r_j o that is on in m_j, $i + 1 \quad j \quad P$, with $r'_P r'_{P \quad 1} ... r' j + 1$ all equal to bits $m_P m_{P \quad 1} ... m_{j+1}$, and thus must have an S-value smaller than any row in E_i. Thus rows in $X_i = G_i\ OR\ (E_i\ AND\ S^i)$ are either in G_i, and therefore have S-values larger than any row in E_i, or in $(E_i\ AND\ S^i)$ and have S-values larger than any other rows in E_i. The rows outside X_i are either in $(E_i\ AND\ NOT(S^i))$ or have S-values smaller than any row in E_i, so clearly X_i consists of the rows with the largest S-values in T. With these preliminaries, we are ready to consider cases.

Now if $n = COUNT(X_i) > k$, this will imply that m_i is on, since there were less than k rows in G_i (m is the k^{th} largest S-value and G contains only rows with S-values larger than m) and more than k when rows in $(E_i\ AND\ S^i)$ were added.

Thus the k^{th} largest S-valued row in T, must be in $(E \ AND \ \ S^i)$, and m_i will be on. Because $n > k$, we set $E_{i \ \ 1} = E_i \ AND \ \ S^i$ in the next line of the algorithm. The new bitmap, $E_{i \ \ 1}$, now has rows with $r_i = 1 = m_i$, and thus contains the appropriate set of rows for pass $i \quad 1$ by induction hypothesis 1, since rows in $E_{i \ \ 1}$ match all bits in m down to m_i. The new bitmap $G_{i \ \ 1}$ is unchanged from G_i, and this is valid for the induction hypothesis 2, since i was not an appropriate value for j in the denition to add new rows to $\ \ G$ with bit m_j o and bit $\ \ r_j$ on.

If $n = COUNT(X_i) < k$, we see that X_i, the set of n rows with the largest S-values in T, does not include the k^{th} largest. But if bit m_i were on, that would not be true, since by construction E_i contains all rows r with $r_P r_{P \ \ 1} ... r_{i+1}$ equal to bits $m_P m_{P \ \ 1} ... m_{i+1}$, and $(E \ AND \ \ S^i)$ would thus include m. Since bit m_i is o, our induction hypothesis 2 requires us to add new rows $\ \ r$ to $G_{i \ \ 1}$ with S-values that have r_i on and bits $r_P r_{P \ \ 1} ... r_{i+1}$ all equal to bits $m_P m_{P \ \ 1} ... m_{i+1}$; in other words we set $G_{i \ \ 1} = X_i = G_i \ OR \ \ (E_i \ AND \ \ S^i)$. This new set $G_{i \ \ 1}$ satises induction hypothesis 2 with $j = i$. Next we set $E_{i \ \ 1} = E_i \ AND \ \ (NOTS^i)$ restricting $E_{i \ \ 1}$ to rows in E_i with $r_i = 0 = m_i$; since all rows r in E_i already have bit representation $r_P r_{P \ \ 1} ... r_{i+1}$ equal to $m_P m_{P \ \ 1} ... m_{i+1}$, it is clear that $E_{i \ \ 1}$ satises induction hypothesis 1 for $\ \ i \quad 1$.

Finally, if $n = k$, then X_i consists of k rows with the largest S-value in T, exactly what we've been seeking. We set $E_{i \ \ 1} = E_i \ AND \ \ S^i$, and break from the loop; on exit we set $F = G \ OR \ \ E$ (the former $X \quad i$), and we will nd that $COUNT(F) \quad k = 0$. In this case, we don't need to continue the loop until $i = \quad 1$.

If we never encounter the case where $n = k$, we continue to loop through $i = 0$, and on exit from the loop (with $i = \quad 1$), we set $F = G_{\ \ 1} \ OR \ \ E_{\ \ 1}$, with $COUNT(G_{\ \ 1} \ OR \ \ E_{\ \ 1}) > k$. But all the S-values of rows in $E_{\ \ 1}$ are now the same (since they all have the same bit representation as m) and as always we know that $COUNT(G_{\ \ 1}) < k$. Thus we simply need to remove some rows of E from F until $COUNT(F) = k$, to nd the desired set $\ \ F$.

4.3 Memory Usage of BSTM

As we have seen in Section 1.2, IRTM algorithms need about 618 Kbytes of memory to hold the hash table of accumulators when there are one million documents in the database. A 6 bit-slice in-memory BSI needs about 750 Kbytes of memory: each bit-slice needs about 125 Kbytes (1 million bits), and 6 125 = 750. Maybe more than 6 bit-slices will be necessary (when query-weights are large), but high-order bit-slices will be very sparse, and low-order slices more dense; so high-order slices may take much less space than a low-order slice. The memory usage for a

BSI used as an accumulator is not xed, it depends on particular queries. But it will not take much more space than the IRTM with hashed accumulators.

With one million documents and 4 Kbytes bitmap segments, we need 32 segments per verbatim bitmap. For most bitmaps used in the experiments included in the following chapter, only one or two 4 Kbytes disk pages were necessary to hold the bitmaps. Most indexed terms are rare, so the bit-sliced indexes and their bit-slices indexing them are sparse. Most segments are held in a RID-list form, and many RID-lists can t in a single disk page.

CHAPTER 5

Experimental Results

I present in this chapter experimental results comparing Information Retrieval Term Matching and Bit-Sliced Term Matching (IRTM and BSTM). Synthetic benchmark tables and queries were generated for the RIDBIT implementation under varying conditions. The experiments were performed on a PC equipped with a AMD Athlon 1.33 GHz CPU, with 256 MB of DDR RAM, an UDMA 100, 7200 rpm, 40 GB, Maxtor IDE drive, running FreeBSD 4.5-PRERELEASE #9, obtained from the STABLE branch on December 31 2001. The kernel is a custom built kernel, with soft updates enabled.

The design of our benchmark tables is based on some of the larger document collections in [PW83], rather small collections by today's standards, but appropriate for our prototype system. In Figure 5.1, we provide a list of notational symbols used in our experiments, along with the values or range of values these symbols represent.

Focusing for the moment on the minimal conguration of Figure 5.1, we see we have $N = 50,000$ documents in our smallest table, with $T_D = 40$ terms for each document (terms are represented by integers because of limitations in our index implementation). This means that the number of term-document pairs contained in index entries is $N \quad T_D = 2,000,000$. Since there are 10,000 distinct terms, we calculate the average number of documents per term to be 200. The number of documents per term grows linearly with the number of documents, for $N = 100,000$

Notational symbol	Values used
N (# rows = # docs)	50K, 100K, 250K, 500K, 750K, 1M
T (# dierent terms)	10,000
T_D (# terms/doc)	40
T_Q (# terms/query)	5, 10, 20, 30, 40, 50, 100, 150, 200
D_Q (avg. # docs/query-term) (approximately linear in N)	0.01 N = 500, 1000, 2500, 5000, 7500, 10000

Figure 5.1: Notation Used in Experiments

we have 400, 1000 for $N = 250,000$, etc. We generated the terms in each document at random, using a Zipan 70-30 distribution skew (a realistic assumption), and then created queries whose terms tended to use the more popular terms, behavior we copied from [PW83]. When the average number of documents per term is 200, the average number of documents per query term is 500, i.e., $D_Q = 500$. In general, we tuned the Zipan function choosing terms of the query so that query terms are 2.5 times more popular than the average document terms; so for $N = 100,000$, when the average number of documents per term is 400, the average number of documents per query term is 1000, i.e., $D_Q = 1000$. The number of rows (or documents) N in the tables and the number of terms per query T_Q are the only independently ranging parameters of Figure 5.1, and we ran experiments with all pairs of values. We randomly generated query runs with $T_Q = 5, 10, 20, 30, 40, 50, 100, 150, 200$ terms, and ran them against implementations of the IRTM and BSTM algorithms on the RIDBIT project for tables of $N =$ 50K, 100K, 250K, 500K, 750K, and 1M rows. In the rst set of runs, query-term weights were all equal to 1. The following set of runs has query-term weights randomly generated, every weights being a power of 2. The last run contains also randomly generated query-term weights, but not limited to powers of 2. For all randomly generated query-term weights, the precision was limited to 6 bits since document-term weights are limited to 6 bits (see Section 4.1). Figures 5.2 to 5.11 present timing results comparing BSTM and IRTM, and a discussion of these follows. The data used to produced these graphs are included in Appendix B.

5.1 Results Analysis

Getting results with query-weights equal to 1 is meaningful since in practice, many queries are initiated by users simply typing some terms in a text eld of a web page (e.g. queries on Google or Yahoo search engines). In [Kir01], it is mentioned that only 10% of users use query syntax, and 1% use advanced search, so most of the time, users type only a simple list of terms. When no query-weights are specied, all query-weights are assumed to be equal, and assuming they are equal to 1 is very natural. BSTM wins over IRTM in this case. Results were also obtained with query-weights equal to powers of 2 and with unrestricted query-weights values. As we will see, results with query-weights equal to powers of 2 are almost identical to results with query-weights equal to 1, but with unrestricted query-weights values, BSTM loses compared to IRTM. The bottleneck with IRTM is the inverted lists lengths, while with BSTM, the problem can be the number of carries to compute. When the binary representation of the query-weights contains many ones, then more carries need to be computed since more additions need to be performed.

5.1.1 Query-Weights Equal to 1

Figures 5.2 and 5.3 show graphs of the query execution time versus the number of query terms, for query-weights equal to 1. Figure 5.3 is a zoom in of Figure 5.2. We can see in Figure 5.2 that BSTM is better than IRTM up to about 100 to 125 terms per query. As the number of documents increases, the crossing point between the corresponding lines (e.g. between the "1M docs IR" and "1M docs BSI" lines) moves to the right. In other words, IRTM seems to be more aected by the number of documents in the database. We can expect this trend to continue since, as mentioned before, IRTM is more sensible to inverted lists length. Having a cross-over point at 100 query-terms or above is not a problem since users of search engines do not type in 100 or more terms to search on; users typing more than 10 terms are rare. In [MZ96], the authors agree since they write *"Queries of perhaps 3-10 terms are the norm for general-purpose retrieval systems."* on page 359. They principally talk about queries of 40 to 50 terms, which is well under the 100 terms limit. BSTM performs well with 40 and 50 query-terms. Also, Steve Kirsch, founder of *Infoseek*, now at *propel*, mentioned during a talk at the SIGMOD 2001 conference, that a query on a WWW search engine, like Google, *"gets faster the longer the query"* because only the *"8 shortest term lists"* (inverted lists) are used to answer the query [Kir01]. If a user asks for more than 8 query-terms, only the 8 less frequent query-terms will be used, and the query-terms in excess will just be ignored. Recall that if a query-term is rare and appears in a given document, then it makes this document more relevant (more similar) to the query than a common term. This document is more "special" and probably more interesting. Also from [Kir01], the average query length is 2.2, well within limits.

The query execution time versus the number of documents graphs are shown in Figures 5.4 and 5.5. Figure 5.5 is a zoom out of Figure 5.4. We can see on these graphs that the slope of the IRTM lines are larger that the slope of the corresponding BSTM lines, except for very large number of terms. The time dierence is not constant between the corresponding lines as the number of documents increases. We can expect this trend to continue with even more documents, but due to the RIDBIT system limitations, we were not able to try it. Note that the *10 terms BSI* case runs faster than the *5 terms IR*, that the *20 terms BSI* case runs faster than the *10 terms IR*, that the *30 terms BSI* case runs faster than the *20 terms IR* and that the *40 terms BSI* case runs almost faster than the *30 terms IR*. The most important we can get from those graphs is that we can expect BSTM to scale better than IRTM.

Figure 5.6 shows the ratio of IRTM execution time and BSTM execution time versus the number of documents graph. We can see immediately that the 10 query-terms case is the best for BSTM. The ratio goes up when the number of documents increases. For 1 million documents, IRTM is 2.5 times slower than BSTM. Similar results for 5 and 20 terms, but here, IRTM is about 2.2 times slower than BSTM. For 30, 40, and 50 terms, it is not bad either: IRTM is 1.5 to 1.7 times slower than BSTM. It gets worse for 100, 150, and 200 terms, but, as mentioned earlier, these cases are not common in practice for the targeted applications. Leading search engines are using at most 8 query-terms anyway.

5.1.2 Query-Weights Equal to Powers of 2

The query execution time versus the number of documents graph is shown in Figure 5.7, and the ratio of IRTM execution time and BSTM execution time versus the number of documents graph is shown in Figure 5.8, both for the case where query-weights are equal to a power of 2. If we compare Figure 5.7 to Figure 5.5 and Figure 5.8 to Figure 5.6, then we can see similar graphs. Before adding a BSI to the sum BSI, we rst need to multiply the BSI by the query-weight, but since that query-weight is equal to a power of 2, only a shallow left shift is needed (see Section 3.3) instead of a general multiplication, and then we can add the BSI to the sum. Shallow left shifting is cheap, and it is not adding much work to the execution time directly, but indirectly, it is adding some work: the weights kept in the sum BSI will be larger, therefore more bit-slices will be needed in the sum BSI, and more carries will need to be computed. The impact is still small, as we can see for the 10 terms, 1 million documents case, IRTM is still 2.3 times slower than BSTM, compared to 2.5 times slower when every query-weight is equal to 1. I omit some graphs for the query-weights equal to powers of 2 case since they are very similar to the query-weights equal to 1 graphs.

5.1.3 Unrestricted Query-Weights

Figures 5.9, 5.10 and 5.11 show graphs for execution times versus the number of terms, execution times versus the number of documents, and the ratio of IRTM execution time and BSTM execution time versus the number of documents, when query-weights can take on any values (limited to 6-bits, as explained above). BSTM is worse because more carries need to be computed since the work is tripled: with 6-bit query-weights, we can expect to have 3 bits on and 3 bits o in the binary representation of the weight, thus the algorithm will do 3 shallow left shifts and 3 BSI addition. Three times the number of additions gives about three times as much work. From the graphs data, we can get that IRTM performances are, on average, about 5% worse in this case compared to the previous case, while BSTM performances are 2.97 times worse on average.

In Figure 5.10, observe how the 150 terms IR and 200 terms IR lines grow quickly for 750 thousand and 1 million documents. This can be observed in Figure 5.11 also. Does IRTM reach (or gets closer to) some limit? With 1 million documents, we have $D_Q = 10000$ documents per query-term, thus when we have 150 query-terms, IRTM needs to handle 1.5 million document pointers. With 200 query-terms, 2 million document pointers. With 750 thousand documents, $D_Q = 7500$ documents per query-term, and IRTM needs to handle 1.5 million document pointers. No other case in the experiments handle that many document pointers. As mentioned a few times before, IRTM is more sensible to inverted list length and the amount of term-document pairs, while BSTM scales better. Because of the test system's limits, we cannot go much higher than 1 million documents, therefore the question: *as IRTM reached (or got closer to) a limit?*, is going to stay open for now and may be answered in future work.

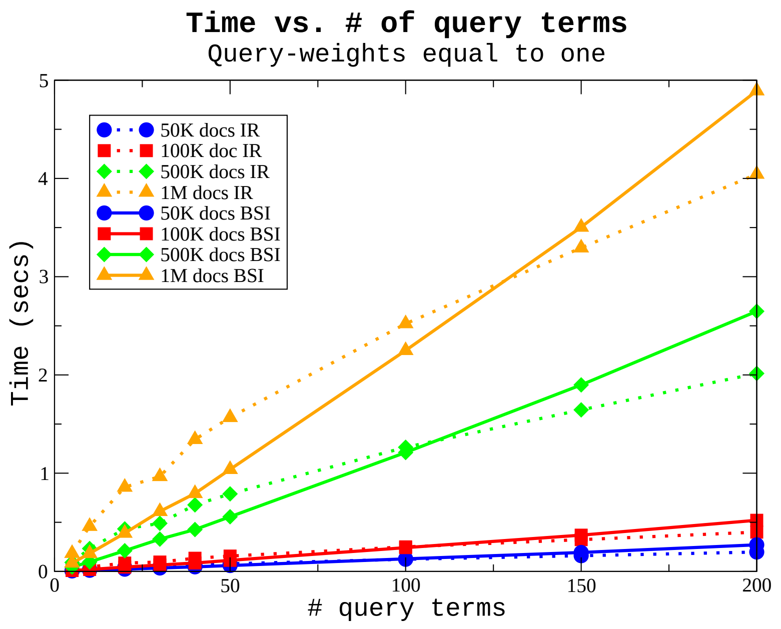

Figure 5.2: Time vs. # of Query Terms

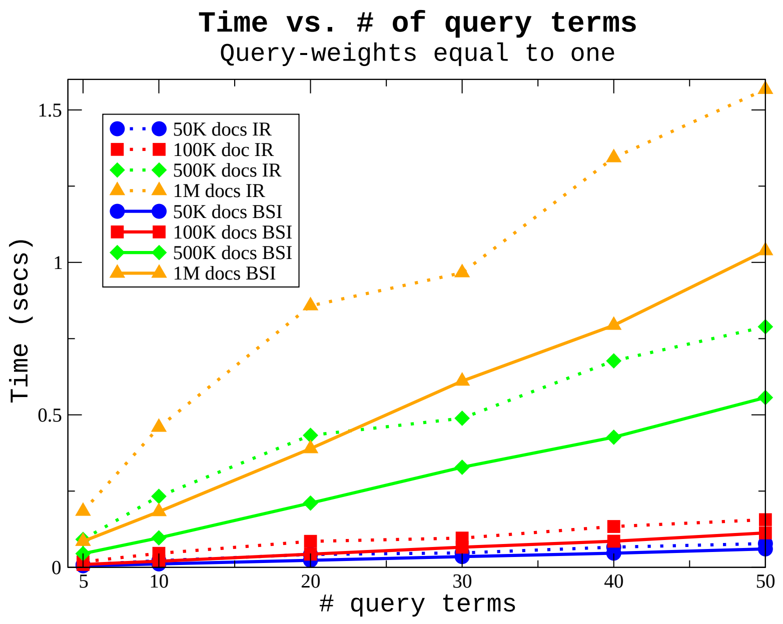

Figure 5.3: Time vs. # of Query Terms

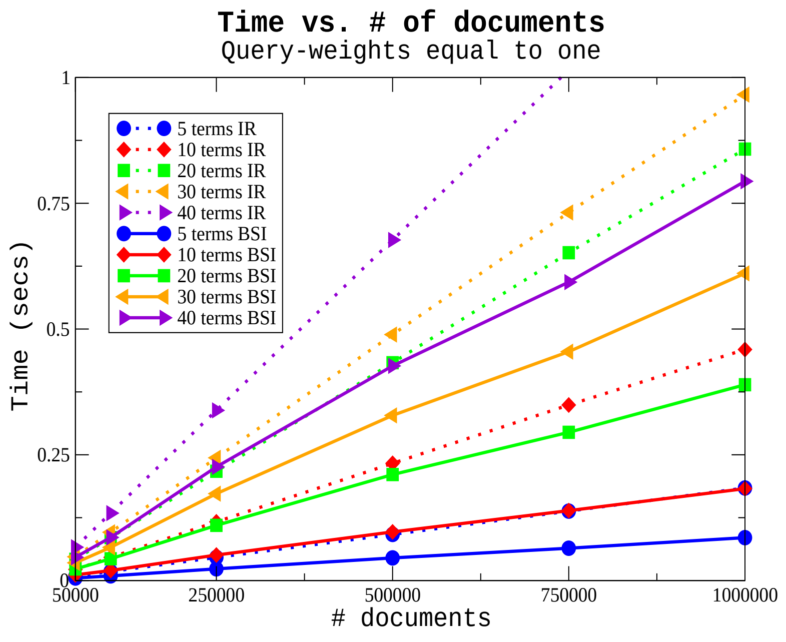

Figure 5.4: Time vs. # Documents

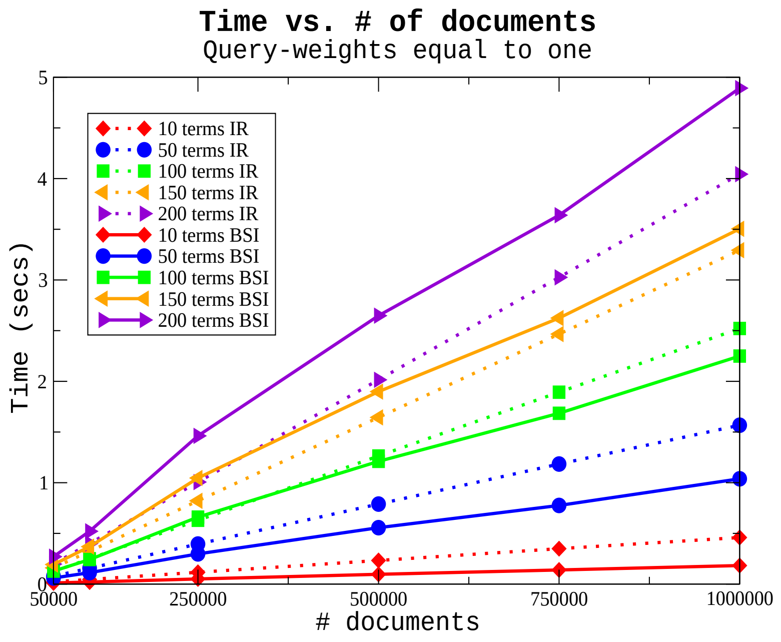

Figure 5.5: Time vs. # Documents

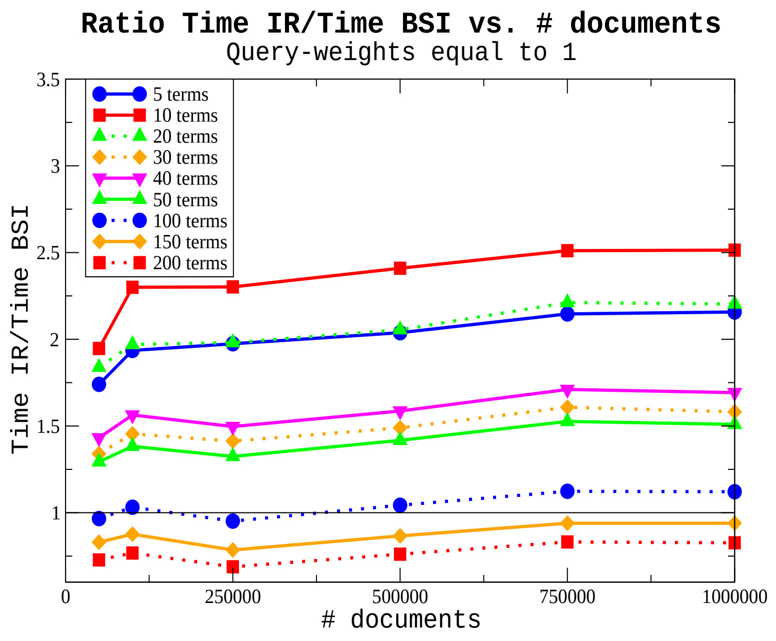

Figure 5.6: Ratio Time IR/Time BSI

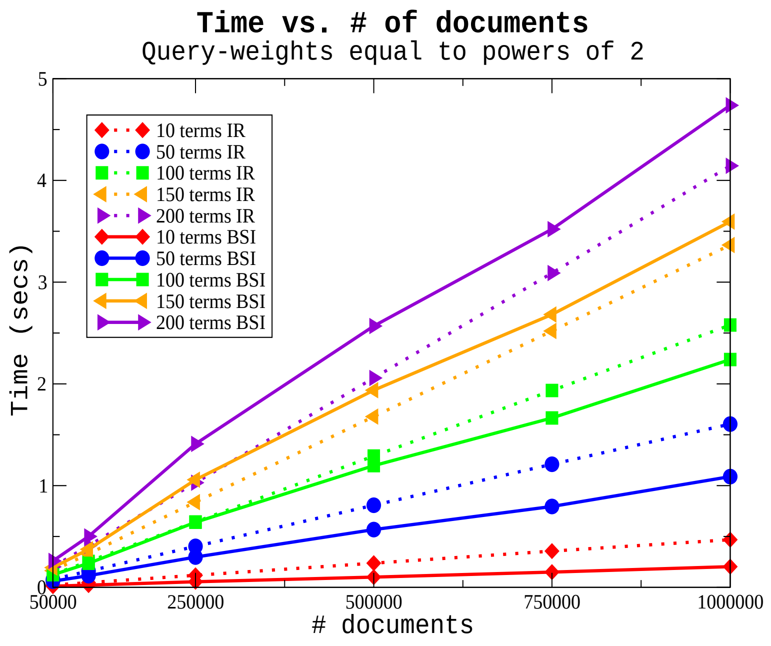

Figure 5.7: Time vs. # Documents

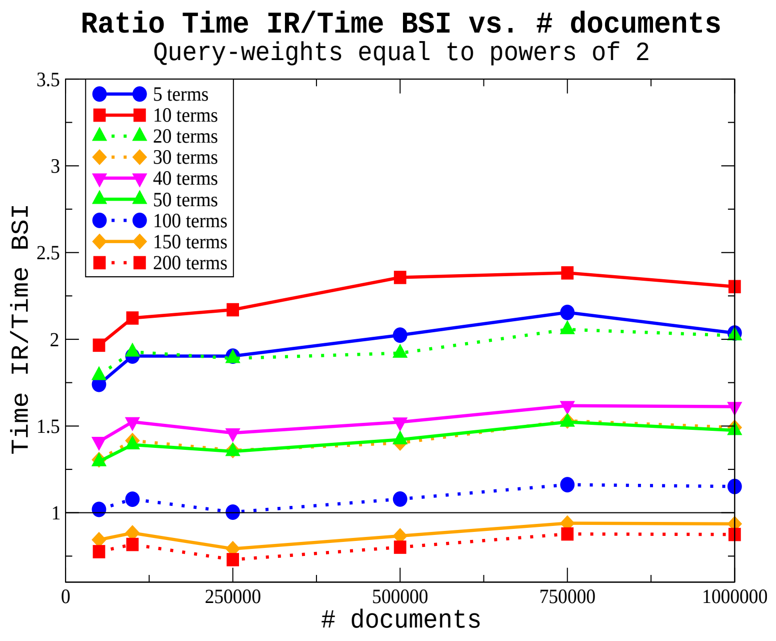

Figure 5.8: Ratio Time IR/Time BSI

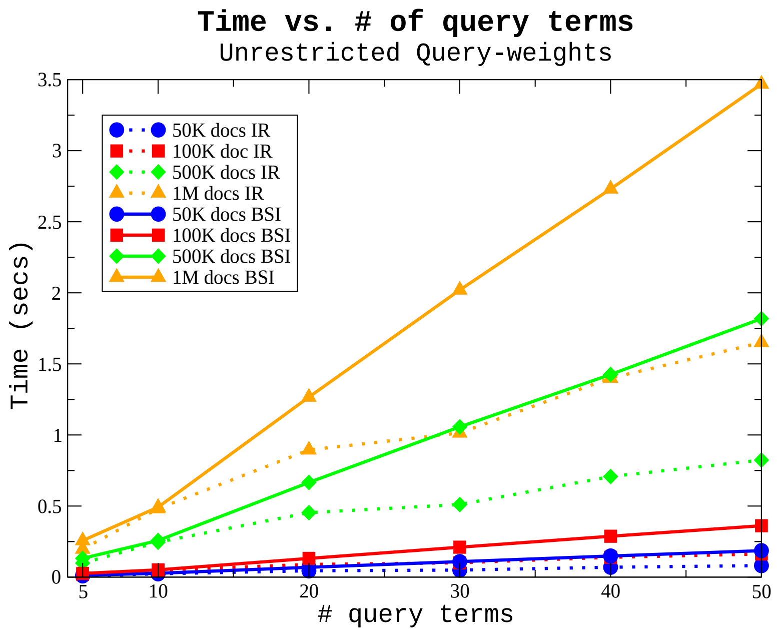

Figure 5.9: Time vs. # of Query Terms

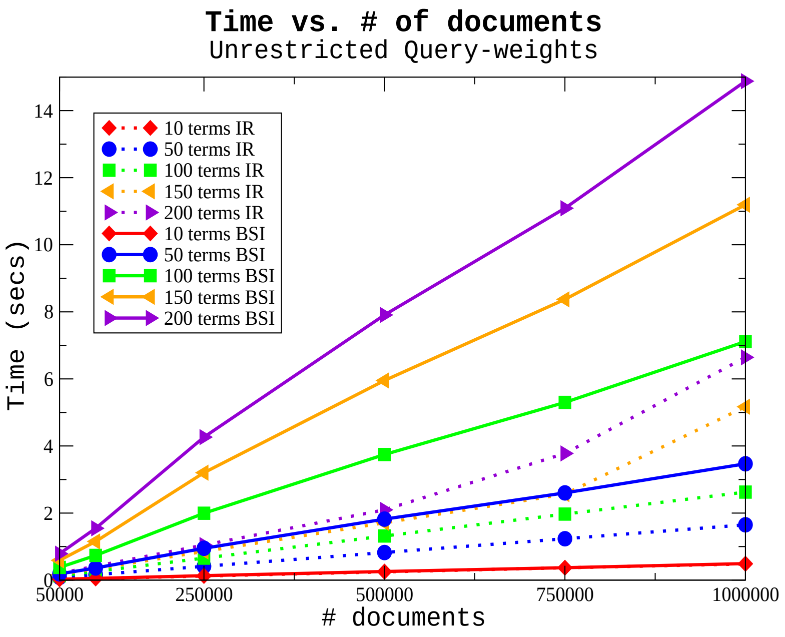

Figure 5.10: Time vs. # Documents

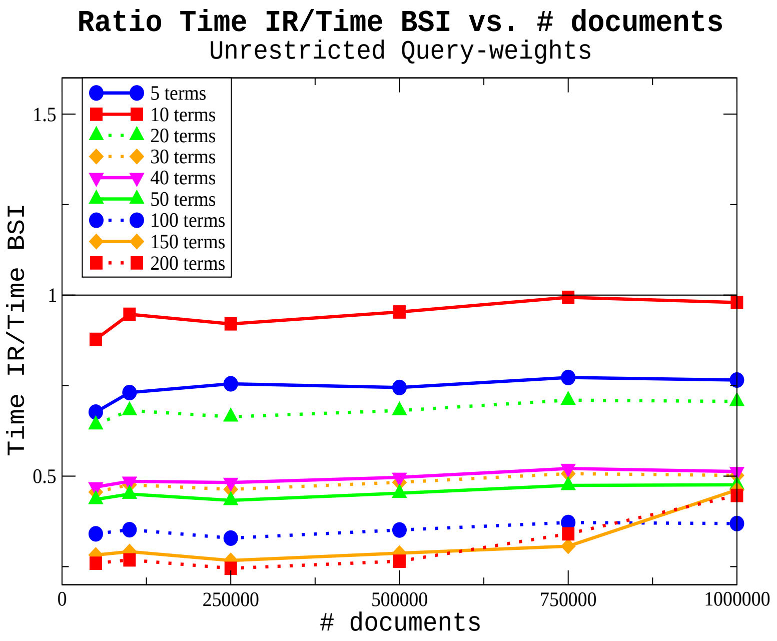

Figure 5.11: Ratio Time IR/Time BSI

CHAPTER 6

Other Applications and Future Work

Users of a search engine should not have to worry about implementation details of the database used to support the search engine. What they want to do is type some terms, click on the *Search* button and get (pertinent) results. What happens behind the scene is more complicated: the text typed by the user is parsed into terms, and the terms are preprocessed as described in Chapter 1 (some terms may be dropped as mentioned in Section 5.1.1), and nally a term matching query is issued to the database. A set of (doc #, weight) pairs are returned by the database to the WWW server, which processes the set to create an HTML le (or some other le format), and sends it to the client to be displayed in a WWW browser (or some other application). The return set can be broken into pieces, so as to return, for example, only 10 results per page. For each document number, a database access is needed to retrieve the document title, perhaps a passage or a summary of the document, and various other information.

But what if we want to allow users to specify query-term weights? We propose a number of syntax alternatives. One option would be to let users specify a query in a form similar to this (using an example similar to Figure 1.1):

Q 6.1 *lentil:10 onion:1 tomato:4 garlic:2 cumin:1*

The user puts a lot of importance on the term *lentil*, so the lentil recipes will score higher, then recipes with *tomatoes* will be favored next, then recipes with *garlic*, *onion* and *cumin*. The weight values should be limited to some realistic range, e.g. between 1 and 10 would make sense. The problem with BSTM, as described in Chapter 5, comes when many bits are on in the query-weights. When there is only one bit on in the query-weights, BSTM is fast.

There is a simple alternative that makes the query syntax easier and that can be set up to use only powers of 2 for query-weights. The idea is to use a syntax similar to what Yahoo is using [Yah]. They only allow Boolean searches, but they allow users to put a + sign in front of a term, to make the term mandatory, so that a query like

Q 6.2 *+lentil onion +tomato garlic cumin* can be specied. In this query, *lentil* and *tomato* are mandatory, and *onion*, *garlic* and *cumin* are optional, but documents containing the optional ingredients will be favored over documents without them.

We could extend the syntax to allow users to put more + signs in front of query-terms. A user could type the query:

Q 6.3 *+++lentil onion ++tomato +garlic cumin*

and would get results similar to query 6.1. No + sign in front of a term would mean the term has a weight of 1, one + sign would mean a weight of 2, two + signs a weight of 4, three + plus signs a weight of 8, etc ... That is, the weight of a query-term would be $2^{n_+(q)}$, where $n_+(q)$ is the number of + signs in front of that query-term. This process allows users to specify query-weights easily, and on the side of the database, searches with query-weights equal to powers of 2 is fast.

A question that can come up easily is: *Is it meaningful to have query-weights growing so fast as the number of + signs grows?* My answer is obviously *Yes*. First, the number of + signs can be limited to 4 or 5, which keeps the largest possible weight to 16 or 32. Second, I don't believe users will be interested in typing more than 4 or 5 + plus signs anyway, and probably most of them will not type more than 1 or 2 plus signs per term. Sometimes, when a term is very important, a user may be willing to type 4 or 5 + signs, but I don't think it would be a common case to do that. Thus, typing 4 or 5 + signs would be a rare case, and it is justied to weight rare cases importantly.

Yahoo also allows users to put a – sign in front of a term, meaning that the associated term must not appear in any document returned to the user. This is not incompatible with BSTM, since a *veto* bitmap can be constructed, consisting of the union of the bitmaps corresponding to the negated terms in the given query. This veto bitmap can be applied against the results obtained from the usual BSTM algorithms presented before. For eciency purposes, this veto bitmap should be applied sooner to potentially save some work. The veto bitmap can be negated and used during the early stages of BSTM in place of the EBM to lter out some documents early . The veto bitmap density will be lower than the EBM density, so the BSI sum density will also be lower than it would be without the veto, and the bitmap Boolean operations could gain speed.

Recall that when a bitmap is negated, it is automatically intersected with the EBM to remove non-existing documents, so the negation of the veto bitmap will contain every existing document not containing any of the vetoed terms

6.1 Multi-Column Queries With BSTM

We can extend the ideas presented above to multi-column queries. Consider a query, similar to query 6.3, but this time, we don't want to search for recipes based only on their ingredients, but based also on their preparation time:

Q 6.4 $\{$*+++lentil onion ++tomato +garlic cumin*$\} \in$ *INGREDIENTS AND PREP_TIME 30*

The steps taken to answer this query would be about the same as for the veto bitmap case, except that here, instead of negating the union of some bitmaps, a range search is performed on the PREP_TIME column to select the recipes which can be prepared in 30 minutes or less. Then term matching can be done on the INGREDIENTS column, but using the results of the range search as the set of potential answers, and executing the BSTM algorithms only on this reduced set. This technique could be extended to any Boolean expression combined with rank computation.

The syntax used in query 6.4 is probably too complicated for an average user to use (even if we change $\in$ by *in* and by <=), so a nicer interface could be created. For example, there could be a text eld for entering ingredients to rank the recipes on, and there could be another text eld where Boolean conditions could be typed in. A user would type the same thing as query 6.3 in the rst text eld, and would type *PREP_TIME <= 30* in the other.

Another interesting question is: *Can we rank recipes based on more than one column?*. Yes. We could imagine having, in the same RECIPES table, a column called QUALIFIERS, in which each recipe could have a set of qualiers, like *vegetarian, dessert, sweet, sour, fatty, lean, spicy*, etc... One could want to make a query similar to query 6.4, but also would like these qualiers: *++++vegetarian +spicy*. This could be achieved with BSTM (and IRTM also), because the BSTM algorithms do not care where the BSIs they work on come from: a list of BSIs are fed into the algorithms, and the top k recipes come out. Care must be taken to have weight BSIs coming from dierent columns with comparable values, e.g. if the INGREDIENTS weights span the range $[0, 100]$ and the QUALIFIERS range is $[0, 1]$ (i.e. the QUALIFIERS BSIs are really bitmaps), since a recipe with *garlic* as one ingredient could gain more weight from the *garlic* than a recipe having *vegetarian* as a qualier. Weights should be scaled in some way to have balanced weights between the two columns. Unfortunately, I do not know exactly how this should be done, since it is probably a case-by-case problem and it depends on which importance the database creator wants to put on each column.

We could go even a step further by pushing the notion of term matching. Until now, the term matching examples I used where based only on set-valued columns of some table (most of the time, on an INGREDIENTS column of a RECIPES table). Query 6.4 could be modied to get this query:

Q 6.5 {*+++lentil onion ++tomato +garlic cumin*} $\in$ *INGREDIENTS, ++(PREP_TIME   30)*

The idea is to rank recipes not just on ingredients, but also on the condition *PREP_TIME   30* by giving it a ++ weight (equal to 4). The range search produces a bitmap to represent the set of recipes which can be prepared in less than 30 minutes, and this bitmap can be left-shifted by two positions into a BSI, to be used in the BSI addition. Of course, this is added work for the database, but users could nd this feature useful. This could be generalized to any operator returning a bitmap or a BSI as a result.

If we go another step further, we could allow weights to be subtracted. Modifying query 6.3, we get:

Q 6.6 *+++lentil onion ++tomato +garlic   cumin*

Notice here two – signs in front of *cumin*. Maybe the user does not particularly like cumin very much, but does not want to veto recipes containing cumin. The terms would be processed in the same way as before, with the cumin BSI being left-shifted rst (by one position since one – sign would mean simply subtract this term, and two – signs would mean subtract even more), but this time, it needs to be negated before being added to the other BSIs. Given that the database can implement BSI negation eciently [†], query 6.6 could be answered almost as quickly as query 6.3.

6.2 Arithmetic Queries

BSI arithmetic can also be used to evaluate more complicated expressions used in SQL (or other kind of) queries. For example, take the expression $T.C_1 + T.C_2 \quad c$. Given that bit-sliced indexes exist on columns C_1 and C_2, a BSI addition can be executed on these two BSIs to get a BSI S, followed by a range search on S. More general expressions like

$$\sum_{i=1}^{n} a_i \quad T.C_i \bigtriangleup c$$

[†]Report to Section 3.4, where it is explained why the RIDBIT project does not do well with BSI negation and multiplication (it does not support varying encoding schemes), and how to x it to support ecient BSI negation.

where the a_is and c are constants, and $\triangle$ is one of $\{<, \quad , =, \neq, \quad , >\}$. We can even go further by allowing any operator working on BSIs to be used (multiplication, subtraction, etc...) instead of addition and by replacing the range search by any other meaningful BSI operator (e.g. sum, average [OQ97]). The performance analysis comparing these expressions implemented with/without BSI arithmetic is left as future work.

6.3 Preference Queries

Preference Queries are a kind of ranked queries, where the user species a *preference function*, evaluated on every tuple of a table to obtain a score for every tuple, and the highest score tuples are returned rst [HKP01]. The goal is to optimize the selection of objects by appropriately weighting the importance of multiple objects attributes. Such optimization problems appear often in operations research and applied mathematics as well as every day life. Given a relation $R(A_1, A_2, ..., A_n)$, the user species the preferences $a_1, a_2, ..., a_n$ (describing the query), over attributes $A_1, A_2, ..., A_n$. The preference function over these attributes is $a_1A_1 + a_2A_2 + ... + a_nA_n$, giving a score to every tuple. Conventional evaluation techniques for such queries require the retrieval and ordering of the entire dataset. The work in [HKP01] is based on the framework introduced in [AW00].

Their approach to implement preference queries is to precompute preference functions and store them as materialized views, and to use those views to answer queries. They provide algorithms to make a good choice of preference functions to materialize, so that the space of preference functions is covered (refer to the paper for details). They provide algorithms to combine dierent views to avoid having to retrieve and order the whole dataset to answer a query. Once a query is issued, they look for a view that is similar to the query, and from that view they extract the tuple with the highest score. Their algorithm is such that a minimum number of tuples are examined to return the top score tuples. They call their system *PREFER*, and it is built as a layer on top of a commercial RDBMS.

They concentrate on the percentage of queries covered in their experimental results, varying dierent parameters, mainly the number of materialized views and the dataset size. They keep for last the only execution times graph, comparing their algorithms with a straightforward DBMS only approach. They use a table with 50,000 tuples and four attributes. The preference values (the a_i coecients) vary between 0 and 1, with a discretization of 0.1, i.e. preferences are taken in the set $\{0, 0.1, 0.2, ..., 0.9, 1\}$. The DBMS they are using is Oracle, and the PREFER system is running on another machine (dual Pentium II, 512 MB of RAM, Windows NT Workstation 4.0). When 10 top score tuples are requested, PREFER executes the queries in about 1 second on average, and when 500 top score tuples are requested, it takes about 19 seconds to execute. The straightforward DBMS only approach takes, respectively, about 40 and 43 seconds to execute. They also evaluate the time taken to build the materialized views. For a discretization of 0.1, and a table with 5 attributes, it takes 210 minutes to build the views. For a discretization of 0.05, it takes 2000 minutes.

So the obvious question is: *Can we use BSI arithmetic to answer preference queries?* From the previous sections of this chapter, it should be clear that yes, it is possible, given that a BSI exists for every attribute a user can specify a preference on. Preference queries are a particular case of arithmetic queries, except that instead of using a range or equality operator, a top_k operator, similar to the TM operator, is used:

$$top_k(\sum_{i=1}^{n} a_i \quad T.C_i)$$

The sum in this expression can be easily computed with at most n BSI multiplications by a constant and by $n \quad 1$ BSI additions[‡]. The preferences in PREFER can take on, usually, only 11 dierent values, but these values are between 0 and 1. Values between 0 and 10 with a discretization of 1 could be used as well; tuple scores will simply be multiplied by 10 and their relative ranks will not be aected. With 50,000 tuples (documents in TM terms), 4 BSIs to add, unrestricted preferences (unrestricted query-weights in TM terms), and top_{10} score tuples to retrieve, BSTM can execute the query in 0.009 seconds. With 500 top score tuples to retrieve, 0.023 seconds are needed by BSTM. Of course, these numbers cannot be directly compared to PREFER numbers since they used a dierent computer and did not specify the CPU speed of their machine, but they used a dual CPU computer and we used a single CPU computer. Also, BSIs in preference queries will be more dense than BSIs in Term Matching problem[§], which will slow down the BSI operations[¶]. But still, 0.009 second compared to 1 second, and 0.023 second compared to 19 seconds, are quite signicant. A more rigorous comparison on comparable systems is needed, but I am condent that BSTM will compete with PREFER. And this is only for 50,000 tuples. How can PREFER perform on 1 million documents? BSTM can answer the same queries, but on 1 million tuples, in respectively 0.167 and 0.185 second.

[‡]I write *at most* because some coecients could be 0, in which case no multiplication nor addition are necessary for this attribute, or 1, in which case no multiplication is necessary for this attribute.

[§]This is why I do not believe IRTM techniques applied to preference queries would perform well, since inverted lists will get very long.

[¶]More disk reads will be necessary since it will be harder to compress bitmaps eciently; more segments will be in verbatim bitmap form.

Also, building bit-sliced indexes will take much less than 210 and 2000 minutes (discretization does not aect bit-sliced index building time). Only a scan of the table, going tuple by tuple, inserting a new entry in a BSI for every indexed attribute, are necessary to build the indexes. The extra disk space used by the BSIs should be a lot less than for the materialized views. Many (between 10 and 100 in PREFER) materialized views can take a lot of space compared to one BSI per indexed attribute. Unfortunately, disk space utilization is not covered in the PREFER paper, so it is dicult to make a comparison. I believe using BSI arithmetic should scale much better than PREFER.

6.4 Nearest Neighbor Searches

Let's consider a POINTS table with three attributes: PID (point ID), X, and Y. To each PID is associated a pair of coordinates (x, y). Suppose we want to know the point that is closest to a point P_1 with coordinates (x_1, y_1). It is necessary to compute the distance between P_1 and every point $P(x, y)$ in POINTS, and then select a point with the minimum distance value. Suppose also that bit-sliced indexes exist on the X and Y columns of POINTS, and that the distance measure used is the Manhattan distance. The distance between P_1 and P is:

$$d_1(P_1, P) = |x_1 \quad x| + |y_1 \quad y|.$$

To answer this query with BSI arithmetic, we can think of P_1 as a query dened similarly as above, with weights equal to 1. Note that x_1 and y_1 are constants, and that BSI subtraction between two BSIs was dened in Section 3.2. BSI subtraction between a constant and a BSI is not much more dicult since it is really a subtraction between an all-x_1 or all-y_1 BSI (i.e. a BSI having only x_1 or y_1 for all its values) and a BSI. Computing the absolute value of a signed BSI[‖] S is not very complicated since we can take the sign bit-slice SBS of S, and for every other bit-slice B of S, do $B = B\ XOR\ \ SBS$. If $SBS[r] = 1$ for some row r, then $S[r]$ is negative and we need to negate it. The exclusive unions will do the job since $0\ XOR\ \ 1 = 1$ and $1\ XOR\ \ 1 = 0$. If $SBS[r] = 0$ for some row r, then $S[r]$ is positive and we need to leave its S-value untouched. The exclusive unions will again do the job since $0\ XOR\ \ 0 = 0$ and $1\ XOR\ \ 0 = 1$. The positive values of S remain unchanged. Then we need to add 1 to every negative values of the original S, so we can simply add SBS to S: $S = S + SBS$. We must of course also drop SBS from S. To select the smallest distance (or, more generally, the k smallest distances) in the distance BSI, it is a matter of modifying the top k algorithm of Figure 4.3 to return the smallest values instead of the largest values.

[‖]Obviously, if S is unsigned, nothing needs to be done.

Can the Euclidean distance be used instead of the Manhattan distance? Yes, but it would be more expensive. Computing the square of a BSI is feasible since the multiplication of two BSIs is, but it would be more expensive than computing the absolute value. Computing the square root of a BSI should be feasible also, but it is going to be expensive to do it. We may not need to compute the square root of a BSI when we need the Euclidean distance in our nearest neighbor searches. We can rst compute:

$$d'_2(P_1, P) = (x_1 \quad x)^2 + (y_1 \quad y)^2$$

and not take the square root immediately. We then nd the k smallest values of d'_2, and before returning the results, take the square root of those k values. This way, only k square root computations are necessary.

Would it be more or less expensive to use BSI arithmetic, with either the Manhattan distance or the Euclidean distance, than using other approaches? This is another question I will leave open, at least for now.

6.5 Other Possible Applications

If nearest neighbor computations using BSI arithmetic of the kind described above prove to be worthwhile, then the next natural question to ask is: *Can clustering algorithms benet from BSI arithmetic?* It seems likely. Much more work is needed before this question can be answered.

Another avenue to explore is the use of bit-sliced indexes in multimedia databases. For example, to index images, the rst step is choosing a suitable feature space: choosing relevant features with respect to the image database, choosing the descriptors of these features and choosing numerical representation of these descriptors (signatures). At the end of this stage, the image database is represented by a cloud of points in a high dimensional feature space [IME]. The second step is building an index to get ecient storage of image signatures. Depending on the feature space used, it could be possible to use bit-sliced indexes to store and query the database based on some features of the feature space.

APPENDIX A

Tools Used to Produce This Thesis

For typesetting, I used Emacs to type LATEX[1] input les. I used the program dvipdfm[2] to produce a PDF le from the DVI le produced by L ATEX. The box-and-arrow gures (e.g. Figure 2.6) were created with Xg [3]. The experimental results gures of Chapter 5 where produced with Grace [4]. To display DVI les I used kdvi[5], and for PDF les, I used both kghostview [5] and Acrobat Reader[6].

The computer I worked on is equipped with an AMD Athlon 1.33 Ghz CPU, with 256 MB of DDR RAM, a UDMA 100, 7200 rpm, 40 GB, Maxtor IDE drive, running FreeBSD[7] (the most recent version used is 4.5-PRERELEASE #9, obtained from the STABLE branch on December 31 2001; synchronization with the STABLE branch of the FreeBSD CVS repository has been made regularly throughout this work). The kernel is a custom built kernel. The window manager used is GNOME[8]. Implementation of the RIDBIT project has been made in part on the computer described above, and on a 333 MHz Sun Ultra-Sparc-IIi, 128MB of RAM, running Sun Solaris OS 5.7. The RIDBIT project has been implemented in C. A program used to successively call the benchmark program of the RIDBIT project to generate some data to produce performance results has been written in Python[9]. I also used the Gnumeric[10] spreadsheet program.

[1]http://www.latex-project.org/
[2]http://gaspra.kettering.edu/dvipdfm/
[3]http://www.xg.org/
[4]http://plasma-gate.weizmann.ac.il/Grace/
[5]http://www.kde.org/
[6]http://www.adobe.com/
[7]http://www.freebsd.org/
[8]http://www.gnome.org/
[9]http://www.python.org/
[10]http://www.gnome.org/projects/gnumeric/

APPENDIX B

Experimental Results Data

The following three gures show tables containing experimental results data used to produce the graphs of gures 5.2 to 5.11. The rst four columns of each gure contain data sorted by the number of documents in the database (data used to produce the query execution time versus the number of query terms graphs), and the other ve columns contain data sorted by the number of query terms (data used to produce the query execution time versus the number of documents graphs and the ratio of IRTM execution time and BSTM execution time versus the number of documents graph).

N	T_Q	$Time_{IR}$	$Time_{BSI}$	T_Q	N	$Time_{IR}$	$Time_{BSI}$	$Time_{IR}/Time_{BSI}$
50000	5	0.0087	0.0050	5	50000	0.0087	0.0050	1.7400
50000	10	0.0220	0.0113	5	100000	0.0180	0.0093	1.9355
50000	20	0.0423	0.0230	5	250000	0.0460	0.0233	1.9742
50000	30	0.0473	0.0353	5	500000	0.0917	0.0450	2.0378
50000	40	0.0663	0.0463	5	750000	0.1380	0.0643	2.1462
50000	50	0.0780	0.0603	5	1000000	0.1840	0.0853	2.1571
50000	100	0.1237	0.1280	10	50000	0.0220	0.0113	1.9469
50000	150	0.1600	0.1927	10	100000	0.0460	0.0200	2.3000
50000	200	0.1973	0.2710	10	250000	0.1167	0.0507	2.3018
100000	5	0.0180	0.0093	10	500000	0.2330	0.0967	2.4095
100000	10	0.0460	0.0200	10	750000	0.3490	0.1390	2.5108
100000	20	0.0853	0.0433	10	1000000	0.4593	0.1827	2.5140
100000	30	0.0960	0.0660	20	50000	0.0423	0.0230	1.8391
100000	40	0.1340	0.0857	20	100000	0.0853	0.0433	1.9700
100000	50	0.1563	0.1130	20	250000	0.2173	0.1097	1.9809
100000	100	0.2497	0.2420	20	500000	0.4330	0.2107	2.0551
100000	150	0.3233	0.3690	20	750000	0.6517	0.2947	2.2114
100000	200	0.4000	0.5210	20	1000000	0.8577	0.3893	2.2032
250000	5	0.0460	0.0233	30	50000	0.0473	0.0353	1.3399
250000	10	0.1167	0.0507	30	100000	0.0960	0.0660	1.4545
250000	20	0.2173	0.1097	30	250000	0.2440	0.1727	1.4129
250000	30	0.2440	0.1727	30	500000	0.4890	0.3283	1.4895
250000	40	0.3383	0.2260	30	750000	0.7317	0.4550	1.6081
250000	50	0.3947	0.2980	30	1000000	0.9660	0.6107	1.5818
250000	100	0.6290	0.6610	40	50000	0.0663	0.0463	1.4320
250000	150	0.8217	1.0467	40	100000	0.1340	0.0857	1.5636
250000	200	1.0063	1.4620	40	250000	0.3383	0.2260	1.4969
500000	5	0.0917	0.0450	40	500000	0.6770	0.4267	1.5866
500000	10	0.2330	0.0967	40	750000	1.0150	0.5933	1.7108
500000	20	0.4330	0.2107	40	1000000	1.3433	0.7937	1.6925
500000	30	0.4890	0.3283	50	50000	0.0780	0.0603	1.2935
500000	40	0.6770	0.4267	50	100000	0.1563	0.1130	1.3832
500000	50	0.7890	0.5567	50	250000	0.3947	0.2980	1.3245
500000	100	1.2633	1.2110	50	500000	0.7890	0.5567	1.4173
500000	150	1.6443	1.8983	50	750000	1.1843	0.7760	1.5262
500000	200	2.0153	2.6477	50	1000000	1.5673	1.0383	1.5095
750000	5	0.1380	0.0643	100	50000	0.1237	0.1280	0.9664
750000	10	0.3490	0.1390	100	100000	0.2497	0.2420	1.0318
750000	20	0.6517	0.2947	100	250000	0.6290	0.6610	0.9516
750000	30	0.7317	0.4550	100	500000	1.2633	1.2110	1.0432
750000	40	1.0150	0.5933	100	750000	1.8930	1.6847	1.1236
750000	50	1.1843	0.7760	100	1000000	2.5217	2.2493	1.1211
750000	100	1.8930	1.6847	150	50000	0.1600	0.1927	0.8303
750000	150	2.4663	2.6240	150	100000	0.3233	0.3690	0.8762
750000	200	3.0257	3.6387	150	250000	0.8217	1.0467	0.7850
1000000	5	0.1840	0.0853	150	500000	1.6443	1.8983	0.8662
1000000	10	0.4593	0.1827	150	750000	2.4663	2.6240	0.9399
1000000	20	0.8577	0.3893	150	1000000	3.2933	3.5040	0.9399
1000000	30	0.9660	0.6107	200	50000	0.1973	0.2710	0.7280
1000000	40	1.3433	0.7937	200	100000	0.4000	0.5210	0.7678
1000000	50	1.5673	1.0383	200	250000	1.0063	1.4620	0.6883
1000000	100	2.5217	2.2493	200	500000	2.0153	2.6477	0.7612
1000000	150	3.2933	3.5040	200	750000	3.0257	3.6387	0.8315
1000000	200	4.0433	4.8923	200	1000000	4.0433	4.8923	0.8265

Figure B.1: Data for Query-Weights Equal to 1

N	T_Q	$Time_{IR}$	$Time_{BSI}$	T_Q	N	$Time_{IR}$	$Time_{BSI}$	$Time_{IR}/Time_{BSI}$
50000	5	0.0087	0.0050	5	50000	0.0087	0.0050	1.7400
50000	10	0.0230	0.0117	5	100000	0.0177	0.0093	1.9032
50000	20	0.0430	0.0240	5	250000	0.0470	0.0247	1.9028
50000	30	0.0483	0.0370	5	500000	0.0937	0.0463	2.0238
50000	40	0.0677	0.0480	5	750000	0.1407	0.0653	2.1547
50000	50	0.0790	0.0610	5	1000000	0.1873	0.0920	2.0359
50000	100	0.1267	0.1243	10	50000	0.0230	0.0117	1.9658
50000	150	0.1640	0.1943	10	100000	0.0467	0.0220	2.1227
50000	200	0.2017	0.2600	10	250000	0.1187	0.0547	2.1700
100000	5	0.0177	0.0093	10	500000	0.2380	0.1010	2.3564
100000	10	0.0467	0.0220	10	750000	0.3567	0.1497	2.3828
100000	20	0.0873	0.0453	10	1000000	0.4683	0.2033	2.3035
100000	30	0.0977	0.0690	20	50000	0.0430	0.0240	1.7917
100000	40	0.1367	0.0897	20	100000	0.0873	0.0453	1.9272
100000	50	0.1597	0.1147	20	250000	0.2217	0.1173	1.8900
100000	100	0.2547	0.2363	20	500000	0.4423	0.2303	1.9205
100000	150	0.3303	0.3740	20	750000	0.6643	0.3230	2.0567
100000	200	0.4090	0.5007	20	1000000	0.8757	0.4333	2.0210
250000	5	0.0470	0.0247	30	50000	0.0483	0.0370	1.3054
250000	10	0.1187	0.0547	30	100000	0.0977	0.0690	1.4159
250000	20	0.2217	0.1173	30	250000	0.2497	0.1837	1.3593
250000	30	0.2497	0.1837	30	500000	0.4983	0.3553	1.4025
250000	40	0.3460	0.2370	30	750000	0.7470	0.4883	1.5298
250000	50	0.4030	0.2977	30	1000000	0.9863	0.6613	1.4915
250000	100	0.6427	0.6403	40	50000	0.0677	0.0480	1.4104
250000	150	0.8380	1.0577	40	100000	0.1367	0.0897	1.5240
250000	200	1.0287	1.4100	40	250000	0.3460	0.2370	1.4599
500000	5	0.0937	0.0463	40	500000	0.6923	0.4547	1.5225
500000	10	0.2380	0.1010	40	750000	1.0377	0.6417	1.6171
500000	20	0.4423	0.2303	40	1000000	1.3727	0.8517	1.6117
500000	30	0.4983	0.3553	50	50000	0.0790	0.0610	1.2951
500000	40	0.6923	0.4547	50	100000	0.1597	0.1147	1.3923
500000	50	0.8070	0.5677	50	250000	0.4030	0.2977	1.3537
500000	100	1.2910	1.1967	50	500000	0.8070	0.5677	1.4215
500000	150	1.6787	1.9380	50	750000	1.2103	0.7947	1.5230
500000	200	2.0577	2.5683	50	1000000	1.6050	1.0883	1.4748
750000	5	0.1407	0.0653	100	50000	0.1267	0.1243	1.0193
750000	10	0.3567	0.1497	100	100000	0.2547	0.2363	1.0779
750000	20	0.6643	0.3230	100	250000	0.6427	0.6403	1.0037
750000	30	0.7470	0.4883	100	500000	1.2910	1.1967	1.0788
750000	40	1.0377	0.6417	100	750000	1.9347	1.6653	1.1618
750000	50	1.2103	0.7947	100	1000000	2.5783	2.2390	1.1515
750000	100	1.9347	1.6653	150	50000	0.1640	0.1943	0.8441
750000	150	2.5200	2.6820	150	100000	0.3303	0.3740	0.8832
750000	200	3.0893	3.5207	150	250000	0.8380	1.0577	0.7923
1000000	5	0.1873	0.0920	150	500000	1.6787	1.9380	0.8662
1000000	10	0.4683	0.2033	150	750000	2.5200	2.6820	0.9396
1000000	20	0.8757	0.4333	150	1000000	3.3647	3.5947	0.9360
1000000	30	0.9863	0.6613	200	50000	0.2017	0.2600	0.7758
1000000	40	1.3727	0.8517	200	100000	0.4090	0.5007	0.8169
1000000	50	1.6050	1.0883	200	250000	1.0287	1.4100	0.7296
1000000	100	2.5783	2.2390	200	500000	2.0577	2.5683	0.8012
1000000	150	3.3647	3.5947	200	750000	3.0893	3.5207	0.8775
1000000	200	4.1437	4.7377	200	1000000	4.1437	4.7377	0.8746

Figure B.2: Data for Query-Weights Equal to Powers of 2

N	T_Q	$Time_{IR}$	$Time_{BSI}$	T_Q	N	$Time_{IR}$	$Time_{BSI}$	$Time_{IR}/Time_{BSI}$
50000	5	0.0090	0.0133	5	50000	0.0090	0.0133	0.6767
50000	10	0.0237	0.0270	5	100000	0.0190	0.0260	0.7308
50000	20	0.0443	0.0690	5	250000	0.0493	0.0653	0.7550
50000	30	0.0497	0.1090	5	500000	0.0983	0.1320	0.7447
50000	40	0.0697	0.1483	5	750000	0.1473	0.1907	0.7724
50000	50	0.0810	0.1860	5	1000000	0.1967	0.2570	0.7654
50000	100	0.1290	0.3787	10	50000	0.0237	0.0270	0.8778
50000	150	0.1663	0.5893	10	100000	0.0483	0.0510	0.9471
50000	200	0.2043	0.7880	10	250000	0.1227	0.1333	0.9205
100000	5	0.0190	0.0260	10	500000	0.2457	0.2577	0.9534
100000	10	0.0483	0.0510	10	750000	0.3677	0.3700	0.9938
100000	20	0.0897	0.1317	10	1000000	0.4820	0.4920	0.9797
100000	30	0.1003	0.2107	20	50000	0.0443	0.0690	0.6420
100000	40	0.1397	0.2877	20	100000	0.0897	0.1317	0.6811
100000	50	0.1627	0.3613	20	250000	0.2270	0.3420	0.6637
100000	100	0.2597	0.7377	20	500000	0.4537	0.6660	0.6812
100000	150	0.3360	1.1530	20	750000	0.6800	0.9580	0.7098
100000	200	0.4147	1.5447	20	1000000	0.8947	1.2663	0.7065
250000	5	0.0493	0.0653	30	50000	0.0497	0.1090	0.4560
250000	10	0.1227	0.1333	30	100000	0.1003	0.2107	0.4760
250000	20	0.2270	0.3420	30	250000	0.2550	0.5503	0.4634
250000	30	0.2550	0.5503	30	500000	0.5103	1.0570	0.4828
250000	40	0.3607	0.7477	30	750000	0.7637	1.5067	0.5069
250000	50	0.4113	0.9500	30	1000000	1.0140	2.0200	0.5020
250000	100	0.6560	1.9960	40	50000	0.0697	0.1483	0.4700
250000	150	0.8553	3.2033	40	100000	0.1397	0.2877	0.4856
250000	200	1.0460	4.2633	40	250000	0.3607	0.7477	0.4824
500000	5	0.0983	0.1320	40	500000	0.7077	1.4253	0.4965
500000	10	0.2457	0.2577	40	750000	1.0597	2.0340	0.5210
500000	20	0.4537	0.6660	40	1000000	1.4000	2.7307	0.5127
500000	30	0.5103	1.0570	50	50000	0.0810	0.1860	0.4355
500000	40	0.7077	1.4253	50	100000	0.1627	0.3613	0.4503
500000	50	0.8233	1.8183	50	250000	0.4113	0.9500	0.4329
500000	100	1.3150	3.7453	50	500000	0.8233	1.8183	0.4528
500000	150	1.7097	5.9527	50	750000	1.2337	2.6010	0.4743
500000	200	2.0947	7.9077	50	1000000	1.6507	3.4683	0.4759
750000	5	0.1473	0.1907	100	50000	0.1290	0.3787	0.3406
750000	10	0.3677	0.3700	100	100000	0.2597	0.7377	0.3520
750000	20	0.6800	0.9580	100	250000	0.6560	1.9960	0.3287
750000	30	0.7637	1.5067	100	500000	1.3150	3.7453	0.3511
750000	40	1.0597	2.0340	100	750000	1.9700	5.3003	0.3717
750000	50	1.2337	2.6010	100	1000000	2.6250	7.1137	0.3690
750000	100	1.9700	5.3003	150	50000	0.1663	0.5893	0.2822
750000	150	2.5653	8.3723	150	100000	0.3360	1.1530	0.2914
750000	200	3.7777	11.0900	150	250000	0.8553	3.2033	0.2670
1000000	5	0.1967	0.2570	150	500000	1.7097	5.9527	0.2872
1000000	10	0.4820	0.4920	150	750000	2.5653	8.3723	0.3064
1000000	20	0.8947	1.2663	150	1000000	5.1707	11.1950	0.4619
1000000	30	1.0140	2.0200	200	50000	0.2043	0.7880	0.2593
1000000	40	1.4000	2.7307	200	100000	0.4147	1.5447	0.2685
1000000	50	1.6507	3.4683	200	250000	1.0460	4.2633	0.2453
1000000	100	2.6250	7.1137	200	500000	2.0947	7.9077	0.2649
1000000	150	5.1707	11.1950	200	750000	3.7777	11.0900	0.3406
1000000	200	6.6437	14.8770	200	1000000	6.6437	14.8770	0.4466

Figure B.3: Data for Unrestricted Query-Weights

References

[AW00] R. Agrawal and E. Wimmers. “A Framework for Expressing and Combining Preferences.” *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 2000.

[CI98] Chee-Yong Chan and Yannis E. Ioannidis. “Bitmap index design and evaluation.” In Laura Haas and Ashutosh Tiwary, editors, *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data: June 1–4, 1998, Seattle, Washington, USA*, volume 27(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pp. 355–366, New York, NY 10036, USA, 1998. ACM Press.

[CI99] Chee-Yong Chan and Yannis E. Ioannidis. “An ecient bitmap encoding scheme for selection queries.” In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data: SIGMOD ’99, Philadelphia, PA, USA, June 1–3, 1999*, volume 28(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pp. 215–226, New York, NY 10036, USA, 1999. ACM Press.

[DM92] Ioan Dancea and Pierre Marchand. *Architecture des ordinateurs*, chapter 7, section 5. Gaetan Morin Editeur, 1992.

[HKP01] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. “PREFER: A System for the Ecient Execution of Multi-parametric Ranked Queries.” In *SIGMOD Conference*, 2001.

[IBM] IBM DB2. “SQL Reference. Full SELECT syntax in IBM DB2 SQL.” http://www.csa.ru/dblab/DB2/db2s0/fullslt.htm.

[IME] IMEDIA. “Research topics at IMEDIA, INRIA Rocquencourt.” http://www-rocq.inria.fr/imedia/English/research.html.

[Joh99] Theodore Johnson. “Performance Measurements of Compressed Bitmap Indices.” In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *Proceedings of the Twenty-fth International Conference on Very Large Databases, Edinburgh, Scotland, UK, 7–10 September, 1999*, pp. 278–289, Los Altos, CA 94022, USA, 1999. Morgan Kaufmann Publishers.

[Kir01] Steve Kirsch. “Internet and Beyond: Database Challenges.” SIGMOD talk. Powerpoint presentation, 2001. http://www.skirsch.com /presentations/sigmod.ppt.

[KZS99] Marcin Kaszkiel, Justin Zobel, and Ron Sacks-Davis. “Ecient passage ranking for document databases.” *ACM Transactions on Information Systems*, **17**(4):406–439, October 1999.

[MZ96] Alistair Moat and Justin Zobel. “Self-Indexing Inverted Files for Fast Text Retrieval.” *ACM Transactions on Information Systems*, **14**(4):349–379, October 1996.

[ON87] P. E. O’Neil. “Model 204 Architecture and Performance.” In *High Performance Transaction Systems, 2nd Int’l. Workshop, Lecture Notes in CS 359*, p. 40. Springer-Verlag, September 1987.

[OO00a] Patrick O’Neil and Elizabeth O’Neil. *Database: Principles, Programming, and Performance.* Morgan Kaufmann/Academic Press, 2000.

[OO00b] Patrick O’Neil and Elizabeth O’Neil. *Database: Principles, Programming, and Performance*, chapter 8, section 6. Morgan Kaufmann/Academic Press, 2000.

[OO00c] Patrick O’Neil and Elizabeth O’Neil. *Database: Principles, Programming, and Performance*, chapter 3, section 6. Morgan Kaufmann/Academic Press, 2000.

[OQ97] Patrick O’Neil and Dallan Quass. “Improved query performance with variant indexes.” In Joan M. Peckman, editor, *Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD 1997: May 13–15, 1997, Tucson, Arizona, USA*, volume 26(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pp. 38–49, New York, NY 10036, USA, 1997. ACM Press.

[PW83] Shirley A. Perry and Peter Willet. “A review of the use of inverted les for best match searching in information retrieval system.” *Journal of Information Science*, **6**:59–66, 1983.

[ROO01] Denis Rinfret, Patrick E. O’Neil, and Elizabeth J. O’Neil. “Bit-Sliced Index Arithmetic.” In *Proceedings, ACM SIGMOD International Conference on Management of Data: SIGMOD*, 2001.

[Wan] Ruye Wang. “Fast Multiplication – Booth’s Algorithm.” http:// mulan.eng.hmc.edu/ rwang/e85/lectures/arithmetic/node9.html.

[WB98] Ming-Chuan Wu and Alejandro P. Buchmann. "Encoded Bitmap Indexing for Data Warehouses." In *International Conference on Data Engeneering 1998*, pp. 220–230, 1998.

[WMB99a] Ian H. Witten, Alistair Moat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition.* Morgan Kaufmann/Academic Press, 1999.

[WMB99b] Ian H. Witten, Alistair Moat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*, chapter 4, section 6, page 206. Morgan Kaufmann/Academic Press, 1999.

[Wu99] Ming-Chuan Wu. "Query optimization for selections using bitmaps." In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data: SIGMOD '99, Philadelphia, PA, USA, June 1–3, 1999*, volume 28(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pp. 227–238, New York, NY 10036, USA, 1999. ACM Press.

[Yah] Yahoo! "Advanced Search Syntax." http://search.yahoo.com /search/syntax?

[ZM98] Zobel and Moat. "Exploring the Similarity Space." *IRFORUM: SIGIR Forum (ACM Special Interest Group on Information Retrieval)*, **32**, 1998.