Bit-Sliced Index Arithmetic

Denis Rinfret
UMass/Boston
Dept. of CS, UMass/Boston
Boston, MA 02125-3393
819-376-3691
drinf@cs.umb.edu

Patrick O'Neil
UMass/Boston & Microsoft Research
Dept. of CS, UMass/Boston
Boston, MA 02125-3393
617-354-6460
poneil@cs.umb.edu

Elizabeth O'Neil
UMass/Boston & Microsoft Research
Dept. of CS, UMass/Boston
Boston, MA 02125-3393
617-354-6460
eoneil@cs.umb.edu

Research for this paper was supported by NSF Grant IRI 97-11374 at UMass/Boston. Microsoft Research, where Patrick O'Neil and Elizabeth O'Neil spent their Sabbatical Year, also provided support.

ABSTRACT

The bit-sliced index (*BSI*) was originally defined in [ONQ97]. 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 efficiently generate new BSI's Z, V, and W, such that Z = X + Y, V = X - Y, and W = 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 Z will be x + y, the value in V will be x - y and the value in W 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 (min) (see [OO00, DB2SQL] for definitions of these clauses). Another contribution of the current paper is to generalize BSI range restrictions from [ONQ97] 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 efficient algorithm to find k documents that share the largest number of terms with some query list Q of terms. A great deal of published work on such problems exists in the Information Retrieval (*IR*) field. The algorithm we introduce, which we call Bit-Sliced Term-Matching, or BSTM, uses an approach comparable in performance to the most efficient 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.

1. INTRODUCTION

The bit-sliced index (*BSI*) was originally defined in [ONQ97], where it was demonstrated how to use a BSI representing column quantities to evaluate SQL aggregate queries (specifically, SUM queries), and to impose range restrictions in a SQL WHERE clause. In the current work, we introduce the concept of BSI arithmetic: addition, subtraction, and min, and show how such BSI operations provide a natural way to determine results of SQL clauses UNION ALL, EXCEPT ALL, and INTERSECT ALL (see [OO000, DB2SQL]), where row sets resulting from subqueries can be combined into multisets (also called bags) of rows: that is, sets with duplicates permitted. For example, Query (1.1) below conforms to ANSI Standard SQL-99 and executes in Microsoft SQL Server to provide a multiset result:

```
(1.1)  SELECT COUNT(*) CT, PRID FROM
         ( SELECT PRID FROM T WHERE Col_1 = const_1
               UNION ALL
           SELECT PRID FROM T WHERE COL_2 = const_2
               UNION ALL
               . . .
           SELECT PRID FROM T WHERE COL_M = const_M)
               AS NEW_T
               GROUP BY PRID;
```

Query (1.1) retrieves the various COUNT(*) multiplicities with corresponding primary key identifiers PRID, from the UNION ALL of the Equal Match predicates in the FROM Clause of the outer Select. The GROUP BY PRID, would normally select only one row in each group of T, but in this case it selects the appropriate multiplicities of individual rows arising from the UNION ALL. No current database product keeps track of these multiplicities using BSI addition, but we will show that BSI addition is extremely efficient for this purpose.

We can also construct examples of queries where multiplicities are subtracted, using EXCEPT ALL, and the minimum multiplicity of two multisets is determined, using INTERSECT ALL. Note that in the case of EXCEPT ALL and INTERSECT ALL, any negative numbers in the result BSI must be replaced with zeros, since rows do not appear with negative multiplicities in SQL.

The current paper also generalizes BSI range restrictions to a non-Boolean form: instead of finding all rows in a table T with a BSI value greater than some constant C (however many rows that might be), we show how to efficiently determine the top k BSI-valued rows, 1 <= k <= |T|, where |T| is the cardinality of T. We require this capability along with BSI addition for our term matching algorithm, BSTM, which we now explain.

Given an object-relational table T of rows representing documents, with a collection type column K representing keyword terms, we model two documents as being *close* if they contain a large number of common terms. A (Maximum) Term Matching algorithm (*TM algorithm*) under this metric finds the k nearest documents from a given document in a collection, i.e., the k documents with the largest number of matching terms. If a query is modeled as a collection of terms, a TM search finds the k documents with the largest number of terms matching the query. Much published work on TM search

ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA

algorithms exists in the Information Retrieval (IR) field (also called the Text Retrieval field). The new algorithm we introduce, called the *Bit-Sliced Term-Matching* or *BSTM* algorithm, uses basically the same approach as the most efficient IR algorithm, the Perry-Willet Term Matching algorithm, *PWTM* [PW83], and has the advantage that it depends only on what we propose as native operations for a DBMS. Some TM algorithms in IR use more complex distance metrics than ours (see [KZS99]), for example by weighting term matches higher for terms that are relatively infrequent. In our concluding section we explain how BSTM can be generalized to more complex metrics. Searching documents for terms in this way is of great interest, of course, as evidenced by Database Vendor products such as the Oracle Cartridge named ConText, and the DB2 Extender product named Text Extender [BYTE97, PCW97].

To illustrate how SQL can provide a statement to retrieve the top k documents in terms of a count of valid equal match restrictions, we modify Query (1.1) to another form, Query (1.2), valid in Microsoft SQL Server, but not in SQL-99:

```
(1.2)  SELECT TOP 10 COUNT(*) CT, PRID FROM
       ( SELECT PRID FROM T WHERE COL_1 = const_1
          UNION ALL
         SELECT PRID FROM T WHERE COL_2 = const_2
          UNION ALL
          . . .
         SELECT PRID FROM T WHERE COL_M = const_M)
            AS NEW_T
            GROUP BY PRID
            ORDER BY CT DESC;
```

Note that the final clause, "ORDER BY CT DESC," guarantees that the top 10 documents with maximum counts of matches on the set of restrictions will be retrieved.

We note in passing that classic SQL queries with Boolean syntax were unable to find rows with the largest number of matching terms from a given set; Gerard Salton [SALT89] pointed out this shortcoming in SQL some years ago. The problem is that there is no Boolean condition that can deal with a count of matches on subsets of conditions. If a query Q provides a list of M keyword terms, and we wish to find k rows with the largest number of matches using classical SQL, we would need to perform a query to look for rows with matches on all M keywords, then multiple queries to look for rows with matches on any subset of M-1 of the keywords, and so on down to subsets of cardinality M-j, with j possibly ranging up to a large fraction F of M, before we find k rows with the maximum number of keyword matches. The number of distinct queries on subsets that would be required is thus:

$$\sum_{j=1}^{F} \binom{M}{M-j}$$

For F = M/2, the number of distinct queries required is about 2^{M-1}. Of course with the addition of the newer UNION ALL and TOP k clauses, Query (1.2) provides the appropriate syntax to perform maximal term matching. The TOP k clause is not in the SQL-99 Foundation Document, but it is implemented in a number of database products.

The plan of the remainder of this paper is as follows. In Section 2 we present previously published fundamental concepts of bitmaps and bit-sliced indexes. In Section 3 we present our new algorithms for bit-sliced arithmetic and show how these apply to SQL clauses UNION ALL, EXCEPT ALL, and INTERSECT ALL. Section 4 introduces our new BSMT algorithm to retrieve k rows with the largest set of matching terms to a list of terms given by a query Q. In Section 5, we explain the optimal PWTM algorithm used in IR, and discuss aspects of comparative performance to our BSTM algorithm. Section 6 provides experimental results comparing the two algorithms, both of which we have implemented in prototype. Finally, Section 7 presents our conclusions and suggestions for future work, including a description of how our BSTM algorithm can be generalized to handle weighted terms.

2. FUNDAMENTAL CONCEPTS

We review a number of previously published concepts below, before presenting new material.

Bitmap Index Definition. [ON87, ONQ97] To create a bitmap index, all N rows of the underlying table T must be assigned ordinal numbers: 1, 2, . . . , 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, . . ., 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. ♦

Variations on this bitmap index definition were studied in [CHI98, CHI99, WUB98, WU99]. Ordinal row-positions 1, . . . , N can be assigned to table pages in fixed size blocks of size J , 1 through J on the first page, J+1 through 2J on the second page, etc., where J is the maximum number of rows of T that will fit 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-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 verbatim bitmaps are more CPU efficient than on compressed forms). In the architecture implemented for this paper, which we call the BITSLICE architecture, bitmap compression simply involves converting sparse bitmap pages into ordered lists of Segment-Relative Ordinal positions called ORDRIDs (defined below). We first describe Segmentation, which was introduced in [ON87, ONQ97] and is used in the BITSLICE architecture.

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 fit on a single disk page. These blocks of rows are called *Segments*, following the MODEL 204 nomenclature of [ON87]. Our BITSLICE

architecture uses 4KByte disk pages, so Segments contain S = 8*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.)

A B-tree index entry for an index value X_i in the BITSLICE architecture has the format shown in Figure 1. (The Seginfo layout is defined below in Figure 2.)

X_i	Seginfo	Seginfo	. . .	Seginfo

Figure 1. Bitmap Index Entry in BITSLICE

The entry in Figure 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 in Figure 1, is shown in Figure 2 to contain a Segment number (*Seg_no*) for the Segment of rows it represents, and the disk position pointer (*DKPTR*) to the *Segment-Relative ORDRID-list* or Bitmap. See the next paragraph for a description of an ORDRID-list. The Seginfo blocks for an index entry are held in order by Seg_no, and if a Segment contains no row for index value X_i, then the Seginfo block for that Segment will be missing in Figure 1. (This fact can be used at an early execution point in a query involving conjunctions to exclude Segments from consideration that have no Seginfo block in one of the index entries.)

Seg_no	DKPTR

Figure 2. Seginfo Layout

ORDRIDs and ORDRID-lists. Since the S bits of a Segment bitmap must fit on a 4 KByte page, $S < 2^{15}$, and a Segment-relative ORDRID will fit in two bytes (in what follows we will refer to a Segment-Relative ORDRID simply as an *ORDRID*). This short length provides a significant 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 * S + k. An ORDRID-list for a Segment of an index entry (pointed to by DKPTR in Figure 2) contains ORDRIDs in ordered sequence. ORDRID lists are also stored in order on disk, and usually many ORDRID-lists will fit 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: $X_i \parallel$ Seg_no. The DKPTR used to address ORDRID-lists has the same format used in row addressing, consisting of (Disk Page #, Slot #), where Slot # addresses an offset directory entry that locates the ORDRID-list on the page.

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 differentiate between bitmap and ORDRID-list when the difference is important to our discussion.

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 verbatim bitmaps B1 and B2, we can create the bitmap B3 = B1 AND B2 by treating memory-resident Segment fragments of these bitmaps as arrays of long ints in C, and looping through the fragments, setting B3[I] = B1[I] & B2[I]. The logic can stream through successive Segment fragments from disk (for B1 and B2) and to disk (B3), until the operation is complete. The bitmap B3 = B1 OR B2 is computed in the same way, and B3 = NOT B1 is computed by setting B3[I] = ~B1[I] & EBM[I] in the loop. Note that the efficiency of bitmap operations arises from a type of parallelism in Boolean operations in CPU registers, specifically SIMD (Single-Instruction-Multiple-Data), where many bits (32, or 64 in some machines) are dealt with in a single AND, OR, or NOT operation occurring in the simplest possible loop. To find the number of rows represented in a bitmap B1, COUNT(B1), 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 another array containing the number of 1 bits in each short int, aggregating these into a count variable.

We perform logical operations AND and OR on two Segment ORDRID-lists B1 and B2 by looping through the two lists in order to perform a merge-intersect or merge-union into an ORDRID-list B3; in the case of OR, the resulting ORDRID-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 B1 is performed by copying the EBM Segment and turning off bits in the list corresponding to ORDRIDs found in B1. To perform AND and OR with a verbatim bitmap B1 in one index Segment and an ORDRID-list B2 in another, the ORDRID-list is assumed to have fewer elements and efficiently drives the loop to access individual bits in the bitmap and perform the Boolean test, in the case of AND, filling in a new ORDRID-list B3, and in the case of OR, initializing the verbatim bitmap B3 to B1 and turning on bits from the ORDRID-list B2.

Bit-Sliced Index Definition. A bit-sliced index B [OQ97], often referred to as a *BSI*, is an ordered list of bitmaps (either verbatim bitmaps or ORDRID-lists), $B^S, B^{S-1}, \ldots, B^1, B^0$; the list of bitmaps is used to represent values (normally non-negative integers) of some column C (although the column C might be a calculated value associated with rows of T, and have no physical existence). The bitmaps $B^S, B^{S-1}, \ldots, B^1, B^0$ are called the *bit-slices*, and provide binary representations of C values for all the rows: B^0 holds the 1's bits, B^1 holds the 2's bits, B^2 holds the 4's bits, etc. More precisely, if we represent the C value of row j (Ordinal position j) by C[j], and the bit for row j in bit-slice B^i by $B^i[j]$, then the values for $B^i[j]$ are chosen so that $C[j] = \sum_{i=0}^{S} B^i[j] \bullet 2^i$. Note that we determine S in advance so that the highest-order bit-slice B^S is non-empty, i.e., it contains an ORDRID for at least one row in some segment. ♦

In [OQ97], a bit-sliced index was also defined to contain a Bit-map B^{nn} representing the set of rows with non-null values in

column C, and a Bitmap B^n representing the set of rows with null values (the redundancy, B^{nn} = NOT(B^n) AND EBM, simply provided extra efficiency). We will not be using these bitmaps in BSI's of the current paper, however, since we will only be dealing with calculated values for rows in T (that is, for all rows included in the EBM for T) with no null values.

3. BIT-SLICED INDEX ARITHMETIC

In this section we demonstrate how we can perform arithmetic on bit-sliced indexes, using SIMD operations on each of the bit-slices. Of course such techniques have been used for years in multi-bit computer operations; see [MANO95].

Consider Figure 3, where each of the bitmaps B_1, B_2, and B_3 represent the found sets of three subqueries that are then combined with UNION ALL clauses in a SQL Query. How are we to calculate and then represent the multiset of rows that results? If we could somehow ADD the bitmaps in the first three rows to generate the SUM of the bottom row, we would solve this problem.

```
B1    00100001000011000010 0...
B2    01001001001001000101...
B3    00001010000101000010 0...
SUM   01102012001113001020 1...
```

Figure 3. A Conceptual Addition of Bitmaps

Of course the SUM on the bottom row of Figure 3 cannot be represented as a bitmap, since it has values other than 0 and 1. It *can*, however, be represented as a bit-sliced index! We need to ask ourselves how we might be able to add three bitmaps to arrive at the BSI SUM of Figure 3. A few observations set this problem in perspective. First, it is easy to add two bitmaps to arrive at a BSI sum. Second, a bitmap is just a BSI with a single bit-slice. This leads us to ask if we can find an efficient algorithm to add any two BSI's, and indeed this turns out to be a simple matter.

First, consider adding the two bitmaps B_1 and B_2 of Figure 3 to arrive at a BSI named BS. Clearly BS must have two bit-slices, BS^0 and BS^1, since we need to represent values 0, 1, and 2. We point out that BS can be generated quite simply with two Boolean operations: $BS^1 = B_1$ AND B_2; $BS^0 = B_1$ XOR B_2. This calculation, along with a row representing the SUM of B_1 and B_2 for comparison, is illustrated in Figure 4.

```
B1    00100001000011000010 0...
B2    01001001001001000101...
SUM   01101002001012001010 1...
BS1   00000001000001000000 0...
BS0   01101000001010001010 1...
```

Figure 4. Two Bitmaps Added to form BS

We note in Figure 4 that interpreting the two BS bit-slices gives the values represented in SUM. The reason is perfectly clear. The operation $BS^1 = B_1$ AND B_2 sets BS^1 to 1 in precisely those bit positions of B_1 and B_2 where both contain 1, and thus where SUM = 2. $BS^0 = B_1$ XOR B_2 sets BS^0 to 1 in precisely those bit positions of B_1 and B_2 where one or the other contains 1, but not both, and thus where SUM = 1. All other positions in BS are 0. To generalize from adding bitmaps to adding BSI's, we merely need to interpret the idea of "carrying" in bit-valued addition to the SIMD situation of Boolean bitmap operations. We illustrate this in Algorithm 3.1.

A "Carry" bit-slice C can arise in Algorithm 3.1 whenever two or three bit-slices are added to form S^i, and a non-zero C must then be added into the next bit-slice S^{i+1}. Note that if C is zero (no bits on), Boolean operations give the expected results, but a flag for zero C can speed up 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.

Negative Numbers in a BSI. We provide an algorithm for subtracting one BSI from another, but first we discuss how to represent negative numbers in a BSI by two's complement

Algorithm 3.1 Addition of BSI's. Given two BSI's, $A = A^S A^{S-1} \ldots A^1 A^0$ and $B = B^P B^{P-1} \ldots B^1 B^0$, 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 $S^{MAX(S, P)+1}$, so that a carry from the highest bit-slice in A or B will have a place.

```
S0 = A0 XOR B0                          -- bit on in S0 iff exactly one bit on in A0 or B0
C = A0 AND B0                           -- C is "Carry" bit-slice; bit on iff bits on in A0 and B0
for (i = 1; i <= MIN(S, P); i++) {      -- While there are further bit-slices in both A and B
 Si = (Ai XOR Bi XOR C)                 -- one bit on (or three bits on) gives bit on in Si
 C = (Ai AND Bi) OR (Ai AND C) OR (C AND Bi)  -- two (or more) bits on gives bit on in C
}
if (S > P)                              -- if A has more bit-slices than B
 for (i = P+1; i <= S; i++) {           -- continue loop until last bit-slice
  Si = (Ai XOR C)                       -- one bit on gives bit on in Si ; note C might be zero!
  C = (Ai AND C)                        -- two bits on gives bit on in C; zero if prior C was zero!
}
else                                    -- P >= S and B has at least as many bit-slices as A
 for (i = S+1; i <= P; i++) {           -- continue loop until last bit-slice
  Si = (Bi XOR C)                       -- one bit on gives bit on in Si; note that C might be zero!
  C = (Bi 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 Bit-slices end
SMAX(S, P)+1 = C                        -- Put Carry into final bit-slice of S, SMAX(S, P)+1 ♦
```

arithmetic. If we're working with a collection of BSI's containing only non-negative numbers (the only kind we've discussed up to now), and the largest positive number that can be represented is 7, then only three bit-slices will be required to represent the largest binary number: 111. However, negative numbers can still arise during subtraction and must be distinguished in two's complement arithmetic by a leading 1 compared to a leading 0 for any positive number. This means that four bit-slices are needed to represent the binary number: 0111. The bit representation for -7 is determined by flipping the bits of 7 (1000) and adding 1: thus -7 is 1001. Now to perform the subtraction -7 -(+7)," we will require yet another high-level bit-slice (5 bit-slices, in all); so we get: -7 -(+7)" = 11001 - 00111 (note we have sign-extended both quantities as we added new high-order bits); then we flip the bits of the right-hand term and add 1 to get 11001 + 11001, then add the two negative numbers by Algorithm 3.1 to get: 10010, or -14.

Whenever two different BSI's are to be subtracted, any BSI representing only positive numbers must have a high-order bit-slice of all zeros adjoined. (This bit-slice will afterward be used in sign-extension.). Then a high-order bit-slice must be adjoined to the BSI with the maximum number of bit-slices to handle overflow during subtraction. Following this, the BSI with the minimum number of bit-slices must be brought up to the maximum number by adjoining new high-order bit-slices. All BSI's must be sign-extended as new high-order bit-slices are adjoined: this is done by copying the most significant bit-slice in adjoining new high-order bit-slices.

We provide Algorithm 3.2 to subtract one BSI from another. Algorithm 3.3 shows how to find the MIN of two BSI's.

We explained in the introduction how algorithms to add bit-sliced indexes can be used in current native SQL to determine row multiplicities arising from UNION ALL clauses. Similarly, we can create queries where multiplicities are subtracted, using EXCEPT ALL, and the minimum multiplicity of two multisets are determined, using INTERSECT ALL. Note that in EXCEPT ALL, any negative numbers in the result BSI D will be replaced with zeros, since rows do not appear with negative multiplicities in SQL; the algorithm to do this will simply find all rows with high-order bits on in D, and mask this found set out of all bit-slices. Similarly, INTERSECT ALL will not need to be concerned with rows that have negative multiplicity.

Algorithm 3.2 Subtraction of BSI's. Given two BSI's, $A = A^S A^{S-1} \ldots A^1 A^0$ and $B = B^P B^{P-1} \ldots B^1 B^0$, we will create a new difference BSI D = A - B, by taking the two s complement of B and adding it to A using Algorithm 3.1. We adjoin bit-slices as specified in the paragraph above, and for simplicity we assume that A, B, and D end up with highest-order bit-slices all the same: MAX(S, P)+2.

```
Add needed bit-slices to A and B,          -- for 2's complement subtraction . . .
   sign-extending A and B if necessary     -- . . . allow for MAX(S, P)+2 bit-slices in D
for (i = 0; i <= MAX(S, P); i++) {         -- loop through all existing bit-slices of B
   B^i = NOT(B^i) AND EBM                  -- one's complement of bit-slice B^i
}                                          -- one's complement complete
D = A + (B + (all 1's bitmap))             -- use Algorithm 3.1;  B + all 1's bitmap is 2's complement  ♦
```

Algorithm 3.3 Min of BSI's. Given two BSI's, $A = A^S A^{S-1} \ldots A^1 A^0$ and $B = B^P B^{P-1} \ldots B^1 B^0$, create new "min"BSI M = MIN(A, B). The following pseudo-code handles only non-negative values. To handle both positive and negative numbers, we would sign extend A or B with any needed high-order bit-slices, and start by differentiating negative and positive values in the highest bit-slice. Then we'd use the pseudo-code below to find MIN(A, B) for the bitmap set of non-negative values, and analogous pseudo-code to find MAX(A, B) for the bitmap set of negative numbers. Considering for now only the special case of non-negative values, the highest-order slice of M will be $M^{MIN(S, P)}$, since the minimum of two numbers x and w represented in row r of the BSI's A and B cannot have more binary digits than MIN(S, P). We assume in the loop below that S >= P (if not we reverse A and B).

```
K = empty set                              -- bitmap K of rows for which we know min
K_A =  empty set                           -- bitmap of rows for which A has lesser value
K_B =  empty set                           -- bitmap of rows for which B has lesser value
for (i = S; i > P; i--)                    -- recall that S >= 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
  X = (A^i XOR B^i) AND NOT(K)             -- rows that differ for the first time in A^i and B^i
  K_B = K_B OR (A^i AND X)                 -- if A^i has 1-bit, new min must be in B
  K_A = K_A OR (B^i AND X)                 -- else B^i has 1-bit & new min must be in A
  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 row 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
  M^i = M^i OR B^i AND K_B                 -- B^i values for rows with (disjoint) bits in K_B
}                                                                                          ♦
```

4. THE BSTM ALGORITHM

We now introduce the BSTM (Bit-Sliced Term-Matching) Algorithm. We are given a query Q with a list of keyword values, Q = <keyword-1, keyword-2, . . ., keyword-|Q|>, which are expected to appear in a multi-valued keyword column K of an object-relational table T. We wish to find the set of k rows that have the largest number of matching keywords with the query list Q. Denote the bitmap representing rows of table T that contains keyword-i in its column K by Bi; these bitmaps will occur as terms of an index KX. It is our task to find the Ordinal positions which have the largest number of matching 1's among all bitmaps B1, B2, . . ., Bm. We use Algorithm 3.1 to ADD these bitmaps resulting in a BSI SUM. All that remains is to find Ordinal positions where the BSI SUM has maximum values. We recall that an algorithm was provided in Section 4 of [ONQ97] by which the set of rows in T with C >= c1, C a value column having a BSI, can be found quite efficiently. In Algorithm 4.1 below we provide a variation of this algorithm to find k rows that have the maximum C values in T.

Finding the rows with the k largest values in a BSI. Given a BSI, $S = S^P S^{P-1} \ldots S^1 S^0$ over a table T and a positive integer k <= |T|, we wish to find the bitmap F (for "found set") of rows r with the k largest S-values, S(r), in T. Algorithm 4.1 accomplishes this in a rather subtle way, explained in the proof of the algorithm, below.

Proof of Algorithm 4.1. We wish to find F, the bitmap of rows with the k largest S-values in T. Denote by m the minimum S-value of any row that lies in F, and assume m has binary representation: $m_P m_{P-1} \ldots m_1 m_0$, 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 4.1 with successively smaller values i.

Variables used in Algorithm 4.1 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 efficiency, and could be dropped from the code.

We wish to demonstrate the defining properties of G (G contains rows r with S(r) Greater than m) and E (E contains rows r with S(r) Equal to m in a specific initial sequence of bits), so we provide an induction hypothesis specifying contents of G_i and E_i, which we define 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-1, and conclude from this the final contents of F.

Induction Hypothesis. Assume for an arbitrary row r in T that the binary representation of S(r) is $r_P r_{P-1} \ldots r_1 r_0$. Our induction hypothesis defines 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 differ in its early bit representation $r_P r_{P-1} \ldots r_{i+1}$ from $m_P m_{P-1} \ldots m_{i+1}$. (2) A row r in T will be in G_i if and only if the early bit representation $r_P r_{P-1} \ldots r_{i+1}$ is greater than $m_P m_{P-1} \ldots m_{i+1}$; this is equivalent to saying that for some bit position j in the range i+1 <= j <= P, bit r_j is on with bit m_j off, and bits $r_P r_{P-1} \ldots r_{j+1}$ are all equal to bits $m_P m_{P-1} \ldots m_{j+1}$.

We now perform induction. The initial test of Algorithm 4.1 guarantees that k <= 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 first 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 <= j <= P to use in the defining 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 differ from bits in m, as required in defining 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 $r_P r_{P-1} \ldots r_{i+1}$ equal to $m_P m_{P-1} \ldots m_{i+1}$. 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 <= j <= P such that bit r'_j is on with bit m_j off, and bits $r'_P r'_{P-1} \ldots r'_{j+1}$ are all equal to bits $m_P m_{P-1} \ldots m_{j+1}$. ($G_i$ can contain no rows until a zero bit shows up in $m_P m_{P-1} \ldots m_{i+1}$.) 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'

Algorithm 4.1. Find k rows with largest values in a BSI.

```
if (k > COUNT(EBM) or k < 0)          -- test if parameter k is valid
   Error ("k is invalid")             -- if not, exit; otherwise, kth largest S-value exists
G = empty set; E = EBM;               -- G starts with no rows; E with all rows
for (i = P; i >= 0; i--) {            -- i is a descending loop index for bit-slice number
   X = G OR (E AND S^i)               -- X is trial set: G OR {rows in E with 1-bit in position i}
   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;                          -- done looping
   }
}                                     -- 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 off n bits from E in F};     -- throw out some ties to return exactly k rows  ♦
```

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'-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 4.1, 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-1} and bit r_i on where bit r'_i is off. Finally, any row r not in G_i or in E_i, since its S-value representation $r_P r_{P-1} \ldots r_{i+1}$ cannot be greater than or equal to $m_P m_{P-1} \ldots m_{i+1}$, must have some bit position r_j off that is on in m_j, i+1 <= j <= P, with $r'_P r'_{P-1} \ldots r'_{j+1}$ all equal to bits $m_P m_{P-1} \ldots m_{j+1}$, 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 $n = \text{COUNT}(X_i) > k$, this will imply that m_i is on, since there were less than k rows in G_i (m is the kth 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 kth 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-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 definition to add new rows to G with bit m_j off and bit r_j on.

If $n = \text{COUNT}(X_i) < k$, we see that X_i, the set of n rows with the largest S-values in T, does not include the kth largest. But if bit m_i were on, that would not be true, since by construction E_i contains all rows r with $r_P r_{P-1} \ldots r_{i+1}$ equal to bits $m_P m_{P-1} \ldots m_{i+1}$, and (E AND S^i) would thus include m. Since bit m_i is off, 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 $r_P r_{P-1} \ldots r_{i+1}$ all equal to bits $m_P m_{P-1} \ldots m_{i+1}$; in other words we set $G_{i-1} = X_i$ (= G_i OR (E_i AND S^i)). This new set G_{i-1} satisfies induction hypothesis (2) with j = i. Next we set $E_{i-1} = E_i$ AND (NOT S^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 $r_P r_{P-1} \ldots r_{i+1}$ equal to $m_P m_{P-1} \ldots m_{i+1}$, it is clear that E_{i-1} satisfies induction hypothesis (1) for i - 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_i), and we will find that COUNT(F) - k = 0. In this case, we don't need to continue the loop until i = -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 = -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 find the desired set F. ♦

Putting Algorithms 3.1 and 4.1 together, we have:

Algorithm 4.2. The BSTM algorithm. We are given an Object-Relational table T with a multi-valued keyword column K having a Bitmap index KX. Given a query list Q of keyword values from K and the task of finding the k rows with the largest number of keyword values in Q, we proceed as follows. Find all the bitmaps for Q's keyword values in KX, and add those bitmaps together using Algorithm 3.1, to create a single bit-sliced index KS. Then apply Algorithm 4.1 to find the set of k rows with the k largest values in the KS index. ♦

5. THE PWTM ALGORITHM

Fionn Murtagh published a term-matching algorithm in [MURT82] that was cited as best in [SALT89], but in [MURT99], Murtagh cites the Perry-Willet Algorithm [PW83] as an improvement on earlier term-matching algorithms used in IR, including his own. The algorithm in [PW83] is straightforward, and we modify its description slightly to use more modern nomenclature from [MZ96, KZS99]. In the Perry-Willet algorithm, an index I is understood to exist on the keyword terms of all documents. For each keyword term t in I, there is a sequence of document identifiers: $(d_1, \ldots, d_n)$. Often these document identifiers have associated weights for the term in the referenced document. (In our nomenclature, the documents are rows in an object-relational table, the index I is an index on the set-oriented keyword column K, and the document identifiers are ORDRIDs. We will ignore weights for now, assuming below that each term match counts as 1.) The Perry-Willet algorithm uses an *Accumulator* variable A_d to accrue weighted matches for each term value found in the document d. A good deal of discussion appears in [MZ96, KZS99] on how Accumulators are to be assigned, whether dynamically as new documents have their first term match, or existing in advance as an array. We assume the pre-declared array form used in Algorithm 5.1.[1]

Algorithm 5.1. The Perry-Willet Algorithm. Q is a list of terms to match, I is the index, the array A represents the Accumulators: A[d] is the accumulator for document d, with a total of N documents.

```
int A[N]     -- Array with N cells
set all N cells of A[ ] to zero
for each term t in Q {
   find in I the list of doc ID's (d1,...,dn)
   for each d in (d1, d2,...,dn)  {
      A[d] = A[d] + 1
   }
}
Find k largest A[d] values (using
  heapsort), and return values of d      ♦
```

[1] [KZS99] studies an approach where only the early, heavily weighted, Accumulators are dynamically materialized, only enough for about 2% of the documents. There is some small time savings, and small loss in Recall (about 15%).

It should be clear that Algorithm 5.1 is close to being optimal for the given task. All needed index information is read from disk exactly once and accumulated efficiently. If the number of terms in Q is |Q|, then assuming that the index I is normally not cached in memory but disk resident, we will need to perform at least |Q| disk reads (some of them possibly long ones) to retrieve all document lists of the various terms. It is possible that more than one I/O per term document list will be required, depending on the size of the list, and this mitigates in favor of long disk reads; the value of multi-page disk reads is understood in the IR field, and we will assume this consideration gives no advantage to database system access. Data compression is also discussed at length in [KZS99], and we assume that data compression gives no advantage either to database system access or to IR.

As successive index lists $(d_1, \ldots, d_n)$ are accessed, the augmentation of the Accumulators A[d] becomes an important performance consideration. Is it possible to keep the set of accumulators so densely packed that memory cache hits affect performance? Or at the opposite end of the spectrum, will there be so many accumulators that there is difficulty holding all of them in memory at once, so that some accumulators might need to reside on disk part of the time? We obviously need to know something about size assumptions for document collections of this kind to evaluate these questions.

In [MZ96] the document collection database used for performance tests was the TREC database for large text collections from [HARM92]. The articles in this text collection vary from around 100 bytes to 2MB, and the authors broke the larger documents into *pages* of around 1000 bytes to help user comprehension. This resulted in 1,743,848 records totaling 2054.5 MB, an average of 191.4 words per record, and 538,244 distinct keyword terms after translating to lowercase and removing variant endings, a process called *stemming*. Note that all document words other than a short list of *stop words* are indexed as keyword terms in IR. The index I comprised 195,935,531 stored pairs of doc ID and weight (in the general case) appearing in term lists. This means that I was over 1GB in size and was not memory-resident, especially for the low-powered machines considered typical by [MZ96]. To give a second example, in [KZS99] about 530,000 documents from TREC-5 [VH96] were used in testing, and these documents were broken into smaller documents of 50-500 bytes each to give a collection of 7.7 million small documents.

From these two examples, we see that CPU cache hits will not be an important performance consideration in accesses to Accumulators in Algorithm 5.1. On the other hand, it is reasonable to assume that the array of Accumulators will fit in memory for most document collections. In [KZS99], the hardware used for experiments was a Sparc20 with 385MB of memory, which easily contained the Accumulator array for the 7.7 million small documents. The point was also made that memory was sufficient to materialize each term-list of document ID's in full. Since Segmentation is not used, this can be an important simplifying factor to avoid special-case code to perform pipelining through memory.

5.1 Comparison to BSTM Algorithm

We rewrite our BSTM Algorithm 4.2 in the form of Algorithm 5.1 for better comparison.

Algorithm 5.2, BSTM Algorithm (restating 4.2). As in the Perry-Willet Algorithm, Q is a list of terms to match, I is the index (of terms appearing in column K), and the BSI A represents the Accumulators for the documents.

```
initialize BSI A
for all Segments 1 through M {
  for each term t in Q {
     find ORDRID-list B or Bitmap B in I
     Add B into A (Algorithm 3.1)
  }
}
Find k largest values in A (Algorithm  4.1) ♦
```

Given a query Q with |Q| terms in its list, we can calculate the maximum number of bit-slice bitmaps, bcount, that are needed in the BSI A, specifically: bcount = CEIL($\log_2$ |Q|). We initialize a BSI by creating structures known as "Segment Anchors" for each of the bitmaps that might be accessed. The Segment Anchors look like the index entries of Figure 1, except that the index value X_i is not needed and the Seginfo structures of Figure 2 are not created in advance.

The outer loop on Segments 1 through M in Algorithm 5.2 is standard, and the loop on all terms within a Segment deals with all bitmap or ORDRID-list additions implied by the query Q before the outer loop passes on to the next Segment.

As outlined in Algorithm 3.1, each addition of B into A will consist of an XOR operation into bit-slice A_0 and a sequence of AND operations to generate Carries that will then be XOR'ed into upper-level bit-slices A_i. Whether the operations involve ORDRID-lists or bitmaps is material only for performance considerations. Operating on two ORDRID-lists may gain from memory cache hits over the PWTM Algorithm 5.1, since the representations are relatively compact. ORDRID-list vs. bitmap operations are in fact identical to the type of operation used in Algorithm 5.1, a lookup in an array, except that individual bits are acted on through AND's and XOR's, and Carries might result. Bitmap vs. bitmap operations, on the other hand, are likely to be more efficient than corresponding steps in Algorithm 5.1, at least on a per bit-slice basis, since the SIMD efficiency of ANDing and XORing multiple bits at once is an advantage. On the other hand, the possibility of Carries from these operations mitigates against the efficiency of Algorithm 5.2 in comparison to Algorithm 5.1, where all carries are handled in one CPU operation. As the number of terms in Q rises and the probability of Carries to higher bit-slices increases in Algorithm 5.2 for later terms, we would expect the efficiency of Algorithm 5.2 to drop off compared to Algorithm 5.1. Thus our discussion seems to show that Algorithm 5.2 will probably have an advantage in performance over Algorithm 5.1 for a small number of terms in Q, but be at a disadvantage for a large number of terms in Q. The final step of this restatement of Algorithm 5.2 is to find the k largest A value rows, a relatively quick task in BSTM using Algorithm 4.1 and in PWTM using a heapsort to extract k terms from the final Accumulator array.

In the next section, we present our experimental results comparing Algorithms 5.1 and 5.2. One other consideration is worth mentioning, however. Algorithm 5.1 is a special-purpose program that has been implemented by IR practitioners in prototype and is available to interested parties [BMWZ95, WMB94]. However, such a program cannot be

expected to perform well as an application on a database, given that massive numbers of SQL statements must be used to retrieve RIDs, a rather heavyweight call interface that detracts from efficiency. Database storage of documents is of great interest, however, with Oracle Cartridges such as ConText and DB2 Extenders such as Text Extender [BYTE97, PCW97]. Algorithm 5.2, which is based on bit-sliced Index operations useful for a large number of query types other than text retrieval, would be natural to implement in native database form. Because of this, we claim that our BSTM algorithm has particular interest to database practitioners.

6. Experimental Findings

To test the performance of the Perry-Willet Algorithm 5.1 against the BSTM Algorithm 5.2 (i.e., 4.2), we created synthetic benchmark tables and queries for our BITSLICE architecture implementation and ran experiments to measure performance under varying conditions. The experiments were performed on a 333 Mhz Sun Ultra-Sparc-IIi, with 128M of memory, running on Sun Solaris OS 5.7.

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 system configuration. In Table 1, we provide a list of notational symbols used in our experiments, along with the values or range of values these symbols represent.

Table 1. Notation Used

Notational symbol	Values used
N (# rows = # docs)	50,000, 100,000, 200,000, 300,000
T (# terms)	10,000
T_D (# terms/doc)	40
T_Q (# terms/Query)	5, 10, 20, 30, 40
D_Q (avg. # docs/Q_term) (approximately linear in N)	0.01*N = 500, 1000, 2000, 3000

Focusing for the moment on the minimal configuration of Table 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 $N*T_D$ = 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 we have 400, 800 for N = 200,000, etc. We generated the terms in each document at random, using a Zipfian 70-30 distribution skew (a realistic assumption), and then created queries whose terms tended to use the more popular terms, behavior we modeled after [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 Zipfian 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 Table 1, and we ran experiments with all pairs of values. We randomly generated query runs with T_Q = 5, 10, 20, 30, and 40 terms, and ran them against implementations of Algorithms 5.1 and 5.2 on the BITSLICE architecture for tables of N = 50,000, 100,000, 200,000 and 300,000 rows. Three runs of each case were performed, with CPU and Elapsed times averaged across the ten queries for each run. An attempt was made to flush UNIX buffers prior to a run, but the result was not perfect so the one requiring the longest elapsed time out of three was dropped in each case as an outlier. We gave the Perry-Willet Algorithm 5.2 the same access to our term index that the BSTM Algorithm 5.1 had, thus providing the advantage of bitmap compression for extremely popular terms. We graph the CPU and Elapsed time for all cases in Figures 5 and 6 below.

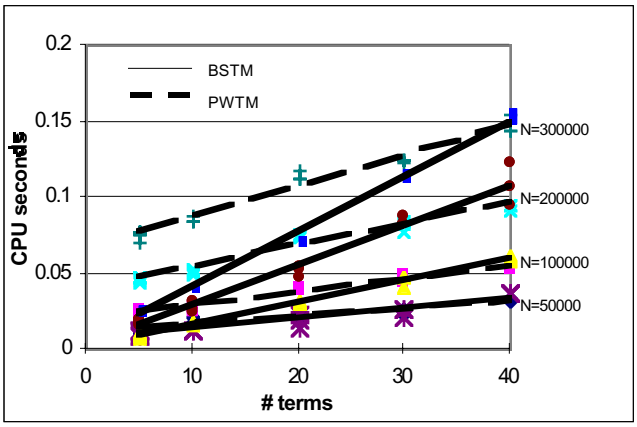

Figure 5. CPU Times Per Query for BSTM (Solid Lines) and PWTM (Dashed Lines) Algorithms

Note that PWTM and BSTM have very similar Elapsed time because they are performing the same I/Os. CPU time is more distinguishing, and tends to support the discussion in Section 5, since BSTM has a slight advantage over PWTM for a smaller numbers of terms and is at a disadvantage for a larger numbers of terms.

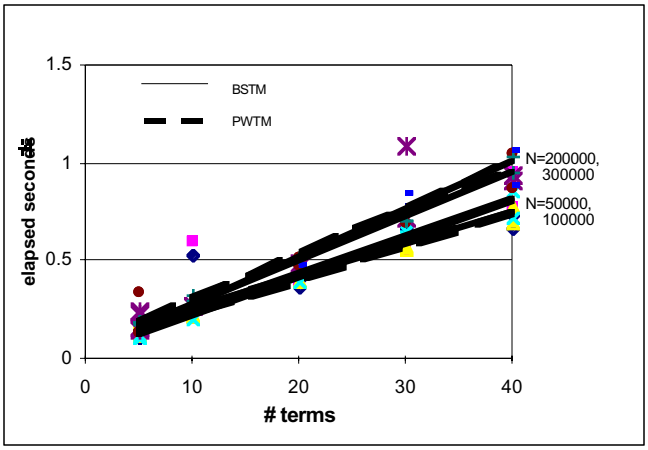

Figure 6. Elapsed Times Per Query for BSTM (Solid Lines) and PWTM (Dashed lines) Algorithms

We take two lessons from these results.

- The BSTM algorithm is comparable in performance with the best IR Term-Matching algorithm PWTM.

- CPU Performance of the BSTM algorithm degrades as the number of Query Terms increases, which we take to be an artifact of the increasing number of Carries required as more terms are added. However, BSTM performance for 30 terms is still comparable to PWTM, and probably quite a bit better than existing Database methods.

7. CONCLUSIONS AND FUTURE WORK

In the current paper we have shown, for any BSI's X and Y on a table T, how to efficiently generate new BSI's Z, V, and W, such that Z = X + Y, V = X - Y, and W = MIN(X, Y). We claim that BSI arithmetic such as this 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 (min). Another contribution of this paper has been to demonstrate how 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 has allowed us to solve a common problem of text retrieval: an efficient algorithm to find k documents that share the largest number of terms with some query list Q of terms.

7.1 Future Work

One avenue of future work is suggested by the fact that our algorithms for BSI arithmetic can be extended by automatically pipelining intermediate results in calculating expressions on multiple BSI's. For example, when we add the three bitmaps from Figure 3, B1 + B2 + B3, instead of creating S_1 = B1 + B2, then writing S_1 out to disk, and later reading it in again to calculate S = S_1 + B3, we can maintain S_1 in memory to be consumed by immediate addition to B3. Pipelining was implemented in our BITSLICE architecture for the simple special case of multiple index term bitmap addition used in Term Matching, but a more general solution would be valuable. It is also worth noting that BSI arithmetic is easily parallelized. Multi-Segment bitmaps can have their Segments partitioned out to be dealt with by different process threads. MODEL 204 [ON87] has long provided pipelining and parallelism of this kind.

Concurrency control to support bitmap indexing and BSI arithmetic presents special problems unless the update frequency is reasonably limited. MODEL 204 has provided a form of concurrency control based on locking for a number of years. But a valuable future task would be to provide a new form of multi-version concurrency (see the Snapshot Isolation discussion in Section 4.2 of [BBG+93]). A particularly challenging avenue which seems feasible would be to implement Snapshot Isolation so as to run queries as efficiently as possible, trading efficient queries for less efficient update transactions when necessary. Up to now, implementers have taken the opposite tack [JAC95].

We now describe how our BSTM approach could be extended in the future to deal with weighted term matching.

7.2 Weighted Term Matching

The IR field has an extremely large number of approaches to evaluating document queries. In [ZM98] there are eight different formulations listed (in Table 1) of similarity measures between a query and a document (e.g., two forms of "inner product", the "cosine measure", two forms of "probabilistic measure", etc.). Each of these similarity measures depends on how weights are assigned to matching terms, and in Table 2, nine different weight functions are listed, including "binary match", "logarithmic", "hyperbolic", two "normalized", and four involving noise and entropy. The simple non-weighted approach we've been dealing with up to now uses the simpler form of "inner product" similarity measure formulation, with the "binary match" weight function.

To illustrate how BSI arithmetic can deal with complex weighted similarity matches, we consider the "Cosine Measure" of similarity between a document d and a query q that was used exclusively in [KZS99]. We define the simplest terms first, and build up to the full similarity function.

- $f_{x,t}$ The number of occurrences (frequency) of the term t in x; x might be either a document or a query.
- $w_{d,t} = \log_e(f_{d,t}+1)$ Weight of term t in document d. Note that if t doesn't appear in d then the weight is zero; the more times t appears in d, the more highly the document is weighted for queries seeking this term.
- $w_{q,t} = \log_e(f_{q,t}+1) \bullet \log_e(N/f_t+1)$ Weight of term t in query q. As before, a higher "frequency" of the term in the query increases weight. Note that f_t is a count of documents that contain the term t, and N is the total number of documents, so rare terms appearing in a query are more highly weighted.

The Cosine measure of similarity of a query q and a document d is symbolized by $C(q,d)$. To evaluate a query, we will need to calculate this measure for a specific q and all documents d, then choose the k documents with the largest measures. The formula for $C(q,d)$ is:

(7.1) $$C(q,d) = \frac{\sum_{t\in q}\left(w_{q,t} \bullet w_{d,t}\right)}{\sqrt{\sum_{t\in d} w_{d,t}^2}}$$ (Note that $\sqrt{\sum_{t\in d} w_{d,t}^2}$ is represented below by W_d for short.)

While formula (7.1) might seem complex, the approach to calculating it using a BSI is relatively straightforward. First, we note that [MZ96] says we can use low-precision document weights of about six bits without significantly affecting retrieval effectiveness. Prior to knowing what terms exist in the query q, we can precalculate $f_{d,t}$, the frequency of each term in *each* document d, then calculate $w_{d,t} = \log_e(f_{d,t}+1)$ and $W_d = \sqrt{\sum_{t\in d} w_{d,t}^2}$. Finally, we can calculate the document weights $w_{d,t}/W_d$, and represent these values as 6-bit integers associated with each term and each document. We construct for each term t a BSI B_t to represent these document weight values $w_{d,t}/W_d$ for all documents in T; thus the BSI B_t is associated with each term in our Index I (as currently a single bitmap is associated with each term). To calculate the BSI $C_{q,d}$ representing $C(q,d)$ by formula (7.1), we begin by deriving the values of $w_{q,t}$ for each term t of the query. From the query q we know $f_{q,t}$, the frequency of each query term, so we can calculate $w_{q,t} = \log_e(f_{q,t}+1)$, and from this we construct 6

bit integer approximations. Then $C_{q,d}$ is derived by calculating $\sum_{t \in q} w_{q,t} \bullet B_t$. This is a sum of BSI's arrived at by multiplying a list of known BSI's B_t from the index by small binary integers $w_{q,t}$. The method of multiplying is simply a matter of left-shifting for 1-bit positions in $w_{q,t}$ and then adding the shifted BSI's B_t.

Most of the calculations above must be performed for any algorithm that solves the given problem; calculations to create the BSI's B_t are used in creating the index for any approach, and in any event do not occur at runtime. The calculation of $\sum_{t \in q} w_{q,t} \bullet B_t$ is the only one that is peculiar to our approach, and while the additions involved are rather time consuming, so is the multiplication needed for the algorithm that accumulates product terms into an array. We leave implementation and performance tests of this algorithm for future work.

8. REFERENCES

[BBG+95] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In Proceedings of the SIGMOD International Conference on Management of Data, May 1995.

[BMWZ95] T. C. Bell, A. Moffat, I. H. Witten, and J. Zobel. The MG Retrieval System: Compressing for Space and Speed. CACM 38, 4 (Apr. 1995), 41-42.

[BYTE97] Byte Magazine, Ann O'Leary. Managing Mission-Critical Text [with ORACLE ConText Cartridge]. http://www.byte.com/art/9709/sec4/art1.htm

[CHI98] Chee Yong Chan, Yannis E. Ioannidis: Bitmap Index Design and Evaluation. SIGMOD Conference 1998: 355-366

[CHI99] Chee Yong Chan, Yannis E. Ioannidis: An Efficient Bitmap Encoding Scheme for Selection Queries. SIGMOD Conference 1999: 215-226

[DB2SQL] Full Select syntax in DB2 SQL Reference Text, http://www.csa.ru/dblab/DB2/db2s0/fullslt.htm

[HARM92] D. K. Harmon, Ed. Proceedings of TREC for Text Retrieval Conference (Washington D.C., Nov. 1992) National Institute of Standards Special Publication 500-207. NIST, Washington, D.C.

[JAC95] K. Jacobs, with contributors: R. Bamford, G. Doherty, K. Haas, M. Holt, F. Putzolu, B. Quigley. Concurrency Control: Transaction Isolation and Serializability in SQL92 and Oracle7. Oracle White Paper, Part No. A33745, July, 1995.

[KZS99] Marcin Kaszkiel, Justin Zobel, and Ron Sacks-Davis. Efficient Passage Ranking for Document Databases. ACM Transactions on Information Systems, Vol. 17, No. 4, October 1999, Pages 406-439.

[MANO95] M. Morris Mano. Digital Design, 2nd Edition, Prentice-Hall, Englewood Cliffs, H.J., 1995.

[MURT82] Fionn Murtagh. A Very Fast, Exact Nearest Neighbour Algorithm for use in Information Retrieval. Information Technology: Research and Development 1982, Vol. 1, Pages 275-283.

[MURT99] Fionn Murtagh. Clustering in Massive Data Sets. Handbook of Massive Data Sets, Kluwer, 2000, J. Abello, P.M. Pardalos and M.G.C. Reisende, Eds. Preprint, August 22, 1999 available from http://www.cs.qub.ac.uk/~F.Murtagh/recent-papers.html.

[MZ96] Alistair Moffat and Justin Zobel. Self-Indexing Inverted Files for Fast Text Retrieval. ACM Trans. on Info. Sys., Vol. 14, No. 4, October 1996, Pages 349-379.

[ON87] Patrick O'Neil. MODEL 204 Architecture and Performance. HPTS Workshop, September 1987, Springer-Verlag Lecture Notes in Computer Science 359.

[ONQ97] Patrick O'Neil and Dallan Quass. Improved Query Performance With Variant Indexes.; Proc. ACM SIGMOD Conf. 1997, Pages 38-49.

[OO00] Patrick O'Neil and Elizabeth O'Neil. Database: Principles, Programming, and Performance, Section 3.6. Morgan Kaufmann/Academic Press 2000.

[PCW97] PCWEEK ONLINE, Timothy Dyck. ConText Gets faster and friendlier. [Also discusses DB2 Text Extender.] http://www8.zdnet.com/eweek/reviews/0414/14cont.html

[PW83] Shirley A. Perry and Peter Willet. A Review of the use of Inverted Files for Best Match Searching in Information Retrieval Systems. J. of Information Science 6 (1083) 59-66.

[SALT89] Gerard Salton. Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer. Addison-Wesley Publishing, Reading, MA 1989.

[VH96] E. Voorhees and D. Harmon. Overview of the Fifth Text REtrieval Conference (TREC-5). Proceedings of the 5th Text Retrieval Conference, Nov. 1996, NIST, http://trec.nist.gov/pubs.html

[WMB94] I. H. Witten, A. Moffat, and T. C. Bell. Managing Gigabytes: Compressing and Indexing Documents and Images. Van Nostrand Reinhold Co., New York, NY, 1994.

[WUB98] Ming-Chuan Wu, Alejandro P. Buchmann: Encoded Bitmap Indexing for Data Warehouses. ICDE 1998: 220-230

[WU99] Ming-Chuan Wu: Query Optimization for Selections Using Bitmaps. SIGMOD Conference 1999: 227-238

[ZM98] Justin Zobel and Alistair Moffat. Exploring the Similarity Space. SIGIR Forum, Spring 1998.