Join Techniques in Relational Databases

Yuping Yang and 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 databases. Substantial research efforts are still underway to enhance its performance due to its extreme importance in the industry. However, in recent years, there has not been a study which comprehensively examines, under a common framework, the performance of a wide spectrum of equijoin techniques available today. Processor speed, memory size, speed of sequential disk access, as well as the time ratio of random to sequential disk accesses, all have increased drastically in the past decade. Results of many previous performance studies of join techniques no longer hold due to this progress. The main contribution of this paper is twofold. First, it surveys a wide spectrum of equijoin techniques available today. Second, it compares their performance under a common framework that reflects the reality of today's computing environments.

Through this survey, we see a trend that indexed join methods are, in general, far better than non-indexed methods for low selectivity joins in large relational databases. Other conclusions of this survey are: (1) a proof that shows that rocking technique in the nested block join provides no speed up; (2) join executions in shared-memory parallel machines with a large number of parallelly operated I/O channels behave more like join executions in shared-nothing parallel machines than that in the shared-memory parallel machines with only a few I/O channels.

Key words: Relational database, query execution, join, equijoin, index, disk I/O, performance, survey.

1 Introduction

Most databases today, especially large corporate databases, are relational databases. Object-oriented databases, though a hot research topic for over a decade, have not made large inroads into the commercial world. The dominance of relational databases is due to the mathematical soundness and simplicity of the relational model.

The query processing in a relational database can be divided into two parts: to find an optimal strategy (query optimization) and to execute (query execution) it. A query execution typically needs much more time than a query optimization does and a join execution is the most expensive part in a query execution. The join execution in relational databases is a mature subject and has been extensively investigated for over two decades. Nevertheless, due to its extreme importance in commercial applications, join execution is still an active research topic and in recent years.

Previously, each join technique was usually compared to one or two existing join techniques to demonstrate its performance advantages. From this type of pairwise comparisons, it is difficult to determine, globally, which join technique perform the best for a particular application. Therefore, we perceive the need for a comparison among a large number of join techniques developed during the last decade under a common framework. In this survey, the discussion is restricted to equijoin execution of query execution only. The main focus of the comparison is the relative performance and the practicality. The main criterion for performance comparison is the number of disk I/Os and for practicality are the ease and the flexibility of implementation.

During the last decade, computer hardware technology has advanced drastically and memory size, CPU speed, as well as the time ratio of a random to a sequential disk access has increased by orders of magnitude. Many past analyses of the performance of various join techniques are no longer valid due to the new progress. For example, in [23], the cost of sorting was computed to be $O(\mid R \mid log_{k-1} \mid R \mid)$, where k is the number of pages in the memory buffer and $\mid R \mid$ is the number of pages of R. This expression is no longer valid since with today's large memory, usually sorting can be accomplished by two scans of each relation [1]. The drastic increase in the processor speed and considerable overlapping between CPU executions and disk accesses mean that the cost of CPU execution is no longer a substantial portion of the cost of the join execution. The ever widening gap between an average random disk access time and a sequential disk access time forces the inclusion of random access factors in all cost formulas of join techniques to ensure truthful comparisons of the join performance.

In this paper, we use different random access factors to reflect different randomness in different join techniques and computing environments. We assumed that large I/O buffers [52] are used for each join techniques, whenever possible, to reduce the randomness in the disk access. Because of the fast processor speed and the overlap between processor executions and disk accesses, only the time for disk I/Os is included in the cost of join execution. This effort results in a series of consistent cost formulas, one for each join technique. These formulas realistically reflect the cost of the join execution in each join technique. Through this survey, we see the trend that indexed join techniques are, in general, faster than non-indexed join techniques for low selectivity joins in large databases. We also proved that rocking has no effect in speeding up the nested block join. Through the cost analysis and comparison of various parallel join techniques, we concluded that terms such as "shared-nothing parallel computer" and "shared-everything (shared-memory) parallel computer" cannot accurately reflect the differences of various parallel machines as far as join execution is concerned. Since the cost of the join execution is mainly the cost of disk I/Os, the cost behavior of the join execution in a shared-memory parallel machine with one I/O channel for each CPU is very similar to the cost behavior of the join execution in a shared-nothing parallel machine, and is far different from the cost behavior of the join execution in a shared-memory parallel machine with only a few I/O channels.

This paper is organized as follows: Section 2 briefly introduces the relational database, assumptions and notations used throughout this paper. A taxonomy of a wide variety of join techniques that are available today results in Section 3. Section 4, 5, and 6 describe loop, sort, and hash based joins, respectively. These are three major classes of traditional non-indexed join techniques. Some mechanisms that can be used to enhance the performance of various join techniques are presented in Section 7. Section 8 describes various parallel join techniques. Section 9 describes some promising new indexing structures that can further speed up join executions. Section 10 gives a performance comparison of many join techniques under a common parameter setting. Section 11 summarizes the paper.

2 Background

2.1 Relational Databases

A relational database system is a software system to store and manage large volume of data according to the relational data model. A *relational database* refers to an instance

of data organized under a relational data model and managed by a relational database system.

A schema in the relational data model can be described as R(A_1:D_1, A_2:D_2, ... , A_n:D_n), where A_i's are *attributes* and D_i's are *data domains* of A_i's. Attributes are names of data categories such as price, telephone number, etc. An attribute's data domain is the set of all values in the same data category as the attribute. For example, attribute price may have data domain [0..$10.00] and telephone number may have the set of all 10 digit strings as its data domain. A relational data model usually has many schemas R_1, ... , R_r.

OFFICE

OID	OBUILDING	OFLOOR
300323	PHYSICS BLD	3
170205	ADMIN BLD	2
024517	CLASSROOM BLD	5
044422	ECONOMICS BLD	4

DEP

DEPID	DNAME
014	PHYSICS
006	ECONOMICS
051	ADMINISTRATION

EMP

EMPID	ENAME	EDEP	EOFFICE
02317	JOHN STEVEN	006	044422
44194	MIKE MILLER	051	024517
68204	JENNIFER TAYLER	014	300323
67279	THOMAS JOHNSON	014	300323
10362	EUGENE GRAY	051	170205

Figure 1: A relational database

Let $S(R_1, R_2, ..., R_r)$ be a relational model; $R_i(A_1 : D_1, A_2 : D_2, ..., A_n : D_n)$ be a particular schema of S; DB be a database with data model S. A *relation* in DB with schema R_i is a set of *tuples* $(a_1, a_2, ..., a_n) \in D_1 \times D_2 \times ... \times D_n$. Each relation has one or more attributes as its key attributes whose values are unique for each tuple. Key attributes can be used to uniquely identify individual tuples in the relation.

Figure 1 shows a relational database with three relations, EMP, DEP, and OFFICE. Relation EMP has five tuples and four attributes: EMPID, ENAME, EDEP, and EOFFICE. EMPID is the key attribute of EMP. Relation DEP has three tuples and two attributes: DEPID and DNAME. DEPID is the key attribute of DEP. Relation OFFICE has four tuples and three attributes: OID, OBUILDING, and OFLOOR. OID is its key attribute. EDEP and DEPID have the same data domain. OID and EOFFICE also have the same data domain.

2.2 The Join Operation

In the example in Figure 1, EDEP in relation EMP and DEPID in relation DEP have the same data domain. If one desires to find the name of the department of an employee, values in attribute EDEP of relation EMP should be compared to values in attribute DEPID of relation DEP for matches, and from matched tuples, values from DNAME in relation DEP are extracted as the answer to the query. The process of finding tuples in both relations with matching EDEP value in EMP and DEPID value in DEP is called the *join operation*, or more precisely, an *equijoin operation*, because the matching criterion is the equality of two values. There are other types of joins such as inequality join which uses "<" or ">" instead of equality as the matching criterion. We will use the term *join* in subsequent discussions with the understanding that we are discussing equijoins. EDEP and EDPID are called *join attributes*, because they are in different relations and share the same data domain, and can be used in a join operation.

The join between two relations is a binary match operation executed between two potentially large lists. Without exploiting any clustering or ordering properties of data and without using any indexes, there can be only one join technique; that is, to compare every tuple in one relation with every tuple in the other relation. The cost of this join technique is roughly the cross product of the number of tuples in two joining relations and is often huge. For example, if $|| R ||=|| S ||= 100,000$, where $|| X ||$ stands for the number of tuples in relation X, then the cost of executing a join is $|| R || /p+ || R || * || S || /p^2$ = 100 million pages, where $p = 10$ is the number of tuples that can be held in one disk page. A simple yet very effective improvement is to read from the disk a large chunk of R into the memory. If $M = 20,000$ pages, this large chunk can be $(M - 2) = 19,998$ pages of R (one page is reserved as the input buffer for S and one page is reserved as the output buffer for joined tuples). Then pages of S are read one at a time into the memory for finding matches. This sharply reduces the number of disk I/Os to $|| R || /p + (|| R || /(p(M - 2))) * (|| S || /p) = 15,001$ pages.

2.3 A Cost Model of Join Executions

It is usually very costly to execute a join in a relational database and often the size of joining relations is much larger than available memory space on the computer. This means a large number of disk I/Os often need to be made during a join execution. Some previous studies such as [34] include CPU time and in-memory move time as part of their estimate of the cost of the join execution, and many others do not include the cost such as the

in-memory move time. This makes their results incomparable to each other.

There are two reasons why we only use disk I/O as the measure of the cost of the join execution. First, on many modern computers CPU executions overlap in time with disk I/Os so that it is not correct to simply add CPU execution time to the disk I/O time as the join execution time. Second, disk I/Os contribute bulk of the time cost in join executions of all join techniques. We exclude the cost of writing out join results in cost formulas because it is the same for all join techniques. In parallel joins, we expect the cost of writing out all join results to be the same among all parallel joins and to be $1/n$ of that of the serial joins.

In the evaluation of a join technique, if some enhancements can be easily incorporated to improve this join technique without changing its basic philosophy, those enhancements are incorporated. This approach allows each join technique to be evaluated to its full potential, and not necessarily in the form it was originally proposed.

Let R and S be two joining relations and $\mid R \mid \leq \mid S \mid$, where $\mid X \mid$ denotes the number of pages in relation X. Let θ be the time of an average random disk access. The time of a sequential disk access is assumed to be 1. To simplify our discussion, we assume R and S have the same number of attributes and each page contains the same number of tuples. Let α be the percentage of the tuples in R that can match with the tuples in S and each matchable tuple of R can match with, on the average, β number of tuples in S. Thus $2\alpha\beta \mid\mid R \mid\mid /p$ is the average size of the join result, i.e., number of pages of all matched tuples in the join.

Let M be the size of the available memory on the computer. To allow a fast sorting algorithm [1] to be implemented for every sort based join technique to ensure a fair comparison among join techniques, we assume $M > \sqrt{\mid S \mid /2}$. The fast algorithm needs two scans over the set to be sorted. The first scan sorts the set into runs of approximately size $2M$ and the second scan merges all sorted runs into a completely sorted list. Because the average number of sorted runs is $\mid S \mid /(2M)$, the above assumption is needed to ensure that there is at least one page in the memory for each sorted run. So, we obtain $\mid S \mid /(2M) \leq M$. This is a realistic assumption and is true in almost all practical cases (based on today's hardware technology). In hash based joins, we assume that the memory is large enough to allow the allocation of at least one page of the memory as a buffer for each bucket. This assumption is usually true in all realistic situations. On the other hand, we are analyzing join techniques for large relations; so both R and S are assumed to be much larger than available memory on the computer.

This paper focus on the analyses of the performance of join executions on large

databases for full (nonselective) joins with low selectivities (small join results). A selective join can usually be regarded as a full join with one or both joining relations being small, because they are the result of some selection operations. Therefore, for most join techniques, there is no need to develop separate cost formulas for full and selective joins. Composite B+ tree (B_c) technique [29] is an exception and this will be discussed later.

The general formula of the join execution cost is:

$$C_{general} = (cost\ \ of\ \ preprocessing) + (cost\ \ of\ \ joining) \quad (1)$$

The cost of preprocessing includes the cost of initial reading of data from the disk and the cost of sorting for sort based joins or hashing for hash-based joins.

3 A Taxonomy of Join Techniques

As stated in the last section, join execution is often very costly and some techniques can reduce join cost significantly. These techniques often rely on some data properties. If data has some very special characteristics and is used for a very special purpose, then a specialized file system, rather than a relational database, can hold the data and do a better job in terms of query performance. However, for a general purpose relational database, few characteristics about the data can be assumed. Clustering and ordering among join attribute values are two properties that can often be exploited for join execution.

There are three categories of traditional join techniques: loop, sort, and hash based joins. Sort based joins exploit ordering properties of join attribute values while hash based joins exploit clustering properties of join attribute values.

The following factors also affect the speed of the join execution: type and amount (e.g., whether to put an index to each attribute, etc.) of indexes, organizing and sequencing of disk I/Os, ways for partitioning and sorting of relations, size of memory buffers, and ways of utilizing buffers.

Table 1 lists join techniques that will be discussed in this paper. Each row represents a join technique and each column represents a mechanism which may be used to facilitate a join execution. A "X" in a row and a column denotes that the mechanism of the column is used by the join technique of the row. Meanings of the first four column titles are self-explanatory: "loops" means nested loops are used and "index" means indexes are used, etc. Column title "large M" means large memory space is crucial for a join technique to perform well. Column title "buffer" means special memory buffer management strategy

is used to gain better performance for a join technique. Column title "parallel" means parallel processing is used. Clustering of tuples on the disk can also be utilized to facilitate join executions and can be applied in a broad spectrum of join techniques. However, it has been thoroughly surveyed in [50]. Therefore, we will not include clustering in our discussion.

Join execution techniques can be classified into seven categories (see Table 1). The first group of techniques are loop based. The simplest technique in this group is nested loop join. Nested block join significantly improves upon the nested loop join by fully utilizing available large memory space. Nested block with rocking [9, 50] tried to improve upon the nested block join by reusing a part of relation S that is already present in the memory. Nested loop with index join uses selection index (such as B+ tree, which was originally designed to facilitate selections) instead of scanning over all tuples of S.

The second group of techniques are sort based. Sort-merge join is the simplest one in this group. Since the disk I/Os spent on sorting is the main cost of sort based join techniques, distributive join [43] saves disk I/Os by only sorting the smaller relation R and partitioning the other relation S. Hybrid join [41] uses sorting for R and selection index for S.

The third group of techniques are hash based. Grace hash join [11, 14] hashes both R and S into pairs of buckets and thus transforms a join between two large relations into a series of joins between buckets in the same pair. Hybrid hash join [17] improves upon the Grace hash join when a large memory is available. It uses extra memory space (beyond what is needed for hash buffers) to execute join during the hashing phase. Hash based joins are very vulnerable to data skew. They perform well when partitioning can produce approximately equal sized buckets and perform badly otherwise. Dynamic destaging hash join, adaptive hash join, and partially preemptible hash join are all aimed at improving the stability of hash based join in case of a data skew and changing memory space (due to activities of other processes) on a computer.

Parallel joins constitute the fourth and fifth groups. The applicability of parallel joins depends on the underlying machine to be used. Shared-nothing parallel computer (SN) [32, 33] is gaining popularity because of its scalability and modularity, but the performance of join executions on a SN is highly vulnerable to data skew (this is very similar to hash based serial joins). Shared-everything (shared memory) parallel computer (SM) [48] is much less sensitive to data skew. To alleviate the sensitivity and to reap benefits of both types of machines, shared virtual memory parallel machines (SVM) [46] can be used. A SVM is an SN with a layer of software constructed such that it appears

to the programmer as an SM. In an SVM, nodes help each other so that the effect of data skew is lessened.

Pointer based techniques [37, 38] form the sixth group of join techniques. The term pointer refers to an extra entry attached to each tuple of R and the entry contains PID (page address) of matching tuples in S. Pointers can be regarded as a special type of cross-relational index; so it has a "X" in the "index" column in Table 1.

The last group of techniques, called "Miscellaneous" techniques, consists of those techniques that do not fall into either one of the above categories. A domain based approach [13] stores all attribute values according to their data domains. This approach speeds up join execution by sacrificing the performance of other operations such as selection. A combined access path structure (CAPS) or a general access path structure (GAPS) [7] is an index that is implemented on a relation such that leaf nodes of the index contain pointers to matching tuples in different relations. A join index [26] is a two-attribute relation which stores join attribute values of both joining relations. A pair of matching attribute values from two joining relations is stored in the same tuple of the join index. A join index is a large data structure (the number of tuples in it can be as many as the cross-product of the number of tuples in both joining relations) and its sheer size prevents it from being used in commercial relational databases. Composite B+ tree (B_c tree) method [29] improves upon the join index method by creating a B_c tree, which is a modified $B+$ tree, on each data domain of the join attributes. Each leaf-node contains pointers pointing to matching tuples in joining relations. Join index technique is conceptually simpler than CAPS, GAPS and B_c tree techniques. CAPS and GAPS are very similar to the B_c tree technique. In [29], B_c tree technique is presented in a simpler way than CAPS and GAPS [7], and it is clearly stated [29] that a B_c tree is created on each data domain of some join attributes.

Strictly speaking, signature filtering method is not a join technique. However, just like a selection index, signatures can be used to speed up a join execution by pruning out some unmatchable tuples early in the join execution.

join techniques	**loops**	**sort**	**hash**	**index**	**large M**	**buffer**	**parallel**
1 *Loop based*							
nested loop (NL)	X						
nested block (NB)	X				X		
NB with rocking	X				X	X	
NL with index	X			X	X		
2 *Sort based*							
sort-merge		X			X		
distributive join		X			X		
hybrid join		X		X	X		
3 *Hash based*							
Grace hash			X		X		
hybrid hash			X		X	X	
dynamic destaging			X		X	X	
Adaptive hash			X		X	X	
part. preemptible			X		X	X	
4 *Parallel join (SN)*							
nested block	X				X		X
sort-merge		X			X		X
hash			X		X		X
hash (SVM)			X		X		X
5 *Parallel join (SM)*							
hash loops	X		X		X		X
Grace hash			X		X		X
hybrid hash			X		X	X	X
6 *Pointer based*							
loops	X			X	X		
sort-merge		X		X	X		
hybrid hash			X	X	X	X	
PID-Partitioning			X	X	X	X	
7 *Miscellaneous*							
signature filtering				X	X		
domain based				X	X		
CAPS and GAPS				X	X		
join index				X	X		
B_c tree				X	X		

Table 1. Join techniques and their underlying strategies

4 Loop Based Joins

Loop based joins are those in which at least one of the joining relations, R or S, or both, are wholely scanned in order to have join attribute values be compared with join attribute values in the other joining relation for matching.

Nested loop join: In a nested loop join [2], the tuples of R are read sequentially from the disk into the memory. For each tuple of R, all tuples of S are read into the memory once to have their join attribute values compared. R is called the outer relation and S is called the inner relation. The entire S is scanned once for each tuple of R. The cost of disk I/Os for a nested loop join is given by the following expression:

$$C_{nested\ loop} = \theta \mid R \mid + \mid R \mid (\theta + \mid S \mid -1) \tag{2}$$

After reading each page of R (a random disk access), the whole relation S is scanned. All readings of S are sequential except the first page. The above formula resulted from the assumption that realistically, usually there are many operations running concurrently on the system and the same disk used to store R and S might be used to store other data as well. So, each time a sequential read (read many disk pages located consecutively) starts, even it starts at the location next to the location of the last read, we assume that the first page to be read is always by a random disk access.

This is the simplest join technique. Obviously, it is very costly and hardly has any practical value for executing joins between large relations. It is mainly useful for join executions between small relations (e.g., the join execution between a small memory-resident partition of R and a small memory-resident partition of S. In that case, there is no disk I/O to speak about), and as such, it is often used along with other join techniques only as an in-memory algorithm.

Nested block join: This join technique [42] significantly enhances the performance over the nested loop join. The outer relation R is divided into large chunks that can be held in the available memory space. Each time a chunk of R is read into the memory, the entire S is read, one page at a time, into the memory and is compared with the tuples of R. The number of times S is read is equal to the number of chunks of R, and this number depends on the ratio $\mid R \mid /M$. In a sense, the nested block join trades using large memory space for the gain of the join execution speed.

One way of organizing the available memory is to allocate $M-3$ pages for B_r, the buffer for memory resident part of R, 1 page for B_s, the buffer for memory resident part of S, and 2 page for B_o, the output buffer used to store matched tuples before writing out to the disk. The disk I/O cost for the nested block join is:

$$C_{nested\ block} = (\mid R \mid -\frac{\mid R \mid}{M-3}) + \frac{\mid R \mid}{M-3} * \theta + \frac{\mid R \mid}{M-3}(\theta + \mid S \mid -1) \tag{3}$$

The first two terms represent the cost of reading R and the last term is the cost of reading S. There are total of $\frac{|R|}{M-3}$ random disk accesses in reading R. This is usually a negligible amount of cost. So the formula can be written as:

$$C_{nested\ block} = \mid R \mid + \frac{\mid R \mid}{M-3}(\theta + \mid S \mid -1) \tag{4}$$

From now on, all formulas will neglect this type of small costs. Compared with formula 2, this formula represents a drastic reduction of the number of disk I/Os. The nested block join is a practical join technique and is used in many commercial relational database systems. For a memory-resident chunk of R and a memory-resident page of S, hashing can be used to reduce the number of comparisons between the tuples of B_r and B_s. Specifically, hashing on join attribute values partitions the memory resident tuples of R in B_r into buckets. Collisions are usually resolved by chaining. The same hash function is used to hash tuples in B_s to determine their appropriate buckets. Comparisons between join attribute values of the tuples of R and S are carried out only between corresponding buckets of R and S. The hash table of B_r only needs to be built once. Usually, the extra CPU cost of hashing can be well justified by the savings in the number of comparisons.

Using rocking to reduce I/O: In the nested block join, the entire inner relation S is read from disk each time B_r is refreshed with the tuples from R. However, beginning from the second time, when start reading pages of S, some tuples of S have already been in B_s and these tuples do not have to be read again if reading is carefully arranged.

An improvement is proposed in [9]: assuming the tuples of S are stored sequentially in a file; in the first pass, S is read forwardly from the top to the bottom of the file; in the second pass, S is read backwardly from the bottom to the top of the file, and tuples at the bottom of the file don't have to be read again. In the third pass, S is read forwardly from the top to the bottom again and tuples at the top of the file don't need to be read. The name rocking comes from the fact that the reading of S is done alternatively from the top and from the bottom of the file.

Another implementation of rocking technique [50] is to read S in a circular manner. This implementation can be useful if forward access is more efficient than backward access, or if backward access is simply impossible. Assume B_s can hold up to 100 pages. In the first pass, S is read in sequence: 1, 2, ... , $\mid S \mid$. In the second pass, S is read in sequence: $\mid S \mid$ - 99, $\mid S \mid$ - 98, ... , $\mid S \mid$, 1 , 2, ... , $\mid S \mid$ - 100. Pages $\mid S \mid$ - 99 through $\mid S \mid$ are reused and no disk read is performed for these pages.

We conducted a study to see, analytically, how much savings of disk I/Os can be obtained from using the rocking technique. Let $\mid B_s \mid = b$ and $\mid B_o \mid = 1$. Then the size of B_r is $M - b - 1$. The disk I/O cost of performing the nested block join with rocking is:

$$\begin{aligned} C_{rocking} &= \mid R \mid + \mid S \mid \\ &+ (\frac{\mid R \mid}{M-b-1} - 1) * (\theta + (\mid S \mid - b) - 1) \end{aligned} \tag{5}$$

The first two terms are the cost of reading R and first scan of S. The third term is the cost of subsequent scans of S. $C_{rocking}$ can be regarded as a function f of b, the number of reused memory resident pages of S in each pass of S, i.e., $C_{rocking} = f(b)$. Optimizing on b and regarding all other variables as constants (by finding the values of b that $\frac{\partial f}{\partial b} > 0$ and applying simple min-max analysis), we obtain the result that if $\mid S \mid > M$, $C_{rocking}$ increases monotonically with the increase of b. This simple analysis proves that $C_{rocking}$ reaches minimum if $b = 0$, i.e., rocking has no effect on the performance of the nested block join.

Using an index to reduce I/O for inner relation: Indexes such as B+ tree can greatly reduce the number of disk I/Os to fetch a particular data value [4]. However, using an index means there is an additional cost of using extra disk storage space for the index itself and using extra disk I/Os for accessing the index.

In the nested loop join, when indexes are available on the join attribute of the entire inner relation S, usually only a few disk reads are suffice to fetch matching tuples from the inner relation for each tuple of the outer relation. This represents a significant saving in the number of disk reads if the join selectivity is low. To further reduce the number of disk reads, memory buffer for S should be made as large as possible. Note that this preference is just the opposite of that in the nested block join, where as much memory space as possible should be given to the buffer of R.

On the negative side, disk accesses through an index are often random. This randomness is reflected by factor $\theta_{indexed}$ in our cost formula below. $\theta_{indexed}$ represents the average time cost of a disk access of a sequence of random and sequential disk accesses.

$\theta_{indexed}$ should be smaller than θ because large memory buffer is allocated to S, i.e., a part of S is in the memory to reduce the number of disk reads. $\theta_{indexed}$ can be estimated by $\theta_{indexed} = \theta(\mid S \mid -0.5 * M)/ \mid S \mid$. This estimate is obtained by assuming only half of the memory is used as the buffer for a portion of S to stay in the memory to reduce the randomness in disk accessing to S; the other half is used for the buffer for R and the memory resident part of indexes. 0.5 is a tentative number and other reasonable number can be used, but the difference is not very critical. There are several reasons behind this. First, if too large a portion of the memory space is allocated to the buffer of S, then there will not be enough memory space for holding indexes, which is also important. Also, for join in large databases (this is what we deal with in this paper), $\mid S \mid$ is much larger than M, so $(\mid S \mid -0.5 * M)/ \mid S \mid$ is very close to 1. The disk I/O cost is:

$$C_{nested\ block\ index} = \mid R \mid + \theta_{indexed} Y_{io}(\alpha\beta \mid\mid R \mid\mid, \mid S \mid, \mid\mid S \mid\mid) \tag{6}$$

where

$$Y_{io}(k, m, n) = Yao(k, m, n) + Yao(k, n/FO, n) + T3(k, n), \tag{7}$$

and

$$T3(k, n) = Yao(Yao(k, n/FO, n), n/(FO * FO), n/FO). \tag{8}$$

FO is the fan out of non-leaf nodes [34]. The formula $Yao(k, m, n)$ is the number of disk pages accessed for retriving k records randomly distributed in n records which reside in m disk pages, without any search overhead, and is given by Yao, S. B. [5]:

$$Yao(k, m, n) = m - m * \prod_{i=1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1}. \tag{9}$$

Signature methods [15, 27, 30] can also be regarded as a type of index and can be used to speed up join executions by filtering out unmatchable tuples. Instead of scanning entire S to find matches with the memory resident tuples of R, the search can start with signatures. Signatures are 0/1 bit patterns which are obtained by hashing data values. Many signatures can be superimposed (bitwise OR) to form a new signature. So, one signature can correspond to many data values and therefore, signature is a way to compactly store information. By examing signatures, possible matching tuples in S can be selected and read into memory and used to probe the hash table of R to find a match. The extra step of going through signatures saves many unnecessary disk reads of unmatchable tuples of S.

Summary for nested loop based joins: The size of total available memory and the size of memory space allocated to various buffers of R, S and output buffer of joined tuples are critical to the performance of various nested block joins.

If the outer relation R can fit entirely in the memory, then the I/O cost of the nested block join execution reduces to a linear scan of R and S:

$$C_{loop} = \mid R \mid + \mid S \mid \tag{10}$$

For large databases, the available memory is often much smaller than each of the joining relations and the performance is not as good as many other fast join techniques. Nested loop/block joins are still widely used in today's commercial databases. The main advantages of these techniques are their simplicity and there are no intermediate relations need to be generated. For joins performed entirely in the memory, such as a join between some small in-memory partitions of relations, these techniques remain competitive and therefore, they can be used along with other join techniques. Also, nested loop/block joins are the only well-known join techniques that can work for complex join predicates [42].

5 Sort Based Joins

Sort based joins are those which have a preprocessing phase (sorting phase) in which at least one of the joining relations, R or S, or both, are sorted according to their join attribute values. The sorted relations are then used in the join and the join execution is expected to be faster due to sorting.

Sort-merge join: This join technique [4, 17] sorts both joining relations on the join attributes and then merges two sorted lists. Joining is performed along with merging. In the merging phase, the tuples of R are read into the memory in sorted order, so are the tuples of S. Suppose both R and S are sorted based on ascending order of their join attribute values, then tuple r_1 which has the smallest join attribute value in R is read into the memory and tuple s_1 which has the smallest join attribute value in S is also read into the memory. If r_1 and s_1 have the same join attribute value, a match is found and a joined tuple (r_1, s_1), which is a concatenation of r_1 and s_1, can be written to the output buffer. If the join attribute value of r_1 is smaller than the join attribute value of s_1, then r_1 is discarded and r_2, the tuple with the smallest join attribute value in the remaining part of

R, is read into the memory. r_2 and s_1 are compared and either a match is found or one of them is discarded. The above process is repeated until all tuples in one of the joining relations are finished and the merging phase stops because all the remaining tuples in the unfinished joining relation have no chance of forming joined tuples. A study by Blasgenm and Eswaran [3] shows that the sort-merge join can almost always outperform the nested loop join when no index is used. However, if there are too many duplicate join attribute values, the performance of the sort-merge join can degenerate to the performance of the nested block join [39].

In the sort-merge join, it is possible to take advantage of prefetching in the merging phase since tuples are sorted. This feature can make a big difference in the performance if the memory space is large and fast disk devices (they are fast in sequential disk accesses) are deployed. The large input buffers of R and S in the memory used during merging phase can prolong the length of sequential disk read and reduce the randomness in the disk access [52].

The sort-merge join is useful for joins between large relations without indexing supports [42], especially when the join predicate does not offer much filtering (i.e., the join result is large). The sort-merge join technique has an additional advantage over the nested block join in that it can efficiently handle inequality join.

The cost of disk I/Os for the sort-merge join is:

$$C_{sort\ \ merge1} = (3 + \theta_{runs}) \mid R \mid + (3 + \theta_{runs}) \mid S \mid + \mid R \mid + \mid S \mid \tag{11}$$

θ_{runs} represents the average time cost of a disk access of a sequence of random and sequential disk accesses in reading sorted runs into the memory for merging into a completely sorted list. A fast sorting algorithm [1] is assumed to be used in the sort-merge join. Both R and S are first sorted into runs of approximately size of $2M$ by a sequential scan and runs are then merged. If available memory is large enough to offer at least one memory page to each run as its own buffer, the merging of the sorted runs can be accomplished within one pass of R and S. There are no more than $\mid S \mid /(2M)$ runs for each relation. So, the minimum memory space needed for implementing the fast sorting algorithm is $M > \sqrt{\mid S \mid /2}$. Disk reads and writes to build sorted runs are largely sequential. But reading sorted runs from the disk for merging is somewhat random (unless each of these runs is served by an individual disk head and we do not consider this case) and its cost is represented by θ_{runs}. If the size of each memory buffer of each run is much larger than one page (this requires M to be much larger than $\mid S \mid /(2M)$), θ_{runs} will be much smaller than θ, but still larger than 1. The reading of runs can be arranged as

follows: (we borrowed this idea from [52]) if memory buffers allocated for runs are large enough, e.g., 100 pages (this means $M > 10\sqrt{\mid S \mid /2}$ is required), each time the pages of a run need to be read, 100 pages of the run are read from the disk in a sequential disk access, except the reading of the first page. The tuples in a run are assumed to be stored consecutively on the disk. The writing out of merged runs to the disk is sequential. In formula 11, the first term is the cost to sort R, including one sequential read to build sorted runs, one sequential write for writing out sorted runs, one random (θ_{runs}) read for reading sorted runs into the memory for merging, and one sequential write for writing out the completely sorted list. Similar explanation can be given to the second term. The third and the fourth terms are the cost of merging and joining the completely sorted lists of R and S.

If $M > \sqrt{\mid S \mid}$ (which is true in most of today's computing environments), we see that it is not even necessary to merge the sorted runs of R and S before joining. There are enough memory pages to offer at least one page to each run of both R and S. After the first scan to build sorted runs, each run of R and S is allocated at least one page (according to the discussion above, allocating more pages for the buffer of each run is preferred in order to reduce randomness in reading the disk) in the memory and tuples in runs of both relations are read into the memory in the sorted order, and are merged and joined, if possible. In this case, the disk I/O cost is:

$$C_{sort\ \ merge2} = (2 + \theta_{runs}) \mid R \mid + (2 + \theta_{runs}) \mid S \mid \tag{12}$$

Compared with formula 11, formula 12 has two less sequential disk I/O cost for each of R and S: the writing out and the reading in of sorted lists are avoided. The disk I/O cost of the sort-merge join is linear with respect to $\mid R \mid + \mid S \mid$. If the size of R and S is very large, the sort-merge join can clearly outperform the nested block join.

Distributive join: Distributive join [43] tries to save the sorting cost by sorting only one of the relations, and only partitioning the other relation. The cost saving occurs at the sorting phase of the join execution.

The Tuples in relation R are sorted by join attribute values and then partitioned in $m+1$ partitions R_1, ... , R_m, R_{m+1}. A join attribute value in a tuple of R_i is less or equal to a join attribute value in a tuple of R_j, if $i < j$. A distribution table is built out of R. If v_i is the maximum join attribute value in R_i, then the distribution table contains v_1, ... , v_m. The distribution table is used to partition S into partitions S_1, ... , S_m, S_{m+1} such

that any join attribute value in a tuple of S_i lies between v_{i-1} and v_i, with $v_0 = -\infty$ and $v_{m+1} = +\infty$. Inside each partition S_i, the tuples are not sorted.

In the joining phase, partitions of R are read into the memory one at a time, assuming that the memory can hold each individual partition of R. While a partition R_i is in the memory, pages of the relation S_i are read into the memory one at a time. At any given moment, only one whole partition of R and one page of S reside in the memory. The join then takes place entirely in the memory between the partition of R and the page of S using nested loop join, benefiting from the fact that the partition of R is sorted. The size of each partition of R should be large enough as long as it can fit entirely in the memory along with at least one page (actually, the more the better) for the input buffer of S and at least one page (again, the more the better) for the output buffer.

The cost of disk I/Os for the distributive join is:

$$C_{distributive} = (4 + \theta_{runs}) * \mid R \mid + (2 + \theta_{part}) * \mid S \mid \tag{13}$$

The first term includes the I/O cost of sorting R (costs $(2 + \theta_{runs} + 1) \mid R \mid$, using a fast sorting algorithm [1]. The sorted list is written back to the disk) and the second term includes the I/O cost of partitioning S (costs $(1 + \theta_{part}) \mid S \mid$) using the distribution table. Also included is the cost of sequentially reading sorted R and partitioned S from the disk for joining. During the partitioning of S, pages of different partitions of S reside in their memory buffers and are flushed out only when the buffers are full. A random disk write followed by a short burst of sequential disk writes is observed for flushing out the content of a buffer. θ_{part} is the average time for a disk access of a sequence of random and sequential disk accesses. θ_{part} should be approximately equal to θ_{runs}. The randomness represented by θ_{runs} is caused by reading pages of sorted runs during merging of sorted runs. Since it is difficult to estimate how many pages should be read for each run, a pre-determined number, for instance, 10 or 100, must be used as the number of pages to be used from each run. In comparison, during the partitioning of S, the largest buffer of runs can be flushed out first and the emptied memory pages can be reused by other buffers. This results in longer span of sequential disk I/Os. From this analysis, θ_{part} can be made smaller than θ_{runs}.

Comparing formulas 12 and 13, cost $C_{distributive}$ can be smaller than cost $C_{sort\ \ merge2}$ only if θ_{part} is substantially smaller than θ_{runs} and $\mid R \mid$ is much smaller than $\mid S \mid$.

Using indexes along with sorting: Hybrid join [41] combines 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 idea is to sort R and access S via the index. If matches are found, then matching tuples of S are fetched. The disk I/O cost is:

$$C_{sort\ \ hybrid} = (4 + \theta_{runs}) * \mid R \mid + \theta_{indexed} Y_{io}(\alpha\beta \mid\mid R \mid\mid, \mid S \mid, \mid\mid S \mid\mid) \tag{14}$$

The first term is the cost of sorting R and reading sorted list of R into the memory for joining. The second term is the cost of retriving pages containing matching tuples from S. As in the nested block join with indexing, $\theta_{indexed}$ reflects the randomness in accessing the pages of S and it can be the same or slightly smaller than θ because a part of S can reside in the memory to reduce the number of disk I/Os. Comparing with cost formula 13, it is clear that the use of index is favored when $\alpha\beta \mid\mid R \mid\mid$ is small.

Summary for sort based joins: The cost of disk I/Os is linear with respect to the size of relations in various sort based joins. Sort based joins perform better than nested loop/block join when both joining relations are large, no indexes are provided, and the join result is large (so the use of indexes is less beneficial).

When the join selectivity is low, most tuples are unmatchable, the efforts invested in sorting and merging of tuples in the joining relations is wasteful. In this case, indexed technique can be a better choice.

6 Hash Based Joins

There are two basic ideas underlying hash based joins. The first is the partitioning of both R and S by hashing using a hash function ϕ_1. Both relations are hashed into the same number of partitions using the same criteria for partitioning. These partitions are in pairs: tuples in one partition of R can only match with tuples in the pairing partition of S. The partitioning should be done such that every partition of R are approximately of the same size and each one can reside entirely in the available memory buffer. This ensures that later during the joining phase, S only needs to be read once. The second idea is that during the joining phase, each time an entire partition of R is read into the memory, a hash table is built from it (i.e., it is further partitioned into many small buckets inside the memory buffer, possibly using a different hash function ϕ_2). The pages of the pairing partition of S are read into the memory one at a time. The tuples in the memory resident page of S are hashed using hash function ϕ_2 to appropriate buckets of the hash table of R to be compared for possible matches.

Hash based joins follow the divide-and-conquer principle and aim to turn the join execution between two large relations into a series of join executions between pairing partitions of the relations.

Simple hash join: Simple hash join is [17] a nested block join with the modification that hashing is used for the in-memory processing. Specifically, first as many pages of R as possible are read into the memory and built into a hash table. Then all pages of S are read into the memory one page at a time. The tuples in the memory resident page of S are hashed and used to probe the memory resident hash table of R. If matches are found, joined tuples are stored in a output buffer and will be flushed out to the disk when the output buffer is full. After a whole scan of S, subsequent pages of R are read into the memory and the same process repeats.

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

Grace hash join: Grace hash join [11, 14] clearly separates two phases of the hash join. In the first phase, each of R and S is partitioned into the same number (k) of buckets by hashing on the join attribute and using a same hash function. These buckets form k pairs. In each pair, one bucket is from R and the other is from S. At least one of the buckets in every pair must fit in the available memory. If not, additional partitionings may be needed until this is true. In the joining phase, nested looping or hashing is used to join two buckets in each pair.

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

$$C_{Grace\ \ hash} = (1 + \theta_{buckets})(\mid R \mid + \mid S \mid) + (\mid R \mid + \mid S \mid) \tag{15}$$

Hashed results are kept in the memory buffers. A random disk write is followed by a short burst of sequential disk writes when flushing out the content of a memory buffer to a disk resident bucket. $\theta_{buckets}$ is the average time of a disk access of a sequence of random and sequential disk accesses. $\theta_{buckets}$ has a value comparable to that of θ_{part} (see discussions on distributive join in Section 5) and smaller than θ_{runs}. $\theta_{buckets}$ can be close to 1 if output buffers of buckets are large. A Grace hash join can generally be expected to

outperform the simple hash join. But, in situations where most of the R can be fit in the memory, the I/O cost of the simple hash (nested block) join reduces to almost linear scan of each relation, and may very well outperform the Grace hash join because the Grace hash join incurs extra cost of partitioning.

Hybrid-hash join: Instead of using extra memory space to set up more buffers as in a Grace hash join, hybrid hash join [17] uses extra memory to execute a join during the hashing phase. R is hashed on the join attribute and partitioned into k buckets R_i, $0 \leq i \leq k-1$. The tuples from R_0 (anchor bucket) is used to build an in-memory hash table and the remaining $k-1$ buckets are written back to the disk. The hash function, bucket criteria, and number k are chosen so that the hash table of each bucket R_i of R, which is slightly larger than $\mid R_i \mid$, can fit in the available memory. The same hash function is also used to partition S into k buckets S_i, $0 \leq i \leq k-1$. The tuples in S_i are joinable only to the tuples in R_i. Except S_0, all other $k-1$ buckets are written to the disk. The tuples belong to S_0 are used to probe the memory resident hash table of R_0 for matches immediately after hashing while scanning S; and are dropped unless they can join with the tuples in R_0. In the joining phase, remaining $k-1$ buckets of R and S are joined through a series of small joins just like a Grace hash join.

The number of bucket pairs, and thus the number of small joins, is usually determined by the size of the smaller relation R, and is independent of the size of S. However, this is true only when partitions are approximately of equal size. An approach like the one used in the Grace hash join can be adopted: after the partitioning phase, for each pair of partitions of R and S, it is always the smaller partition to be read entirely into the memory and the other one is read into the memory one page at a time. For a pair of partitions of R and S, recursive partitioning may be needed if neither one in the pair can be fit in the memory.

The cost of disk I/Os of the hybrid hash join is:

$$\begin{aligned} C_{hybrid\ \ hash} &= \mid R \mid + max\{0, (\theta_{buckets}+1)(\mid R \mid - \mid R_0 \mid)\} \\ &\quad + \mid S \mid + max\{0, (\theta_{buckets}+1)(\mid S \mid - \mid S_0 \mid)\} \end{aligned} \tag{16}$$

The first two terms represent the cost to read in R, write out hashed result of $(R-R_0)$ to hash buckets, and read in buckets of $(R-R_0)$ to participate in the join. The writing of buckets is a somewhat random disk access and this is reflected by $\theta_{buckets}$. $\theta_{buckets}$ can be reduced by reserving larger memory space for output buffers. However, the output

buffer size is restricted by the fact that R_0 takes up much of the memory space. Similar explanation can be given for the second term.

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 hashing phase. So there is a trade off between the size of R_0 and the size of the buffers of buckets. Though less critical, it is also preferable that the size of the rest of buckets of R is large. This affects the number of times to re-initialize hash tables in the memory.

A Grace hash method is relatively insensitive to the size of available memory since the amount of the memory affects only the number of times hash tables are re-initialized and does not affect disk I/Os. In comparison, the hybrid hash join is much more sensitive to the size of the available memory because it takes advantage of a large memory space. If the memory space is small, the performance of the hybrid hash join may degrade to that of the Grace hash join. The hybrid hash join is also very sensitive to data skew because if the anchor bucket is too small or too large, either the use of the anchor bucket becomes ineffective or a bucket overflow occurs.

There are some other variations of the hybrid hash join. Most of them aim at improving the hash join in case a severe data skew results in very uneven partitions.

Dynamic destaging strategy (DDS): In the standard hybrid hash join, the anchor bucket is pre-determined. This strategy could have problems since the anchor bucket may be too large to fit in the memory or too small to be effective. Because usually data distribution is unknown before a join, the decision to choose the anchor bucket should be delayed as much as possible [28].

In DDS, the intended bucket size is chosen to be much smaller than the intended partitions to avoid bucket overflows. In fact, no bucket size is 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 the memory. Each bucket is allocated one page in the memory to begin with and it grows as scanning progresses. Whenever the memory becomes full, the content of some pages in the memory buffer should be paged out. The bucket to be chosen for output is the one that has already been paged out, or in case such a 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 a partially filled page. This process continues until all tuples in R and S are scanned in the hashing phase. The decision of which bucket or set of buckets serves as the anchor bucket is made dynamically. After the partitioning of

R and S, buckets are grouped into the partitions of approximately the equal size. The joining 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 benefits are the possible time saving due to a reduction of bucket overflows, reduction of the number of re-initialization of hash tables, and an appropriately sized anchor bucket, all are made possible by bucket tunning.

Adaptive hash join (ADH): Adaptive hash join technique [39] is similar to DDS, and can be considered as a generalization of it. This join technique can adapt to the change of size of available memory during a join execution.

In ADH, the number of buckets is large, pre-determined, and never changes. Initially, each bucket is allocated to a buffer of size B. A buffer may have several buckets. The large number of buckets is compensated by the sharing of buffers; so the number of buffers is comparable to that in the hybrid hash join. A group of buckets that share the same buffer constitutes one cluster. A buffer is the memory resident part of a cluster.

First, the pages of R are read. If a buffer grows beyond its limit B, and if enough memory is available and the cluster has not done a page out before, B more pages of the memory are allocated to the cluster. If not enough memory is available or the cluster has a page out before, the contents of the buffer are output and all but the last B pages of buffer are freed. If all clusters are output and still the memory is not enough even for a B page buffer of each cluster, then B is decreased. After writing out the content of the buffer (because memory is not enough to hold all buffers), the cluster (on the disks) that just received output pages is checked. If the cluster can not fit into a memory of size M and the cluster contains more than one bucket, then the cluster is split into two smaller clusters. This means two clusters share the same buffer in the memory. However, new memory pages to be allocated to buffers are not shared. If a cluster is too large but contains only one bucket, then do nothing.

After scanning of R is finished, scanning of S starts. Now, there are buffers of R in the memory and these buffers collectively serve as the anchor bucket in the hybrid hash join. If a tuple of S falls in the anchor bucket, this tuple is used to probe the anchor bucket of R for matches. If matches are found, joined tuples are formed and written into the output buffer. The contents of the output buffer are flushed to the disk when it is full. If no matches are found, this tuple is discarded.

ADH follows the same philosophy as that of DDS. The splitting of clusters is the main new feature which better suits the situation when the size of available memory

decreases during the join execution.

Partially preemptible hash join ($PPHJ$): $PPHJ$ refers to a class of hash joins [45] and both DDS and ADH can be regarded as special cases of $PPHJ$. In this paper, we only discuss the most sophisticated version of $PPHJ$, i.e., with dynamic expansion and contraction of memory buffers and prioritized spooling policy. This technique is more flexible than either DDS or ADH. In addition to writing excess pages of tuples out to disk when memory is full, $PPHJ$ also reads in the pages of tuples when more memory space becomes available.

$PPHJ$ sets the number of partitions to be $\sqrt{F* \mid R \mid}$, where F is a fudge factor which represents a slight increase in the space requirement of R due to the overhead of making it into a hash table. If R is partitioned evenly, the size of each partition is about $\sqrt{F* \mid R \mid}$.

Both DDS and ADH allow only the buckets of R to be moved out of the memory during the scanning of R. $PPHJ$ also allows this to be done during the scanning of S, when the memory availability decreases. In addition to that, $PPHJ$ also moves buckets of R into the memory when more memory pages become available. Unlike DDS or ADH, $PPHJ$ does not combine buckets of R into large partitions and it lets each bucket itself to be a partition. This decision affects the number of pages used as buffer space in the memory.

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 pages be flushed out first) spooling strategy. In contrast, $PPHJ$ uses prioritized spooling strategy. The pages from R are more likely to be memory resident than the pages from S, i.e., when memory space is not enough, the pages of S are more likely to be flushed out of the memory and when more memory space becomes available, the pages of R are more likely to be read into the memory.

Experiments [45] show that if memory availability fluctuates, the hybrid join does not have a satisfactory performance. Unless memory availability fluctuates too fast, $PPHJ$ has a substantially better performance than DDS and ADH for a wide variety of parameter settings. The reduction in the execution time comes mostly from dynamic expansion and contraction of buffers. The priority spooling also has some effect on the reduction of the execution time.

Shin's join algorithm ($SHIN$): In $SHIN$ [49], both joining relations are repeatedly divided by up to five statistically independent hash functions. The number five is not a

magic number and other appropriate numbers can be used. Buckets obtained from applying the first hash function are partitioned 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 same sequence of hash functions and using the same criteria. If B_r is empty, then all the tuples in B_s can be safely dropped. The dropping can happen at any stage of the hashing process and the goal 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 hash functions 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 [49] it is recommended to use a linked list structure for tuple storage in buckets. $SHIN$ does not need a preprocessing step to estimate hash table sizes. It sets up a fixed number of hash table buckets. $SHIN$ is simple to implement. However, it could happen that after repeated hashing, there are almost no or almost all buckets are dropped. In either case, $SHIN$ does not give a good performance. So $SHIN$ is sensitive to data distribution.

Summary for hash based joins: If tuples are initially fully or partially sorted, sort based joins may very well outperform hash based joins since in this case, sort based join execution reduces to one linear scan of each relation.

Hash based joins can usually outperform sort based joins [8, 14, 17, 19, 22, 25]. However, in most cases, the performance difference is not large [47].

Because we included random access factor $\theta_{buckets}$ in our cost formulas, we concluded that even in the hybrid hash join (it needs a large portion of the memory to hold the anchor bucket), the size of output buffers of each bucket in the memory should not be too small. Otherwise, the writing out of the content of the memory buffers of buckets will be too random.

In [47], Graefe et al. presented a detailed comparison between sort based joins and hash based joins. Their conclusion was that these two classes of join techniques are to a large extent complementary to each other. The duality stems from the fact that hashing is the opposite of merging. Contrary to previous beliefs, their conclusion is that the criterion to choose between a sort based join and a hash based join is neither the relation size nor the memory size, but the following:

- The relative size of the two relations. Hashing makes a number of pairs of buckets and in each pair, one bucket is from R and the other from S. Hashing stops as soon as at least one bucket in every pair can fit in the memory. In case of sorting, the ratio of $\mid S \mid$ to the size of available memory dominates the cost of sorting.

- Possibility that partitions generated by hashing are uneven. This is not a concern for the sort based joins.

- The number of duplicate values [39]. If there is a large number of duplicate join attribute values, the sort-merge join is at least able to degrade to the performance of the nested block join while the hash based joins can perform even worse than the nested block join due to high overhead caused by heavy hash collisions.

  Since the number of duplicate values in join attributes are positively correlated to the size of join result, an estimated large join result can also be used to indicate the favorable condition for use of the sort based joins.

- Opportunities to explore interesting ordering, such as that the tuples on the disk are already sorted based in the order of join attribute values. Sorting is the primary source of the disk I/O cost in a sort based join execution. If sorting has been done, the sort based join execution reduces to a linear scan of two relations. In addition, a request that the join result be sorted also favors the use of the sort based joins.

Hash based joins usually outperform sort based joins by some noticeable margin, especially when the smaller relation R can be partitioned evenly and is much smaller than S. However, in cases of heavy data skew, i.e., buckets generated by hashing have very different size, the sort based joins can outperform the hash based joins.

7 Some Speed up Techniques for Join Executions

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

Bit vector filtering: Bit vector filtering [8, 16, 21, 24, 33] speeds up a join execution by discarding unmatchable tuples from the larger relation S. First, each join attribute value of R is hashed into a 0/1 bit pattern during the preprocessing (sorting of sort based joins or hashing of hash based joins) of R, and the pattern is used to set some bits in

an initially all-zero bit vector by bitwise OR. During the preprocessing of S, every join attribute value of S is hashed using the same hash function and the hashed 0/1 pattern is compared with the bit vector. If there is one or more bits on which the 0/1 pattern from the tuple of S cannot be matched by the bit vector, the tuple of the 0/1 pattern is discarded because it is a unmatchable tuple. If R is much smaller than S (so the bit vector is created from not too many values from R and is less likely to be saturated by bits of 1's), and a large number of the tuples of S can be filtered out by checking with the bit vector, the saving of the disk I/O by employing the bit vector is significant.

The filtering can be done very fast if the bit vector is implemented in the hardware [8]. For a large relation, an array of bit vectors can be used to do the filtering and this array can be kept in the memory while scanning S. This method actually has another name, i.e., the signature method [12, 20].

A special type of the bit vector is called Babb array [8]. It is a boolean array that is built during the scan of R and each bit in the array corresponds to a hash bucket. A bit in the Babb array is set when there are tuples of R hashed into its corresponding bucket. During the scan of S, a tuple of S is discarded if its corresponding bit in the Babb array is not set.

Semijoin: The semijoin was initially proposed in distributed database environments to reduce the cost of data transmission between sites when executing a join [10]. However, it can be applied to centralized systems as well. The following is a semijoin of R with S:

1. project R on its join attribute, obtain $\pi(R)$.
2. join $\pi(R)$ with S, obtain $semi(S)_R$ ($\subseteq S$).

After this, $semi(S)_R$ can be used to join with R to achieve the effect of an equijoin. Assume that $\pi(R)$ can fit in the memory, then the disk I/O cost of a join execution using a semijoin is:

$$C_{semi} = 2 \mid R \mid + \mid S \mid \tag{17}$$

The first term include the cost of scanning and projecting R to obtain $\pi(R)$ and also the cost of scanning R to join with $semi(S)_R$ after $semi(S)_R$ becomes available. The second term is the cost of scanning S to obtain $semi(S)_R$. Note that all disk I/Os are sequential.

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 scanning S. This is a similar but more generalized method than the semijoin. Other more general methods than the semijoin can be found in [18, 25].

Eliminating random I/O: A hash join has two phases: partitioning and joining. In [52], disk I/Os are divided into four phases: PR (partitioning phase read), PW (partitioning phase write), JR (joining phase read), JW (joining phase write). Hash buckets are comprised of bucket segments. Each bucket segment is a physically contiguous storage area on a disk. several bucket segments of the same bucket form a write group.

The available memory space is divided into three parts: input buffer b_i, staging buffer b_s, and the output buffer b_o. The staging buffer holds bucket contents in the partitioning phase and hash table in the joining phase.

Before partitioning, no memory pages are initially allocated to b_s. As tuples from either relation are read into b_i and are hashed into the buckets, the memory pages are being allocated to b_s. When a new tuple is hashed to a full bucket in b_s and no free memory pages are available for allocation, some pages in some buckets have to be written out to the disk. The key technique adopted in [52] is to write out as much a contiguous collection (a write group) of buckets as possible. This reduces randomness as much as possible during PW and JR.

We believe that this technique is universally applicable to almost all join techniques. As the disk technology advances, the time ratio of a typical random disk access to a sequential disk access increases. So, today, there is a greater need to reduce the number of random disk accesses in the join execution. Also, the size of memory becomes increasingly larger, this makes the use of larger input and output buffers (I/O buffers) in memory more feasible. We assume that large I/O buffers are used whenever possible, and develop cost formulas accordingly throughout this paper to ensure a fair comparison between join techniques. The reduced randomness through the use of large I/O buffers is reflected in cost formulas by various random factors (θ, θ_{bucket}, etc). Our experiments show that the effect of applying this technique to a join technique, if applicable, is usually noticeable.

8 Parallel Joins

The type of the parallel join technique that can be employed depends very much on the structure of the underlying hardware. We focus on join techniques based on two classes

of architectures: shared-nothing and shared-everything parallel computers. The current trend in technology is that the share-nothing architecture is winning among different parallel architectures because of its good scalability and high bandwidth (up to 200MB/s) in interprocessor network [46].

We compare different parallel joins by the maximum bandwidth requirement (MBR) and the time cost. The MBR is the maximum bandwidth needed to ensure that the parallel join is executed at its highest speed, i.e., no congestion in the network. The time cost is the time to execute the join parallelly. When the MBR exceeds the bandwidth of the network, the time cost of a join technique becomes worse than what is indicated by its time cost formula.

8.1 Shared-Nothing Parallel Machines

Shared-nothing parallel machines are also called the distributed memory or the message passing parallel machines. Each processor has its own local memory and disk [21] as shown in Figure 2, and the time spent on interprocessor data communication is significant enough to be considered in the overall time cost. Each processor in a parallel machine has a much smaller processing capacity as compared to a processor in a single processor machine, but the collective processing capacity of all processors in the parallel machine can be much greater than that of a single processor machine.

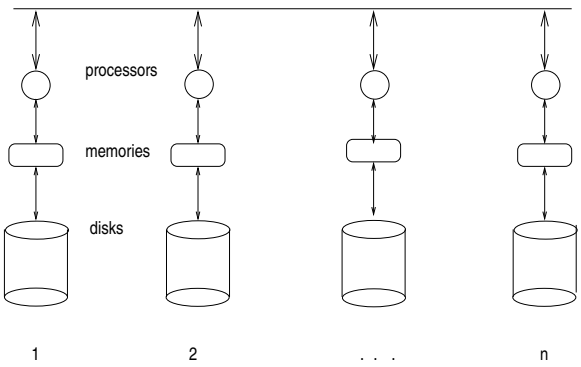

Figure 2: A share nothing parallel machine

Factors that affect the speed up in parallel joins: The interprocessor data communication time and the synchronization overhead of various parallel join techniques critically affect the performance of parallel join techniques. The interprocessor data com-

munication time can be analyzed statically. The synchronization cost is sensitive to data skew and the stochastic nature of the join execution [32].

Comparison of three classes of parallel joins: We now compare the performance of three join techniques, i.e., parallel nested block join, parallel sort-merge join, and parallel hash join. In the parallel nested block join, each processor first reads its local partition of R from the local disk, and then reads in the entire S from the local disk as well as the disks of all other processors to do the join. Let γ be the time spent on sending a page of data over the network and n be the number of processors. Assume that the local memory of each processor is large enough to hold the local partition $(1/n)$ of R. Then, the time cost is:

$$C_{parallel\ nested} = \mid R \mid /n + \mid S \mid /n + (1+\gamma)(1-1/n) \mid S \mid \tag{18}$$

The first and the second terms are the cost of reading local partitions $(1/n)$ of R and S into the local memory respectively. The third term is the cost of reading non-local part $(1-1/n)$ of S and sending it over the network into the local memory, assuming the local memory can accept only one incoming data communication thread, i.e., all incoming data from other processors have to be queued for entering local memory if their transmission time overlap. The above cost is $O(\mid S \mid)$ and this order does not depends on n. Thus, it is advisable to choose the larger relation as R and have the pages of the smaller relation to be sent over the network.

In case all data communications can proceed parallelly, all incoming data have to be first written into the local disk and then to be read sequentially into the local memory. So, the second term becomes $(1+\gamma+1) \mid S \mid /n + (1-1/n) \mid S \mid$. This cost is also in the order of $O(\mid S \mid)$. Surprisingly, parallel data transmission does not make much difference for the parallel nested block join. The MBR for the network is $C_{nested\ MBR\ total} = n(1-1/n) \mid S \mid$, and the local MBR for each processor (provided by the part of the network connected to this processor) is $C_{nested\ MBR\ local} = (1-1/n) \mid S \mid$.

In parallel sort-merge join, first each processor sorts local partitions of R and S in its local disk and then the sorted partitions are merged with sorted partitions from other processors to produce larger sorted lists. The merging is repeated until the total number of merged (and sorted) lists of R and S is less than the number of available memory pages in one of the processors. At this moment, all sorted lists of R and S are read in sorted order into the memory of this processor to be joined.

In merging, the local memory needs to reserve two pages for each participating processor: one for the tuples of R and one for the tuples of S. So, total of $M/2$ processors can participate in a merge. Let $P = (\mid R \mid + \mid S \mid)/n$. The time cost as well as the beginning and ending sizes of each sorted list in each phase, is:

Time cost	*beginning size*	*ending size*
$P(1+\gamma) + P(M/2)$	P	$P(M/2)$
$P(1+\gamma)(M/2) + P(M/2)^2$	$P(M/2)$	$P(M/2)^2$
$P(1+\gamma)(M/2)^2 + P(M/2)^3$	$P(M/2)^2$	$P(M/2)^3$
...	...	...
$P(1+\gamma)(M/2)^(k-1) + P(M/2)^k$	$P(M/2)^(k-1)$	$P(M/2)^k$

Table 2. Time cost and the total size of sorted lists in each merging stage.

The first term in an expression in the left column gives the time to read non-local data and send it over the network, and the second term of the same expression is the time to write out merged list to the local disk. The number of stages k satisfies $P(M/2)^k = Pn$, which is equal to $\mid R \mid + \mid S \mid$, because the last merged list must have the size of $R \cup S$. So, $k = log_{(M/2)}n$ and the total time cost of merging is (obtained by summarizing the cost of k merging stages):

$$C_{merge} = P(1 + \gamma + M/2)(n-1)/(M/2-1) \tag{19}$$

The time cost of initial sorting of local partitions of R and S is $P(3 + \theta_{runs})$ (see discussions on serial sort-merge join in Section 5). Hence, the time cost of the parallel sort-merge join is:

$$C_{parallel\ sort1} = P(3 + \theta_{runs}) + P(1 + \gamma + M/2)(n-1)/(M/2-1) \tag{20}$$

$C_{parallel\ sort1}$ is roughly Pn which is equal to $\mid R \mid + \mid S \mid$. Note that this cost does not depend on n. If in each stage, the merged list is not written to the local disk and directly sent to the memory of other processors for further merging, the initial cost of disk read is P. The time in merging phase i is $P\gamma(M/2)^i$. So, the total cost reduces to:

$$C_{parallel\ sort2} = P(4 + \theta_{runs}) + P\gamma(n-1)/(M/2-1) \tag{21}$$

This is a drastic improvement. However, it adds difficulties to the system implementation as the operation of different processors have to be precisely synchronized. Even with the use of disks to store intermediate merging results, it is difficult to achieve the near perfect synchronization needed during merging to achieve the performance indicated

in 20. The MBR for the network is $C_{sort\ MBR\ total} = Pn$ and the local MBR for each processor to do merging is $C_{sort\ MBR\ local} = P(M/2-1)$. Although $C_{sort\ MBR\ total}$ is much smaller than $C_{nested\ MBR\ total}$, $C_{sort\ MBR\ local}$ can be much larger than $C_{nested\ MBR\ local}$ if $(M/2)$ is much larger than n. This means the transmission capability of the network is not fully used. Our analytical result is supported by [6], which shows that the parallel sort-merge join often perform inferior to the parallel nested block join because the parallel sort-merge join cannot fully utilize all processors and memories at the later stage of merging/joining.

In parallel hash join, each processor hashes its local partitions of R and S in its local disk into buckets. Then, globally, each processor is assigned one bucket. For each processor, all buckets but one are sent to appropriate processors. Then, each processor executes a join between the local buckets of R and S. Unlike the other two parallel joins, there is no data communication over the network in the joining phase. The time cost of the parallel hash join is:

$$C_{parallel\ hash} = P(4 + \theta_{buckets} + \gamma) \tag{22}$$

This cost includes the time to hash local partitions of R and S in each processor ($P(1 + \theta_{bucket})$, see discussions on serial hash join in Section 6), to read and send data over the network and write to the disk of the destination processor ($P(1 + \gamma + 1)$), and then to read all local data (P) to execute the join. The time cost of the parallel hash join is much smaller than the two parallel joins discussed above and the parallel hash join is the only parallel join whose speed increases linearly with the number of processors. The MBR for the network is $C_{hash\ MBR\ total} = (\mid R \mid + \mid S \mid)$ and local MBR for each processor is $C_{hash\ MBR\ local} = (1 - 1/n)(\mid R \mid + \mid S \mid)$. Both global and local MBR's are smaller than those in the other two types of parallel joins. In case the system tends to be communication bound, the parallel hash join perform even better [21].

Vulnerability of parallel hash joins: The parallel hash join in the shared-nothing parallel machines has been demonstrated to be very sensitive to data skew and the stochastic nature of the processing time (caused by randomness of data placement on the disk, random delay in data communication, the random nature of locations of disk heads, etc.) in the shared-nothing parallel machines. It was shown [32] that with 5% data skew, even if the collective processing power of a parallel machine is 10 times that of a single processor machine, there can be no performance advantage of using the parallel hash join.

The stochastical nature of the join processing time also exacts a similar penalty on the performance of the parallel hash join.

However, one needs to be cautious about the conclusion in [32]. In [32], total of 300 processors with 2 MIPS each were used in a parallel machine and the performance is compared to that of a single processor machine of 60 MIPS. An even distribution means each processor should have about 0.33% of the total data. 5% data skew means that instead of 0.33%, one of the processors has 5% of the total data, 15 times higher. They used a large number of very small processors to compare with a very fast processor. Had the number of processors been lowered and the processing capacity of each processor been increased, the sensitivity as shown in their experiment would have been much reduced. Take into account the current hardware technology, it is very conservative to assume the collective processing capability in a parallel machine with 300 processors is only 10 times that of a single processor machine.

Using semijoin as a preprocessing step: Let R_i and S_i be the local portions of R and S on processor p_i and A is the join attribute of R. The semijoin as a preprocessing step for the parallel hash join can be implemented on each processor as follows [36]:

1. Read and project R_i on its join attribute values, obtain $\pi(R_i)$. $\pi(R_i)$ is sorted since projection involves sorting to eliminate duplicates.
2. Send $\pi(R_i)$ to all other processors.
3. Wait to receive all join attribute values of R from all other processors.
4. Merge all the sorted lists and eliminate duplicates, obtain $R.A$
5. Read S_i.
6. Restrict S_i by $R.A$, obtain $semi(S_i)_R$ (or $semi(R.A)_{S_i}$).

If $semi(R.A)_{S_j}$ of every processor j is sent over the network to complete the join, then in each processor i, the union of all received sets $semi(R.A)_S = \cup_{\forall j} semi(R.A)_{S_j}$ is used to select the tuples from R_i to obtain $semi(R_i)_S$. After that, $semi(R_i)_S$ is sent over the network to all other processors to form joined tuples.

Instead of sending $semi(R.A)_{S_i}$, all $semi(S_i)_R$'s can be sent over the network to complete the join and there is no data communication after $semi(S_i)_R$'s are sent. The disadvantage of doing this is that $\sum_{\forall j} \mid semi(S_j)_R \mid$ may be much larger than $\sum_{\forall j} \mid semi(R_j)_S \mid$.

Using the semijoin as a preprocessing step is also very sensitive to data skew. It is shown that with even 5% data skew, using the semijoin has only a marginal effect (figure 14 in [36]). The reason for this is that data skew tends to shift the bottleneck of processing performance from the network to individual processors. Again, one needs to be cautious about the conclusion in [36] 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 lowered the number of processors.

8.2 Shared-Everything Parallel Machines

A share-everything parallel machine is also called a shared-memory parallel machine (SM). Contention for shared-memory is the major limiting factor of join performance in SM. Works on parallel join techniques on SM in the open literatures are far less than those 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.

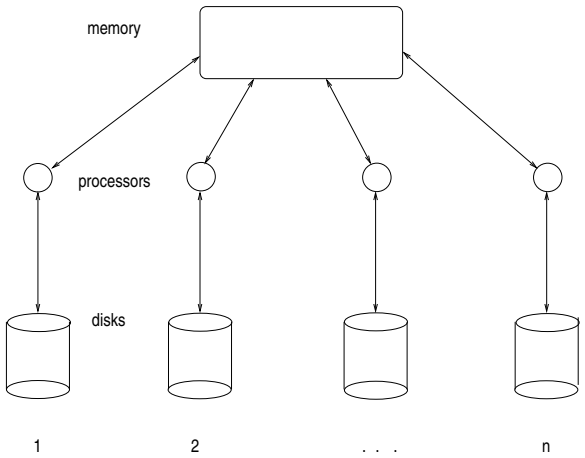

Figure 3: A shared memory parallel machine with separate I/O channels

Separate I/O channels: Since the I/O cost is the dominant cost of executing join, the I/O device structure 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 I/O channels each controlled separately by a processor as shown in

Figure 3. We assume that no disk head contention takes place. This is an ideal situation. Statistical analysis of the disk head contention is beyond the scope of this paper. The analysis in the previous section shows that the parallel hash join generally outperforms the nested or sort based parallel joins. So we only concentrate on various parallel hash joins.

In [48], a SM with 10 processors was used to test various parallel joins: hash loop join, Grace hash join and hybrid hash join. In the parallel hash loop join, in the beginning as many pages of R as possible are read into the memory. Then the followings is done in parallel on each processor: join attribute values of tuples are hashed and pointers to tuples (not tuples themselves) are inserted in buckets of a hash table. There is only one hash table and it is used by all processors. Then the pages of S are read into the memory and used to probe the hash table of R. Matched tuples are written to output buffers and unmatched tuples of S are dropped. This constitutes one pass to S. If there are still pages of R left unexplored, the above process is repeated until all the pages of R are processed. The time cost is:

$$C_{sh_{mem}\ hash\ loop} = \frac{\mid R \mid}{M-n}(\frac{M-n}{n} + \frac{\mid S \mid}{n}) \tag{23}$$

The first factor is the number of partitions to be made out of R due to limited local memory, and is also the number of times to re-initialize the hash table in the memory. $M-n$ represents the need to set aside n pages of the memory as the input buffer for each of the n processors. The first term of the second factor reflects the time to read into the memory a partition of R in parallel by the n processors. The second term of the second factor reflects the time to read S into the memory by the n processors. The time cost of a parallel Grace hash join is:

$$C_{sh_{mem}\ Grace\ hash} = P(2 + \theta_{buckets}) \tag{24}$$

The factor 2 represents one pass to read R and S from the disk to do partitioning and one pass to do joining. $\theta_{buckets}$ reflects the somewhat random disk I/Os to write out the hashing result during the partitioning phase (see discussions on serial Grace hash join in Section 6).

A parallel hybrid hash join improves upon the parallel Grace hash join in that during the partitioning phase, a large portion of the memory is reserved for the first partition R_0 of R. While scanning S, the tuples of S that fall in S_0 will be used to probe the hash

table of R_0 immediately after hashing and do not have to be written out to the disk. The time cost is:

$$C_{sh_{mem}\ hybrid\ hash} = \begin{array}{l} \mid R \mid + \mid S \mid /n + max\{0, (\theta_{buckets} + 1)(\mid R \mid - \mid R_0 \mid)/n\} \\ +max\{0, (\theta_{buckets} + 1)(\mid S \mid - \mid S_0 \mid)/n\} \end{array} \quad (25)$$

The first term is the time to read in the tuples of R and S. The second term is the time to write out buckets of R after hashing, and then read all the buckets again to perform join. R_0 is the anchor bucket and its tuples do not need any disk I/O once they are in the memory. The similar explanation can be given for the third term.

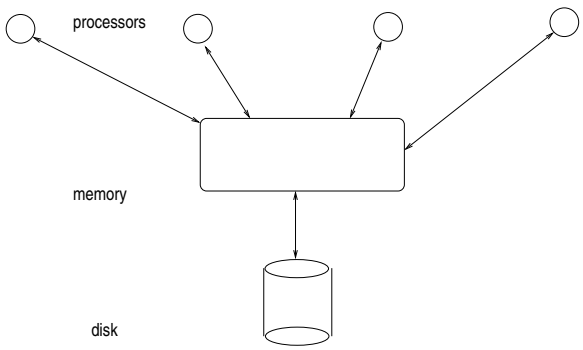

Figure 4: A shared memory parallel machine with shared I/O channels

Shared I/O channels: If a SM has only one or a few shared I/O channels, as shown in Figure 4, the behavior of the parallel machine during join execution is not much different from a single processor machine with a very fast processor. The join execution is likely to be I/O bound due to the high collective processing capacity of many processors. It was found [48] that there is little effect on the performance when the number of processors is increased – of course, it is I/O bound. Martin et al. [48] found that the response time of the hash loop join and the hybrid hash join reduces substantially when the amount of available memory space increases while this increase has virtually no effects on the Grace hash join – this is very similar to the behavior of these joins on a single processor machine. Nevertheless, some observations from [48] are worth mentioning:

- 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 machines. Locking granularity should be as small as possible in space and as short as possible in time.

It is observed [48] that the processing time is even increased when more memory is added in parallel hybrid hash join. This is because in their test, the whole R_0 used a single lock, and the size of R_0 is fixed at 50% of the available memory. Had R_0 been split into many smaller buckets and separate locks were used, the contention would be much less.

- The parallel hybrid hash join is faster than the parallel hash loop join and the parallel Grace hash join for relatively small (less than 20% of $\mid R \mid$) memory sizes.
- The parallel hash loop join has the best performance when the size of the memory varies from 20% to 80% of $\mid R \mid$. This is because in the parallel Grace hash join and the parallel hybrid hash join, the time spent in the partitioning phase is more than the saving of time during the joining phase.
- Effects of data skew to the parallel Grace hash join and the parallel hybrid hash join are small. Compared with [36], this is not only because of the difference between the architectures of a SM and a SN, but also because of the fact that in [48], fewer processors are used in the parallel machine.

8.3 Shared Virtual Memory Machines

Main reasons of using shared virtual memory machine [31, 40, 46] (SVM) are the ease of coding and the ease of conceptualization of shared data structures.

Knowing the sensitivity of the parallel hash join on a SN to data skew, Shatdal et al. [46] proposed a parallel join technique using shared virtual memory to alleviate the effect of data skew in S. The novelty lies in the joining phase. A processor p_1 which just finished its own share of work ahead of others randomly picks a busy processor, say p_2, to help out. The stream of the tuples of S read from the local disk at p_2 is split between p_1 and p_2. p_1 uses the 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 the hash table from p_2's local memory to p_1's local memory. As pages of the hash table in p_2 be copied to p_1, the page fault will be less and eventually stop. p_1 is unaware of the locality of the pages of the hash table during this process due to the shared virtual memory, and this greatly simplifies the programming complexity. The sharing of the hash table is in the read-only mode and hence, there is no conflict between the executions of p_1 and p_2.

To reduce the effect of data skew in the partitioning phase, Shatdal et al. [46] suggested to use the idea proposed in [44], that is to use range partitioning instead of hashing to

partition the tuples of R. The cutoff values of ranges that ensure approximately even partitioning can be obtained by sampling.

8.4 Summary for Parallel Joins

Parallel joins gain performance advantage in join execution by sheer power of more hardware. Parallel computers are commonly classified as shared-nothing and shared-everything (shared-memory) parallel machines. However, this classification is not very helpful in understanding the behavior of parallel joins because the dominant factor in the execution time of parallel joins is the number of disk I/Os, not the number of CPU executions. So, join executions in shared-memory parallel machines with large number of I/O channels, e.g., one I/O channel per CPU, are more similar in behavior to join executions in shared-nothing parallel machines, and far different from join executions in shared-memory parallel machines with only a few I/O channels. In fact, a computer with a large number of parallelly executable CPUs and a common shared I/O channel is not much different from a computer with a very fast single CPU, as far as join execution is concerned, if the size of joining relations are much larger than available memory. For this reason, in this section we mainly focus on discussion of parallel joins in shared-nothing parallel machines and shared-memory parallel machines with large number of I/O channels.

Through cost formulas developed in this section, it can be seen that the performance of both the parallel nested loop join and the parallel sort-merge join are not scale well with the number of parallel processors. Parallel hash joins are, in general, the best parallel join techniques because their performance scales almost linearly with the number of processors. Parallel hash joins on shared-nothing parallel machines are vulnerable to data skew. To use the shared virtual memory parallel machine (it is actually a shared-nothing parallel machine with a layer of software which creates an illusion to the programming interface that a shared-memory exists among processors) is a recommended way of solving this problem.

9 Special Structures for Join Executions

9.1 Access Path Structures (CAPS and GAPS)

Ideas to design index structures specially for facilitating join executions came as early as 1978 [7] in the form of a binary link. A binary link connects together matching tuples from

both joining relations and is often implemented as a chain of TIDs or physical pointers (storage addresses). A binary link reduces the execution cost of a join because a selection in the binary link can replace an expensive join execution between joining relations. In a sense, a binary link is a data structure embodying a pre-computed join.

A combined access path structure ($CAPS$) [7] is an implementation of the binary index and is an index (e.g., a $B+$ tree) that is implemented on an attribute, with leaf nodes populated not only with TIDs or physical pointers of the tuples of this relation, but also with TIDs or physical pointers of matching tuples of other relations. A generalized access path structure ($GAPS$) [7], in addition, stores values of the join attribute along with TIDs or physical pointers. These data structures can greatly speed up the join execution. However, these are very large data structures. In the worst case, number of entries (pointers, TIDs, values) stored can be the cross product of the numbers of tuples in two joining relations.

9.2 Join Index

The idea of access path structures reappeared in 1987 in the form of join index [26]. A join index is a two-attribute relation. Each tuple in a join index contains two TIDs of two matching tuples from two joining relations. A join index is also a pre-computed join and can turn a join execution into a selection. To facilitate search in a join index, it can have two copies, each clustered along one of the attributes of the join index. For example, the copy of the join index clustered on a join attribute of R facilitates join executions from R to S.

The main difference between access path structures and a join index is that in the access path structures, the connection between the matching tuples are, as claimed in [7], implemented by chaining of TIDs or physical pointers while the join index implements the connection by simply making a two-attribute relation between each pair of joinable relations in the database.

Both the access path structures and the join index serve the purpose of transforming a join execution into a selection and thus achieve the desired speed up, which can be far greater than that obtained by clustered index or inverted index which were originally designed for facilitating selections. However, a drawback is that both structures may require a very large storage space.

9.3 Domain Based Internal Schema

Each attribute in a database has a data domain. Therefore a database corresponds to a set of data domains. Domain based internal schema stores each value along with pointers pointing to the tuples in the attributes that have this value [13]. This data structure can be expected to have a better performance of the join execution than a join index. However, this structure may favor the join execution at the expense of other operations.

9.4 Composite B+ Trees

A composite B+ tree (B_c tree) [29] can be regarded as an improvement of the access path structure approach [7] and the domain based internal schema approach. A B+ tree is created for each data domain of some join attributes. Many attributes in the database share the same data domain and values in these attributes are indexed by the same B+ tree. A leaf-node of the B+ tree of a data domain has pointers that point to tuples in different relations of the database. These relations have join attributes that share this data domain. A pointer of a value in the data domain only points to a tuple which has this value as its join attribute value. Such a B+ tree is called a B_c tree. Our analysis shows that the B_c tree technique has an excellent performance, either for a full join (all the tuples in both joining relations participate the join execution), or for a selective join with very low join selectivity.

The B_c tree is a simpler approach than the access path structure and its performance could also be better due to the smaller size of the data structure. In the B_c tree approach, the number of B_c trees in a database is equal to the number of different data domains. In a 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 the worst case, each join index could have the number of tuples equal to the cross product of the numbers of tuples in the two joining relations. So, the B_c tree approach usually has a much smaller data structure than that of the join index approach. The cost to execute a full join using a B_c tree is:

$$\begin{aligned} C_{B_c\ full} = \quad & Node_{sz}(\| R \| + \| S \|)/p \\ & +\theta Y_{io}(\alpha\beta \| R \|, \| R |, \| R \|) \\ & +\theta Y_{io}(\alpha\beta \| R \|, \| S |, \| S \|) \end{aligned} \tag{26}$$

The first term is the cost to read all leaf nodes (they constitute the bulk of the volume of B_c) from the disk. The second and the third terms are the cost to selectively read matching tuples from R and S. $Node_{sz}$ is the size of each leaf node which containing

a join attribute value and one or more pointer values which are addresses of matching tuples in R and S. p is the average number of tuples in a disk page and is assumed the same for both R and S. The cost to execute a selective join using a B_c tree is:

$$\begin{aligned} C_{B_c\ selective} = \quad & \theta * Y_{io}(s * p, Node_{sz}(\| R \| + \| S \|)/p, \| R \| + \| S \|) \\ & + \theta Y_{io}(\alpha\beta \| R \|, \| R \|, \| R \|) \\ & + \theta Y_{io}(\alpha\beta \| R \|, s, s * p) \end{aligned} \tag{27}$$

The first term is the cost to select relevant leaf nodes of the B_c into the memory. The selection is assumed to be done on S and the selection result has total of s pages. $Node_{sz}(\| R \| + \| S \|)/p$ is the number of pages of all leaf nodes and there is roughly about $\| R \| + \| S \|$ leaf nodes. The last two terms are the cost to selectively read matching tuples from R and and a subset (the selection result) of S. Though the number of disk pages to be read can be even smaller than that in a full join, the disk reads are random.

9.5 Pointer Based Joins

Although the pointer based join techniques are mostly considered for object-oriented databases, they can also be used to enhance the performance of relational databases [37, 38]. In pointer based joins, each tuple of R, if it is joinable with a tuple of S, contains a pointer, which can be either a logical or a physical [35] address of a matching tuple in S. These pointers are stored along with tuples of R. Each tuple can have multiple pointers.

Each of nested loop join, sort-merge join and hybrid hash join has its own counterpart in the pointer based join techniques.

Pointer based nested loop join (PNB): The tuples of R are read into the memory one page at a time. At any given time, only one page of R resides in the memory. Let the page address part of the pointer value be denoted as PID. PIDs of the tuples of R are examined and pages of their matching tuples of S are read into the memory as needed. A large portion of the memory is allocated to the buffer of S to reduce the disk reads. The reading of R is sequential (if R and S reside in different disks and each page of R corresponds to only a few or none tuples in S, then the scanning of R is reasonably fast and is likely to be accomplished as a long sequence of sequential disk accesses) and the reading of S is largely random (θ). The disk I/O cost of PNB is:

$$C_{PNB} = \| R \| + \theta Y_{io}(\alpha\beta \| R \|, \| S \|, \| S \|) \tag{28}$$

Pointer based sort-merge join (PSM): PNB generates random disk I/Os when reading S. To reduce the randomness, the tuples of R can be sorted by their PIDs before the joining starts [37, 38]. After sorting, the tuples of R are placed physically as close as possible to each other if their pointed tuples in S are close to each other. The sorting incurs extra disk I/Os but the reading of S can be more sequential. Even after sorting, there may still be some random disk accesses (reflected by θ_{sort}) when reading the tuples of S. θ_{sort} should be much smaller than θ and that is the whole benefit to involve sorting. Only R is sorted. Because a tuple of R that is joinable to some tuples of S contains the PIDs of those tuples, lack of PIDs in a tuple of R indicates that this tuple is not joinable to S and can be safely discarded. So, only $\alpha \mid\mid R \mid\mid$ tuples of R are written to the disk in sorted runs. The disk I/O cost of PSM is:

$$C_{PSM} = \mid R \mid + (1 + \theta_{runs})\alpha \mid R \mid + \theta_{sort} Y_{io}(\alpha\beta \mid\mid R \mid\mid, \mid S \mid, \mid\mid S \mid\mid) \tag{29}$$

The first term is the cost of reading R. The second term is the cost of writing out sorted runs of joinable tuples (in all, $\alpha \mid R \mid$ pages of them) of R, which costs $\alpha \mid R \mid$, and then read them back into the memory for merging and joining, which costs $\theta_{runs}\alpha \mid R \mid$. θ_{runs} has been discussed in Section 5. The third term is the cost of fetching matching tuples of S. The performance is better than that of the PNB due to less randomness ($\theta_{sort} < \theta$) when reading tuples from S. The hope is that the reduction in randomness can offset the extra cost of sorting. If all joinable tuples of R can reside in the memory, then the second term is zero.

Pointer based hybrid hash join (PHH): The clustering of tuples in R according to their matching tuples in S can also be achieved by hashing [37, 38]. The values to be hashed are PIDs. R is first partitioned through hashing by using function ϕ_1 into R_0, R_1, ... , R_k. All R_i's except R_0 are written to disk. R_0 is then built into a hash table in the memory through hashing on PIDs by possibly a different hash function ϕ_2. The hash table is built in such a way that all tuples referencing the same page in S are hashed into the same hash entry. After hashing, matching tuples of S are read into the memory through pointers to join with the tuples of R_0. This process is repeated for all R_i's. The disk I/O cost is:

$$\begin{aligned} C_{PHH} = \; & \mid R \mid + max\{0, (\theta_{buckets} + 1)(\alpha \mid R \mid - \mid R_0 \mid)\} \\ & + \theta_{hash} Y_{io}(\alpha\beta \mid\mid R \mid\mid, \mid S \mid, \mid\mid S \mid\mid) \end{aligned} \tag{30}$$

The first term is the cost to read R from the disk. The second term is the cost to write out hashed buckets to disk and then read them in to do the join. $\theta_{buckets}$ reflects the randomness in writing out the hashed result to buckets (see discussions about $\theta_{buckets}$ in Section 6). Note that only joinable tuples of R need to be written out. The third term is the time to fetch matching tuples of S into the memory. θ_{hash} reflects the residual randomness in fetching tuples of S after hashing and should be about the same scale as θ_{sort}.

Pointer based PID-partitioning join (PPP): Instead of tuples, in PPP [38], list of PIDs of S , L, is read into the memory and sorted. L is divided into $k+1$ partitions L_i $(0 \leq i \leq k)$. Let S_i be range of pages in S corresponding to L_i. k should be chosen such that each S_i, $0 \leq i \leq k$ may fit entirely in the memory. Note that data skew problem such as that might happen in PSN or PHH will not happen in PPP. L remains memory resident after being partitioned. Then, S_0 is read into the memory. All tuples of R are read one page at a time into the memory. If the PID of a tuple of R falls into S_0, then this tuple is used to probe S_0 for a possible match. If the PID of the tuple falls into S_i, $i > 0$, this tuple is placed in the buffer i (k buffers are reserved in the memory for this purpose). The content of a buffer is flashed out to the disk when it is full. All tuples that have PIDs in S_0 constitute R_0 and all tuples fall in buffer i, $i > 0$, constitute R_i. Note that $\mid \cup \forall i R_i \mid = \alpha \mid R \mid$. After R has been scanned, R is partitioned and R_0 is joined with S_0. Then each S_i, $i > 0$, is read into the memory and the tuples of R_i are read one page at a time into the memory to execute the join. This process repeats until all R_i's and S_i's are processed. The disk I/O cost is:

$$C_{PPP} = \mid L \mid + \mid S_0 \mid + \mid R \mid + \theta_{bucket} \alpha \mid R \mid + (\mid S \mid - \mid S_0 \mid) + \alpha \mid R \mid \tag{31}$$

The first three terms are the cost of reading L, S_0, and R. The fourth term is the cost to write out joinable tuples of R. θ_{bucket} is the randomness in disk write and was discussed in Section 6. The fifth term is the cost to read all the pages of S, one S_i at a time, $i > 0$, into the memory. The last term is the cost to fetch all matching tuples of R, one R_i at a time, into the memory. We can see that the disk access in PPP are mostly sequential. If join selectivity is low, then the fourth term and the last term will be zero. It was claimed [38] that PPP can often outperform the standard and the pointer based hybrid hash joins. However, PPP can only be used for executing full joins [38].

In general, the pointer based joins perform better than the traditional joins if R is equal or smaller than S, and they perform worse than the traditional joins if R is larger

than S [38]. This is simply because the pointers are one-directional and only point from R to S. Also, as any indexing based techniques, the small join result is a favorable condition for applying pointer based join techniques.

10 A Performance Comparison

The scope and setting of the comparisons: In this section, we compare the performance of various join techniques under a uniform parameter setting. Two types of nested block joins are compared: the nested block join and its variation that uses index. Three types of sort based joins are compared: the sort-merge join, the distributive join, and the hybrid join. In hash based joins, only hybrid hash join is compared. We do not compare dynamic destaging strategy, adaptive hash join, and partially preemptible hash joins because these techniques mainly enhance the stability of join execution performance in the environment of changing memory size during the join execution and skewed data distributions over the hybrid hash join.

For shared-nothing parallel machines, we only include the parallel hash join in the comparison since it is the best join technique in that category. For shared-everything parallel machines, only join techniques for machines with separate I/O channels are included since the behavior of join execution on machines with shared I/O channels is not much different from that on a fast single CPU machine. The B_c tree approach is usually better than the join index approach and is included in the comparison.

Parameters used in the performance comparison: Parameter values are varied over a wide spectrum to test the relative sensitivity of performances to relation size, memory size, and the random to to sequential disk access time ratio. In the hybrid hash join, the size of R_0 depends on data distribution. If the data is not too skewed, $\mid R_0 \mid$ may be around 50% to 90% of M. We assume $\mid R_0 \mid = 80\%M$ and $\mid S_0 \mid = \mid S \mid * \mid R_0 \mid / \mid R \mid$.

$\theta_{indexed}$ in the hybrid join and $\theta_{indexed}$ in the nested block join with indexing are all set at $\theta * (\mid S \mid - 0.5M) / \mid S \mid$ because they represent reduced randomness in disk access due to the fact that a part of S, which is equal to a large portion of the available memory. Let's assume that is 50% of M (see discussions about θ_{bucket} in Section 4) is used for the buffer of S. The size of PID list in pointer based PID-partition join is set at $(1/20) \mid S \mid$ (this is the value obtained by assuming that S has 20 attributes, which is a reasonable number in practice, and the data volume in each attribute takes about the same space). For parallel joins, the number of processors is set at 10.

Now let us estimate θ, θ_{runs}, $\theta_{buckets}$, θ_{part}, θ_{sort}, and θ_{hash}. Each sequence of sequential disk accesses has a limited length. Let p be the time span for a "true" sequential disk access. If the typical length of a sequence of sequential disk I/O accesses is m pages, then $p + \theta/m$ is the average time for a sequential disk access, taking into account the time of the initial random disk access. The flushing out (or the reading) of the content in a memory buffer of m pages can be regarded as one sequence of sequential disk accesses. If the total buffer size is fixed and the buffer is divided among b partitions (which are runs in sort based joins and buckets in hash based joins), then the average length of a sequence of sequential disk accesses is m/b. So the average time for such a somewhat random disk access is $p + (b\theta)/m$. Clearly, the fewer partitions the output buffer has, the less average time an average disk access takes since θ is much larger than p. We use θ between 10 and 100.

Let the disk page size be 2,048 bytes and 1 MB (= 500 pages) of the memory is allocated as the space to hold various output buffers. In the sort based techniques, tuples are first sorted into runs of size 2M, and then runs are merged into a completely sorted list. So the number of runs is approximately $\mid R \mid /(2M)$ for R and $\mid S \mid /(2M)$ for S. Each run needs a buffer in the memory for the merge operation. The number of buffers is (if runs of R and S are merged in one pass): $b = (\mid R \mid / + \mid S \mid)/(2M)$ and $\theta_{runs} = 1 + \theta * b/500 = 1 + \theta * ((\mid R \mid + \mid S \mid)/2M)/500$. $\theta_{buckets}$ and $\theta_{PID-part}$ reflect the randomness in writing the disk. The largest buffer can be selected to be written out first and its space can be freed and then used by other buffers. So the actual randomness in disk writing is less than that indicated in above computation.

θ_{sort} and θ_{hash} in pointer based joins are more difficult to estimate since their values depend on many factors (page size, the ratio of tuples that cross page boundaries, etc). However, we know that these values should in general be fairly close to 1 due to sorting or hashing. Each page in R and S is assumed to contain five tuples on the average.

Observed Performance: Figures 6 through 11 are plotted as the ratio of execution time between nested block join and the indicated join technique. Although the cost of writing out join results are uniformly dropped in all cost formulas for a simpler representation, it is uniformly added back in plotting these figures for a more realistic comparison. The horizontal axis is the size of S, which varies from $\mid R \mid$ to 10 $\mid R \mid$ in each figure. Three sets of figures are plotted. From the first set of figures (Figure 6 and 7) to the second set of figures (Figure 8 and 9), the size of R and S are increased 20 times. The size of R and S in the third set of figures (Figure 10 and 11) is the same as that in the

second set of figures. The difference between the second and the third set of figures is that the available memory size is also increased 20 times. So, in the third set of figures, the $\mid R \mid /M$ is the same as in the first set of figures. In the figures, the performance curves of the nested block join with index and the pointer based nested loop join are almost overlapped.

We see that Figure 6 is very similar to Figure 10 and Figure 7 is very similar to Figure 11. This shows that it is the ratio $\mid R \mid /M$, not the absolute size of R, that determines the relative performance between different join techniques. This conclusion is further evidenced in the second set of figures. In the second set of figures, the ratio $\mid R \mid /M$ is 20 times larger than in the other two sets of figures. All techniques in the second set of figures perform much better as compared to the nested block join.

The first and the second figure in each set of figures has $\theta = 10$ and $\theta = 100$ respectively. In every set of figures and for each join technique, its performance is better in the first figure as compared to that in the second figure. Looking more closely, the performance degradation from the case when $\theta = 10$ to the case when $\theta = 100$ for each indexed technique, e.g., the B_c tree, is larger than the performance difference for a non-indexed technique, e.g., the hybrid hash join. This can be explained by the more randomly disk accesses employed in the indexed techniques.

The relative performance of nearly every indexed join technique is better at the right end of each figure. Since the join selectivity (in terms of number of pages) is fixed in each figure and $\mid S \mid$ is larger at the right end of each figure, this shows that a small selectivity (in term of %) favors the use of indexed techniques.

Overall, parallel joins have good performances. Only 10 processors are used here for the parallel joins. If more processors were used, the performance advantage of parallel joins would be more prominent. The sort-merge join, the hybrid hash join and the distributive join perform comparable or even worse than the nested block join when the ratio $\mid R \mid /M$ is small (Figure 6, 7, 10, and 11).

11 Summary Remarks

Loop based joins (late 70's), sort based joins (late 70's), and hash based joins (early 80's) are three classes of non-indexed join techniques that have been widely used in commercial relational databases. None of these techniques is the best and each has special situations where it can outperform the others. In general, loop based joins offer most robust performance in all situations and sort based joins usually perform better than loop based join

when joining relations are large. Hash based joins usually outperform sort based joins when tuples in R can be evenly partitioned, and/or the ratio of $\mid S \mid / \mid R \mid$ is high. However, compared to sort based joins, hash based joins are much less robust, i.e., much more sensitive to data skews. Rocking [9, 50] in the nested block join is proved to have no effect on the performance of join execution.

From our performance study, we observed that index based joins offer the best performance when the join result is small. The conditions that favor the use of indexes are: small join result (compared to the total size of relations), larger data unit (longer tuples and larger size of average data entries), and not too large speed ratio between random disk access and sequential disk access. Early index based joins used selection indexes, which were originally designed to speed up selections but nevertheless, can be used to speed up join executions. Though the join index (late 80's) offers good potential to further speed up the join execution, it has not been deployed to commercial relational databases except in spatial databases where the huge size of spatial data and the complexity of join predicates justify the use of such a large data structure. B_c tree [29] approach improved upon the join index in that it uses a modified B+ tree to speed up disk accessing, and it drastically reduces (compared to the join index) the size of data structure used. Another group of index based joins is the pointer based joins. A drawback of this class of join techniques is that like the join index, it also may have huge data structures.

We considered the time ratio of random to sequential disk accesses in each join technique. Reducing randomness in disk access is an important factor in enhancing the performance of the join execution. The idea proposed in [52], i.e., to use large buffers to increase the length of the sequence of sequential disk accesses, is applied for each join technique, whenever possible, to reduce the randomness in the disk I/O. The approach can improve the performance of many join techniques and make our performance comparison more realistic (it is a simple approach that can be used, especially in the large memory of today's computers, and therefore will be used, in the real implementation of the database software). However, many join techniques have different degrees of randomness in their disk I/Os, even if the size of input or output buffers reserved in the memory is the same for all the join techniques. We deal with the different degrees of randomness in the disk I/O by using different random access factors for different join techniques in our efforts to achieve a more realistic performance comparison between different join techniques. Based on this, we developed a consistent set of cost formulas to compare the performance of a large number of join techniques fairly and realistically.

Conceptually, a parallel join offers an easy way to enhance the performance by the

sheer power of more hardware. However, the use of many parallelly executing processors can turn the join execution into a heavily I/O bound execution. Shared-nothing (message passing) parallel machines are gaining popularity due to the ease of the scalability of hardware and software. In general, the parallel hash join is the best join technique on a shared-nothing parallel machine. However, the performance of the parallel hash join on a shared-nothing parallel machine is very sensitive to data skew as compared to loop based or sort based parallel joins. One way to alleviate this problem is to use a shared virtual memory and encourage workload sharing among processors. Parallel hash is also the best on a share-everything (shared-memory) machine and is fully scalable with regard to the number of processors. One advantage of parallel joins on the shared-everything machine is the relatively small sensitivity to data skew as compared with joins on shared-nothing parallel machines. Note that the conclusion about joins on shared-everything parallel machine is drawn on machines with separate I/O channels for each processor. However, due to manufacturing complexity, most shared-everything parallel machines have only a few I/O channels shared by all processors. On this type of parallel machines, parallel joins only have a very limited performance advantage over joins on single processor machines because join executions are most likely to be I/O bound.

[32] concluded that the performance of the parallel hash join is very sensitive to data skew in a shared-nothing parallel machine while [48] claimed that the effect of data skew on parallel hash join is small in a shared-everything parallel machine. However, in [32], a large number (300) of processors were used in the parallel machine while in [48], only 10 processors were used. We have reason to believe that they have different conclusions about the sensitivity of data skew not only because of the difference between shared-nothing and the shared-everything parallel machines, but also because of the large difference in the number of processors in their parallel machines.

Future databases are likely to be very large and query result from very large databases are likely to be small (relatively, compared to the size of the database), and this phenomenon is especially true for many "online" databases (as in mobile and internet environment) that directly answer customer queries. From our analyses, we conclude that when conditions are favorable to use indexes, e.g., the average join result is small, a parallel hash join with assistance from a good cross-relational index, such as an improved join index, or a B_c tree index, will perform the best in large relational databases.

References

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

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

[3] M. Blasgen, K. Eswaran. On the Evaluation of Queries in a Data Base System. *IBM Res. Lab., San Jose, CA, Res. Report RJ1745, 1976.*

[4] M. Blasgen, and K. Eswaran. Storage and Access in Relational Databases. *IBM Systems Journal. Vol 16, No. 4,* 1977.

[5] S. B. Yao. Approximating Block Accesses in Database Organizations. *Commun. ACM 20, 4(Apr. 1977), 260-261.*

[6] F. P. Preparata. New Parallel-Sorting Schemes. IEEE. Trans. Comput. C-27 (July 1978).

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

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

[9] 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.

[10] A. P. Bernstein, 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.

[11] 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.

[12] G. H. Gonnet, P. A. Larson. External Hashing with Limited Internal Storage. Proc. of the ACM Symposium on Principles of Database Systems, ACM, New York, 1982, pp. 256-261.

[13] M. Missikoff. 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.

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

[17] D. J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, 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] 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.

[23] G. M. Sacco. Fragmentation: A Technique for Efficient Query Processing. ACM Transactions on Database Systems, Vol. 11, No. 2, June 1986, Pages 113-133.

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

[25] 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.

[26] P. Valduriez. Join Indexes. ACM TODS, V12, N2, June 1987,

[27] 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.

[28] 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.

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

[30] D. L. Lee, C.-W. Leng. Partitioned Signature File: Design Issues and Performance Evaluation. *ACM Transactions on Information Systems, Vol. 7, No.2, April 1989, pp. 158 - 180.*

[31] K. Li, P. Hudak. Memory Coherence in Shared Virtual Memory Systems. ACM Transaction on Computer Systems, 7(4), November 1989.

[32] 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.

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

[34] J. A. Blakeley, N. L. Martin. Join Index, Materialized View, and Hybrid-Hash Join: A Performance Analysis. IEEE. Proc. of 6th Int'l Conf. on Data Engineering, 1990.

[35] L. Hass et al., Starburst Mid-Flight: As the Dust Clears. IEEE Transactions on Knowledge and Data Engineering, March 1990.

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

[37] 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.

[38] E. J. Shekita. High-Performance Implementation Techniques for Next-Generation Database Systems. Ph.D thesis. Dept. of Computer Science, the University of Wisconsin-Madison, 1990.

[39] 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.

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

[41] 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.

[42] J. L. Wolf, B. R. lyer, K. R. Pattipati, J. Turek. Optimal Buffer Partitioning for the Nested Block Join Algorithm. IEEE. Proc. of 7th Int'l Conf. on Data Engineering. April 8-12, 1991, Kobe, Japan.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

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

[50] E. P. Harris. Towards Optimal Storage Design for Efficient Query Processing in Relational Database Systems. Ph.D thesis, 1995, Department of Computer Science, The University of Melbourne, Parkville, Victoria 3052, Australia.

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

[52] M.-L. Lo and C. V. Ravishankar. Towards Eliminating Random I/O in Hash Joins. IEEE. Proc. of 12th Int'l Conf. on Data Engineering, 1996.

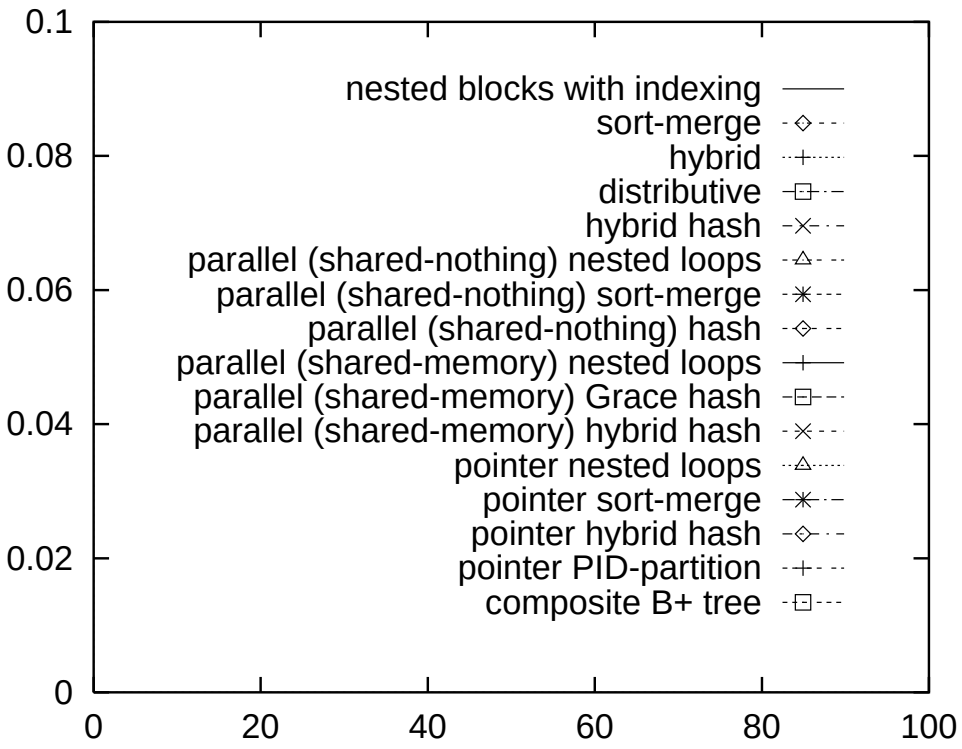

Figure 5: Legends used in performance figures

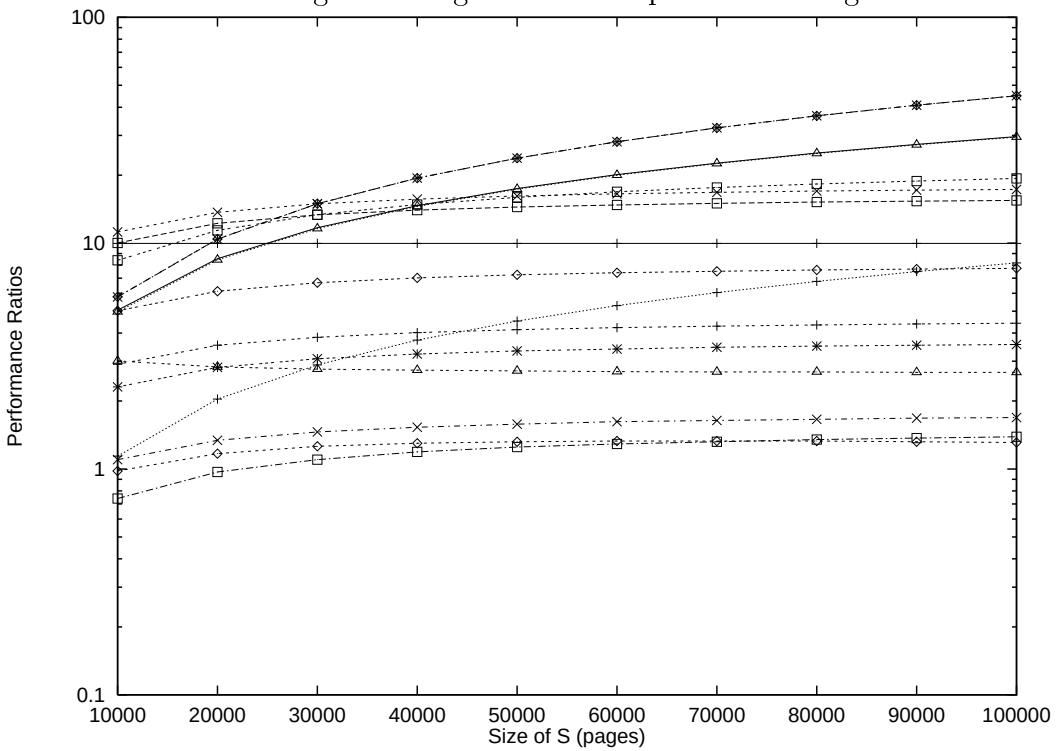

Figure 6: R=10,000, S=10,000 $\rightarrow$ 100,000, M=2,000, $\theta = 10$

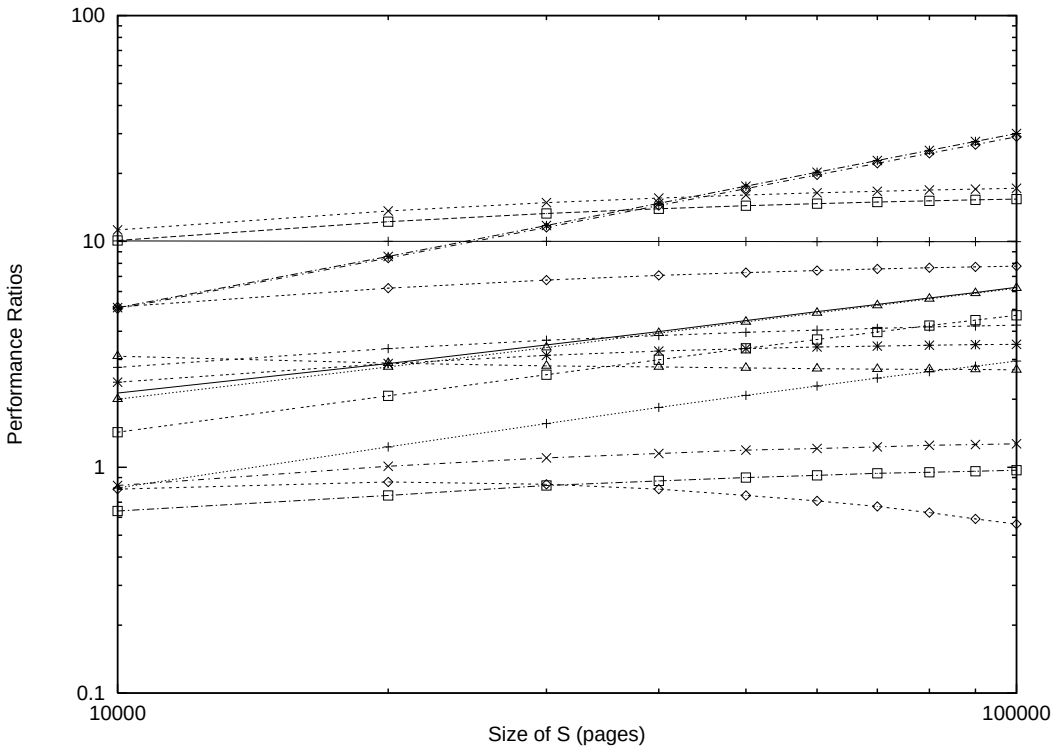

Figure 7: R=10,000, S=10,000 $\rightarrow$ 100,000, M=2,000, $\theta = 100$

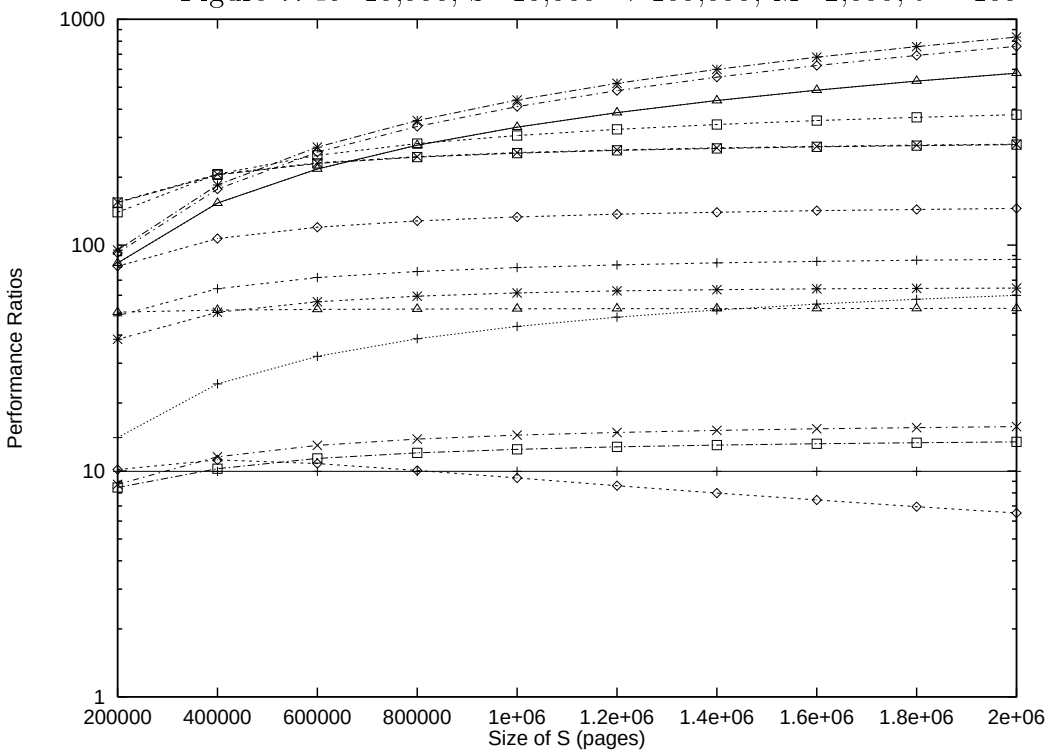

Figure 8: R=200,000, S=200,000 $\rightarrow$ 2,000,000, M=2,000, $\theta = 10$

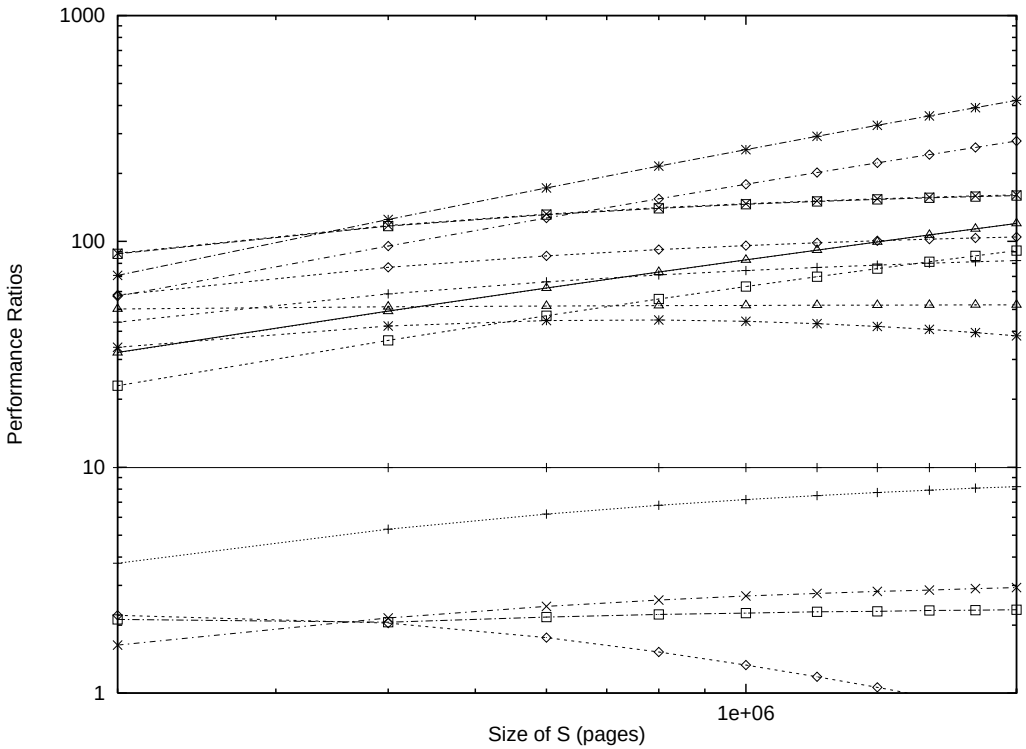

Figure 9: R=200,000, S=200,000 $\rightarrow$ 2,000,000, M=2,000, $\theta = 100$

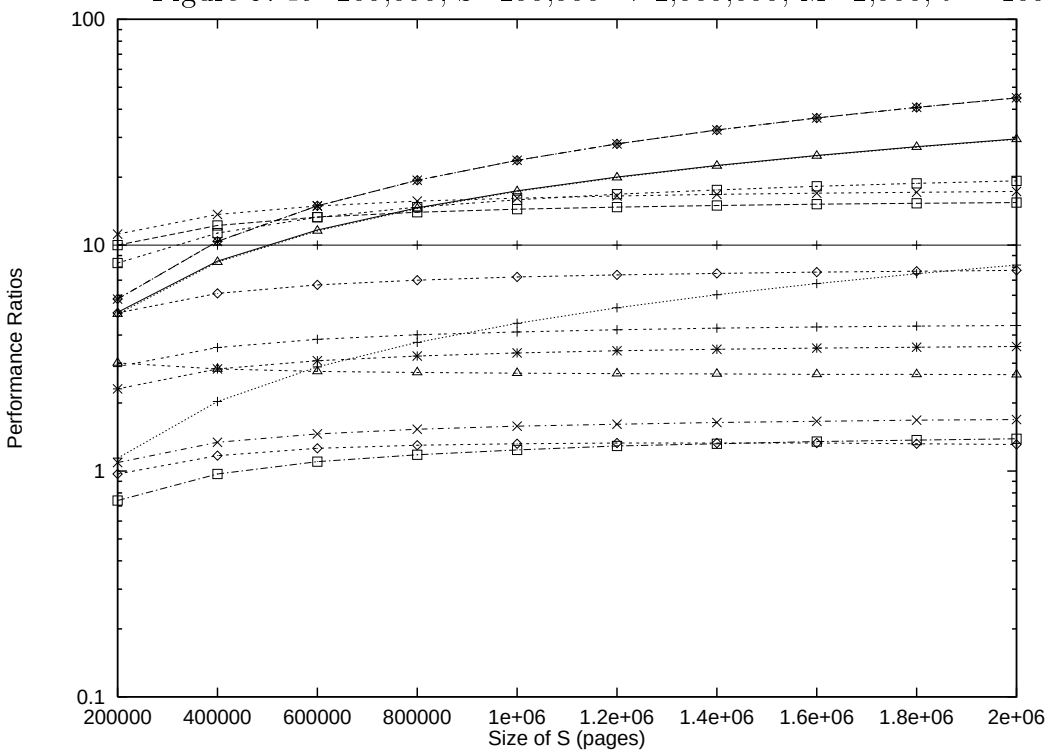

Figure 10: R=200,000, S=200,000 $\rightarrow$ 2,000,000, M=40,000, $\theta = 10$

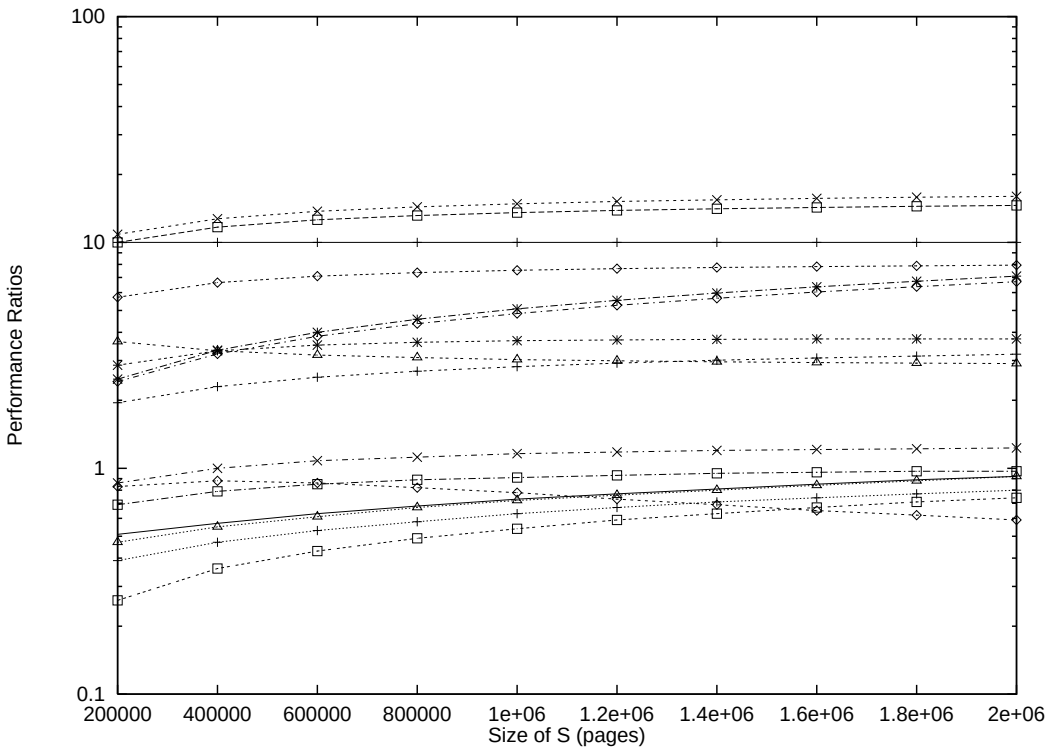

Figure 11: R=200,000, S=200,000 $\rightarrow$ 2,000,000, M=40,000, $\theta = 100$