A Comprehensive Survey of Join Techniques in Relational Databases

Yuping Yang, Mukesh Singhal
Department of Computer and Information Science,
The Ohio State University
Columbus, OH 43210
{yangy,singhal}@cis.ohio-state.edu

Abstract

Equijoin between two relations is one of the basic operations in relational database and a large volume of research have been devoted to it. However, in recent years, there hasn't been a survey which objectively compares a wide spectrum of various join techniques in their relative performances. This survey compares performance and practicality between various join techniques. Main criteria for performance comparisons are disk I/Os. For comparing of practicality, criteria used are easiness and flexibility of implementation. When comparing join techniques, each join technique is evaluated in its full potential, which means that if other techniques are available to enhance this join technique while retaining the main philosophy of it, the later techniques are applied.

The main contribution of the paper is that it confirms the believe that no dramatical performance improvement of three major join techniques (nested loops, sort-based, and hash based) of relational database can be made. The future of join performance improvement in relational database systems lies in more radical approach: parallel join, join index, composite index, and layered database.

Key words: Relational database, query execution, join, join index, equijoin, band join, selection, index, access path, disk I/O, performance.

1 Introduction

There are two parts in query processing in relational database: to find an optimal strategy (query optimization), and to execute (query execution) it. Usually query execution takes much more time than query optimization and the most expensive part in query execution is join. Join execution techniques in relational database is a mature subject and have been extensively investigated for over 20 years, and yet in recent years, due to its extreme importance in commercial applications, much research have been done in this area and some new progresses have been made.

Common join techniques in relational database can be classified into three classes: nested loops join, sort-merge join, and hash join. There have been much discussions on their relative merits. Some new database architectures and join techniques have been proposed in recent years which hold hopes to further improve the performance of join execution in relational database. These techniques include signature method [15, 31, 34], clustering of data tuples and partition of relations [59], join index [28, 61], composite index [33], layered relational database [52], etc. Essentially, the improvements these new techniques made are either better organizing of physical storage of data on the disk to take advantage of some special cases such as sequential disk accesses, or adding some new clever indexing schemes to gain faster access to specified data values. In the proposals of most new techniques, usually some special circumstances are assumed or emphasised which either simplify the comparison or favor the application of the new techniques and favorable performance results are obtained.

This is a valid way of conducting research because of the limit in the scope of the research. This phenomenon brings up the need to objectively compare a wide spectrum of join techniques in order to make clear the relative merits of each technique and future research direction, and to facilitate the selection of join techniques by database system designers and implementors.

In this paper discussions are restricted in join execution of query execution only. The focus of comparisons is the relative performance and the practicality. Main criterion for performance comparison is the amount of disk I/Os for each join technique and the criterion for practicality is the easiness and flexibility (which often means stability in a dynamic environment) of implementation.

Band join as a terminology was introduced in 1991 [43] and can be regarded as an extension of equijoin in ordered data domain. This is discussed in Section 10 due to its similarity to equijoin.

2 Assumptions

When comparing and evaluating join techniques, many techniques are evaluated according to their principal philosophies. This means that in evaluation of a join technique, if other tricks and methods are available to improve this join technique, these tricks and methods are assumed to be incorporated in the join technique, as long as the incorporations are faithful to the original philosophy of the join technique. So each join technique is evaluated by its full potential, not necessarily in the exact same form as originally proposed. Such is the case of hybrid hash join, we assume that dynamic change of roles of two joining relations based on sizes of partitions as that in Grace join can be readily used in hybrid hash join.

In many papers that proposes a join techniques and/or evaluate performances between two or three join techniques, many particular assumptions are made to make the comparison as accurate as possible. Since this paper evaluates and compares a large number of join techniques, we feel the best strategy is to keep it simple: only essential assumptions are made, and the assumptions are consistent across difference join techniques to ensure reliable and fair comparisons.

Let $\mid R \mid$ be the number of pages in relation R. Assuming R and S are two joining relations and $\mid R \mid<\mid S \mid$. Let $R.A$ and $S.B$ be join attributes between R and S, i.e., $R.A$ and $S.B$ share the same data domain. For simplicity of discussion, it is assumed that there is only one join attribute in each relation.

An equijoin between R and S is defined as for each pair of tuples t_r and t_s, $t_r \in R$, $t_s \in S$, if $t_r[R.A] = t_s[S.B]$, then tuples t_r and t_s are concatenated into one joined tuple. Duplications among joined tuples are removed.

Let $\mid R \mid$ be the number of pages in relation R and $\mid S \mid$ be the one in S, and θ be the ratio of the execution times between random and sequential accesses of the disk. Tuples in R and S are of approximately the same length. Let it be that $\mid\mid R \mid\mid$ is the number of tuples in R and $\mid\mid S \mid\mid$ is the one in S. Roughly, $\mid\mid R \mid\mid / \mid\mid S \mid\mid = \mid R \mid / \mid S \mid$.

α is the percentage of tuples in R that can match with tuples in S. $beta$ is the average number of tuples of S that each matchable tuple of R can match, So, $\alpha\beta \mid R \mid$ is the number of pages in the join result.

Let M be the size of memory, and L be the size of the larger relation, then it is assumed that for sort-based joins, at least $M > \sqrt{L}/2$ and for hash-based joins, each bucket or partition can be allocated at least one page in memory as its own buffer area. On the other hand, both R and S are assumed to be far larger than available memory.

If there is no further explanation, it is assumed that $\mid R \mid= 50,000$, $\mid S \mid= 500,000$, $\theta = 200$, $\alpha = 0.05$, $\beta = 3$, and $M = 20,000$.

To distinguish indexes that can help to do selection and are defined on one or several attributes of one relation to more complicated ones such as join index, the former are called selection index. No indexes are assumed unless explicitly stated.

In performance evaluation, only disk I/Os are compared between various join techniques. The unit of disk I/O is disk page, which usually ranges from a half a kilobyte to a few kilobytes. For easiness of comparison, it is assumed that one tuple occupies exactly one page. One reason for this decision is that in current technology, CPU speed usually far exceeds the disk I/O speed and is still growing fast. Another reason is that in today's computer, CPU operations and disk I/O operations can be done parallelly and the execution times are often overlapped. Since in this paper the emphasis is on comparing differences of major cost factors, we can afford to concentrate our attention on disk I/Os only.

Random disk access time is usually orders of magnitudes bigger than sequential disk access time and this difference cannot be ignored. This decision is reflected in various cost formulas and analyses throughout this paper.

The writing out step of the first phase, which is sorting for sort based join techniques and hashing for hash based join techniques, and the reading in step of the the second phase, the joining phase, are assumed to be sequential. If each write operation can write a sufficient large amount of data out to disk, the sequential read/write assumptions are close to the reality. In Section 7, a paragraph is dedicated to discuss this issue.

3 A Taxonomy of Join Techniques

Essentially, join between two relational tables are a binary match operation executed between two potentially very large lists. Without exploiting any clustering in the data and without using any indexing, there can be only one join technique, that is, to compare join attribute value of every tuple in one relation to join attribute value of every tuple in the other relation. This is called the nested loops join technique. The cost of this join technique in terms of disk I/Os is the cross product of the numbers of tuples in the two joining relations and it is often huge: for two relations with $10,000$ tuples each, the cost of nested loops join is $10,000 \times 10,000 = 100,000,000$ tuples. If each tuple occupies exactly one page, this means 100 millions of disk I/Os.

Any other join technique becomes possible only by exploiting some special characteristics of the data and join attribute values distribution. For a general purpose database, there are very few special characteristics of the data can be assumed. One characteristic that often can be assumed is that there is an order among join attribute values and another is that the join attribute values are either clustered or can be partitioned into groups. The first assumption makes possible sort based join techniques and the second assumption makes possible hash based join techniques.

	Looping	Sorting	Hashing
	Nested loop join	Sort-merge join	Hash join
Buffer Tricks	△	GSM	Hybrid hash join Dynamic Destaging Adaptive hash join Partially preemptible join
Block I/O	Nested block join	△	△
Relation partition	Join by fragmentation	Distributive join	Shin's join
Selection index	○	Hybrid join	○
Join index	○	△	△
join page index (pointer-based)	Page indexing nested loop	Page indexing sort-merge	Page indexing hybrid hash Page indexing Hash-Loops Page indexing PID-Partitioning
Signature filtering	△	△	△
Layered approach	△	△	△

Figure 1: The join combination table

Therefore, there are three basic join techniques: nested looping, sorting and hashing. These techniques are somewhat exclusive with respect to one another. Nested looping compares every tuple from one relation to every tuple from the other relation, so in the same execution doing sorting and hashing are obviously wasted efforts. Sorting makes it possible to compare two lists of values in sequence, so no longer is it necessary to compare each tuple from one relation with every tuple from the other relation (nested loops join). Hashing would disrupt the order created by sorting and void the result of sorting.

On the other hand, these techniques can be used together in a join execution: they can be applied for different relations and at different stages of join execution.

Besides characteristics of data, there are other factors that can affect the speed of join execution: existence of indexes and the types of indexes; organization and sequencing of disk I/O; partitioning of joining relations and the methods employed to make the partitioning; sorting of data values and the methods employed to do the sorting; size of memory buffer and the way to use the buffer. However, all these factors would not have any effect without assuming the existence of at least one of the two basic characteristics of the join attribute values as mentioned above.

Figure 1, let's call it the join combination table, is organized in three columns, representing three major join techniques that are essentially exclusive with respect to one another. The join techniques at the cross point of two techniques is the combination of the two techniques. For example, Hybrid join applys both sorting and selection index. For each basic join technique there are many other techniques that can be combined: memory management (Buffer Tricks), I/O management (Block I/O), disk storage management (Relation partition), and indexing. All join techniques in the middle column assume the existence of order among join attribute values. All join techniques in the right column assume the possibility of clustering or grouping among the join Attribute values. In the left column, only the nested loops join that does not combine with any other technique doesn't make use of any assumption on the characteristics of the data. All other join techniques in the left column assume either order exist among attribute values or grouping or partition is possible.

The distinct advantage of the hybrid hash join is that it utilizes memory buffer as much as possible. Since hashing and sorting are complimentary methods, in [54], the same idea used in hybrid hash join was applied to sort based join techniques. So observing relationships between various join techniques and understanding the reason behind the relationship could provide insights into the rule of designing join techniques and give hints on the possible existence of new join techniques. In this sense, the join combination table is very useful. It is by no means exhaustive. The places marked with triangles are possible places to design new join techniques or some join techniques have already been in existence but this survey doesn't cover or simply it is not yet published at all.

4 Nested Loops Join

Simple nested loops join: In this technique [2], tuples of R are read sequentially from disks into memory. For each tuple of R, all tuples of S are read in memory at least once to have their join attribute values compared. R is called the outer relation and S is called the inner relation. R can be scanned (tuples to be read into memory and have their join attribute values compared) sequentially and only need to be scanned once. S is scanned once for each tuple of R.

The disk I/O cost of disk I/O fo a simple nested loops join is:

$$C_{simple\ nest} = \mid R \mid + (\mid R \mid * \mid S \mid) + \alpha\beta \mid R \mid \tag{1}$$

Plug in numbers that are assumed in Section 2, the I/O cost is:

$$C_{simple\ nest} = 50,000 + (50,000 * 500,000) + 0.05 * 3 * 50,000 = 2.5 \times 10^{10} \tag{2}$$

This is a very slow join technique. There are many improvements of it have been proposed and major improvements have been discussed in the rest of this section.

Blocked nested loops join: It is sometimes called the nested block join [46]. It is a very effective and straightforward improvement over the simple nested loops join. The number of times the inner relation S is read into memory is equal to the number of times the memory is refreshed with tuples of the outer relation R. The strategy of nested block join is to hold as many tuples of the outer relation R in memory as possible so that number of times memory has to be refreshed with tuples of R can be minimized. When the available memory of the computer is large but not so large to hold any of the joining relations, this technique can significantly reduce the disk I/Os over that of the nested loops join [46]. The word "block" in nested block join refers to the chunk of tuples from either relation that are read and reside in memory in batch, and should not be confused with the block that is the unit of disk storage. For clearness of discussion, in the following discussion, "block" refers to the unit of disk storage and "partition" refers to the chunk of tuples of either relation that are read together into memory. In nested block join, relations are usually evenly partitioned.

The available memory of the computer can be divided into three parts. One part is allocated as a buffer area for storing tuples from R and is denoted as B_r. Another part is allocated as a buffer area for storing tuples from S and is denoted as B_s. Still another part is used as an output buffer area for temporarily storing joined tuples before they are written out to disk, and is denoted as B_o. One way of organizing the available memory is to allocate $M - 2$ pages for B_r, 1 page for B_s, and 1 page for B_o.

Then, $M - 2$ blocks of tuples from the outer relation R can be held simultaneously in memory. The disk I/O of reading R is not affected by the size of B_r. The number of times to read the inner relation S is greatly reduced by the relative size of B_r and $\mid R \mid$:

$$C_{block\ nest} = \mid R \mid + (\mid R \mid / (M - 2) \mid S \mid + \alpha * \beta * \mid R \mid \tag{3}$$

Plug in numbers that are assumed in Section 2, the I/Os cost is:

$$C_{simple\ nest} = 50,000 + (50,000 * 500,000)/(20,000 - 2) + 0.05 * 3 * 50,000 = 1.25 \times 10^6 \quad (4)$$

Using hashing to reduce the number of comparisons: To reduce the number of comparisons between tuples from B_r and B_s, hashing can be used. Hashing partitions memory resident tuples into buckets and comparisons only to be carried out between tuples of two relations that fall into the same bucket. Only tuples from B_r need to be built into a hash table. The hash table is built by hashing tuples from B_r on their join attribute values and place them into appropriate hash buckets. Collisions in a bucket are usually resolved by chaining of tuples in the bucket. The tuples from B_s are hashed on their join attribute values by using the same hash function. Comparisons between join attribute values of tuple from R and S can be carried out right after each tuple in B_s be hashed and its corresponding bucket in hash table of B_r is known. The hash table of B_r needs only to be built once for the entire S. Usually, the extra CPU cost of hashing can be well justified by the savings in number of comparisons.

Using rocking to reduce I/O: Rocking is proposed by Kim [8]. In nested block join, the entire inner relation S is read from disk into buffer B_s each time the buffer B_r is refreshed with tuples from R. However, beginning from the second time, when start reading blocks of S, there are tuples of S already in buffer B_s and these tuples don't have to be read again if reading is carefully organized.

A simple arrangement can save some disk readings of S: in the first pass to read S file, it is read forwardly from the top to the bottom of the file. In the second pass, it is read backwardly from the bottom to the top of the file, and tuples at the bottom of S file don't have to be read again. In the third pass, S file is read forwardly from the top to the bottom and tuples at the top of S file don't need to be read again. The name rocking comes from the fact that the reading of S file is done alternatively from the top and from the bottom.

Another implementation of the rocking technique [59] is to read S file in a circular manner. This implementation could be useful if forward access is more efficient than backward access, or if backward access is simply impossible. Assume there are m blocks in S file and buffer B_s can hold up to 100 blocks. The first pass is to read blocks 1, 2, ... , m. The second pass is to read blocks $m-99$, $m-98$, ... , m, 1, ... , $m-100$. The blocks $m-99$ through m are reused and no actual reading is performed for them. The third pass is to read blocks $m-199$, $m-198$, ... , m, 1, 2, ... , $m-200$, and no actual reading is performed for blocks $m-199$ through $m-100$.

Suppose there are total M pages of available memory, the size of B_r is $M-1-p$, the size of B_s is p, and the size of B_o is one page. Then, the disk I/O of performing a nested block join is:

$$C_{block\ rocking} = \mid R \mid + (\frac{\mid R \mid}{M-1-p}) * (\mid S \mid - p) + \mid S \mid \tag{5}$$

Regarding all other variables as constants and optimizing on p, one gets:

$$[(\frac{\mid R \mid}{M-1-p}) * (\mid S \mid - p)]' = \frac{\mid R \mid (\mid S \mid - M + 1)}{(M-1-p)^2} \tag{6}$$

the above derivative is always positive. So to minimize the I/O cost, the value of p should be chosen as small as possible, i.e., $p = 1$. However, take into account that sequential access is much fast than random access, p should be chosen somewhat bigger and in this case, rocking makes sense. However, the effect of employing rocking in nested block join is marginal.

Using indexing to reduce I/O for inner relation: Indexes such as B+ tree can greatly reduce the number of disk access to fetch a particular data value if data values are ordered in some way. In general, indexes can help speed up join execution with the cost of using extra disk storage space for index itself.

When indexes are available on the join attribute of the entire inner relation in the nested block join, no longer the entire inner relation has to be scanned for each partition of the outer relation. With index, usually only a few disk read are suffice to fetch matching tuples from the inner relation for each tuple of the outer relation. This could mean a great saving on disk readings if $\mid R \mid$ is far smaller than $\mid S \mid$, and/or join result is small. On the negative side, disk accesses to and through an index are often random and cost at least a few random disk reads for each tuple of R (If $\mid S \mid$ is large). If two joining relations have approximately the same size, this means that disk I/O is at least a few times the size of the outer relation, and this is very costly in time.

The disk I/O is:

$$C_{block\ nest\ index} = \mid R \mid + \theta(\mid R \mid * f)/(M-2) + \alpha * \beta * \mid R \mid \tag{7}$$

where f is a factor that on average, for each tuple of R, f tuples of S are fetched for comparison. If $f = 5$, then,

$$C_{block\ nest\ index} = 50,000 + 200(50,000 * 5)/(20,000 - 2) + 0.05 * 3 * 50,000 = 6 \times 10^4. \quad (8)$$

Much better than nested block join.

Concluding remarks for nested loops join: One of the critical factors in determining the performance of nested block join is the buffer size allocated to each relation. The simple analysis above seems to favor either maximize or minimize one of the buffers and adjust the size of other buffers accordingly. This is not completely true since often the disk storage of a relation is clustered and disk head seek time makes up a large part of disk access time. In order to reduce disk head seek time, it is beneficial to give either buffer of the two relations a comparable size, or to have two relations reside on two separate disks.

Generally speaking, nested loops join techniques are not fast join techniques. The nested block join can perform better than simple nested loops join if large memory is available. If the outer relation R can fit entirely in memory, then the I/O cost of the join execution reduces to just a linear scan of both relations:

$$C = \mid R \mid + \mid S \mid + \alpha\beta \mid R \mid \quad (9)$$

In this special case the nested block join becomes very efficient in comparison with sort based and hash based join techniques.

However, for large databases, very often available memory space of the computer is much smaller than either of the joining relations and in this case, compared to other join techniques, even nested block join is not a fast join technique.

There are advantages of these join techniques and they are still widely used in today's commercial database. One of the main advantages of these techniques is their simplicity and no intermediate relations need to be generated. For joins performed entirely in the memory, such as joins between small in-memory fragments of relations, these techniques remain competitive and therefore, nested loops join or nested block join can be used along with other join techniques such as sort-merge join and hash join. Also, nested loops join (or nested block join) is the only well-known join technique that works for complex join predicates [46].

For many applications, there is no partial order among values of join attributes, therefore neither sort-merge join nor hash join, nor any indexing scheme can be effectively used, and in this case the nested loops (block) join is the only available method. In [46], two kinds of applications were given as examples of such applications. One is the fuzzy comparison between names or addresses, the other is the document recognition [46].

5 Sort-Based Join

sort-merge join This join technique [4, 17] sorts both joining relations on the join attributes into two sorted lists and then merges these two sorted lists. Tuples are joined on the fly in the merging phase. As soon as two tuples from the two relations have their join attribute values match, these two tuples are joined and outputted and merging resumes in the tuples (they are sorted and therefore are ordered by their join attribute values) that follow. Otherwise, the tuple with the smaller attribute value between the two is dropped because it has no hope to be joined with other tuples. A study by M. W. Blasgenm and K. P. Eswaran [3] shows that the sort-merge join can almost always outperform the nested loops join when no index is used. However, if there are too many duplicated join attribute values, sort-merge join can degenerate to the performance of nested loops join [41].

In sort-merge join, it is possible to take advantage of prefetching in the merging phase since tuples are sorted. This feature makes difference in performance if the memory space of the computer is large and fast disk devices are deployed.

Sort-merge join is useful for joins between large relations without indexing supports [46], especially when the join predicate does not offer much filtering (the join result is large). Sort-merge join technique has an additional advantage over nested loops join in that it can efficiently handle inequality join.

The I/O cost of the sort-merge join is [63]:

$$C_{merge} = 2 \mid R \mid log \mid R \mid + 2 \mid S \mid log \mid S \mid + \mid R \mid + \mid S \mid + \alpha\beta \mid R \mid \tag{10}$$

The first two terms are the cost to sort the two relations and the last two terms are the cost to merge the two relations. The factor 2 in the first two terms reflects the fact that the sorted relations have to be written back to disk before merging. Read and write are done mostly sequentially. If $M > \sqrt{\mid S \mid}/2$, a faster sorting technique [1] can be used. Both R and S can be sorted into runs of approximately size of $2M$ by a sequential scan. Then these runs are merged. If enough memory is available, each run can have its own buffer in memory and merging of these ordered runs can be accomplished within one pass. The I/O cost of the sort-merge join becomes:

$$C_{merge} = 4 \mid R \mid +4 \mid S \mid + \mid R \mid + \mid S \mid +\alpha\beta \mid R \mid \tag{11}$$

The first term represents the cost of two passes to sort R into ordered runs (each pass needs both read and write). The second term represents that cost for S. The third term and fourth term represent disk accesses to read in ordered tuples of R and S to merge. The fifth term is the cost to write out joined tuples. All the I/Os are sequential. The large memory is needed to give at least one page of buffer space to each sorted run.

If $M > \sqrt{\mid S \mid}$ (recall that $\mid R \mid < \mid S \mid$), the ordered runs of R and S do not even need to be merged into one ordered list before joining with tuples of other relation. After first scan, each run of R and S is allocated one buffer in memory and tuples in runs are read in memory in sorted order. There are enough memory pages to make this allocation possible because the total number of runs of R is approximately $M/2$, and this number is also $M/2$ for S. In memory, merging happens first within buffers of R, and separately within buffers of S. These two in-memory merging processes can supply tuples of R as S as two ordered lists for merging join. All these can be done in one scan. In this case, the I/O cost is:

$$C_{merge} = 3 \mid R \mid +3 \mid S \mid +\alpha\beta \mid R \mid -2min(\mid R \mid + \mid S \mid, M - \sqrt{\mid S \mid}) \tag{12}$$

The last term is disk I/O saving due to availability of extra memory, i.e., if after allocating at most B buffers there are still extra memory pages, sorted results in these extra pages don't have to be written out to temporary file during sorting phase and then read back in memory again during merging phase. This is the mirroring of the same idea used in hybrid hash join.

Plug in numbers assumed in Section 2, and assume $M = \sqrt{\mid S \mid}$, the I/O cost is:

$$C_{merge} = 3 \times 50,000 + 3 \times 500,000 + 0.05 \times 3 \times 50,000 = 1.5 \times 10^6 \tag{13}$$

Distributive join: Distributive join is proposed by N. Negri and G. Pelagatti [47]. This method tries to save some sorting work by only sorting one of the relations completely, and the other relation is only partitioned. The cost saving is at the sorting phase, not at the joining phase.

First, relation R is completely sorted by join attribute values of its tuples and partitioned in $m+1$ partitions $R_1, \ldots, R_m, R_{m+1}$. Any join attribute value in a tuple of R_i is less or equal to the join attribute value in a tuple of R_j, if $i < j$.

A distribution table is built during sorting and partitioning of R. Let us assume that v_i is the maximum join attribute value in R_i, then the distribution table contains join attribute values $v_1, \ldots, v_m$. The distribution table is used to partition S into partitions $S_1, \ldots, S_m, S_{m+1}$. Any join attribute value in a tuple of S_i is between the value v_{i-1} and v_i, with $v_0 = -\infty$, $v_{m+1} = +\infty$, for $1 \leq i \leq m+1$. Inside each partition S_i, the tuples are not sorted.

In the joining phase, the partitions of R is read into memory one at a time, assuming that the memory can hold each individual partitions of R. When a partition R_i is in memory, blocks of partition S_i are read into memory one at a time. In any given moment, only one whole partition of R_i and one block of S_i reside in memory. The join then takes place entirely in memory between R_i and a block of S_i that reside in memory using nested loops join, benefiting from the fact that R_i is sorted. The size of R_i's should be large enough as long as each of them can fit entirely in memory along with one page as the input buffer of the inner relation and one page as the output buffer.

The disk I/O cost of the distributive join is:

$$C_{merge} = 5* \mid R \mid +3* \mid S \mid +\alpha\beta \mid R \mid \tag{14}$$

The first term includes I/O cost of sorting R (2 passes, sorting results are written back to disk) provided that enough memory is available to give at least one page to each run of R and the second term includes the I/O cost of partitioning S using the distribution table. The first and the second term also include the I/O cost of reading sorted R and partitioned S in memory for joining. Compared with formula 11 and 12, it is clear that distributive join outperforms sort-merge join if memory is less than $\sqrt{\mid R \mid}$ and R is much smaller than S.

Combining indexing with sorting A hybrid join technique was proposed [44] to combine the advantages of indexing and sorting. It is assumed that R is unsorted and there is an index on the join attribute of S. The basic idea is to sort R and access S via index. TIDs are fetched along with join attribute values of the S. This can be done without extra access to disk since TIDs can be kept in the index (index is assumed to be sufficiently small to be kept in memory). If matches are found, then matching tuples of

S are fetched. By combining sorting and indexing, performance improves over pure sort base join when join result is small or $midR \mid$ is much smaller than $\mid Smid$.

The disk I/O cost is:

$$C_{merge} = 5* \mid R \mid + \theta f \mid R \mid + \alpha\beta \mid R \mid \tag{15}$$

f represents the number of nodes to be visited in index for each value accessed through index. Note that this formula is very similar to that of the Distributive join. By comparing these two formulas, it is clear that using index is especially good when $\mid R \mid$ is much smaller than $\mid S \mid$.

Concluding remarks of sort-based join: In general, sort-based join performs better than nested loops join when both joining relations are large, no indexes are provided and especially the join result is large.

When the join selectivity is low, i.e., most tuples fetched are unmatchable, involving these tuples in sorting and merging is wasteful. In this case, nested block join with indexing could be a better choice.

6 Hash Join

There are two basic ideas in hash join. The first idea is the partitioning of both relations R and S by hashing using a hash function ϕ_1. Two relations are hashed into exactly the same number of partitions using exactly the same criteria for partitioning. These partitions are in pairs: each partition of R corresponds to one partition of S and tuples in the partition of R can only match tuples of S in the same pair. The partitioning should be done such that partitions of R are approximately the same size and each individual partition of R can reside entirely in the available memory buffer. This ensures that after partitioning, during the joining phase the other relation S only needs to be read once. The second idea is that in each step an entire partition of the smaller relation R be read into memory and be built into a hash table, i.e., further partitioned into many small buckets inside the memory buffer, possibly using a different hash function ϕ_2. The blocks of the corresponding partition of the other relation S are read in memory one by one and are hashed by hash function ϕ_2 to appropriate buckets of the memory resident hash table of R to be compared for possible matches.

In spirit, hash join follows the divide and conquer principle and aims to turn the join execution between two relations into a series of join executions between corresponding partitions of the relations.

Simple hash join: This is the simplest implementation of hash join [17] and is basically a nested block join with modification that hashing is used for the in-memory processing. To be specific, first as many tuples of R as possible are read in memory an built into a hash table. Then S is scanned and its pages are read in memory one by one. The memory resident tuples of S are hashed and be used to probe the memory resident hash table of R. If matches are found, joined tuples are written out. After scanning S, the remaining part of R is read in the memory and the same process repeats.

The performance of simple hash join is slightly better than the nested block join because of saving in CPU executions and in memory moves of data, but disk I/O cost of the two techniques are the same.

Grace hash join: A Grace hash join [10, 14] technique clearly separates two phases of hash join. In the first phase, each of two relations R and S is partitioned into the same number (N) of buckets by hashing on the join attribute and using the same hash function ϕ_1. These buckets form N pairs. In each pair, one bucket is from R and the other is from S. At least one of the buckets in each pair must be able to fit in available memory of the computer. If not, additional partitionings may need until this is true. In the second phase, in-memory operations as nested looping or hashing is used to perform join between two buckets in each pair.

To ensure an even partitioning, part of memory can be set as output buffers for a large number of buckets with individual bucket size much smaller than available memory. After partitioning has been done, buckets are combined to form larger partitions whose size are close to the available memory size. This is called bucket tuning. The I/O cost of the Grace hash join is:

$$C_{hash} = 2(\mid R \mid + \mid S \mid) + (\mid R \mid + \mid S \mid) + \alpha\beta \mid R \mid \tag{16}$$

If both R and S are much larger than available memory space, a Grace hash join can be expected to outperform simple hash join. But in case that most of the R can be fit in memory, the I/O cost of simple hash join is $\mid R \mid + \mid S \mid$ or $\mid R \mid +2 \mid S \mid$, and may very well outperform the Grace hash join because the Grace hash join has the extra cost of partitioning.

Hybrid-hash join: Instead of using extra memory space to set up more buffers as in a Grace hash join, a hybrid hash join [17] uses extra memory to do joining while partitioning. The outer relation R is hashed on the join attribute and be partitioned into N buckets R_i, $0 \leq i \leq N-1$. Tuples from R_0 (anchor bucket) is used to build an in-memory hash table and the rest $N-1$ buckets are written back to disks. The hash function phi_1 and bucket criteria are so chosen that each bucket of R can fit in available memory. phi_1 is also used to partition S into N buckets S_i, $0 \leq i \leq N-1$. Tuples of S_i is only joinable to tuples in R_i. Again, except S_0, all other $N-1$ buckets are written to disks. Tuples belong to S_0 are used to probe the memory resident hash table for matches immediately after hashing. Tuples of S_0 are dropped unless they can form joined tuples with tuples of R_0. This means that all tuples of S_0 don't have to be memory resident at the same time and a large portion of available memory can be used for R_0. In the second phase, the remaining $N-1$ buckets of R and S are joined through a series of small joins. In each small join, a whole R_i is read in memory and built into a hash table and then blocks of S_i are read one by one in memory to probe the hash table. Again, all tuples of S_i don't have to be memory resident at the same time and a large portion of available memory can be used to hold R_i. These small joins can use, for in-memory processing, either nested loops join or hash join on a smaller scale.

The number of buckets, and thus the number of small joins is determined by the size of the smaller relation R, and independent of the size of S. However, this is true only when partitions are approximately even. In the case that a few partitions of R cannot fit in available memory while all partitions of S can, roles of R and S can be reversed after partitioning phase. Recursive partitioning can be used if neither all individual partitions of R or S can fit in memory. An approach similar to that in a Grace hash join is that in each step of the joining phase, if $\mid R_i \mid < \mid S_i \mid$, then the whole R_i are read in memory and build into a hash table. Otherwise, the whole S_i are read.

The I/O cost of the hybrid hash join is:

$$C_{hybridhash} = (3 \mid R \mid -2 \mid R_0 \mid) + (3 \mid S \mid -2 \mid S_0 \mid) + \alpha\beta \mid R \mid \tag{17}$$

The first term is the cost to read in R, write out hashed result of size $R - R_0$, and read in buckets of size $R - R_0$ to participate in the join. Similar explanation can be given to the second term an the third term is the cost to write out join results.

In a hybrid hash join, it is preferable that the size of bucket R_0 is close to the available memory because the larger the size of R_0, the more tuples can be joined during the partitioning phase. Though it is also preferable that the sizes of the rest of buckets of R

are large, it only affects the number of times to initialize hash tables in memory and is not that critical. If there are total N pairs of buckets after partitioning and M pages of available memory space on the computer, because each bucket of R must have at least one page as output buffer during partitioning of R, it must be that $\mid R_0 \mid \leq M - N - 1$ (one extra page is needed as input buffer for R), and $\mid R_i \mid \leq M - 1$, for $1 \leq i \leq N$. To ensure more evenly partitioning of R, bucket tuning such as that used in the Grace hash join can be used.

A Grace hash method is relatively insensitive to the size of available memory since the amount of memory mainly affects only the number of times hash tables be initialized and does not affect disk I/O. In comparison, hybrid hash join is much more sensitive to the size of available memory of the computer. A hybrid hash join takes advantage of a large memory space and if the memory space is small, the performance of the hybrid hash join may degrade to that of the Grace hash join. Hybrid hash join is also very sensitive to data skew for if the anchor bucket is too small or too large, in such cases either the use of the anchor bucket is ineffective or bucket overflow occurs.

There are some other variations of hash join, most of them aim at improving hash join in case severe data skew result in very uneven distribution of tuples in different buckets.

Dynamic destaging strategy (*DDS*): In standard hybrid hash join, the anchor bucket is pre-determined. This strategy could have problem since the anchor bucket may be too large to fit in memory or too small to be effective. Because usually the data distribution is unknown before join, the decision to choose the anchor bucket should be delayed as much as possible [32].

In *DDS*, the intended bucket size is chosen to be much smaller than the intended partitions to avoid bucket overflow. Actually in implementation not bucket sizes are set, rather, the number of buckets are set so that if data is distributed evenly, the bucket size will be small enough to fit in memory. Each bucket is allocated one page in memory to begin with and will grow as scanning progresses. As memory becomes full, some filled pages of memory buffer should be paged out. The bucket to be chosen for output is a bucket who already has page out, or in case such bucket cannot be found, e.g., the first time when this happens, the largest bucket is chosen. All but one page of the chosen bucket are paged out. The retaining page serves as the output buffer for this bucket and could be an empty page or an partially filled page. This process is repeated during partitioning phase until all tuples in R and S are scanned. The decision of which bucket or set of buckets serves as the anchor bucket is made dynamically. After partitioning of

R and S, buckets are grouped into partitions of approximately the same size. The join phase is the same as that of the hybrid hash join.

The cost formula of DDS is the same as that of the hybrid hash join. The difference is the possible time saving due to reduction of bucket overflows and reduction of re-initialization of hash tables, both are made possible by bucket tunning, and the time saving due to a more appropriately sized R_0, which should be as large as possible, but within the limit of available memory.

Adaptive hash join(ADH): This method is proposed by Zeller and Gray [41] and is similar to DDS, and can be considered to be a generalization of it. This method allows change of memory size during join execution.

In ADH, the number of buckets is fixed and is quite large. Each bucket is allocated to a buffer in memory and a buffer may have several buckets. The number of buckets and the limit of buffer size are predetermined. The large number of buckets are offset by the sharing of buffers so the number of buffers are not too large and initially it is possible to offer each buffer at least one page of memory space. A group of buckets that share the same buffer constitutes one partition. A buffer is the memory resident part of the partition and a partition may also have a disk resident part.

In the first phase, R is scanned and buffers grow as scanning progresses. If a buffer grows beyond its limit, it can be split in two smaller buffers. No new pages in memory are allocated for the newly split buffers: the same memory pages that are used by the old buffer are now used by the two newly split buffers. If a buffer overflows but contains only one bucket, no split is done.

When a buffer overflows (but not exceed its limit yet) and no buckets have been paged out before, and enough memory is available, a new page is allocated to the buffer.

Eventually, buffers used up all available memory and some of the data in memory has to be written out to disk. If no buffer has been outputted before, the largest buffer in memory is chosen to be the buffer to output. All but one page of this buffer are written out to disk. The retaining page serves as the output buffer of this partition. If there is a partition which already has some disk resident blocks, then buffer of this partition is chosen as the buffer to be outputted.

After the scanning of R is finished, scanning of S starts. In this phase, there are buckets of R in memory. Some of these buckets should be written out to disk to make room for output buffers of buckets of S. After that, there are still some buckets of R in memory. These buckets collectively serve as the anchor bucket in a standard hybrid hash join. If a tuple of S falls in the range of memory resident buckets of R, this tuple is used to probe the buckets of R for matches. If matches are found, joined tuples are formed and written to the result output buffer and the contents of the result output buffer are flushed to disk when it is full. If no matches, this tuple is dropped. If this tuple falls within the range of a disk resident bucket of R, this tuple is written to the output buffer of that bucket and the contents of the buffer are flushed to disk when it is full.

This stage of ADH works just like hybrid hash join, except that there may be several memory resident buckets of R instead of just one, and the number of memory resident buckets of R can adjust depending on the availability of memory.

ADH basically follows the same philosophy as that of DDS. Split of buffers is the main new feature added.

Partially preemptible hash join ($PPHJ$): A class of partially preemptible hash joins were proposed in [50]. DDS and ADH can be regarded as restricted versions of $PPHJ$. Here we only discuss the most sophisticated version of it, i.e., with dynamic expansion and contraction of memory buffers and prioritized spooling policy, and name it $PPHJ$. This technique is more flexible than either DDS or ADH. In addition to writing excess blocks of tuples out to disk when memory is full, $PPHJ$ also reads in blocks of tuples when memory space become more available.

As Grace hash join, $PPHJ$ explicitly sets the number of partitions to be $\sqrt{F* \mid R \mid}$, where F is a fudge factor which represents a slightly increase of the space requirement of R due to overhead of making it into a hash table. The purpose of this choice is to strike a balance between the number of partitions (so the number of buffers) and the size of each partition. If R is partitioned evenly, the size of each partition is about $\sqrt{F* \mid R \mid}$.

Like DDS or ADH, $PPHJ$ moves the overflowing buckets of R out of memory. Both DDS and ADH only allows buckets of R to be moved out of memory during scanning of R. $PPHJ$ also allows this to be done during scanning of S (when memory availability decreased). In addition to that, $PPHJ$ also moves buckets of R into memory when memory becomes more available. Unlike DDS or ADH, $PPHJ$ does not combine buckets of R into large partitions and let each bucket itself to be a partition. This decision affects the number of buffers in memory.

$PPHJ$ uses a large number of partitions so the number of buffers is also large. When there is not enough memory to hold all the buffers of R, DDS and ADH flush out some of the buffers using LRU (the least recently used flush out first) spooling strategy. In contrast, $PPHJ$ uses prioritized spooling strategy. Pages from R have higher priority than pages from S and are more memory resilient, i.e., when memory space is not enough, pages of S are more likely to be flushed out of memory and when memory becomes more available, pages of R are more likely to be read in memory.

Experiments [50] shows that if memory availability fluctuates, hybrid hash does not have a satisfactory performance. Unless memory availability fluctuates too fast, $PPHJ$ has a substantial better performance than DDS and ADH in a wide variety of parameter settings. The reduction of execution time comes mostly from dynamic expansion and contraction of buffers. The priority spooling also has some effect on reduction of execution time.

Shin's join algorithm:($SHIN$) In $SHIN$ [56], both joining relations are repeatedly divided by up to five statistically independent hash functions. The number five is not a magic number and it can be adjusted to other appropriate numbers. The buckets obtained from applying the first hash function are divided again by applying the second hash function, and so on. The main idea is that after applying several statistically independent hash functions, the buckets are very small and many of them are empty. Let B_r and B_s be two correspondent buckets of R and S, that is, they are tuples from R and S obtained by applying the exact same number of hash functions in exactly the same sequence and using exact the same criteria. If B_r is empty, then, all tuples in B_s can be safely dropped. The dropping can happen at any stage of the hashing process and the idea is to drop unmatchable tuples as early as possible and when the last hash function is applied, nearly all unmatchable tuples are filtered out. To achieve this, the number of buckets to the power of the number of pairs of hash tables should be sufficiently large. For example, if each hash table has 256 buckets and there are five hash functions, then 256^5 is a sufficiently large number.

Because $SHIN$ uses an extensive and repeated hashing process, in [56] it is recommended to use linked list instead of array data structure for tuple storage in buckets. $SHIN$ does not need preprocess to estimate hash table sizes. It sets up the fixed number of fixed hash table buckets. Clearly the performance of $SHIN$ is linear with respect to the size of the relations, and it is simple and likely to be a stable algorithm. But in worst case, after repeated hashing it could happen that almost no buckets are dropped. In this case, $SHIN$ has cost $10 \times (\mid R \mid + \mid S \mid)$. The relative performance of $SHIN$ compared to other join techniques obviously very much depends on data distributions.

Concluding remarks for hash join: If initially tuples are already fully or partially sorted, sort based join techniques may very well outperform hash join since for sorted data, sort based join execution reduces to one linear scan of each relations.

Hash join can generally expect to outperform sort-merge join [7, 14, 17, 19, 23, 26]. However, mostly their performances differ by percentages rather than factors [54].

In [54], Graefe et al. presented a quite detailed comparison between sort based join and hash join. Their conclusion is that these two classes of join techniques are to a large extent complimentary to each other. The duality stems from the fact that hashing is the opposition of merging. Contrary to previous believes, their conclusion is that criteria to choose between sort based join and hash join is neither relation sizes nor memory size, but the followings:

- the relative sizes of the two relations. This is because hashing stops as soon as every partition of either relation can be fit in memory. In case of sorting, the available memory space vs. the size of the larger relation S is the dominant factor that determines the cost of sorting.

- possibility that partitions generated by hashing are uneven. This is because in hash base join, for all the individual partitions of one relation to be fit in memory, the maximal partition of that relation must be able to fit in the memory. This is not a concern for sort based join.

- number of duplicated values [41]. If there are a large number of duplicated join attribute values, at least the sort-merge join is able to degenerate to the performance of the nested block join while hash join can perform even worse than nested block join due to high overhead caused by heavy hash collisions.

  Since the number of duplicated values in join attributes are positively correlated to the size of join result, an estimated large join result can also be used as a indication that favors use of sort based join.

- opportunities to explore interesting ordering, such as the case that the physical storage of tuples on disk are already be sorted based on the order of join attribute values. Sorting is the primary source of disk I/O in sort based join execution and if sorting has been done, the sort based join execution reduces to a linear scan of two relations. In addition, the request that join results to be sorted also favors the use of sort base join.

In most cases, hash join outperforms sort-merge join by some noticeable percentages, especially when the smaller relation R can be partitioned evenly and is much smaller than the larger relation S. However, in case of heavy data skew, i.e., the hash buckets generated by hashing have very different sizes, sort-merge join could outperform hash join.

7 Some Speed Up Techniques for Join Executions

This section discusses some general techniques that can be used along with other join techniques to speed up join executions.

Bit vector filtering: Join execution can be improved by using bit vectors [7, 16, 21, 25, 38] for small or medium sized joins. Before performing join, an vector of n bits is initialized and each bit is set to 0. During the first phase (sorting phase for sort-merge join and hashing phase for hash join), each join attribute value in R is hashed and the result is used to set bits in the bit vector. Then, during scan of S, join attribute values in relation S is hashed using the same hash function. The hashed results are compared with the bits in the bit vector. If at least one bit in the hash result can't match the bits in the bit vector, the corresponding tuple of S is dropped. The purpose of doing this is to filter out tuples of S that will not match tuples of R in an early stage of join execution. If R is much smaller than S (this means bit vectors are created from not too many values from R), and a large number of tuples of S can be filtered out by bit vectors, the saving of disk storage and processing by using bit vector filtering is significant.

The effect of this filtering depends on data quantity, data distribution, the choice of hash function, and whether it is implemented in hardware or software (bit vectors as arrays in memory). If only one or a few bit vectors are used and implemented in hardware [7], the filtering may be very effective for small or medium sized data files, and may not be effective if the sizes of two relations or the number of distinct attribute values are too large. This is because if the size of data files are very large, the bit vectors could be inundated by set bits. For large data, array of bit vector can be used to do the filtering and this array can be kept in memory while scanning S. This method actually has another name called signature method [11, 20]. This is slower than implementing filtering using special hardware [7]. Still, using signature method can achieve significant reduction of the size of S and hence speed up the join execution.

A special type of bit vector is called Babb array [7]. A boolean array is built while scan R and each bit in the array corresponds to a hash bucket. The bit is set when there are tuples of R be hashed into its corresponding bucket. While scan S, tuples are checked against the bits in the Babb array and dropped if the corresponding bit is not set.

Semijoin: Semijoin is initially proposed in the distributed database environment to reduce the transmission of data between sites when performing join [9]. However, it can be applied to centralized system as well. The semijoin of S with R is performed as follows:

1. project R on its join attribute, obtain $\pi(R)$.
2. join $\pi(R)$ with S. The result $semi(S)_R$ of the join is the set of tuples of S that joinable to tuples of R.

After this, $semi(S)_R$ can be used to join with R to achieve the effect of an equijoin. If the size of $semi(S)_R$ is small, this practice can substantially reduce the number of tuples of S to be further processed.

Assume that $\pi(R)$ can fit in memory, then the I/O cost of join using semijoin is:

$$C_{semihash} = 2 \mid R \mid + \mid S \mid + \theta\alpha\beta \mid R \mid \tag{18}$$

The first term include the cost to scan R and project it to obtain $\pi(R)$ and also the cost to scan R to join with $semi(S)_R$. The second term is the cost to scan S and obtaining $semi(S)_R$. The third term is the cost to write out join result.

Not only join attribute values themselves, but also signatures made from join attribute values can be used to filter out unmatchable tuples of S while scan S. This is a similar but more generalized method than semijoin. Other more general methods of semijoins can be found in [18, 26].

Eliminating random I/O: Hash join has two phases: partitioning and joining phases. In [62], disk I/Os are divided into four phases: PR, PW, JR, JW. PR represents partitioning phase read, PW represents partitioning phase write, JR represents joining phase read, and JW represents joining phase write. For convenience of comparison, in this paper we assume PW and JR are sequential. Actually, this assumption is not totally true. The proposed optimization targets reducing random disk I/Os on two phases: PW and JR.

In [62], hash buckets are comprised of bucket segments. Each bucket segment is a physically contiguous storage area. Several bucket segments can form a write group.

The memory for performing join operation are divided into three parts: IB, the input buffer, SB, the staging buffer, and OB, the output buffer. The staging buffer holds bucket contents in the partitioning phase and hash table in the join phase.

Initially, before partition, no memory pages are allocated to SB. As tuples from either relation be read into IB and be hashed into buckets, memory pages are being allocated to SB and being linked to buckets. When a new tuple is hashed to a full bucket and no free memory pages are available for allocation, some pages in some buckets have to be written out to disk so that those memory pages can be reclaimed for further allocation. The key technique adopted in [62] is to write out as much a collection of contiguous buckets as possible so that during the joining phase read, as much blocks as possible can be read sequentially. All buckets are organized into write groups. Each write group may have one or more buckets. On disk, each write group is allocated a contiguous area. When SB becomes full, one or more write groups are chosen to be written out to disk in one batch write. Usually, all bucket segments in a bucket of a write group are chosen to be written if the write group is chosen. To extend the length of sequential read as much as possible in joining phase read, usually, the largest write group is chosen to be written out.

Let n_{wg} represents the number of write groups, H_s represents the number of buckets, A_{ran} represents the cost of an average random access to disk, A_{seq} represents the cost of an average sequential access to disk. For convenience of analysis, it is assumed that all write groups are about the same size. Then, the cost to read one write group into memory in the joining phase is [62]:

$$A_{ran} + (\frac{\mid R \mid}{n_{wg}} - 1) * A_{seq} \tag{19}$$

Let $\theta = A_{ran}/A_{seq}$. The cost to read relation R into memory in the joining phase is [62]:

$$C(JR_R) \leq \sum_{t=1}^{H_s} C(read\ \ write\ \ group\ \ of\ \ t) = H_s(\frac{\mid R \mid}{n_{wg}} + \theta - 1) * A_{seq} \tag{20}$$

It is claimed in [62] that the performance of their technique is several times better than that of Dynamic-Hashing GRACE Hash-join [32, 35].

Orders of executions: It has long been accepted that if a query has projection, selection and join operations, it is usually more efficient to perform projection and selection before performing join. This practice may greatly reduce the size of data entering join and since join is the most expensive operations among all relational algebraic operations, the overall execution time of query may be greatly reduced.

Only recently the order of executions between join and group-by has been investigated [58]. The benefit to push down group-by is similar to push down executions of selection and projection.

A difficulty that is special to exchange order of group-by and join is that the exchange may not always be valid and the condition under which the exchange is valid could be very expensive and even impossible to test [58]. So, in [58] a fast algorithm to test a sufficient condition of the validity of order exchange is developed.

8 Parallelism and Multiprocessors

The parallel join techniques can be employed depends very much on the hardware architectures. Here the discussions are limited on join techniques in two classes of machines: shared-nothing and shared-everything parallel machines. The current trend in technology is that the share-nothing architecture is wining up hand among different parallel architectures because the easiness in scalability and the high bandwidth (up to 200MB/s) in interprocessor network [51].

In analyzing parallel join, we distinguish between I/O cost and time cost. I/O cost is the amount of I/Os that needs to be performed, and time cost is the time to perform that amount of I/Os, possibly parallelly by many processors.

8.1 Shared-Nothing Parallel Machine

It can also be called distributed memory or message passing parallel machine. It is assumed that each processor has its own local memory and disk [22] as shown in Figure 2, and pairwise interprocessor communication cost ($PICC$) is significant enough to be considered in the overall join cost. Each processor in a parallel machine has much less processing capacity ($MIPS$) compared to the processor in a single processor machine, but the collective processing capacity can be much higher than that of the single processor machine.

Factors that affect the speed up in using parallel join: $PICC$ and the synchronization overhead of various join techniques play very important role in determining

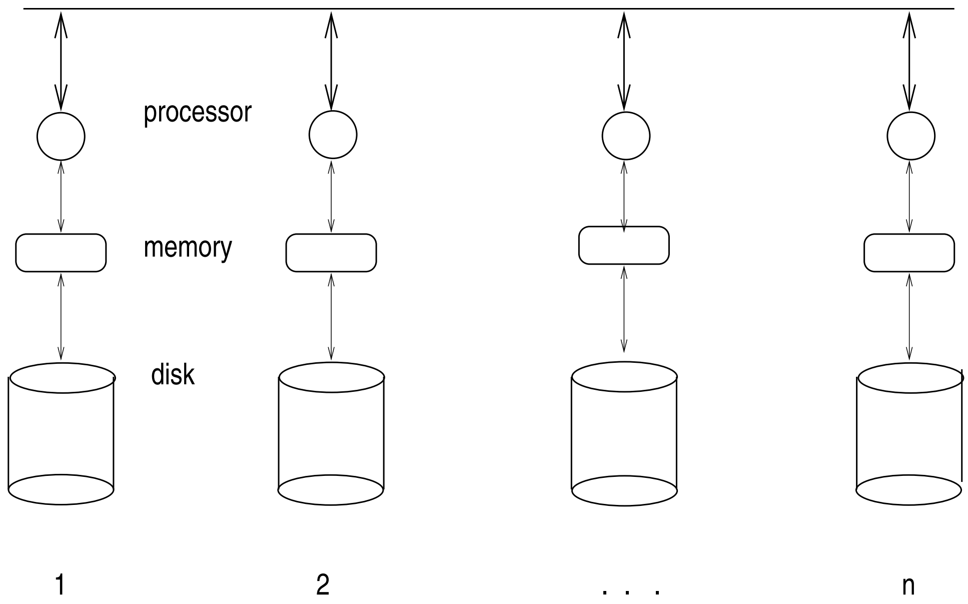

Figure 2: A share nothing parallel machine

relative performances of various parallel join techniques. For shared nothing parallel machine, not only time cost is important, but also communication cost. Communication cost is the amount of communication that needs to be conducted. If there is enough bandwidth in the communication network of the parallel machine, communication cost can be ignored. However, if the communication bandwidth isn't enough, a join technique that has larger communication cost is more likely to be slowed down due to congestion in the communication network of the parallel machine. These factors can be analyzed in static. Other important factors are sensitivity to data skew in the join attribute values and the stochastic nature of join executions [37]. These factors are statistic in nature and are usually analyzed through simulations.

Comparisons of three classes of parallel joins: The nested loops join can be fully parallelized. The speed up of join execution is almost linear with respect to the number of processors. In parallel nested loops join, first each processor read its own part of R from local disk, and then read in entire S from local disk as well as disks of all other processors to do the join. Assume γ is the ratio of time spent on transporting data over communication network between processors and disk I/O, both are assumed to be sequential. Let n be the number of processors. Then, the time cost, including both transporting data and disk I/O of parallel nested loops join is:

$$C_{parallel\ nest\ time} = \mid R \mid /n + (1+\gamma) \mid S \mid + \alpha\beta \mid R \mid /n \tag{21}$$

The communication cost, i.e., the total amount of messages that have to be transported over the network is:

$$C_{parallel\ nest\ communication} = n(1-1/n) \mid S \mid \gamma \tag{22}$$

The significance of the communication cost is: if the speed and topology of the communication network allows sufficient bandwidth compared to the communication cost, time cost is roughly what the time cost formula predicates. Otherwise, the time cost can be much bigger due to congestion in the network.

In parallel sort-merge join, first each processor sort parts of R and S in its local disk, then the sorted run from local disk are merged with sorted run from other processors. The merge is repeated until the total number of sorted runs of R and S in the join is less than the number of pages in memory of one of the processors in the system. At this time, all sorted runs of R and S are read in sequence in memory of this processor to be joined.

Assume $M > \sqrt{\mid S \mid /n}$, the time cost of parallel sorted merge join is:

$$C_{parallel\ sort\ time} = 2(\mid R \mid + \mid S \mid)/n + (2+\gamma) log_{M/2}(\mid R \mid + \mid S \mid)/n + \alpha\beta \mid R \mid /n \tag{23}$$

The first term is the time to sort tuples in local disk of each processor and the second term is the time to read and transport sorted runs from individual processors to do $M/2$ way merge repeatedly, and the third term is to write out join result to the local disk of a processor which is selected as the one to do the final joining of tuples.

The communication cost over the network is:

$$C_{parallel\ sort\ communication} = (\mid R \mid + \mid S \mid) + (log_{M/2} \frac{\mid R \mid + \mid S \mid}{2M})\gamma \tag{24}$$

This is much lower than that of parallel nested loops join because $\mid R \mid$ is usually smaller than $\mid S \mid$, and $log_{M/2} \frac{|R|+|S|}{n}$ is usually a small number. So parallel sort-merge join is likely to be faster than parallel nested loops join when data size is very large, to be more specific, parallel sort-merge join is less likely to be communication bandwidth bounded.

In case of small data sizes, parallel sort-merge join often perform inferior to parallel nested loops join [5] because in this case, the system is not communication bounded and the parallel sort-merge join cannot fully utilize all processors and memories at the later stage of joining.

In parallel hash join, first each processor hashes its part of R and S in its local disk into buckets, and then sends all buckets except one to disks of appropriate processors. After that, each processor joins local buckets of R and S. Since after partition, all possible joining tuples are between local buckets of each processor, no more communication is needed.

The time cost of parallel hash join is:

$$C_{parallel\ hash\ time} = (4+\gamma)(\mid R \mid + \mid S \mid)/n + \alpha\beta \mid R \mid /n \tag{25}$$

The first term include the time to hash R and S locally in each processor, assuming the hashing is being done separately for R and S. The hashing needs one read and one write for each tuple. The first term also include the time to read, transport and write data tuples to disks of appropriate processors. The time cost of parallel hash join is lower than that of the parallel sort-merge join.

The communication cost is:

$$C_{parallel\ hash\ communication} = (\mid R \mid + \mid S \mid)\gamma \tag{26}$$

It has even less communication overhead than parallel sort merge join. So in case that the system tends to be communication bandwidth bounded, parallel hash join performs even better.

Parallel hash join is the best among three major classes of parallel join techniques [22] in case of large data and little data skew.

Vulnerability of parallel hash join: Parallel hashing join in shared-nothing parallel machine has been demonstrated to be very sensitive to data skew and the stochastic nature of the processing time (caused by random placement of original data set on the disk, random delay in communication, random nature of locations of disk heads, etc.) in shared-nothing parallel machine. In [37], it is shown that with 5% data skew, even if the collective processing power of a multiprocessor architecture is 10 times that of a single processor architecture, there can be no performance advantage of using parallel hash join. In [37], total of 300 processors and each processor has 2 MIPS were used in a parallel architecture. This is compared to a processor of 60 MIPS in a single processor architecture. An even distribution means each processor should have about 0.33% of total data. 5% data skew means that instead of 0.33%, one of the processor has 5% of total data, 15 times higher than an average figure. The 5% data skew also means that a processor in a parallel architecture with only 1/30 processing power is to process 1/20 of data. All these figures are compared to a single processor architecture. The work of the parallel architecture is not complete unless this processor's work is complete.

The stochastical join processing time also exact a similar penalty on performance of parallel hash join. One needs to be cautious about the conclusion of very high performance sensitivity to data skew and variations of processing time obtained in [37]. They used a large number of very small processor to compare with a very fast processor. If the number of processors be lowered and the number of processing capacity of each processor be increased, the sensitivity as shown in their experiment can be much reduced. Actually, a top notch fast processor costs dramatically higher than a reasonable fast processor, so it is conservative to assume the collective processing capability in a parallel machine is only 10 times that of a single processor machine, and it is also conservative to assume in a parallel architecture, each processor has only 1/20 of processing capacity as a good processor in a single processor architecture.

Effects of using semijoin as preprocessing: Semijoin used in a shared-nothing parallel machine for preprocessing in parallel hash join can be implemented as follows [39]. Let R and S be two relations in the join and $\mid R \mid \leq \mid S \mid$. In disk of each processor i resides R_i which is a partition of R and S_i which is a partition of S. For each processor,

1. read R_i.
2. project the join attribute values from R_i and sort them. Sorting takes no extra work since projection involves sorting to eliminate duplicates.
3. send local set of join attribute values to all other processors.
4. wait to receive all join attribute values from other processors.
5. merge all the sorted sequences and eliminate duplicates, obtain $R.A$.
6. read S_i.
7. restrict S_i by $R.A$, obtain $semi(S)_R$.

After this step, R and S are joined by sending either $semi(S_i)_R$ or $semi(R.A)_{S_i}$ to all other processors.

If $semi(S_i)_R$ is sent, upon receiving it, the join can be readily performed. If $semi(R.A)_{S_i}$'s are sent, upon receiving them, $semi(R_i)_S$ can be obtained. If on average $semi(R_i)_S$ is much small than $semi(S)_R$, then it is better to send $semi(R_i)_S$ over the network to do the join. In the second approach, the saving of amount of communication is traded by an extra round of sending $semi(R.A)_{S_i}$ over the network. If the system is not communication bounded, the first approach is faster. Otherwise, the second approach wins.

The benefit of using semijoin preprocessing is also very sensitive to data skew. It is shown that with even 5% data skew, the use of semijoin has only a marginal effect (figure 14 in [39]). The reason for this is that data skew tends to move the bottleneck of processing from communication network to individual processors that are overburdened. Again, here one needs to be cautious about the conclusion made in [39] since it is drawn upon a parallel machine with a large number of very small processors. The sensitivity would be less had they increased processing capacity of each processor and decreased the number of processors.

8.2 Shared-Everything Parallel Machine

It can also be called the shared-memory multiprocessor parallel machine(SM). Instead of communication costs in a shared-nothing parallel machine (SN), contention for shared-memory is the major limiting factor of performance in SM. Work on parallel join techniques on SM in open literatures is far less than that on SN. Parallelism in SM is achieved by executing independent and parallel threads of processes on available processors. Since processors share a common memory space, parallel join techniques on SM are much less sensitive to data skew than those on SN.

Separate I/O channels Since we are analyzing I/O cost, the I/O part of the assumption of the parallel machine is essential to the evaluation. Let's assume that each processor can read and write the disk in parallel with other processors. In other words, there are n processors and n I/O channels, each control separately by a processor as shown in Figure 3. Also, for the following analysis, it is assumed that no disk head contention takes place. This is an ideal situation and can be actually achieved by using many physically different disks, each for one I/O channel.

The main difference in performance between shared-nothing and shared-everything

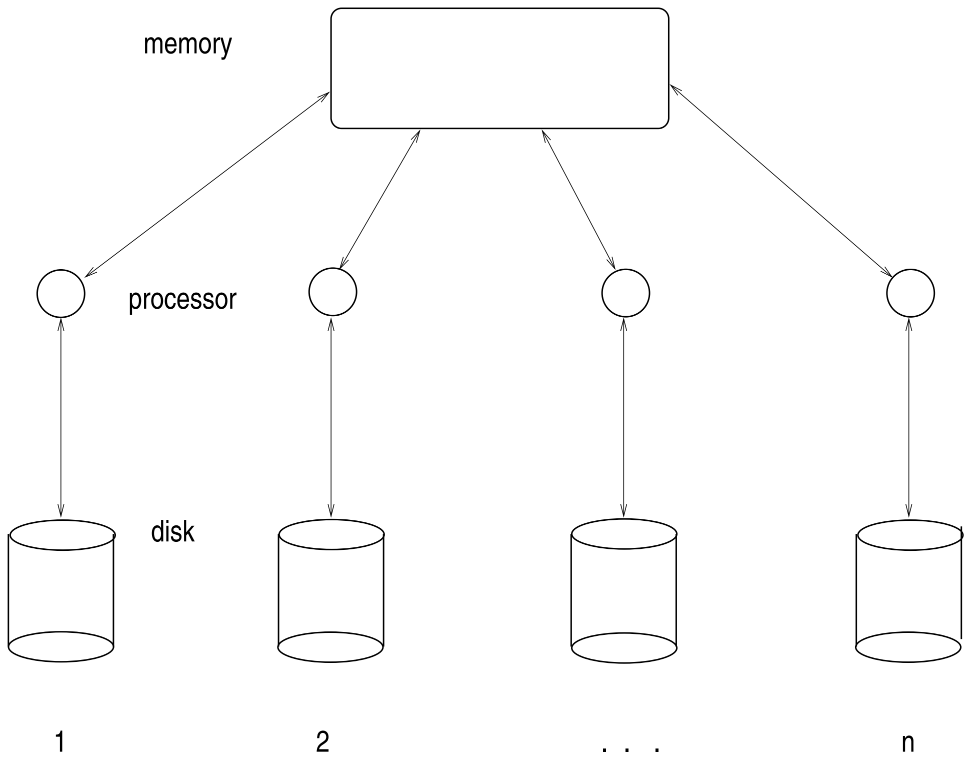

Figure 3: A share memory parallel machine with separate I/O channels

parallel machines is the communication cost. The above analysis shows that with or without sufficient communication bandwidth, parallel hash join generally outperforms other two major parallel join techniques. So here we concentrate our attention only on parallel hash join.

In [55] a SM with up to 10 processors was used to test parallel join techniques. Three parallel hash join techniques were tried: Hash loops, Grace and Hybrid hash.

Parallel Hash loops join reads as many blocks of R (the smaller relation) as possible in memory. The following is done in parallel: on each processor, join attribute values of tuples are hashed and pointers to tuples are inserted in buckets of a hash table. There is only on hash table and it is used by all processors. Then, blocks of S are read in memory and be used to probe the hash table of R. Matched tuples are written to output buffers and unmatched tuples of S are dropped. This constitute one pass to S. Then, if there are still blocks of R remain unexplored, this process is to be repeated until the whole R is processed.

The time cost is (only I/O times are counted):

$$C_{sh_mem\ hash\ loops} = \frac{\mid R \mid}{M-n}(\frac{M-n}{n} + (\frac{\mid S \mid}{n}) + \alpha\beta \mid R \mid /n \tag{27}$$

The first term has two factors. The first factor is the number of partitions to be made on R due to limited memory, and is also the number of times to re-initialize hash tables. $M-n$ in the denominator represents the need to set aside n pages of memory for each of the n processors. The first term of the second factor is the time to read in memory a partition of R by n processors. The second term of the second factor is the time to read S in memory by n processors. The second term is the time cost to output results.

Grace hash join has four phases: the first two phases are to partition R and S by hashing in the memory, and the last two phases work as that in hash loops join for each partition. In each phase in-memory processing are being done parallelly.

The time cost is (only I/O times are counted):

$$C_{sh_mem\ Grace\ hash} = 3\frac{\mid R \mid + \mid S \mid}{n/2} + \alpha\beta \mid R \mid /n \tag{28}$$

It is assumed that during partitioning phase, half of the processors are working to partition R and the other half is to partition S. In the first term, the factor 3 represents one read from disk to do partition, one write to disk to write the partition results, and one read from disk in the joining phase. The second represents I/O to write out the join result. In Grace hash join, it is assumed that memory is enough for each processors to set up its own input and output buffer pages.

Hybrid hash join improves upon Grace hash join in that a large portion of memory is reserved for the first partition R_0 of R. When scan S, tuples of S that fall in S_0 will be used to probe hash table of R_0 immediately and do not have to be written out to disk.

The time cost is (only I/O times are counted):

$$C_{sh_mem\ Hybrid\ hash} = \frac{3 \mid R \mid -2 \mid R_0 \mid}{n/2} + \frac{3 \mid S \mid -2 \mid S_0 \mid}{n/2} + \alpha\beta \mid R \mid /n \tag{29}$$

The first term is the time to read in tuples of R, write out partitions after hashing, and then read in tuples of R again to do join. R_0 is the anchor partition and tuples in R_0 do not need any more I/O once they are in memory, so amount of $2 \mid R_0 \mid$ I/Os are subtracted from the total amount of I/Os. It is assumed that only half of all processors are involved in I/Os for R, thus the denominator is $n/2$. The similar explanation can be given to the second term. The third term is the time to write the join results.

Shared I/O channels Although CPU time were considered in [55], it is still the I/O time that dominate the overall time of join. If a shared-memory parallel machine has only one or a few shared I/O channels as shown in Figure 4, the general behavior of the parallel machine when performing join is not that different from a single processor machine with a very fast processor. The join execution is for most part I/O bounded due to high combined processing capacity of using many processors. This is confirmed by [55]. With their parameter setting, they found that there is little effect on performance in increasing number of processors–of course, it is I/O bounded. The response time of Hash loops join and Hybrid hash join reduced substantially when the amount of memory increases while this increase has virtually no effects on Grace hash join– this is exact as the behavior of a single processor machine.

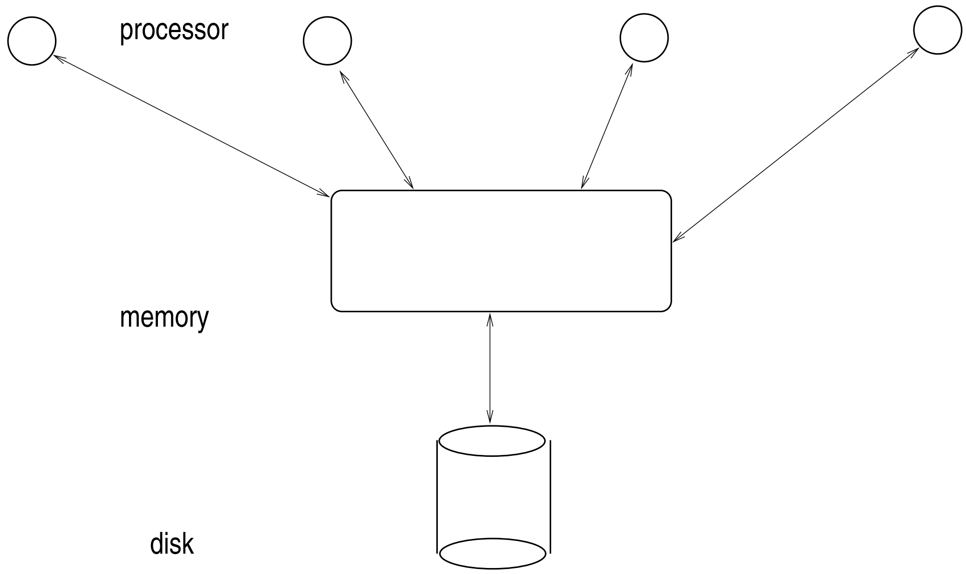

Figure 4: A share memory parallel machine with shared I/O channels

Nevertheless, some very useful observations from [55] are worth mentioning with regard to their influences on performance:

- it is important to use pointers as much as possible to reduce moving tuples in the memory.
- locking granularity is an extremely important issue in SM. Locking granularity should be as small as possible in space and as short as possible in time. It is observed that the processing time (other than I/O) is even increased with more

memory added in parallel Hybrid hash. This is because that the whole R_0 used a single lock, and they fixed the size of R_0 to be 50% of the available memory. Had R_0 be split into many smaller buckets and separate locks were used, the contention would be much less.

- Hybrid hash join has the best performance for relatively small (less than 20% of $\mid R \mid$) memory sizes.
- surprisingly, Hash loops algorithm has the best performance when memory size varies from 20% to 80% of $\mid R \mid$. This is because in Grace hash algorithm and Hybrid hash algorithm, the time spent in the partitioning phase overweighted the saving of time during joining phase due to partition of relations.
- with their parameter setting, they found the effects of data skew to Grace hash join and Hybrid hash join are small. Compared with [39], the insensitivity to data skew reported in [55] not only comes from the difference between architectures of SM and SN, but also comes from the fact the in [55], fewer processors are used in the parallel machine.

8.3 Shared Virtual Memory

Shared virtual memory [36, 42] SVM provides a single virtual address space in a shared-nothing parallel machine. The main reasons that SVM is useful are [51]:

1. ease of coding
2. ease of conceptualization of shared data structures.

Knowing the sensitivity of parallel hash join to data skew, Shatdal et al. proposed a parallel join technique using shared virtual memory [51] to alleviate the effect of data skew in S, the probing relation. The novice is in the joining phase. For a processor p_1 finished its own share of work ahead of others, it randomly pick a busy processor, say p_2, to help out. Stream of local tuples of the inner relation S read from local disk at p_2 then be split between p_1 and p_2. p_1 uses tuples of S from p_2 to probe hash table of R at p_2. Initially, this causes a page fault to read a page of hash table from p_2's local memory to p_1's. As pages of hash table in p_2 be copied to p_1, the page fault will decrease and eventually stop. p_1 is unaware of the locality of the pages of the hash table during this process and this greatly simplifies the programming complexity. The sharing of hash tables is in read-only mode and therefore there is no conflicts between executions of p_1 and p_2.

To reduce the effect of data skew in the partitioning phase, in building relation R, Shatdal et al. suggested to use the idea proposed in [48], which is to use range partitioning instead of hashing. The cutoff values that ensures approximate equal number of tuples in each range can be obtained by sampling. However, this practice add extra processing time if data skew is not severe in R.

9 Structures Specially Designed for Join

9.1 Generalized Access Path Structure

Idea to design special indexing structure to help join operation came as early as 1978 [6] in the form of a binary link. A binary link connects together matching tuples from both operand relations and often be implemented as chains of TIDs or physical pointers (storage addresses). The binary link reduces execution cost for a join in that a selection in the binary link can replace expensive join operation between operand relations. In a sense, a binary link is a data structure of a pre-computed join.

Let R be a relation with attribute A_1, ... , A_n. In [6], an image is defined on a individual attribute A_i as "a mapping from values in A_i to those tuples in R which have those values for the ith attribute". An image is implemented by an (clustered or inverted) index. In [6] the concept of image and binary link were combined to propose a combined access path structure and a generalized access path structure. A combined access path structure is an index, e.g. a $B+$ tree, implemented image on an attribute of a relation, with index nodes populated with not only TIDs or physical pointers of the tuples in this relation, but TIDs or physical pointers of matching tuples of other relations as well. The difference between combined access path structure and generalized access path structure is that the later also stores the value of the join attribute, along with TIDs or physical pointers.

This is a data structure that essentially stores a pre-computed join and can vastly speed up join execution. However, it is also a very heavy data structure: each node of it contains a lot of information and in the worst case, the number of nodes could be the cross product of the numbers of tuples in two joining relations.

9.2 Join Index

Similar idea re-appeared in 1987 in the form of join index [28], which is simpler than the generalized access path structure. Join index is a relation with only two attributes. Each tuple in the join index contains two TIDs of two matching tuples from the two operand relations. Join index is also a pre-computed join and can turn join operation into selection operation. A join index needs to be clustered for fast access if it cannot be fit into memory. Let R and S be two joining relations and a tuple of the join index is in the form of (r, s), where r is a TID from R and s is a TID from S. The clustering should be done on either r or s. If more space is allowed, a better solution is to have two copies of join indexes, each clustered along one of the attributes r or s. The copy of the join index clustered on r (or s) makes join execution from R (S) to $S(R)$ very fast. To further speed up the performance, each copy can be implemented by an index, for example, a B_+-tree.

The difference between a binary link and a join index is that in the binary link, the connection between the matching tuples are, as claimed in [6], often implemented by chaining of TIDs or physical pointers while join index does not restrict the form of implementation. Join index can be characterized as a concept at an appropriate abstraction level. Th join index does not restrict the implementation details. In fact, the combined access path structure or generalized access path structure are, in essence, join index added with clustered or inverted indexes.

Both generalized access path structure and join index serve the purpose of turning a join execution into a selection and thus achieve the desired speed up. The achieved speed up can be far greater than using clustered index or inverted index, which are primary designed for speed up of the selection operation. However, the drawback is that both the generalized access path structure and the join index could require very large storage space.

9.3 Join Page Index

This technique is a generalization and modification of the pointer-based join techniques proposed in [40]. Though pointer-based join techniques are mostly be considered for Object-oriented databases, they could be used to improve the performance of relational databases. In relational databases, a pointer to a tuple can be just the physical address of the tuple. Let this address be called a physical TID, or simply TID. Such pointers could exist in forms of foreign key-primary key relationship. In [40], it is claimed that pointer-based join techniques are not suited for select-project joins in which the selection predicate on one of the relation is a more restrictive one, since in this case, the most efficient way to execute join is first to perform selection, and then travel opposite to the direction of pointers to perform join. This becomes a restriction because they assumed that pointers are one-directional and only pointing from tuples of one relation to tuples of the other. If bi-directional pointers are used, either in the form of actual pointers in both relations, or in the a separate data structure such as the join index, this restriction can be removed.

For uniformity of discussion, let's assume that pointers in a relation are page addresses of matching tuples in the other relation, denoted as PID. These PIDs are stored along with the tuples of this relation and such is the case for both relations. Each tuple can have multiple PIDs. Let this PID data structure be called the join page index to emphasis the similarity with the join index [28]. Of course, join page index can be stored separately from the relations, just like a join index.

Each of nested loops join, sort-merge join and hybrid hash join has its own counterparts in page indexing join techniques.

Page indexing nested loops join(PNL): Tuples in one of the relations, the outer relation, are read into memory and their PIDs are examed. The pointed tuples in the other relation, the inner relation, are read into memory when needed. The reading of outer relation is sequential and could be totally random for the inner relation.

Assume α is the percentage of R tuples that match with the other relation and each tuple in R has on average β matching tuples in the other relation. The random access factor is θ. Then, the disk I/Os of PNL is:

$$C_{PNL} = \mid R \mid + (\theta + 1)(\alpha\beta \mid R \mid) \tag{30}$$

In the second term, $\alpha\beta \mid R \mid$ is the size of the join results. A factor of $(theta + 1)$ exists because these matching tuples are read into memory using random disk accesses and write out disk in sequential. Note that the I/O cost is not affected by the size of S. If $\mid R \mid$ is $50,000$, α is 10%, β is 1.2, and θ is 200, then the I/O cost is:

$$\begin{aligned} C_{PNL} &= 50,000 + (200 + 1) * 0.1 * 1.2 * 50,000 \\ &= 1,256,000 \end{aligned} \tag{31}$$

This is tens of thousands times faster than the simple nested loops join. This is no wonder since it is a nontrivial work to construct the page indexes and to store them along with the tuples. However, if the size of join result is large, i.e., α and β are large, the comparison may not be so favorable to PNL. As nested loops join, PNL does not require large amount of memory to operate.

Page indexing sort-merge join (PSM): PNL generate a lot of random disk I/Os when reading S. To improve upon this, the tuples of the outer relation R could first be sorted by their PIDs instead of join attribute values before matching starts. After sorting, the tuples of the outer relation R are physically placed as close as possible if their pointed tuples in the inner relation S are close to each other. The sorting could cost many disk I/Os but the reading of the inner relation S could be made more sequential. The outer relation should be chosen as the smaller one of the two relations to reduce sort time. Let the random access factor in this case be θ_{sort}. The best sorting algorithm takes $4 \mid R \mid$ disk I/Os: first sort R in runs of size approximately $2M$ and this can be done in one scan. Then merge these runs into a single sorted list. This can also be done in another scan. For both scans the read and the write are sequential. Finally, tuples in sorted sequence are read in memory to be joined. The overall I/O cost is:

$$C_{PSM} = 5 \mid R \mid -2 \mid R_0 \mid +(\theta_{sort} + 1)(\alpha\beta \mid R \mid) \tag{32}$$

Tuples in R_0 are sorted and R_0 is part of R. In the final stage of sorting, if enough memory is available, some of the sorted tuples don't have to be written out to disk and therefore no need to be read in during the joining phase. This is an idea similar to that of the hybrid hash join and can save up to $2 \mid R_0 \mid$ disk I/Os. θ_{sort} should be much smaller than *theta* and that's the whole purpose of involving sorting. If $\mid R \mid$ is $50,000$, $\mid S \mid$ is $500,000$, $\mid R_0 \mid$ is $10,000$, α is 10%, β is 1.2, and θ_{sort} is 10, then the I/O is:

$$\begin{aligned} C_{PSM} &= 5 * 50,000 - 2 * 10,000 + (10 + 1) * 0.1 * 1.2 * 50,000 \\ &= 236,000 \end{aligned} \tag{33}$$

The performance is better than that of the PNL due to less randomness when reading tuples from S. Compared with formula (31), it is clear that the advantage of sorting will be more prominent if the join result is large, i.e., α and β are large. For PSM, large memory is good: if the whole R can reside in memory, the sorting is reduced to a linear scan of R.

Page indexing hybrid hash join (PHH): The clustering of tuples in R according to their pointed tuples in S can also be achieved by hashing. The hashing can be done as in standard hybrid hash join, except hash values are TIDs of tuples in S instead of join attribute values. First R is partitioned by hashing (function ϕ_1) into R_0, R_1, ... , R_k, and all R_i but R_0 are written out to disk. R_0 is then built into a hash table in memory by possibly a different hash function ϕ_2. When S is hashed by ϕ_1, it is partitioned into S_0, S_1, ... , S_k. If a tuple of S belongs to S_0, this tuple is then hashed by ϕ_2 and the appropriate bucket in the hash table are probed for a match. Thus the matching of R_0 and S_0 is done along with partitioning of S. Then, R_i's are read into memory one by one to build into hash tables and corresponding S_i's are read to find matches.

The difference between the performance of PSM and PHH is the difference between sorting and hashing and resembles that between standard sort-merge join and the standard hybrid hash join. The cost to read and write R during hashing is $2 \mid R \mid - \mid R_0 \mid$, and the cost to read the remaining part of R other than R_0 is $\mid R \mid - \mid R_0 \mid$. All the reading and writing are basically sequential. SO the disk I/O cost is:

$$C_{PHH} = (3 \mid R \mid -2 \mid R_0 \mid) + (\theta_{hash} + 1)\alpha\beta \mid R \mid \tag{34}$$

θ_{hash} should be comparable in value to θ_{sort} and much smaller than θ. If $\mid R \mid$ is $50,000$, $\mid R_0 \mid$ is $10,000$, α is 10%, $beta$ is 1.2, and $theta_{hash}$ is 10, then the I/O cost is:

and $theta$ is 200, then the I/O cost is:

$$\begin{aligned} C_{PHH} &= (3 * 50,000 - 2 * 10,000) + (10 + 1) * 0.1 * 1.2 * 50,000 \\ &= 196,000 \end{aligned} \tag{35}$$

The performance is slightly better than that of PSM.

Page indexing PID-partitioning join (PPP): To use this technique [40], the PIDs of tuples of S need to be stored separately from S. First the PIDs of S are read into memory and sorted, and then partitioned into $B + 1$ partitions L_i $(0 \leq i \leq B)$ by their PID values. If $i < j$, any PID in L_i is smaller than any PID in L_j. PIDs remain memory resident after partitioning. The set of tuples in S with their page addresses fall into L_i will be denoted as S_i. After partition, S_0 is read into memory. B pages in memory are set up as buffers for R_i, $1 \leq i \leq B$, one for each buffer. Then, tuples of R is read into memory block by block. If a tuple of R whose PID pointers fall into partition L_0, then S_0 is probed for matches. Otherwise, the tuple is placed in the appropriate output buffer of R_i, and the whole buffer will be written out to disk if it is full. After R be scanned and R_0 joined with S_0, the rest is similar to hybrid hash join. In each step all tuples of R_i are read into memory and tuples of S_i are read block by blocks. This process repeats until all the R_i's and S_i's be joined.

The performance analysis of PPP is very similar to that of PHH, except the cost of sorting PIDs of S needed to be added and the random access factor $\theta_{partition}$ in this case should in general be equal or smaller than θ_{hash}. Assume the size of the PIDs list of S is $\mid L \mid$, the disk I/O cost is:

$$C_{PPP} = 4 \mid L \mid +3 \mid R \mid -2 \mid R_0 \mid +(\theta_{partition} + 1)\alpha\beta \mid R \mid \tag{36}$$

Since $\mid L \mid$ is much smaller than $\mid R \mid$, it is possible that entire list of L can fit in memory. In this case, the factor 4 in the first term can be eliminated:

$$C_{PPP} = \mid L \mid +3 \mid R \mid -2 \mid R_0 \mid +(\theta_{partition} + 1)\alpha\beta \mid R \mid \tag{37}$$

If the cost of sorting PIDs is small, PPP can generally be expected to outperform PHH due to reduced randomness and PPP is more resistant to data skew.

9.4 Domain Based Internal Schema

Another special data structure designed to facilitate join execution is a data structure to store each join attribute domain values separately, and store pointers associated with the values in this data structure pointing to tuples of relations that have this value in attributes of this domain [12]. This data structure can achieve speed up of join execution slightly more than a join index do. However, this method favors join operation at the expense of other operations. Not only that, if there are many relations that are mutually joinable in a relational database, join attribute values has to be duplicated many times over.

9.5 Composite B+ Tree

A composite B+ tree, also called a B_c tree by author [33], can be seen as a generalization of the generalized access path structure approach [6] and this approach is an improvement over the domain based internal schema approach. The idea is to create one B+ tree for each data domain. Many attributes in the database share the same data domain, so values in these attributes are indexed by the same B+ tree. The B+ tree of a data domain is further modified in its leaf node to have pointers to all values in all different attributes that share this data domain. Such a B+ tree is called a B_c tree. A B_c tree can be used to do join executions just like a join index with the added advantage of B+ tree access. Using a B_c tree to do join can increase performance several times than a tradiitonal join method when at least one full scan over each joining relation is needed.

B_c tree is a much simplified approach than the generalized access path structure and therefore is much easier to implement and the performance could be better due to smaller size of overhead structures. The size of data structure in B_c tree approach is also much smaller than that in the join index approach. In B_c tree approach the number of B_c trees in a database is the number of different data domains. In join index approach, the number of join indexes can be as high as C_2^m, where m is the number of relations in the database. In worst case each join index could have number of tuples as many as the cross product of the numbers of tuples in two joining relations. So B_c tree approach has a much smaller data structure than that of the join index.

10 Band Join

An ordered data domain is one that order between data values makes sense. Examples of ordered data domain are numerical data domain, string data domain in lexicographical ordering, etc. In ordered data domain, the join predicate can be relaxed using inequalities. Let c_1 and c_2 be two non-negative integers, not both equal to zero, $t_r \in R$ and $t_s \in S$ are two tuples from R and S, respectively. A join predicate is characterized by its predicate which can be stated as $t_r[R.A] - c_1 \leq t_s[S.B] \leq t_r[R.A] + c_2$. A match in a band join is defined as a join that attribute values from one relation fall within a specified range of attribute values of the other relation.

In most commercial databases today, the job of a query with a band join can be accomplished by breaking down the query into two queries each has an inequality join, and then combine the results of these two joins. As band join is very similar to the equijoin both in concept and in execution, it is likely that band join will become a standard operation in future commercial relational databases.

While a large number of papers have been presented for processing equijoins, only a few papers [43, 49, 60] are devoted for processing band joins. Most of the band join algorithms require sorting of at least one of the relation. Truncating-hash band join [49] is an attempt trying to do away with the sorting and has considerably improved the speed of band join processing. The essential idea of the truncating-hash band join is to group data values that are near one another. For example, if the query is between R and S, join attributes are $R.A$ and $S.B$, and the query predicate is $R.A - c_1 \leq S.B \leq R.A + c_2$, where $c_1 \geq 0$ and $c_2 \geq 0$ are two constants that cannot be both zero, then, the truncating can be done as the join attribute values divided by $c_1 + c_2$.

As its counterpart equijoin, truncating-hash band join has two phases: the building phase and the joining phase. In the building phase, a tuple t_R from R first has its join attribute value $t_R.A$ be truncated [49]:

$$r_i = t_R.A - ((t_R.A) mod(c_1 + c_2)) \tag{38}$$

into bucket h_i of a hash table. Collisions are resolved by chaining values in the same bucket.

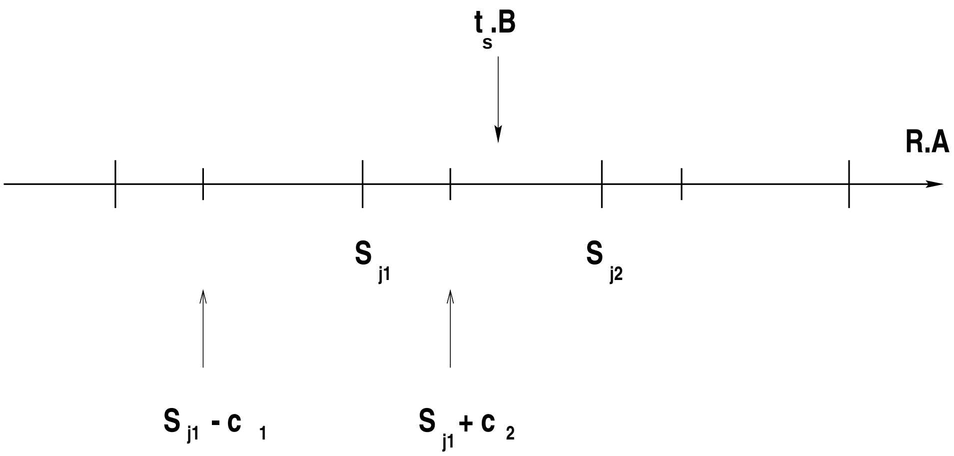

Figure 5: Choose buckets

In the joining phase, tuples from S are hashed using the same method as that for R, except right after hashing, appropriate buckets in R's hash table are found and values in the buckets are probed for a match.

Let t_S be a tuple from S. It's join attribute value $t_S.B$ is truncated as:

$$s_{j1} = t_S.B - ((t_S.B) mod(c_1 + c_2)) \tag{39}$$

s_{j1} is an index into a hash bucket of R. Another bucket of R is chosen based on following criterion:

$$\begin{array}{ll} if & t_S.B < (s_{j1} + c_2) \\ then & s_{j2} = s_{j1} - (c_1 + c_2) \\ else & s_{j2} = s_{j1} + (c_1 + c_2) \end{array}$$

The reason to use this criterion is that if $t_S.B < (s_{j1} + c_2)$, then, some R's attribute values in the bucket $s_{j1} - (c_1 + c_2)$ could possibly satisfy the join predicate, but not any attribute value in the bucket $s_{j1} + (c_1 + c_2)$. Otherwise, it must be that $t_S.B \geq (s_{j1} + c_2)$, i.e., $t_S.B \geq (s_{j1} + (c_1 + c_2) - c_1)$, and bucket $s_{j1} + (c_1 + c_2)$ may have some tuples satisfy the join predicate, but not bucket $s_{j1} - (c_1 + c_2)$. This is shown in Figure 5.

It is a well known fact that for equijoin, hash based algorithms are in general better than sort-merge join and nested block join [17, 23, 26]. The truncating-hash band join is an attempt trying to carry this performance advantage over to the band join and analytical comparisons show that this approach is successful [49]. In essence, this is a hash join facilitated by truncating.

A later improvement of the sort-merge band join [60] by H. Lu and K.-L. Tan appeared to offer competitive performance. It is essentially a sort-merge technique. Instead of merging two sorted lists of R and S, it merges $m_r + m_s$ sorted runs, where m_r sorted runs are from R and m_s sorted runs are from S. This technique makes better use of the large memory available to speed up join execution. After sorting, substantial number of tuples from one relation is kept in memory to form a window. The window is characterized by its upper limit and lower limit. Tuples from the other relation are read in memory to probe the window to find match. Important issues to be addressed in this technique are: (1) which relation should be the window relation ? (2) how to divide memory space to various buffers, especially to a window, to achieve the best performance ? (3) how to adaptively adjust window size ?

Let the join predicate be $R.A - c_1 \leq S.B \leq R.A - c_2$, D_R be the number of distinct values in the join attribute of R and D_S be such a number for S. The criterion to decide the window relation is [60]:

$$\begin{array}{ll} R & if \frac{||R||}{D_R} < \frac{||S||}{D_S} \\ S & if \frac{||R||}{D_R} > \frac{||S||}{D_S} \\ R|S, & otherwise \end{array}$$

where $||\ .\ ||$ represents number of tuples, not pages. For R, the number of tuples in the band is about $\frac{||R||}{D_R} \times (c_1 + c_2)$. If R is the window relation, it is desirable that at least this many tuples should be in the window. The same argument is true for S with the number be replaced by $\frac{||S||}{D_S} \times (c_1 + c_2)$. Without knowing (or computing) precisely the available buffer space for holding the window, a reasonable choice is to pick the relation that gives the smaller number between the two as the window relation.

A too small window lowers performance because often the tuples from the other relation needs to compare tuples that are outside the window and thus cause tuples in window be repeated read from the disk. A too large window lowers performance because this may cause the buffer space too small for the other relation, and tuples from the other relation may need to be read repeatedly.

The goal of buffer allocation is try to have a sufficient window size that (most likely) can store all tuples in a band for the window relation, which is $\frac{|R|}{D_R} \times (c_1 + c_2)$ if R is the window relation, and yet left enough space for sorted runs of R and S. At least one page of buffer should be allocated to each run of R or S. If there are total B pages and w is allocated to the window of R, then there should be no more than $B - w$ runs from R and S.

If memory is not enough for the window and buffers of runs of R and S, several small (means with less tuples) runs can be merged and their buffer pages be freed except one page that is left for the merged run. This adjustment may be needed during the join execution due to data skew or change of available memory.

11 Summary

Nested loops join (late 70's), sort-merge join (late 70's) and hash join (early 80's) are mature join techniques that have been widely used in commercial relational database in the 90's. Each of them has an area that it can perform better than others. In general, nested loops join offer most robust performance in all situations, sot-merge join usually performs better than nested loops without indexing, especially if the join result is large. Hash join can usually be expected to perform even better than sort-merge join, especially when join result is small and data distribution is even. However, hash join is much less robust in performance than sort-merge join.

Join indexes (late 80's) is a relatively new join technique and offer much greater potential for further speed up of join execution. But ut has not been deployed to commercial relational databases, at least in popular ones. Before this new idea can be used in commercial databases, some problems must be solved. Size of the join index is perhaps the biggest problem needs to be addressed.

Join page index, i.e., pointer-based join, was proposed in 1990. It is mainly used in object-oriented database but can be used in relational database as well. It uses pointers in tuples to materialize join connection without actually duplicate joined tuples. The obvious drawback of this join technique is the same as that of the join index, i.e., it has a huge data structure.

B_c tree was proposed in 1989 by Bipin C. Desai. B_c tree serves the same purpose as that of the join index, i.e., to materialize a join but not to store all joined tuples away from joining relations. The size of the data structure in B_c tree approach can be in general expected to be much smaller than that of the join index.

Multi-layer concept (early 90's) on relational databases are relatively new. However, the term "multi-layer" has different meaning in different papers. Many used multi-layer as a way for improving database security. some have used multi-layer to improve modeling capability, few paper used multi-layer as a way to speed up query execution. However, none has used multi-layer to do join. Conceptually, multi-layer approach offers greatest potential for further speed up of join execution.

The use of parallel machines and parallel join techniques can turn join execution into a heavily I/O bound execution, so join execution can be speeded up. Types of parallel join techniques are very much dependent on the types of parallel machine used. Shared-nothing (message passing) parallel machines are gaining popularity because of the easiness of scalability of hardware and software. However, join executions on shared-nothing parallel machines are very sensitive to data skew. On way to alleviate this problem is to use shared virtual memory and encourage workload sharing. Shared-everything (shared-memory) parallel machines are more complicated in scaling, both in hardware and software. However, they show less sensitivity to data skew as compared with shared-nothing parallel machines.

Conceptually, a parallel join offers an easy way to enhance performance by sheer power of the hardware. However, a parallel join is more sensitive to data skew. A large part of research in parallel join are devoted on alleviate the impact of data skew.

Band join is not a hot research branch. However, because of its usefulness and similarity to equijoin, it could become a standard feature in commercial relational database and several approaches are discussed and compared here.

Adaptivity seems the ultimate improver of every join technique: every join technique eventually evolves into an adaptive one. This is especially clear in the evolution of the hash join technique. The reason is simple: a DBMS has to share the hardware, especially, disk heads, memory space and processors on the computer, with many other software processes, and the ultimate response time to the user not only to share the hardware with many other software and processes, and the ultimate join response time appears to the user is heavily affected by the fluctuation of the hardware (disk heads, memory space, processors, etc.) and software (operating system process scheduler, for example) resources.

The ultimate performance winner of join technique seems to be a parallel, adaptive join technique with fast indexes (join index, multiple layer DB).

References

[1] D. Knuth. The Art of Computer Programming: Sorting and Searching, Vol. 3, Addison-Wesley, Reading, Mass, 1973.

[2] Gotlieb L. R., Computing Joins of Relations. ACM SIGMOD Conf., San Jose, California, May 1975.

[3] Blasgen M., Eswaran K.. On the evaluation of queries in a data base system. *IBM Res. Lab., San Jose, CA, Res. Report RJ1745, 1976.*

[4] Blasgen, M. W., and Eswaran, K. P. Storage and access in relational databases. *IBM Systems Journal. Vol 16, No. 4,* 1977.

[5] F.P. Preparata. New parallel-sorting schemes. IEEE. Trans. Comput. C-27 (July 1978).

[6] Theo Haerder. Implementing a Generalized Access Path Structure for a Relational Database System. ACM TODS, Vol3, N3, Sept. 1978, p 285-298.

[7] Babb E., Implementing a Relational Database by Means of Specialized hardware. ACM TODS, V4, N1, March 1979, pp. 1-29.

[8] W. Kim. A new way to compute the product and join of relations. In Proc of the 1980 ACM SIGMOD Int'l Conf. on the Management of Data, pp. 179-187, 1980.

[9] Bernstein, A. P., et al. Query Processing in a system for distributed databases (SDD-1). ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, pp. 602-625.

[10] J. R. Goodman. An investigation of Multiprocessor Structures and Algorithms for Database Management. Technical Report ECB/ERL, M81/33, University of California at Berkeley, 1981.

[11] G. H. Gonnet, P. A. Larson. External hashing with with limited internal storage. Proc. of the ACM Symposium on Principles of Database Systems, ACM, New York, 1982, pp. 256-261.

[12] Missikoff, M. A domain based internal schema for relational database machines. In ACM SIGMOD Int'l Conf. on Management of Data, Orlando, Fla., June 2-4, 1982. ACM, New York, 1982, pp. 215-224.

[13] Nicholas Roussopoulos. View Indexing in Relational Databases. ACM Transactions on Database Systems Vol. 7, No. 2, June 1982.

[14] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Application of Hash to Data Base Machine and its Architecture. New Generation Computing, vol 1, No.1, pp. 66-74, 1983.

[15] R. Sacks-Davis, K. Ramamohanarao. A two level superimposed coding scheme for a partial match retrieval. *Information Systems, Vol. 8, No. 4, 1983*, pp. 273-280.

[16] Bratbergsengen, Kjell. Hashing Methods and Relational Algebra Operations. Proc. of the 1984 Very Large Database Conf., 1984.

[17] DeWitt, D. J., Katz, R., Olken, F., Shapiro, L., Stonebraker, M. and D. Wood. Implementation Techniques for Main Memory Database Systems. Proc of the 1984 ACM SIGMOD Int'l Conf. on Management of Data, pp. 1-8, Boston, MA, USA, June 1984.

[18] W. Kiessling. Tunable dynamic filter algorithms for high performance database systems. Proc. of the Int'l Workshop on High Level Computer Architecture, May 84, 6.10-6.20.

[19] K. Bratbergsengen. Hashing methods and relational algebra operations. In Proc. of the Tenth Int'l Conf. on Very Large Data Bases, pp. 323-333, Singapore, August 1984.

[20] P. A. Larson, A. Kaila. File organization: Implementation of a method guaranteeing retrieval in one access. Commun. ACM 27, 7(1984), pp. 670-677.

[21] P. Valduriez, G. Gardarin. Join and Semi-Join Algorithms for a Multiprocessor Database Machine, ACM Transactions on Database Systems, Vol 9, No. 1, 1984.

[22] P. Valduriez, G. Gardarin. Join and Semijoin Algorithms for a Multiprocessor Database Machine.

[23] D. J. DeWitt, R. Gerber. Multiprocessor Hash-Based Join Algorithms. Proc. of the 11th Int'l Conf. on Very Large Data Bases, Stockholm, 1985, pp. 151-164.

[24] Arie Segev. Optimization of join operations in horizontally partitioned database systems. ACM Transactions on Database Systems, Vol. 11, No. 1, March 1986, Pages 48-80.

[25] L. Mackert and G. Lohman. "R* Optimizer Validation and Performance Evaluation for Local Queries", Proc. of 1986 ACM SIGMOD Conf., Washington, DC, May 1986.

[26] L. D. Shapiro. Join Processing in Database Systems with Large Main Memories. ACM Transactions on Database Systems, Vol. 11, No. 3, September 1986, pp. 239-264.

[27] Sacco, G. M. Fragmentation: A technique for efficient query processing. ACM Transactions on Database Systems, Vol. 11, No. 2, 1986, pp. 113-133.

[28] Patrick Valduriez. Join Indexes. ACM TODS, V12, N2, June 1987,

[29] A. Kumar and M. Stonebraker. The Effect of Join Selectivities on Optimal Nesting Order. SIGMOD Record, Vol16, N1, p 28-41. March 1987.

[30] H. Kang, N. Roussopoulos. Using 2-way Semijoins in Distributed Query Processing. IEEE. Proc. of 3rd Int'l Conf on Data Engineering, 1987.

[31] R. Sacks-Davis, A. Kent, K. Ramamohanarao. Multikey Access Methods Based on Superimposed Coding Techniques. *ACM Transactions on Database Systems, Vol. 12, No.4, 1987*, pp. 655-696.

[32] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In Proc of the 14th Int'l Conf. on Very Large Data Bases, pp. 468-478, Los Angeles, California, USA, August 1988.

[33] B. C. Desai. Performance of a Composite Attribute and Join Index. *IEEE Transactions on Software Engineering, Vol. 15, No. 2, 1989.*

[34] Dik Lun Lee, Chun-wu Leng. Partitioned Signature File: Design Issues and Performance Evaluation. *ACM Transactions on Information Systems, Vol. 7, No.2, April 1989, pp. 158 - 180.*

[35] M. Kitsuregawa, M. Nakayama, and M. Takagi. The effect of bucket size tuning in the dynamic hybrid grace hash join method. Proc. of the 15th Int'l Conf. on Very Large Data Bases, pp. 257-266, Amsterdam, 1989.

[36] K. Li, P. Hudak. Memory coherence in shared virtual memory systems. ACM Transaction on Computer Systems, 7(4), November 1989.

[37] M. S. Lakshmi, P. S. Yu. Limiting Factors of Join Performance on Parallel Processors. IEEE. Proc. of 5th Int'l Conf. on Data Engineering, 1989.

[38] D. Schneider and D. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. Proc. ACM SIGMOD Conf., 1989.

[39] M. S. Lakshmi, P. S. Yu. Effectiveness of Parallel Joins. IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 4, December 1990.

[40] E. J. Shekita, M. J. Carey. A Performance Evaluation of Pointer-Based Joins. Proc. of 1990 ACM SIGMOD Int'l Conf. on Management of Data, 1990.

[41] H. Zeller and J. Gray. An adaptive hash join algorithm for multiuser environments. In Proc of the 16th Int'l Conf. on Very Large Data Bases, pp. 186-197, Brisbane, Australia, August 1990.

[42] J.B. Carter, J.K. Bennet, W. Zwaenepoel. Implementation and performance of Munin. Proc. of the 1991 Symp. on Operating System Principles, 1991.

[43] D. DeWitt, J. Naughton, D. Schneider. An Evaluation of Non-Equijoin Algorithms. Proc. of 17th Int'l Conf. on Very Large Data Bases, Barcelona, Spain, September 1991, pp. 443-542.

[44] J. Cheng, D. Haderle, R. Hedges, B. R. Iyer, T. Messinger, C. Mohan, Y. Wang. An Efficient Hybrid Join Algorithm: A DB2 Prototype. IEEE. 7th Int'l Conf. on Data Engineering, 1991.

[45] Eugene Shekita. High-Performance Implementation Techniques for Next-Generation Database Systems. Ph.D thesis, Computer Science Technical Report #1026, 1990, University of Wisconsin at Madison.

[46] Joel L. Wolf, Balakrishna R. lyer, Krishna R. Pattipati, John Turek. Optimal Buffer Partitioning for the Nested Block Join Algorithm. IEEE. Proc. of 7th International Conference on Data Engineering. April 8-12, 1991, Kobe, Japan.

[47] M. Negri and G. Pelagatti. Distributive Join: A New Algorithm for Joining Relations. ACM Transactions on Databases, Vol. 16, No. 4, December 1991, pp. 655-669.

[48] D.J. DeWitt, J.F. Naughton, D.A. Schneider et al. The Gamma database machine project. IEEE Transactions on Knowledge and Data Engineering, 2(1), March 1990.

[49] Valery Soloviev. A Truncating Hash Algorithm for Processing Band-Join Queries. IEEE. Proc. 9th Int'l Conf. on Data Engineering, 1993.

[50] H. Pang, M. J. Carey, and M. Livny. Partially preemptible hash joins. In Proc of the 1993 ACM SIGMOD Int'l Conf. on the Management of Data, pp. 59-68, Washington, DC, USA, May 1993.

[51] A. Shatdal and J. F. Naughton. Using Shared Virtual Memory for Parallel Join Processing. Proc. of 1993 ACM SIGMOD Int'l Conf. on Management of Data, 1993, pp. 119-128.

[52] J. Han, Y. Fu and R.T. Ng. Cooperative Query Answering Using Multiple Layered Databases. *Proc. of the 2nd Int'l Conf. on Cooperative Information Systems,* Toronto, Canada, May 1994. pp. 47-58.

[53] Goetz Graefe. Sort-Merge-Join: An Idea Whose Time Has(h) Passed ? IEEE. 10th Int'l Conf. on Data Engineering, 1994.

[54] G. Graefe, A. Linville, L. Shapiro. Sort vs. Hash Revisited. IEEE. Transactions on Knowledge and Data Engineering, Vol.6, No.6, December, 1994.

[55] T.P. Martin, P.-A. Larson, and V. Deshpande. Parallel Hash-Based Join Algorithms for a Shared-Everything Environment. IEEE Transactions on Knowledge and Data Engineering. Vol. 6, No. 5, October 1994.

[56] D. K. Shin, A. C. Meltzer. A New Join Algorithm. ACM SIGMOD Record, Vol. 23, No. 4, December 1994.

[57] J. Vaghani, K. Ramamohanarao, D. B. Kemp, Z. Somogyi, P. J. Stuckey, T. S. Leask, and J. Harland. The Aditi deductive database system. The VLDB journal, 3(2):245-288, 1994.

[58] W. P. Yan, P.-A. Larson. Performing Group-By before Join. IEEE Proc. of 10th Int'l Conf. on Data Engineering, 1994.

[59] Evan Philip Harris. Towards Optimal Storage Design for Efficient Query Processing in Relational Database Systems. PhD thesis, 1995, Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia.

[60] H. Lu and K.-L. Tan. On Sort-Merge Algorithm for Band Joins. IEEE. Transactions on Knowledge and Data Engineering, Vol. 7, No. 3, June 1995.

[61] Z. Li, K. A. Ross. Fast Joins Using Join Indices. *Technical Report, CUCS-032-96, 1996, Columbia University.*

[62] Ming-Ling Lo and Chinya V. Ravishankar. Towards Eliminating Random I/O in Hash Joins. IEEE. Proc. of 12th Int'l Conf. on Data Engineering, 1996.

[63] Raghu Ramakrishnan. Database Management Systems. Beta Edition, 1996, ISBN 0-07-052522-6. McGraw-Hill Companies, Inc.