Bucket Skip Merge Join: A Scalable Algorithm for Join Processing in Very Large Databases using Indexes

Mohan Kamath and Krithi Ramamritham

CMPSCI Technical Report 96-20

February 1996

Bucket Skip Merge Join: A Scalable Algorithm for Join Processing in Very Large Databases using Indexes †

Mohan Kamath and *Krithi Ramamritham*
Computer Science Technical Report 96-20
Department of Computer Science
University of Massachusetts
Amherst MA 01003
{*kamath,krithi*}*@cs.umass.edu*

Abstract

Join processing algorithms play a critical role in efficient query processing. Popular join processing algorithms like merge join and hash join typically access all the data items in the datasets to be joined. In this paper we present a new join algorithm called *bucket skip merge join* which does not always access all the data items of the join sets. The basic idea is to divide the datasets into buckets that contain monotonically increasing values and maintain high and low values for each bucket. During join processing, the algorithm uses these values to skip a whole bucket or parts of a bucket when it can be determined that a match will not be found for the rest of the data items in the bucket. This considerably reduces the number of memory accesses, disk accesses and CPU time. Results of the performance tests on a prototype system indicates that our algorithm outperforms other popular join algorithms. Specifically, they scale better with the size of the database and the degree of join. Hence our algorithm will be very useful for applications like information retrieval, data warehousing and decision support systems that access very large databases.

Keywords: Join Processing, Merge Join, Query Optimization, Performance, Information Retrieval, Digital Libraries, Decision Support Systems, Data Warehousing

† Supported by NSF grant IRI-9314376 and a grant from Sun Microsystems Labs

Contents

1 Introduction

Joining related data items between two or more sets or relations is a basic operation that is supported by all database systems. Several join processing techniques have been proposed and investigated. As databases grow in size, *scalability* of join processing schemes is very essential. Current database trends indicate that databases of the future will be very large and used extensively for applications like decision support systems and digital libraries. In such applications, the read to write ratio of data items is very high. Hence indexes are to be maintained to speed up accesses. Also a recent article focusing on join facilities offered by commercial products for data warehousing and decision support systems applications [BS96], emphasizes the cost efficiency of join algorithms and motivates the need for exploring new join schemes for emerging applications. These factors have prompted us to explore new join techniques that can scale and perform better than techniques in vogue.

Early query processing schemes were based either on nested loop join or merge join [BE76]. Both these schemes are expensive since nested loop join performs a lot of disk I/Os and merge join requires sorting of data prior to the join. Hence hash join was proposed as a better alternative and has since been enhanced to improve performance [Bra84, DKO+84, NKT88, KNT89, Sha86]. Query processing schemes in commercial products currently use both merge join or hash join. Note that if the data is stored in a sorted order or if an index (like B-tree) exists on the datasets, then data need not be sorted prior to merge join. Similarly hash indices can be maintained for efficiently performing hash joins. In any case, both the merge join scheme and the hash join scheme typically access all the data items of the datasets involved (indexes or the records themselves) while performing the join. If intelligent schemes can be designed that can skip data items when it is known that they will not produce a match, join processing performance can be considerably enhanced. Such schemes are very essential for emerging applications that require the join of huge datasets.

In this paper we present a new join processing scheme called *bucket skip merge join* (BSMJ) that attempts to skip data items whenever it knows a match will not be produced with those data items. The basic idea is to divide the datasets into buckets that contain monotonically increasing values and maintain high and low values for each bucket. During join processing, the algorithm uses these values to skip all or parts of a bucket when it can be determined that a match will not be found for the rest of the data items in the buckets. This considerably reduces the number of memory accesses, disk accesses and CPU time. Buckets can be created and managed easily if an index like B+ tree exists. The join fields need not be numeric. Our scheme works differently from the partition band join scheme proposed in [DNS91a] and is intended for general purpose joins rather than band joins. To handle datasets that do not have indexes, we have also developed extensions to the BSMJ algorithm. We illustrate the use of the BSMJ

algorithm for join processing in relational databases and query processing in text databases.

To evaluate and compare the performance of the BSMJ algorithm with hash join and merge join algorithm, we have conducted a variety of performance tests on a prototype system. Results of the performance tests on a prototype system indicates that our algorithm outperforms these popular algorithms. Specifically, they scale better with the size of the database and the degree of join. Hence the BSMJ algorithm will be very useful for applications like information retrieval, digital libraries, data warehousing and decision support systems that typically read data items from very large databases.

During the last decade, join processing research has primarily focused on schemes that try to determine the best way to manage memory and schedule page fetches such that the number of I/Os is minimized. In contrast, our work focuses at a more fundamental level to reduce the number of data items to be considered for join by using ordering information in the datasets. Recently, [HR96] has emphasized the need to consider disk seek time, data transfer time and CPU time rather than merely considering the number of I/Os for determining the cost of join processing. Hence accessing all the data items can create a lot of unnecessary overheads. This clearly shows that the benefits that accrue from our scheme can be enormous since all these costs are reduced due to fewer comparisons of data items and fewer page fetches. We believe approaches like ours that exploit knowledge about data items are needed to get better scalability in join processing.

The contributions of the paper are as follows:

- development of a new high performance scalable join algorithm,
- analysis of the best and worst case performance of the algorithm, and
- a detailed performance evaluation to compare the algorithm with previous schemes like merge join and hash join.

The rest of the paper is organized as follows: Section 2 discusses related work and motivates the problem in more detail. The BSMJ algorithm is presented in section 3. Techniques for creating and manipulating buckets are described in section 4. We perform best and worst case analyses of the BSMJ algorithm in section 5. The use of the BSMJ algorithm for join processing in relational database systems and query processing in text database systems is illustrated in section 6. Section 7 discussed the nature of performance tests and section 8 analyzes the results of the tests. Section 9 concludes the paper with a summary.

2 Related Work and Motivation

To motivate the need for new join algorithms, we now trace through previous work in join processing.

One of the first studies on join processing algorithms [BE76] recommended merge join as *the most* optimal scheme for large databases. In merge join, the join inputs are first sorted on the join attribute and tuples that match are determined efficiently by simultaneously scanning both inputs. Since this scheme loads pages from both the inputs sequentially as necessary, it does not require much memory and is also independent of the input sizes. However sort is an expensive operation.

Hash indices were already being used for quick access to data [SWKH76, FNPS79]. This sparked interest in hash join schemes as an alternative to merge join schemes [Bra84, DKO+84]. The basic principle is to build an in-memory hash table for the smaller of the join inputs and probe this table for items in the large input. To overcome the memory size limitation while handling large inputs (table size greater than the size of the memory available), the hash join scheme was enhanced into the partition or hybrid hash join schemes [DKO+84, NKT88, KNT89, Sha86] where both inputs are first partitioned into disjoint subsets, and pairs of subsets, one from each relation, are matched using the basic hash join scheme.

With the advent of parallel processing, there has been growing interest in exploiting parallelism for sort [Men86, LY89, STG+90, DNS91b] and hash [DG85, FKT86, SD90] based join schemes. Comparisons of various parallel join schemes has also been performed [SD89]. Just like all other join schemes, parallelism can be exploited for the schemes we are proposing in this paper to achieve better performance. We do not discuss parallelism issues in the rest of the paper since it is orthogonal to the main focus of our work.

There has also been a lot of interest in improving buffer management and page fetching schemes for join processing [MKY81, Chr83, GP89, Omi89]. The schemes typically try to avoid duplicating page fetches when the pages that are to be brought into memory for performing the join is known. To improve the efficiency of joins, use of special index structures called join indices [Val87] has been studied. The elements of a join index are tuples that store the surrogate (tuple ID) pairs of the tuples which match on the join attribute. However when the database is appended frequently, join indexes incur a lot of overheads. The performance of join indices have also been compared with other schemes in [BM90]. This study concludes that join indices are good when the join selectivity is low and updates/appends are not frequent. Join schemes for relational databases have also been adapted for object-oriented database systems [SC90, DLM93]. Several issues related to join processing are surveyed in [ME92]. The issue of which one is the better among merge join and hash join was debated a lot until a recent comprehensive study concluded that there exist dualities between the two schemes [Gra94, GLS94]. Commercial database products currently support both the merge join and hash join schemes [CHH+91].

Thus it can be observed that there has been no work to improve join performance by maintaining additional information and skipping data items during join processing.

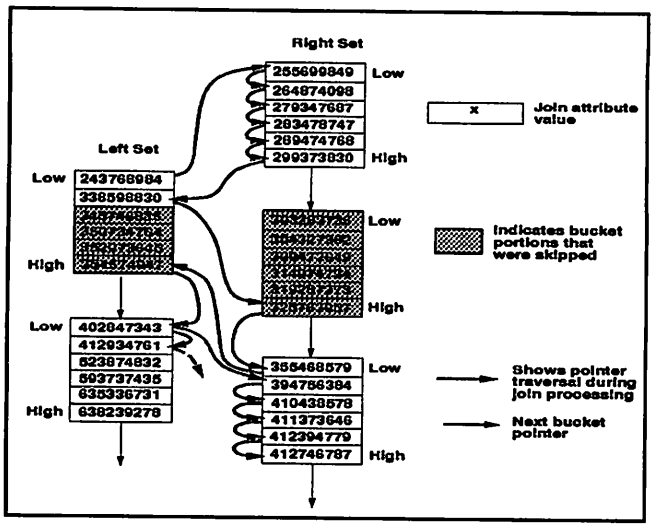

Figure 1: *Bucket comparisons in the BSMJ algorithm*

3 Bucket Skip Merge Join Algorithm

In this section we present the basic philosophy behind the BSMJ algorithm and explain the functioning of the algorithm.

The main goal of the algorithm is to skip as many data items as possible in the datasets being joined. The merge join algorithm forms the basis for the algorithm. It maintains a pointer to items within the datasets and moves to the next item if a match cannot be found. Instead of moving to the next element it would be nice if the algorithm could use some "intelligent scheme" to *jump* to the next item in the dataset for which a match may be found. Since this is not always possible, we exploit information about ordering of items in the datasets to determine where the next jump should be within the data sets. This is achieved by partitioning each dataset into *buckets* which store data items in an ascending order. The lowest value and the highest value of the data items in the buckets are also maintained. The attribute itself need not be numeric as long as a comparison function exists for the attribute.

Figure 1 shows how the BSMJ algorithm works. At any point in time, joining is attempted between a bucket in the first (left) dataset and a bucket in the second (right) dataset. The algorithm starts by considering the first two buckets of each dataset. A *cursor* is maintained for each dataset, which is initially the lowest value in that dataset's first bucket. The cursor value of one dataset is compared with the high value of the current bucket of the other set. This helps determine if the current buckets *overlap*, *i.e.*, the high value of this bucket is less than the cursor value of the other bucket, there is an overlap in the range of the values in the two current buckets. If the buckets do not overlap, then the next bucket in the dataset that contained the smaller value is considered. This process is repeated until an overlap is found. When there is an overlap between the buckets, then a merge-join is performed between the items of the two

```
1.    cursor1=left.low; cursor2=right.low;
2.    while (last bucket of any list has not been reached) {
3.       while ( (cursor1 > right.high) OR (cursor2 > left.high) ) { /* buckets do not overlap*/
4.          while (cursor2 > left.high)
5.             left=left.next;
6.          cursor1=left.low; cursor1.pos=1; /* first item in bucket */
7.          while (cursor1 > right.high)
8.             right=right.next;
9.          cursor2=right.low; cursor2.pos=1;
10.      }
11.      /* buckets now overlap */
12.      while (cursor1 != left.high AND cursor2 != right.high) {
13.         if (cursor1 < cursor2)
14.            while (cursor1 < cursor2) cursor1=left.val[cursor1.pos++]; /* cursor1.pos is position of cursor1*/
15.         elseif (cursor2 < cursor1)
16.            while (cursor2 < cursor1) cursor2=right.val[cursor2.pos++];
17.         if (cursor1 == cursor2) /* match found */
18.            output cursor1; cursor1=left.val[cursor1.pos++]; cursor2=right.val[cursor2.pos++];
19.         else
20.            if (cursor1 > right.high)
21.               cursor2 = right.high;
22.            elseif (cursor2 > left.high)
23.               cursor1 = left.high;
24.      }
25.      /* read next bucket for bucket(s) whose end was reached*/
26.      if (cursor1 == left.high)
27.         left=left.next;
28.      if (cursor2 == right.high)
29.         right=right.next;
30.   }
```

Figure 2: *Pseudo code for BSMJ Algorithm for joining two datasets — left & right*

buckets that overlap until the end of one of the buckets is reached. This merge-join is slightly different than the traditional merge-join: Suppose the cursor of the left dataset is smaller than the cursor of the right dataset. This cursor is incremented until it is greater than or equal to the other cursor. If the values are equal then a match has been found and it is added to the list of matched items, and the cursors of both the buckets are incremented. If the values are not equal, a check is performed to see if it exceeds the high value of the current bucket of the right dataset. If so, the right cursor is set to the last data item in the current bucket, thus completing the merge-join between the pair of buckets under consideration. Once the end of a bucket is reached, the next bucket is retrieved from the corresponding dataset. The whole process described above repeats and the algorithm terminates when all the buckets from one of the datasets have been processed. The pseudo-code for the BSMJ algorithm is given in figure 2.

An important consideration in the BSMJ algorithm is the size of the two datasets. If one of the datasets is extremely large compared to the other, several items of the large dataset will be examined on an average while looking for a match. Since our objective is to minimize the number of item accesses, we introduce the notion of a *bucket skip factor*. The idea is that when

there is a mismatch in the sizes of the datasets, instead of proceeding to the next data in a set, we can skip several items before checking for a match again. If we find we have skipped too much, we undo the last step (the skip) and move to the next item. This bucket skip factor indicates how many data items it should jump within a bucket instead of reading the next data item. By choosing an appropriate skip factor, the average number of items accessed can be reduced considerably. This skip factor can be specified statically or determined dynamically based on the size of the datasets and hence involves a tradeoff. We return to this issue later in the context of the performance tests. To incorporate the bucket skip factor into the BSMJ algorithm, additional code is inserted before line 16 in figure 2 as follows.

```
while (cursor2 < cursor1) cursor2=right.val[cursor2.pos+skip_factor];
cursor2=right.val[cursor2.pos+skip_factor];
```

The skipping is done for the buckets on the right because we sort the datasets such that the left set is the smallest and the right one is the largest. Basically this scheme ensures that larger jumps are taken when the current value of the larger bucket is much less that the current value of the smaller bucket and when the values are close, we jump just to the next data item.

The dataset partitioning scheme and the BSMJ algorithm are different from the *partition band join* [DNS91a] scheme. In band joins, the join attribute of the first input should fall in a specified range above or below the values in the join attribute of the other input. Both the inputs datasets are partitioned carefully into equal number of partitions using a sampling based scheme (instead of a sorting scheme). The partitions are such that items of a partition of one input fall within a "range" of the highest and lowest items of the corresponding partition of the other input. This is required to ensure that the pages are never re-read during the join of the two corresponding partitions that is performed using a sort-merge band join. In contrast to this scheme, the BSMJ algorithm does not have restrictions as to how the data is partitioned into buckets. The manner in which the join is processed is totally different since a bucket from one dataset does not have to fall in the range of a bucket from the other set and the buckets are loaded independently one after the other as exaplained in figures 1 and 2, The BSMJ algorithm is meant for general purpose joins and aims at avoiding fetching pages whenever possible.

In summary, the BSMJ algorithm correctly determines all the items that match while avoiding accesses to as many data items as possible in each set. Since many instances of partial bucket skips exist, the number of memory accesses is considerably reduced. Also some buckets may be skipped completely. By suitably mapping buckets to disk pages (as described in the next section), the skipping of full buckets can be translated to skipping page fetches from the disk, thus saving I/O overheads.

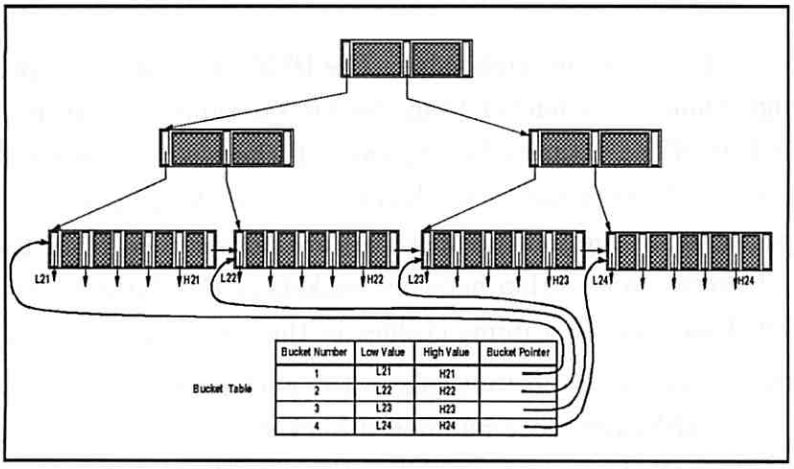

Figure 3: *Creating and managing buckets via bucket table using leaf nodes of B+ tree*

4 Creating and Managing Buckets

In this section we describe how the buckets are created and managed. Specifically, we discuss how they can be maintained using an existing index scheme.

The B+ tree is a popular indexing scheme for databases. It is a variation of the B tree and all pointers to physical records exist and the leaf level nodes. The leaf level nodes are also chained from left to right such that it is easy to make a pass through the entire data set (table) using the index. We make use of this structure by mapping a leaf node to a bucket. Since we also need to know the low and high values of a bucket, we maintain a *bucket table* as shown in figure 3. Each row in the table corresponds to an individual bucket and contains the low and high value of the bucket and a physical pointer to the bucket, which in our case is one of the leaf level nodes of the B+ tree. This bucket table is updated whenever a change occurs to the B+ tree that affects its structure or the low and high values of a bucket. Hence if a pointer internal to the leaf level node (not the left most or the right most) is added or modified or deleted within a leaf level node, the modification does not affect the bucket table. If just the left most or right most pointer in a leaf node is changed, the low or the high value of a bucket has to be changed in the bucket table. If a leaf split/merge occurs then an additional row is added/deleted in the bucket table and the low and high values of the old and the new nodes are changed accordingly. Handling data skew is an important consideration in join algorithms [GLS94]. Since the B+ tree is balanced, in our case data skew does not have an effect on bucket maintenance. Note that the overhead for maintaining buckets via the bucket table is very minimal compared to the cost of maintaining the B+ tree itself.

The advantage of having a separate bucket table to store information about the low and the high values of a bucket is that during join processing, it is not necessary to fetch the buckets

themselves just to examine their low and high value. The BSMJ algorithm we presented in figure 2 reads the low and high values of a bucket from the bucket table to determine if there is an overlap between the buckets of the datasets being joined. If there is no overlap, then the bucket, *i.e.*, the corresponding leaf node need not be fetched from the disk. This results in considerable savings since useless pages (from the join perspective) are not fetched from the disk unlike in other join algorithms. If there is an overlap between buckets of the datasets, then of course the buckets have to be fetched and the data items (values in the leaf nodes) examined.

The size of the buckets plays an important role in the performance of the BSMJ algorithm. If the buckets are too small, although more number of buckets can be skipped completely, more time will be spent comparing the highs or lows to check for overlaps. If they are too large, then the number of buckets skipped will be less. In practice however, since the savings depend more on the reduction in the number of I/Os, we have mapped buckets to the granulaity of physical pages.

If there are duplicate values in the join attributes, they will handle appropriately by the B+ tree index management scheme. Hence no special schemes are needed to maintain the buckets. However an additional check is to be performed in the BSMJ algorithm while incrementing the bucket cursors after a match (lines 20-23 in figure 2) and while checking for bucket overlaps (line 3-10 in figure 2) to see if there are duplicates and handle them based on the policy for handling duplicates, *e.g.*, output one copy or all the duplicates.

Thus the advantages of the BSMJ algorithm over the traditional join schemes are as follows:

- accesses to several data items within a bucket can be avoided, thus saving in memory access time (*i.e.*, partial buckets are skipped).
- when buckets do not overlap, buckets need not be fetched from the disk reducing I/O overheads (*i.e.*, buckets are skipped completely).
- if there is disparity in the size of the datasets, using a bucket skip factor many data items can be skipped in the larger dataset while looking for a match, thus avoiding even more on memory accesses.
- several unnecessary comparisons are avoided, reducing CPU overheads

5 Best and Worst Case Analysis of BSMJ Algorithm

In this section we analyze the best and worst case in terms of the number of data items accessed by the BSMJ algorithm. Based on this, the cost of the BSMJ algorithm can be determined in terms of the number of memory accesses, disk I/Os and CPU time. A comparison is also performed with respect to the traditional merge join (MJ) scheme. The system parameters used in the analysis are shown in the figure 4.

Factor	Notation
Number of buckets in dataset a	N_a
Number of buckets in dataset b	N_b
Average number of items in a bucket	n
Number of bytes in a row of bucket table	b
Page size in number of bytes	P

Figure 4: *Parameters used for best and worst case analysis*

The best case for the MJ scheme occurs when all the values of the join attribute of one set is less than (or greater than) the values of the join attribute of the other set. In this case, MJ will examine all the pages of the index of one dataset and only the first page of the index of the other dataset. This is because the cursor is advanced only for one of the datasets. Assuming the cursor is advanced through dataset B, the number of pages fetched is N_b, the number of items accessed is $N_b \times n$ for dataset B and 1 for dataset A, the number of comparisons is $N_b \times n$. The BSMJ algorithm will examine the high value of the buckets of only one data set. Hence no index pages are fetched into memory and only one page from the bucket table for dataset A is fetched whereas all the pages of the bucket table for dataset B ($\frac{N_b \times n}{P}$ pages) are fetched into memory. The number of memory accesses is $N_b + 1$ and the number of comparisons is N_b.

The worst case for the BSMJ algorithm occurs when all the buckets have to be fetched from the disk and all the items within the buckets are to be examined. This happens when the all the buckets overlap (join selectivity is high), when the bucket skip factor is 1 for the larger dataset and the data items are so close that even partial buckets skips are not possible. This also coincides with the worst case of the MJ scheme. The costs are determined in a similar manner as the best case.

The cost for the different cases are summarized in the table in figure 5. Let us assume the join attribute takes j bytes, a page pointer (page address) takes p bytes and the bucket

Case	Join Scheme	Cost Component			
		DISK		MEMORY	CPU
		Bucket Table Page Fetches	Index Page Fetches	No. of Memory Accesses	No. of Comparisons
Best Case	BSMJ	$1+\frac{b \times N_b}{P}$	—	$N_b + 1$	N_b
	MJ	—	$1 + N_b$	$N_b \times n + 1$	$N_b \times n$
Worst Case	BSMJ	$\frac{b \times (N_a + N_b)}{P}$	$(N_a + N_b)$	$(N_a + N_b) \times n$ $+(N_a + N_b)$	$((N_a + N_b) \times n) - 1$ $+(N_a + N_b) - 1$
	MJ	—	$(N_a + N_b)$	$(N_a + N_b) \times n$	$((N_a + N_b) \times n) - 1$

Figure 5: *Cost summary for different cases*

number takes t bytes. Hence the value of b, the number of bytes in a row of the bucket table is $(2 \times j + p + t)$ which is small compared to the page size P. Hence the fraction b/P is small. From the cost analysis table in figure 5, it can be observed that in comparison with the MJ scheme, the BSMJ scheme has *substantial* lesser total cost in the best case and a slightly higher total cost in the worst case.

Let the cost for fetching a page be C_p, accessing memory be C_m and performing a comparison be C_c. The savings for BSMJ scheme in the best case and the extra costs in the worst case can be determined by substituting some realistic values for the byte sizes. Assuming j=10, p=8, t=8, P=1024, n=64 (approx $\frac{P}{j+i}$, where i is the size of a pointer address within a page and needs 1 byte; thus n= 93 but we have been conservative) and both relations have a million items each, $N_a = N_b = 1,000,000/64 = 15,625$. The equations combining all cost components are as follows:

$$\textit{Best Case Savings}: \quad 15,224 \times C_p + 984,375 \times C_m + 984,375 \times C_c \tag{1}$$

$$\textit{Worst Case ExtraCosts}: \quad 793 \times C_p + 31,250 \times C_m + 31,250 \times C_c \tag{2}$$

It can be seen that the best case savings from all the components is orders of magnitude higher than the extra costs incurred in the best case for the BSMJ scheme over the MJ scheme. This clearly indicates that on average, considerable savings can be obtained by using the BSMJ algorithm over the MJ algorithm. We do not compare BSMJ with the hash join scheme since hash join works on a different principle and a lot more parameters are to be considered which are complex and beyond the scope of this paper. However we have experimentally compared the performance of the hash join scheme with the BSMJ scheme and have shown that the BSMJ scheme is superb.

6 Query Processing using BSMJ Algorithm

In this section we discuss how the BSMJ algorithm can be used for query processing in relational databases and in text databases. Specifically, in the context of relational databases we discuss how multiway joins can be performed and how joins can be performed by generating the buckets on the fly in absence of indexes. In the context of text databases we discuss how the BSMJ algorithm can be used to retrieve documents that satisfy an AND query, *i.e.*, query that retrieves documents that contain *all* the given a set of keywords.

Relational Databases

Most joins in relational databases are two way joins on a common attribute. The BSMJ algorithm can be directly used when there are indexes on both the attributes to be joined. It is also possible to have a multiway join on the same attribute. In this case, using the BSMJ

algorithm instead of creating temporary relations, the join can be performed directly among all the sets, *i.e.*, if an item qualified for join in the two leftmost tables, the BSMJ algorithm is used to check if that item exists in the rest of the tables as well. Such a case is shown in figure 6 and we will describe the details later in the context of text databases.

Although we expect most large databases of the future to have indexes due to a high read to write ratio on data items, we have also designed an enhancement to the BSMJ algorithm to work in the absence of indexes. An obvious choice would be to build a complete B+ tree index on the datasets or sort the entire dataset first as in merge join. However we have come up with something more interesting. Knowing the low and the high values of the join attribute (this is determined using a single pass through the table) and the number of tuples in the table, it is possible to approximately determine the range of values each bucket should have to partition the data into buckets of equal sizes. Then using a single pass through the table, data is partitioned into buckets and the bucket table is created. Notice that the data within the buckets is not yet sorted. Hence the overhead we have incurred thus far is only for partitioning the data into buckets, which is not much compared to the cost of sorting the entire table. When the BSMJ algorithm determines that there is an overlap between the buckets (based on high and low values of buckets), then the corresponding bucket is loaded and the data inside the bucket is sorted before moving the cursor within the bucket. Thus the data is *sorted on demand*, *i.e.*, sorting is done only if and when it is necessary. The performance of this sort-on-demand scheme should be better than that of merge join and comparable to that of hash join.

Text Databases

Now we discuss the use of the BSMJ algorithm for processing queries in text databases. In text databases, documents are usually retrieved by specifying a set of keywords connected by logical operators like AND and OR [HFBYL92]; of these, AND queries are the most popular form of queries in any search system. Since joins are analogous to the AND operation, we focus only on AND queries, *i.e.*, queries that have AND connectors between the keywords. The techniques we describe here are important, especially with the current trend to integrate information retrieval (IR) systems with DBMSs [DDS+95]. It is also important in the case of image-databases (and digital-libraries), where a join might have to be performed on several attributes (color, texture, shape, patterns) to locate images that match a specified query.

Although boolean query processing has been used in IR systems, the probabilistic query processing approach has become popular more recently [HFBYL92]. Each keyword is associated with an index entry called *inverted list*, which contains the document IDs of the documents that contain that keyword. During query processing this index entry is used to determine the documents that qualify for a query. In boolean query processing, a document is retrieved only if

its ID appears in the index entries of all the keywords specified in the query. Thus the results are more precise in the sense that all the keywords specified are contained in the documents. On the other hand in the probabilistic approach, documents are assigned scores based on the number of occurrences of the specified keywords in the documents and the importance of the keyword. The documents are sorted based on the scores and the top few documents are returned to the user. The documents returned need not contain all the keywords specified in the query. Also to cut down the query execution time, optimization schemes have been proposed [Bro95] where a *best list* for each keyword is maintained that contains document IDs of the top 1000 documents that have the maximum occurrences that keyword. During query processing, the document IDs from the best list of each keyword specified in the AND query are aggregated into a set called the *candidate set.* For each document ID in the candidates set, information about the keywords specified in the query (like the number of occurences of each of the keywords in the document) is extracted from the respective signature files. This information is processed to compute an aggregate score for the document which indicates the relevancy of the document to the specified query. The document IDs in the candidate set are sorted by this aggregate score and the top few (usually 100) documents are returned to the user.

Note that in the case of boolean query processing, it is possible to maintain a count of the keyword occurences in the documents and assign scores to the documents that qualify from the boolean search. The qualifying documents can then be ranked based on this score (similar to probabilistic search) and the top 100 documents returned to the user. These top 100 documents are the best documents that can be retrieved for the given query.

The problem with boolean query processing used to be that it may not be able to find 100 documents that contain all the keywords. However current trends in text databases and IR systems indicate that the size of the document database will be huge. That being the case it would be possible on almost all occasions to retrieve 100 top documents that satisfy a query. The only problem is that an efficient scheme is necessary for boolean query processing (for AND queries) with large datasets. The BSMJ algorithm provides such a scheme.

We have illustrated the use of the BSMJ algorithm to perform a boolean query on four datasets (which are the inverted lists of the four keywords) in figure 6. The figure shows the sequence of pointer traversal during join processing. A detailed explanation is not possible due to space limitation but the figure can be interpreted in a similar manner as figure 1. As seen in the figure the datasets are already partitioned into buckets. This is done as the documents are inserted into the database and the inverted lists are updated. The values (document IDs) are assigned a monotonically increasing value and hence values are typically appended. The shortest dataset is the leftmost and the longest is the rightmost (this improves the performance). A join is started between the first two datasets. If a match is found a join is attempted with this

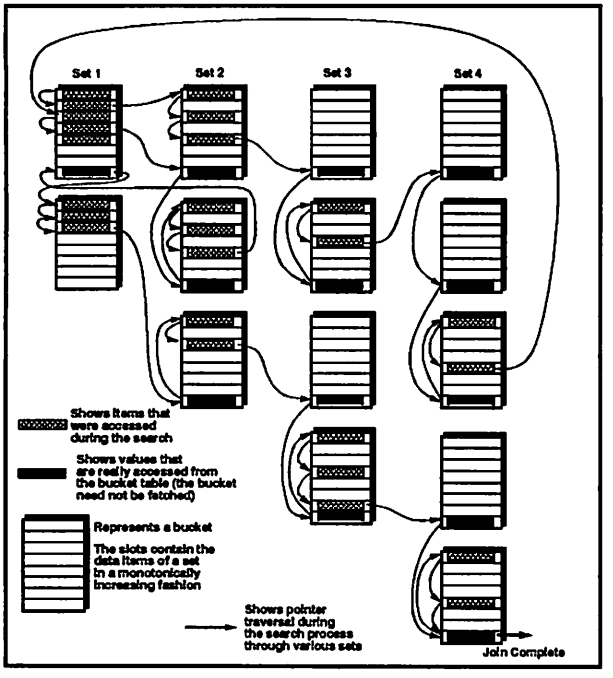

Figure 6: *Working of the BSMJ Algorithm for joins on multiple sets*

matched item and the third dataset. If a match is also found in the third dataset, a match is attempted in the fourth dataset and so on. Notice that there is no need to store the data items in a temporary table. Here data is effectively pipelined from one join to the next [GLS94] and we refer to this method as piplelined join in this paper. This scheme can achieve sublinear scaling with number of keywords while giving more precise answers.

7 Performance Tests

In this section we briefly describe the prototype system we used for the tests and the nature of the performance tests.

Prototype System

The prototype systems is an extention of the query processing infrastructure we had developed for the database bibliography search system which has been in use on our world wide web site[1] for a few months. The system has been built on a SunSparc 20 workstation equipped with 96 MB of main memory, 2 SCSI disks of 2GB capacity each and running Solaris 2.3 version of Unix.

We have implemented the merge join scheme, hash join scheme and the BSMJ scheme. The merge join scheme is simple. It reads the index pages sequentially from both the datasets and

[1](at URL `http://www-ccs.cs.umass.edu/db/bib.html`).

moves the respective cursors through the pages as it performs the join. The BSMJ scheme makes use of the buckets that have been created using the same indexes that are used by the merge join scheme. Enhancements discussed to the BSMJ scheme like the bucket skip factor have also been implemented. For the hash join scheme, a parition hash join scheme has been implemented. Data is already partitioned and during the join, the hash table is created in memory for the partition from the left dataset and items from the right dataset are considered sequentially and hashed to check if the value exists in the left table as well. We have considered various sizes of the datasets and they are indicated along with the results of the tests.

Nature of Tests

The first set of tests compare the schemes in the context of relational databases. Our primary focus has been on comparing the scalability of the join schemes for large databases. Two important parameters for the tests are the size of the input datasets for the join and the join probability. Various sizes have been considered for the input datasets, varying from 100,000 records to 1,000,000 records. The join probability indicates the probability of finding a match between two identical ranges (absolute value) and is used as follows to synthetically generate the datatests: if the join probability is P_i, we generated datasets such that for an absolute range of 100, e.g. 5100-5200, the common items between the two datasets will be in the range 5100-$(5100 + P_i \times 100)$ and the other items in the left and right datasets will be in the ranges $(5100 + P_i \times 100)$-$(5200 - \frac{1-P_i}{2} \times 100)$ and $(5200 - \frac{1-P_i}{2} \times 100)$-5200 respectively. Hence the join probability is also an indication of the overlap that can occur between the two datasets. The two input datasets are generated simulateneously by incrementing the absolute range, starting from a range of 0-100. The number of items generated in each dataset within this absolute range is dependent on the ratio of the sizes of the two datasets, *e.g.*, if the left dataset needs 100,000 records and the right dataset needs 200,000 records; for every value generated in the left dataset, two values are generated in the corresponding range in the right dataset. The number of common items between the entire dataset is $(P_i \times S)$ where S is the size of the left dataset. The metric used for comparison is the *query response time* (join processing time). We have also compared the schemes for three way joins on the same attribute. Also to study the efficiency of the BSMJ algorithm, we report on the percentage of buckets skipped during join processing.

The second set of tests are performed in the context of text databases. We compare the performance of boolean query processing and the BSMJ algorithm in terms of the quality of documents returned and the response time of the queries. Then we compare the three different join schemes, *i.e.*, merge join, hash join and BSMJ for boolean query processing by changing a variety of parameters — number of keywords from 2 to 5 and number of documents (size of the database) from 20,000 documents to 1,000,000 documents. The join datasets here (indexes) are

actual datasets generated by analyzing the text documents. These tests verify the performance of the BSMJ scheme on real world datasets. The metrics used to compare the performance are the *response time of queries* and the *number of comparisons* performed (only between merge join and BSMJ scheme). We have also performed tests to determine the influence of bucket size and bucket skip factor on the performance of the BSMJ scheme.

8 Results

In this section we analyze the results of the performance tests. We first discuss tests done in the context of relational database and then focus on tests in the context of text databases.

Relational Databases

Here we study response time for 2 way and 3 way joins (on the same attribute) with respect to the join probability, scalability with respect to input size and degree of join and finally response time for joining unsorted data.

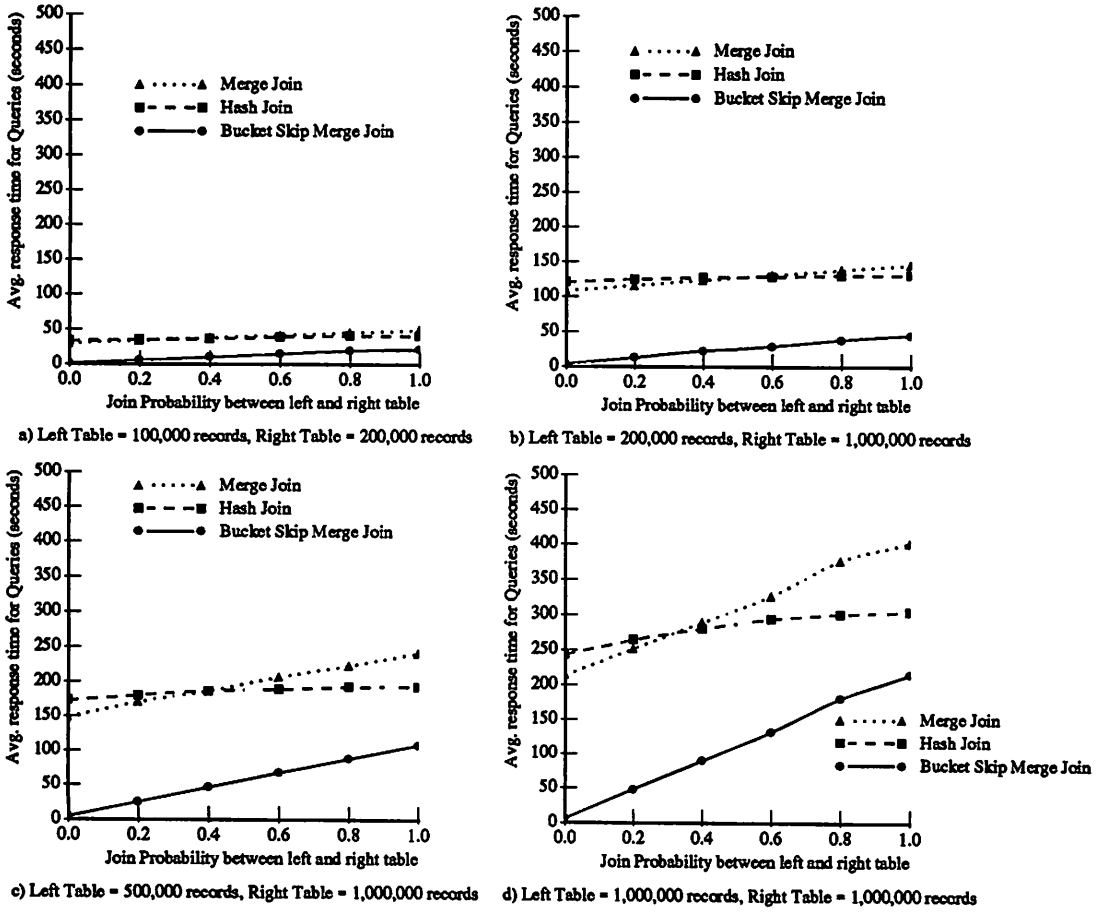

Figure 7: *Response time of 2-way join queries*

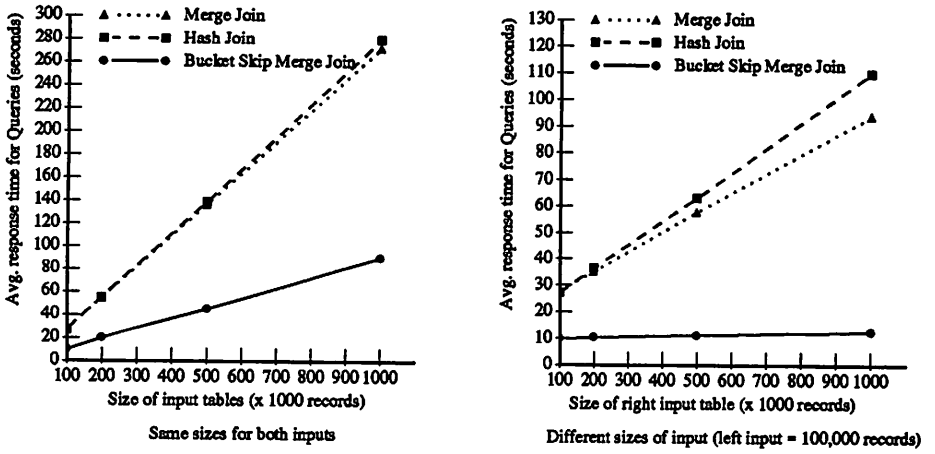

Figure 8: *Scalability of join schemes with respect to input sizes*

BSMJ has lower response time for 2 way joins: The response time of queries for two way joins using the three different schemes are shown in figure 7 for various combinations of input sizes. The X axis represents the join probability. It can be observed that the behavior is similar for all inputs except that the hash join curve is relatively flat. This is because the hash join scheme is not affected by the join probability but only the merge join schemes are affected and as the probability increases, the response time also increases. The BSMJ algorithm performs the best in all cases. On average, the BSMJ algorithm performs 2 to 3 times faster than the other schemes for the sizes we have considered. Notice that even when the join probability is 1.0, *i.e.*, the values in both the tables fall in the same range, BSMJ performs better. This is because BSMJ is still able to skip comparisons and parts of buckets whenever possible. Note that we have not considered smaller input sizes as the performance of all the schemes will be almost the same.

BSMJ scales very well with the input sizes: The scalability of the schemes is compared in figure 8. The join probability is fixed at 0.4 for these tests. The graph on the left shows the response time for input tables of the same size. The X axis here represents the size of the input tables. Eventhough all schemes are linear with respect to the input size, the BSMJ scheme has a smaller slope and hence is sublinear. The graph on the right side shows the response time for input table of different sizes. The size of the left table is fixed at 100,000 records and the size of the right table is varied. The X axis shows the sizes of the right table. The BSMJ scheme performs extremely well and scales sublinearly with respect to the size of the right table. Both these graphs clearly show that the BSMJ scheme scales very well with input sizes.

BSMJ scales very well with the degree of join: In a multiway join, it is possible for two or more tables to be joined on the same attribute. Pipelining as described in figure 6 can be used to

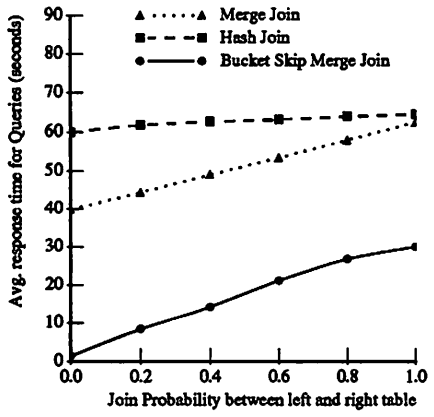

Figure 9: *Response time of 3-way join (same attribute) queries*

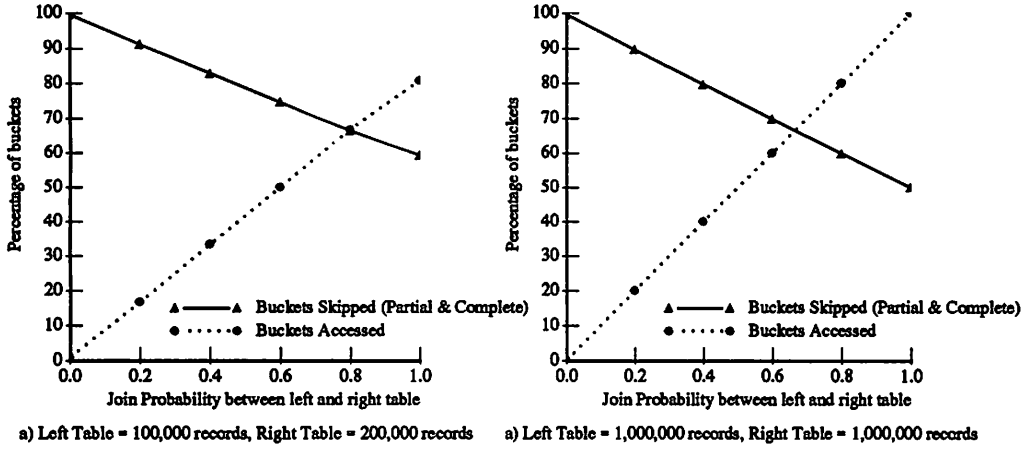

Figure 10: *Percentage of Buckets Skipped and Accessed in 2-way joins queries*

perform the join efficiently. We performed tests to determine the benefits of the BSMJ scheme in 3 way joins. As seen in figure 9, the BSMJ scheme performs the best. Since the BSMJ scheme skips buckets and also skips items within a bucket based on the value of the bucket skip factor, it is ideally suited for pipelined joins on the same attribute. If the input sizes are increased further, the difference between the response times of the schemes will increase as the number of bucket skips and jumps within a bucket will be more for the BSMJ scheme. Also comparing figures 7(a) and 9, it can be observed that while the difference in the response time between a 2 way join and a 3 way join is less in the case of the BSMJ scheme, it is considerable for the other schemes. This indicates that the BSMJ scheme scales better than the other schemes with respect to the degree of join. This is another significant result of our tests. This result will be clearer when we examine results of similar tests later in the case of text databases.

BSMJ skips buckets even when the join probability is high: During the tests, we traced the number of buckets accessed and skipped (partial and complete) by the BSMJ algorithm. The results are plotted in figure 10 as a percentage of the total number of buckets for both the inputs put together. The X axis represents the join probability. Note that the percentage of buckets accessed and skipped are not complements of each other since there are some buckets which are accessed and skipped partially. The graph on the left depicts the case when the two inputs are of different sizes. Here less number of buckets are accessed since many buckets are skipped from the larger table. On the other hand, the graph on the right depicts the case when both the inputs are of the same size. In this graph, the slope of the buckets accessed curve is steeper than the other graph, indicating that more buckets are accessed on average. Also for a join probability of 1.0, we find that all the buckets are examined when the input sizes are equal. However for the same case, the number of buckets skipped is almost the same as in the other graph. This indicates that, if the inputs sizes are equal, more buckets are accessed and as the probability of common band increases, the number of partial bucket skips exceeds the number of complete bucket skips. This confirms the observation we made in the scalability graphs plotted in figure 8.

With unsorted data, BSMJ works well when join probability is not high: Recall that when the inputs are not sorted and indexes do not exist on the tables to be joined. The BSMJ scheme first partitions the data into buckets and then sorts the buckets on demand, *i.e.*, only if it is necessary to examine the items in the buckets. This gives the BSMJ algorithm a significant edge over the merge join algorithm and even helps it perform competitively with the hash join algorithm. We have performed two sets of tests as shown in figure 11. The graph on the left shows the case when both inputs are not sorted and the one on the right shows the case in which only the left input is sorted. We observe that in both cases, for a range of probabilities, the

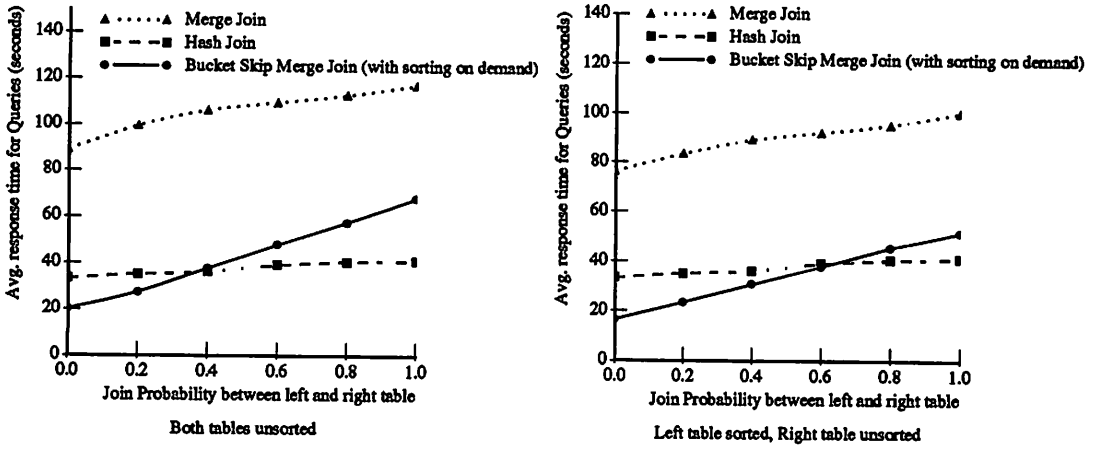

Figure 11: *Response Time for 2-way joins queries on unsorted data*

BSMJ scheme performs better than the hash join scheme — upto a probability of 0.4 when both tables are unsorted and upto a probability of 0.6 when the left table is sorted and the right table is unsorted. This clearly indicates that when data is not sorted, the BSMJ scheme outperforms the merge join scheme and is to be opted over the hash join scheme when the join probability is not high.

Text Databases

Here we study the performance of boolean and probabilisitic schemes on AND queries in text databases, response times of the three different join schemes (for boolean query processing) with respect to the number of keywords specified in the query and the effect of bucket size and bucket skip factors on the performance of the BSMJ scheme.

For AND queries on text databases, BSMJ produces better results than probabilistic schemes in a shorter time: Our first set of tests compare the performance of boolean and probabilistic information systems in terms of quality of the documents retrieved and the response time of the queries. In probabilistic systems, the performance critically depends on the size of the best list for each keyword. The results of our tests are plotted in figure 12. The text database has 1,000,000 documents. Our metric for determining the quality is to determine the percentage of the top 100 documents returned by the query that contain *all* the keywords specified, which we refer to as the best documents. This is plotted in the left graph. The other metric is the response time and is plotted in the right graph. The BSMJ scheme always returns the best 100 documents since it determines all documents that have all the keywords specified in the query and then ranks them based on the number of occurences of the keywords. In contrast, the probabilistic scheme is restricted by the size of the best list. From both the graphs it can be

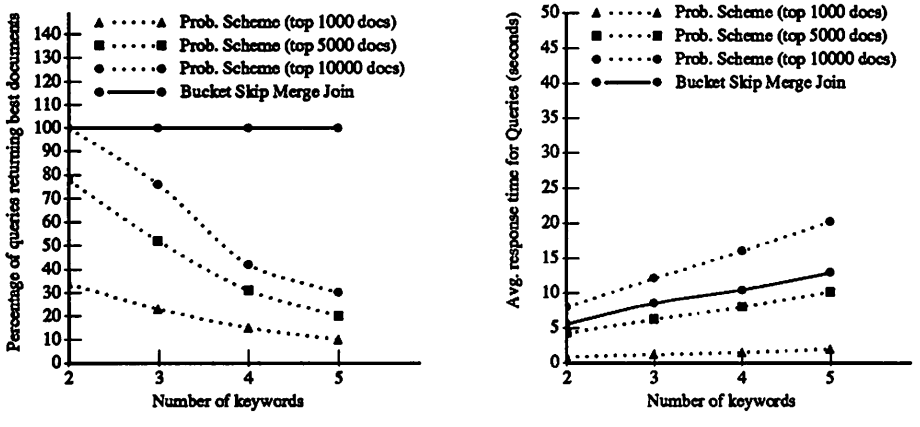

Figure 12: *Comparison between boolean and probabilistic processing*

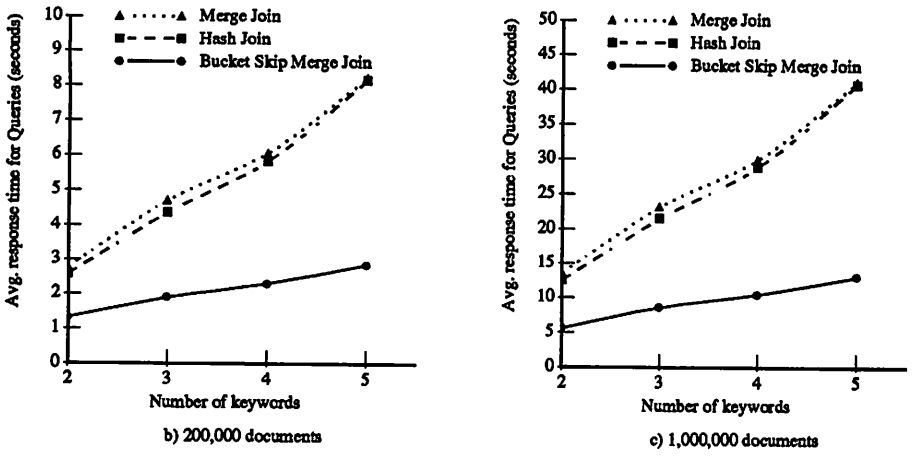

Figure 13: *Response time of queries as a function of no. of keywords*

observed that if the size of the best list is small (1000 documents), queries have a better response time at the cost of sacrificing the quality whereas if it is large, the quality improves but the response time increases. Thus it is clear that for AND queries, a boolean search using the BSMJ algorithm will give the best results with a short query response time. Just as a query optimizer can choose between the different join algorithms, the BSMJ algorithm should be available for processing AND queries in an information retrieval system. Our database bibliography search system[2] now uses the BSMJ scheme and is performing superbly. The query response time has been reduced by 5 times over that of the previous version which used a hash join scheme. This clearly shows the practical utlity of the BSMJ scheme.

BSMJ scales very well with the number of keywords (degree of join): Since merge join and hash join can also be used to find the common documents between sets, we have compared them with the BSMJ scheme for boolean query processing of AND queries. The results are plotted in figure 13. Clearly the BSMJ scheme dominates the other schemes. This test also shows the scalability of the BSMJ algorithm with the degree of join. Here the BSMJ scheme is used in a pipelined fashion to perform the joins of sets numbering 2 to 5 and we observe that the BSMJ scheme scales sublinearly.

Medium sized buckets and dynamically determined bucket skip factors give the best performance for BSMJ: Our final sets of tests determines the effects of bucket size and the bucket skip factor on the performance of BSMJ algorithm. The results are plotted in figure 14. Figure 14(a) indicates that the bucket size is important and that as long as the buckets are large enough to contain around 40-200 items, good performance can be guaranteed. If the size of buckets is too small, although more number of buckets may be skipped, the number of comparisons

[2](at URL `http://www-ccs.cs.umass.edu/db/bib.html`).

made between the high values and the low values of the buckets will be very high and hence the performance is bad. Figure 14(b) shows the effect of the bucket skip factor on the response time. It shows that the performance is best when the bucket skip factor is dynamically determined rather than being fixed at a particular value. The bucket skip factor is determined dynamically for each set as the ratio between the size of that set over the size of the smallest set. This is because the smallest set occupies the leftmost position and the largest the rightmost position in the way we perform the pipelined join. The other two graphs plot the ratio of the real number of comparisons made vs. the ideal number of comparisons. The ideal number of comparisons is the minimum number of comparisons to be made to determine the number of matches between the given input sets and is the product $S \times (n-1)$, where S is the size of the smallest set and n is the number of input sets. The smaller the ratio, the better the scheme. The graphs indicate that this ratio is small as long as the buckets are large enough to contain 40-200 items and the bucket skip factor is determined dynamically based on the sizes of the input sets.

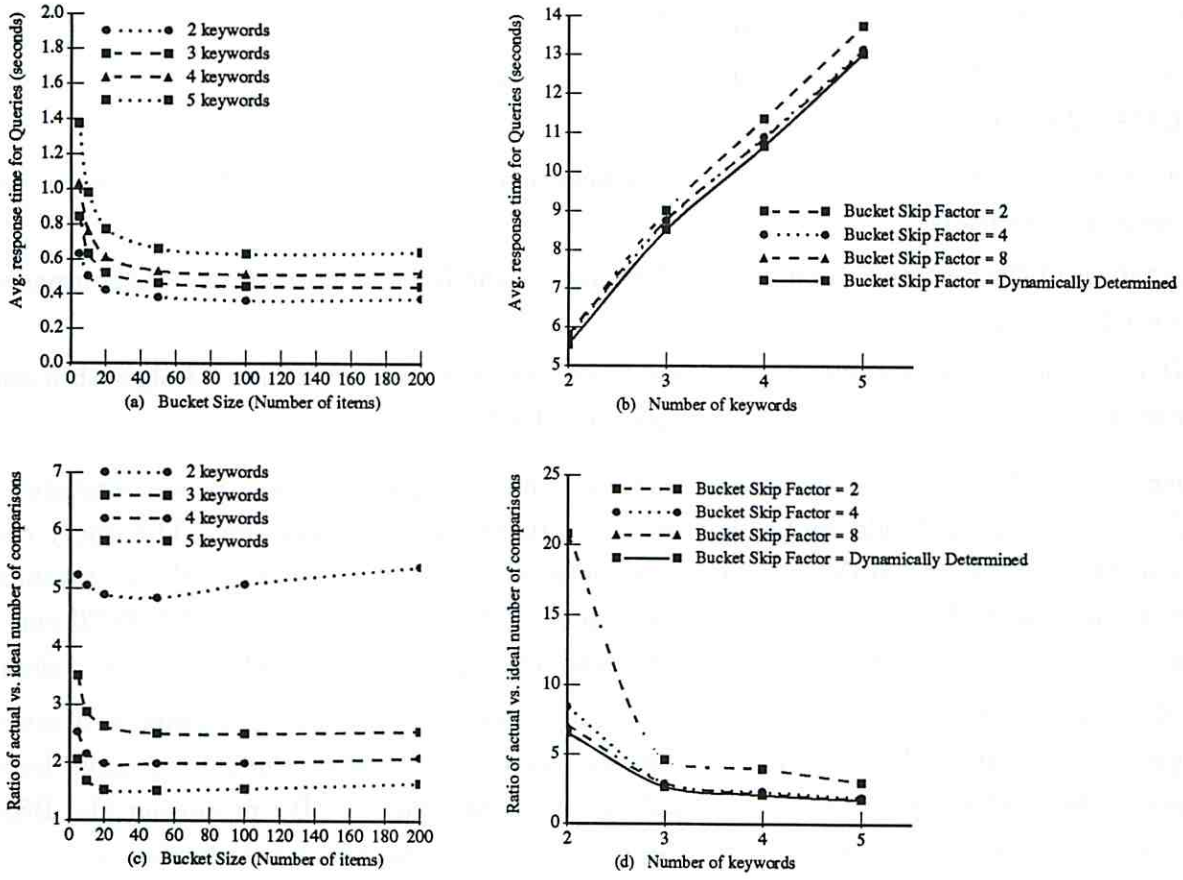

Figure 14: *Effect of bucket size and bucket skip factor*

Hence in all the tests we performed for relational databases using the BSMJ scheme, the bucket skip factor was determined dynamically. The reason we performed the test on bucket skip factor for text databases instead for relational databases is that because there are more variations in the size of the sets in text databases. We used a bucket size of 128 for all the other tests.

9 Conclusions

Join processing is a costly operation and critically determines query performance. Current database trends indicate that databases used for emerging applications like information retrieval, digital libraries and decision support systems will be large and the read to write ratio of data items will be very high, requiring indices for fast accesses. To exploit this scenario, we presented the BSMJ algorithm, a new join scheme that performs better than traditional join techniques by avoiding accesses to data items and reducing page fetches whenever possible. The CPU overheads are also reduced in turn. All this is achieved by inexpensively maintaining some extra information about the data items using buckets.

The specific contributions of the paper are as follows:

- BMSJ algorithm and its extensions
- Schemes for efficiently managing the buckets using the traditional B+ tree index with minimal overheads
- Analysis of the best and worst case performance of the BSMJ algorithm and its comparison with the merge join scheme
- Detailed performance tests that indicate the superb performance of BSMJ algorithm compared to standard schemes like merge join and hash join

Since the BSMJ algorithm has a tremendous impact on join processing performance, we strongly believe that it should be incorporated into future database systems. The query optimizer can then choose the BSMJ scheme over the merge join and hash join scheme whenever indexes are available. Even if indexes are not available, if the join selectivity [HNSS93] can be determined not to be high, the BSMJ algorithm should be preferred over the hash join scheme.

As database systems are being increasingly used to handle the data requirements of several new applications, established database techniques have to be re-examined to achieve better performance by exploiting the data and application characteristics. By proposing the BSMJ scheme, we have taken a step in this direction to enhance join processing performance.

References

[BE76] M. W. Blasgen and K. P. Eswaran. On the evaluation of queries in a database system. Technical report, RJ-1745, International Business Machines (IBM), San Jose, April 1976.

[BM90] J. A. Blakeley and N. L. Martin. Join index, materialized view, and hybrid-hash join: A performance analysis. In *Proc. IEEE CS Intl. Conf. No. 6 on Data Engineering*, February 1990.

[Bra84] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of the 10th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Singapore*, August 1984.

[Bro95] E.W. Brown. Fast Evaluation of Structured Queries for Information Retrieval. In *Proc. of SIGIR Intl. Conf. on Research and Development in Information Retrieval*, 1995.

[BS96] C. Bontempo and C. Saracco. Join processing: The relational embrace. *Database Programming and Design Magazine*, 9(1), January 1996.

[CHH+91] J. Cheng, D. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, and Y. Wang. An efficient hybrid join algorithm: A DB2 prototype. In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 171, Kobe, Japan, April 1991.

[Chr83] S. Christodoulakis. Estimating block transfers and join sizes. In *Proc. ACM SIGMOD Conf.*, page 40, San Jose, CA, May 1983.

[DDS+95] S. DeFazio, A. Daoud, L. Smith, J. Srinivasan, B. Croft, and J. Callan. Integrating IR and RDBMS Using Cooperative Indexing. In *Proc. of SIGIR Intl. Conf. on Research and Development in Information Retrieval*, pages 84–92, 1995.

[DG85] D. J. DeWitt and Gerber.R. Multiprocessor hash-based join algorithms. In *Proceedings of the 11th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Stockholm*, 1985.

[DKO+84] D. J. DeWitt, R. H Katz, F. Ohlken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memory databases. In *ACM SIGMOD*, July 1984. Also published in/as: UCB, Elec.Res.Lab, Memo No.84-5, Jan.1984.

[DLM93] D. DeWitt, D. Lieuwen, and M. Mehta. Pointer-based join techniques for object-oriented databases. In *PDIS-93, San Diego*, January 1993.

[DNS91a] D. DeWitt, J. Naughton, and D. Schneider. An evaluation of non-equijoin algorithms. In *Proc. Int'l. Conf. on Very Large Data Bases*, pages 443–452, 1991.

[DNS91b] D. DeWitt, J. Naughton, and D. Schneider. Parallel sorting on a shared-nothing architecture using probabilistic splitting. In *Proc. Int'l. Conf. on Parallel and Distr. Inf. Sys.*, Miami Beach, FL, December 1991.

[FKT86] S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the systems software of a parallel relatational database machine: GRACE". In *Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Kyoto*, August 1986.

[FNPS79] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing — A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3), September 1979. Also published in/as: IBM, Res.R. RJ2305, Jul.1978.

[GLS94] G. Graefe, A. Linville, and L. D. Shapiro. Sort versus hash revisited. *IEEE Trans. on Knowledge and Data Eng.*, 1994.

[GP89] D. Gardy and C. Puech. On the effect of join operations on relation sizes. *ACM Transactions on Database Systems*, 14(4), December 1989.

[Gra94] G. Graefe. Sort-merge-join: An idea whose time has(h) passed? In *Proc. IEEE Int'l. Conf. on Data Eng.*, page 406, Houston, TX, February 1994.

[HFBYL92] D. Harman, E. Fox, R. Baeza-Yates, and W. Lee. *Information Retrieval: Data Structures & Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1992.

[HNSS93] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Fixed-precision estimation of join selectivity. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, page 190, Washington, DC, May 1993.

[HR96] E.P. Harris and K. Ramamohanarao. Join algorithm costs revisited. *VLDB Journal*, 5(1), January 1996.

[KNT89] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid GRACE hash join method. In *Proceedings of the 15th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Amsterdam*, August 1989.

[LY89] R. A. Lorie and H. C. Young. A low communication sort algorithm for a parallel database machine. In *Proceedings of the 15th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Amsterdam*, August 1989. Also published in/as: IBM TR RJ 6669, Feb.1989.

[ME92] P. Mishra and M. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[Men86] J. Menon. A study of sort algorithms for multiprocessor DB machines. In *Proceedings of the 12th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Kyoto*, August 1986.

[MKY81] T. H. Merrett, Y. Kambayashi, and H. Yasura. Scheduling of page-fetches in join operations. In *Proceedings of the 7th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Zaniolo and Delobel(eds)*, September 1981.

[NKT88] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 468, Los Angeles, CA, August 1988.

[Omi89] Edward R. Omiecinski. Heuristics for join processing using nonclustered indexes. *IEEE Transactions on Software Engineering (SE), ; ACM CR 8912-0899*, 15(1), January 1989.

[SC90] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In *Proc of ACM SIGMOD Conf. on the Management of Data, Atlantic City*, May 1990.

[SD89] D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proc. ACM SIGMOD Conf.*, page 110, Portland, OR, May-June 1989.

[SD90] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proc. Int'l. Conf. on Very Large Data Bases*, page 469, Brisbane, Australia, August 1990.

[Sha86] L. D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3), October 1986.

[STG+90] B. Salzberg, A. Tsukerman, J. Gray, M. Stewart, S. Uren, and B. Vaughan. Fastsort: An distributed single-input single-output external sort. In *Proc. ACM SIGMOD Conf.*, page 94, Atlantic City, NJ, May 1990.

[SWKH76] M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementation of INGRES. *ACM Transactions on Database Systems*, 1(3), September 1976. Also published in/as: UCB, Elec.Res.Lab, Memo No.ERL-M577, Jan.1976.

[Val87] Patrick Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2), June 1987.