Performance Study of the Index-based Parallel Join

Byeong-Soo Jeong
Edward Omiecinski*

Abstract

The index file has been used to access database records effectively. The join operation in a relational database system requires a large execution time, especially in the case of handling large size tables. If the indexes are available on the joining attributes for both relations involved in the join and the join selectivity is relatively small, we can improve the execution time of the join operation. In this paper, we investigate the performance trade-offs of parallel index-based join algorithms where different indexing schemes are used. We also present a comparison of our index-based parallel join algorithms with the hash-based parallel join algorithm.

1 Introduction

The join operation has received much attention in the research community due to the fact that it gives semantic importance and also requires a large execution time in a relational database system. Generally, three major paradigms (i.e., *nested-loops*, *sort-merge* and *hash-based*) have been widely used to implement such a join operation.

Though many algorithms have been proposed to process the join operation as efficiently as possible, the join operation remains an expensive operation in relational database systems. One way to achieve significant improvements in response time of a join operation is to parallelize the join processing, that is, to use multiple processors to join two relations concurrently. Similar to other parallel processing algorithms, the parallel processing of a join involves three steps:

1. decompose the join into n tasks,
2. assign those tasks to p processors, and
3. assemble the results as the join result.

* Georgia Institute of Technology

In order to achieve the best performance for parallel join algorithms, a balanced task decomposition (step 1) is very important. Several techniques (such as *Bucket Size Tuning* in parallel hash-based join, etc.) have been suggested to solve the load balancing problems in parallel join algorithms.

Another way to improve the execution time (reduce I/O cost) of a join operation is to use an index. If the indexes (primary or secondary) are available on the joining attribute(s) for both relations involved in the join, we can extract the information about joining tuples by scanning the index files only, without reading the entire data files. Since the index files are much smaller than the data files in most cases, it will reduce I/O time significantly.

In uniprocessor environments, several performance studies of join algorithms using indexes have been done in the paradigm of the nested-loop approach, but not much work has been done in a parallel database environment. In this paper, we study the aspect of parallel join processing algorithms when the indexes are available on the joining attribute(s) for both relations involved in the join. As a target indexing scheme, we consider our parallel indexing schemes which are illustrated in [2]. We investigate the performance trade-offs of parallel index-based join algorithms where different indexing schemes are used. We also present a comparison of our index-based parallel join algorithms with the parallel hash-based join algorithm.

In the next section we discuss the previous work that has been done with respect to the parallel join algorithms. In section 3, we describe the parallel index-based join algorithms which use the parallel index files illustrated in [2] In section 4, we present an analytical cost model intended to predict the performance of each parallel join algorithm. In section 5, we discuss the optimal buffer allocation strategies in the case of bucket partitioning and N-way merging for the parallel join operations. We also describe the experimental results from our analytical cost model while varying several parameter values in section 6. Finally, we discuss our conclusions in section 7.

2 Previous work

Since the join is the most frequently used and the most time-consuming operation in a relational database system, much work [3, 4, 5, 8, 9, 10] has been directed to finding efficient parallel join algorithms under several multiprocessor architectures. The previous work can be roughly divided into two groups; one group deals with the basic parallel join methods and their performance, and the other group emphasizes the handling of data skew in the join operation in order to further improve the performance.

In [8], DeWitt et al. investigated the performance of four parallel hash join algorithms while implementing them on GAMMA, a shared-nothing parallel database machine. Omiecinski proposed

a load balancing join algorithm for a shared-memory multiprocessor and analyzed its performance in [5]. In [9, 10], Wolf et al. considered the performance of the parallel hash join and sort-merge join algorithms in the presence of data skew.

To the best of our knowledge, only [1, 7] deal with index-based join processing in a parallel database environment. In [7], Omiecinski suggested heuristic algorithms for reducing I/O costs when the buffer space is limited. His work considered the shared-memory multiprocessor architecture for a target system. In [1], DeWitt et al. presented four variants of index-based parallel join algorithms in a shared-nothing environment. They also compared the performance of four variants by analytical modeling and implementation in the Gamma parallel database system.

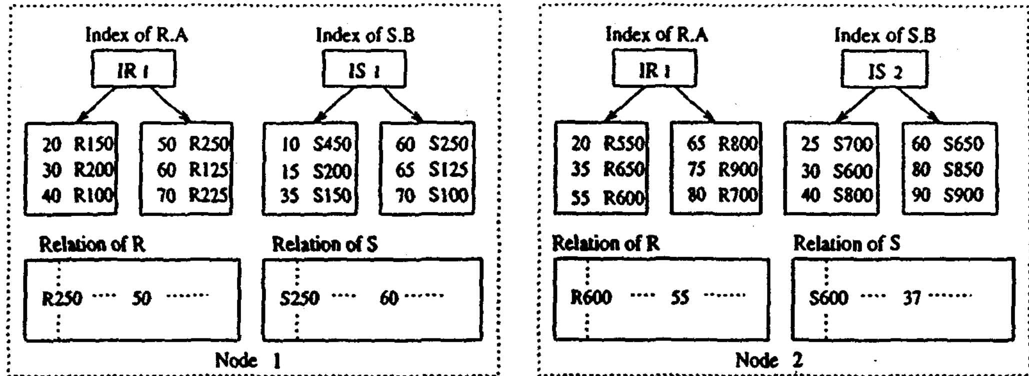
Our work can be differentiated from the above work by the following aspects.

- We consider a shared-nothing environment (different from [7]).
- We assume that indexes are available on each relation's attributes (different from [1]).
- We consider our parallel indexing schemes (different from [1, 7]).

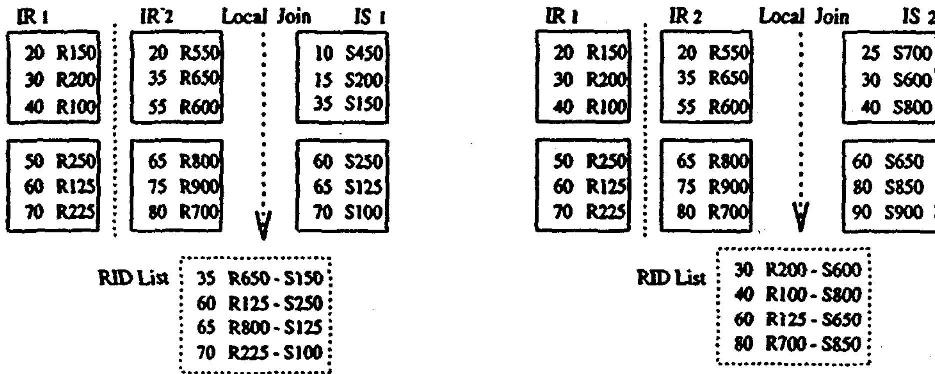
3 Index-based Parallel Join Algorithms

In this section, we describe index-based parallel join algorithms which use the parallel index files illustrated in [2]. As in [2], we consider a shared nothing architecture where processing nodes are connected by a scalable interconnection network. We also consider the equi-join operation with the convention that the two relations being joined are R and S, and that the join condition is R.A = S.B. We assume that the relations R and S are declustered over all the nodes on attributes other than R.A and S.B. We also assume that the indexes are available on the joining attributes for both relations (R.A and S.B) involved in the join. Typically, the indexes are implemented as B+ trees. The leaves of B+ tree contain entries that consist of a key value and the addresses (RIDs) of all tuples in a relation which contain that key value. By using the joining attributes' indexes of the two relations we can determine the RID list that contains pairs of RIDs which show the tuples to be joined. In uniprocessor systems, an index-based join algorithm looks very simple. By comparing the leaves of B+ tree indexes built on the joining attributes of two relations, the RID list is generated. Then retrieval of joining tuples is performed from the RID list. At this time, the page connectivity graph model [7] can be applied for reducing I/O cost in the case of limited buffer space.

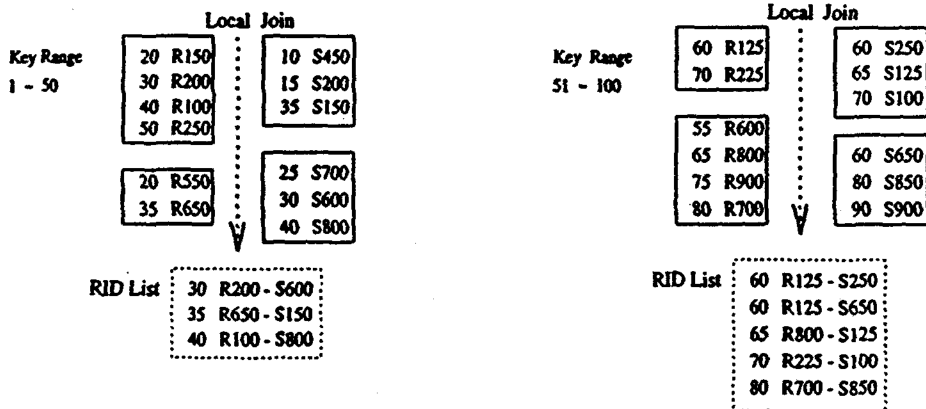
In a shared nothing parallel system, we should consider declustered indexes and tables (relations) over the nodes in order to get the correct RID list and retrieve the joining tuples which would be



(a) Declustered Relations and Indexes



(b) Local Join using Index Fragment Replication



(c) Local Join using Index Fragment Redistribution

Figure 1: Index-based Parallel Join using Local Indexes

scattered over the nodes. Our index-based parallel join algorithm consists of several processing steps. First, in order to make sure that all pairs of index entries are examined, the index entries are redistributed or replicated at each node. After this, the local join is performed in parallel with the redistributed or replicated index fragments. For the local join, we can use traditional join algorithms (such as sort-merge and hash join). The RID lists are generated from the local join at each node. Figure 1 describes this step.

As previously mentioned, tuples pointed by the RID list are scattered over the nodes. Therefore, we need to redistribute the entries of the RID list based on the value of the RID (we assume that the RID has node information, e.g., node 1 holds R001 ~ R500 and node 2 holds R501 ~ R1000) so that we can read the joining tuples locally. In order to optimize the page access of reading the joining tuples, the RID list is sorted by the RIDs of the relation which is intended to be read. Figure 2 illustrates this step.

Depending on the parallel indexing schemes, detailed procedures of each processing step would be different. We will consider our two parallel indexing schemes in following sections.

3.1 Using Local Indexes

In the local indexing scheme, we can think of two possibilities for getting a correct RID list, *index fragment replication* and *index fragment redistribution*. The most obvious solution is to broadcast all index fragments (leaves of B+ tree index) of the smaller relation (R) to every node and join them with the local index fragment of the S relation (see Figure 1(b)). The other alternative is that index fragments of R.A and S.B are redistributed, such that for any index entry of R.A that joins with some index entry of S.B, both index entries are at the same node (see Figure 1(c)).

Index fragment replication has the disadvantage that all index fragments of R.A are shipped to nodes where there may be no matching index entries of S.B. This cost may be non-trivial if the size of R.A's index fragments is relatively large. On the contrary, index fragment redistribution can avoid this wasted effort by eliminating unnecessary pairs of index fragments through the redistribution. In the case of index fragment redistribution, it is important that equal sized portions of the indexes be assigned to each node for load balancing. As a way of redistribution, a hash function and key value range can be applied.

From another view, index fragment replication does have some good points; since a local index stores all the key values and RIDs for the relation fragment which is stored locally, joining tuples of S can be retrieved locally. However, in the case of index fragment redistribution, the retrieval of joining tuples requires additional communication cost since the tuples can be scattered over the

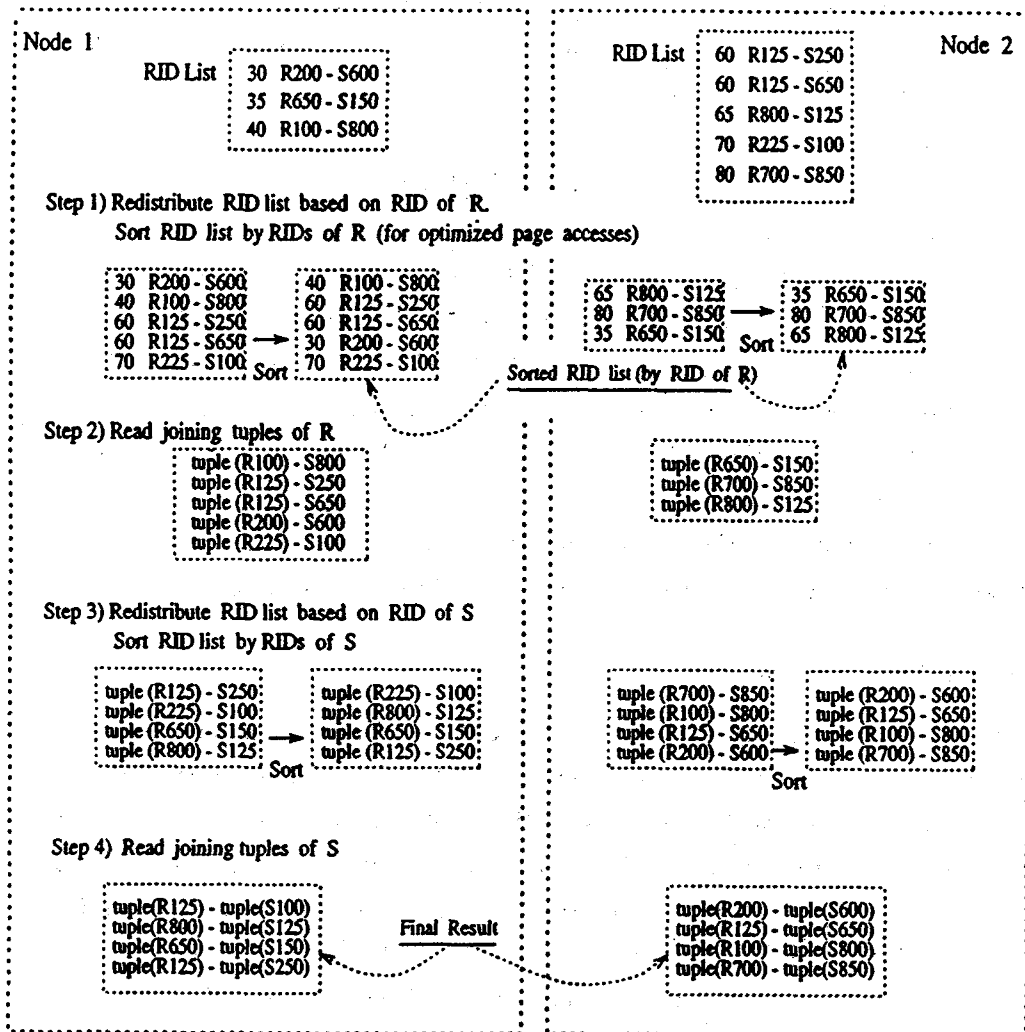


Figure 2: Read the Joining Tuples of R and S

nodes. We will see such tradeoffs by using the analytical cost model in section 6.

3.2 Using Partitioned Global Indexes

In the partitioned global indexing scheme, the global index tree is partitioned by several criteria (such as the range value and hash function value) and distributed across the nodes. We can take advantage of this information for the index-based parallel join operation. The best case is that, if the indexes of R.A and S.B are partitioned and distributed by the same criteria, there is no need to replicate and redistribute the index fragments. Even though the partitioning criteria are different in both indexes, only one of the index fragments (R.A and S.B) needs to be redistributed according to the partitioning criterion of the other index.

In the partitioned global indexing scheme, the tuples pointed by one index fragment can be scattered across the nodes. For this reason, the index fragment replication method does not give any advantage in terms of execution cost. Thus, we will not consider index fragment replication in the case of the partitioned global indexing scheme.

4 An Analytical Cost Model

In this section we describe an analytical cost model intended to compare the performance of each algorithm. We include the cost model of the parallel hash join algorithm (Grace-hash) in order to see the break point where the index-based algorithm performs better. We separate the cost of the algorithm into I/O, CPU, and Communication while assuming that CPU and I/O processing are not overlapped. For simplicity, we assume that the relations (R and S) and the joining tuples are evenly distributed across the nodes. We also assume that R is the smaller relation. Throughout the analysis the parameters shown in Table 1 are used.

4.1 Index Fragment Replication with Local Indexes

(1) Index Replication

For the index replication, each node reads an index fragment of R.A (IR_i) into the *input buffer* (B_I), broadcasts it to all the nodes through the *communication buffers* (B_C), and writes the broadcasted index fragments of R.A to each local disk through the *output buffer* (B_O). Let B_I, B_O, and B_C be the sizes (in number of pages) of the input, output, and communication buffers respectively. At this step, the following I/O cost and communication cost will be needed at each node.

Table 1: Parameters in the Cost Model

Parameters	Description
n	Number of nodes in a system
R	Size of relation R in number of pages ($R_i = R/n$)
S	Size of relation S in number of pages ($S_i = S/n$)
r	Number of tuples in R ($r_i = r/n$)
s	Number of tuples in S ($s_i = s/n$)
IR	Size of leaf pages of B+ tree index on R.A ($IR_i = IR/n$)
IS	Size of leaf pages of B+ tree index on S.B ($IS_i = IS/n$)
M	Number of memory pages at each node
RID	Size of RID list in pages ($RID_i = RID/n$)
rid	Number of RID pairs in RID list ($rid_i = rid/n$)
SR, SS	Semijoin selectivity factors
JS	Join selectivity factor $JS = 10 \cdot max(\frac{SS}{r}, \frac{SR}{s})$ [6] ($rid = JS \cdot r \cdot s$)
$Y(k,m,n)$	Optimal number of page I/O's for accessing k records randomly distributed in a file of n records stored in m pages [6].
T_{access}	Average disk access time (seek time + rotational latency)
$T_{transfer}$	One page data transfer time
$comm$	Time in sending/receiving a packet
$comp$	Time in comparing two tuples and moving a tuple
$hash$	Time in computing the hash value of a tuple and moving a tuple
$probe$	Time in probing hash table

I/O cost is

$$I/O_{cost} = \lceil \frac{IR_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil \frac{IR}{B_O} \rceil \cdot (T_{access} + B_O \cdot T_{transfer})$$

under the condition that

$$IR_i = IR/n \;\; and \;\; B_I + B_O + (n-1) \cdot B_C \leq M$$

In order to minimize the above sequential I/O cost, the values of B_I, B_O, and B_C should be carefully chosen. The problem of optimal buffer allocation for sequential I/O is discussed in section 5.
Communication cost is

$$COMM_{cost} = comm \cdot (n-1) \cdot IR_i$$

(2) Local Join by using Index Fragments (Generate the RID list)
For the local join, we can use the sort-merge technique since the leaf pages of the B+ tree index are already sorted by the key value. Thus, we need to read IS_i and IR once and write the RID list onto the disk.
I/O cost is

$$\begin{aligned} I/O_{cost} &= \lceil \frac{IS_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil \frac{IR}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) \\ &+ \lceil \frac{RID_i}{B_O} \rceil \cdot (T_{access} + B_O \cdot T_{transfer}) \end{aligned}$$

with the condition that

$$RID_i = RID/n \;\; and \;\; (n+1) \cdot B_I + B_O \leq M$$

CPU cost is

$$CPU_{cost} = comp \cdot r$$

(3) Read the Joining Tuples of S from the RID list
(a) Sorting the RID list by S.B: At this step, we need to first sort the RID list by S.B values for the optimized access of S data pages. We assume that the RID list is not too big (i.e., join selectivity is small) to fit in memory. By this assumption, the sorting cost can be calculated as follows.

I/O cost is

$$I/O_{cost} = \lceil \frac{RID_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer})$$

CPU cost is

$$CPU_{cost} = 2 \cdot comp \cdot s_i \cdot SS \cdot \lceil \log_2 s_i \cdot SS \rceil$$

(b) Accessing the data pages of S at each node

$$I/O_{cost} = Y(s_i \cdot SS, S_i, s_i) \cdot (T_{access} + T_{transfer})$$

(4) Read the Joining Tuples of R from the RID list

Since the joining tuples of R are scattered over the nodes, we need to send the RID list to the corresponding node. Then, the retrieval of the R tuples is performed as the above. The cost of this step is

$$COMM_{cost} = comm \cdot (n-1) \cdot RID_i$$

(a) Sorting the RID list by R.A:

$$I/O_{cost} = \lceil \frac{RID_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer})$$

$$CPU_{cost} = 2 \cdot comp \cdot r_i \cdot SR \cdot \lceil \log_2 r_i \cdot SR \rceil$$

(b) Accessing the data pages of R at each node:

$$I/O_{cost} = Y(r_i \cdot SR, R_i, r_i) \cdot (T_{access} + T_{transfer})$$

4.2 Index Fragment Redistribution with Local Indexes

(1) Index Redistribution

As previously mentioned, we use the hash function value or key range value for the index redistribution. In the case of range partitioning, the prior information about the key value distribution is needed to evenly distribute the index fragments among the nodes. We assume that the hash function is used for the index redistribution. At this step, we need to read the index fragments of R.A and S.B, apply the hash function to the key value, and then, distribute the index entries to the corresponding node.

Therefore, I/O cost is

$$
\begin{aligned}
I/O_{cost} &= \lceil \frac{IR_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil \frac{IR_i}{B_O} \rceil \cdot (T_{access} + B_O \cdot T_{transfer}) \\
&+ \lceil \frac{IS_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil \frac{IS_i}{B_O} \rceil \cdot (T_{access} + B_O \cdot T_{transfer})
\end{aligned}
$$

with the condition that

$$B_I + B_O + n \cdot B_C \leq M$$

CPU cost is

$$CPU_{cost} = hash \cdot (r_i + s_i)$$

Communication cost is

$$COMM_{cost} = comm \cdot (n-1) \cdot (IR_i + IS_i)$$

(2) Local Join by using Index Fragments (Generate the RID list)

For the local join, we use a hash join since the sorted sequence of leaf pages can not be kept during the redistribution. Since we consider a large size of input relations, it is possible that the index fragments (leaf pages) will not fit in memory for a hash table. Thus, we partition the index fragments into small buckets which fit in memory and then perform a bucket join.

(a) Bucket Partition:

I/O cost is

$$
\begin{aligned}
I/O_{cost} &= \lceil \frac{IR_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil \frac{IR_i}{B_O} \rceil \cdot (T_{access} + B_O \cdot T_{transfer}) \\
&+ \lceil \frac{IS_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil \frac{IS_i}{B_O} \rceil \cdot (T_{access} + B_O \cdot T_{transfer})
\end{aligned}
$$

with the condition that

$$B_I + h \cdot B_O \leq M \quad (h = number \ \ of \ \ buckets)$$

CPU cost is

$$CPU_{cost} = hash \cdot (r_i + s_i)$$

(b) Bucket Join:

I/O cost is

$$I/O_{cost} = \lceil \frac{IR_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil \frac{IS_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer})$$

$$+ \lceil \frac{RID_i}{B_O} \rceil \cdot (T_{access} + B_O \cdot T_{transfer})$$

with the condition that

$$B_I + B_O + HASH \leq M \quad (HASH = size\ of\ hash\ table)$$

CPU cost is

$$CPU_{cost} = hash \cdot r_i + probe \cdot s_i$$

(3) Read the Joining Tuples of R and S from the RID list

In this case, the tuples of R and S given in the local RID list can be found anywhere. So, we need to redistribute the RID list based on the RID of r (or s). I/O cost is

$$I/O_{cost} = \lceil \frac{RID_i}{B_I} \rceil \cdot (T_{access} + B_I \cdot T_{transfer})$$

with the condition that

$$B_I + n \cdot B_C \leq M$$

CPU cost is

$$CPU_{cost} = comp \cdot r_i \ \ or \ \ comp \cdot s_i$$

The remaining cost will be the same as the case of the index fragment replication.

4.3 Index Fragment Redistribution with Partitioned Global Indexes

For the partitioned global indexing scheme, the cost model will be almost the same as in the case of using the local indexing scheme except that the index redistribution cost should be eliminated from the total cost.

4.4 Parallel Hash Join

In this section, we include the cost model for the parallel hash join with the assumption that the indexes are not available on the joining attributes and the input relations (R and S) are very large (compared to the available memory size). We use a parallel grace hash join for the cost model.

(1) Redistribution

Initially, the tuples of R and S have to be redistributed so that each node can perform a local join operation independently.

I/O cost is

$$\begin{aligned} I/O_{cost} &= \lceil\frac{R_i}{B_I}\rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil\frac{R_i}{B_O}\rceil \cdot (T_{access} + B_O \cdot T_{transfer}) \\ &+ \lceil\frac{S_i}{B_I}\rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil\frac{S_i}{B_O}\rceil \cdot (T_{access} + B_O \cdot T_{transfer}) \end{aligned}$$

CPU cost is

$$CPU_{cost} = hash \cdot (r_i + s_i)$$

Communication cost is

$$COMM_{cost} = comm \cdot (n-1) \cdot (R_i + S_i)$$

(2) Bucket Partition

In order to prevent hash table overflow, R and S are divided into several buckets that fit in memory.
I/O cost is

$$\begin{aligned} I/O_{cost} &= \lceil\frac{R_i}{B_I}\rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil\frac{R_i}{B_O}\rceil \cdot (T_{access} + B_O \cdot T_{transfer}) \\ &+ \lceil\frac{S_i}{B_I}\rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil\frac{S_i}{B_O}\rceil \cdot (T_{access} + B_O \cdot T_{transfer}) \end{aligned}$$

with the condition that

$$B_I + h \cdot B_O \leq M \quad (h = number\ of\ buckets)$$

CPU cost is

$$CPU_{cost} = hash \cdot (r_i + s_i)$$

(2) Bucket Join

I/O cost is

$$I/O_{cost} = \lceil\frac{R_i}{B_I}\rceil \cdot (T_{access} + B_I \cdot T_{transfer}) + \lceil\frac{S_i}{B_I}\rceil \cdot (T_{access} + B_I \cdot T_{transfer})$$

with the condition that

$$B_I + HASH \leq M \quad (HASH = size\ of\ hash\ table)$$

CPU cost is

$$CPU_{cost} = hash \cdot r_i + probe \cdot s_i$$

5 Buffer Allocation for Sequential I/O

Sequential I/O is effectively used in certain database operations such as join and sort, which read a large number of contiguous data pages from disk. When a parallel join operation is applied to very large relations by a hash-based method, it requires a bucket partitioning step which distributes input relations into several buckets. In the case of the parallel sort-merge join operation, a multi-way merging step is needed for sorting large input relations. The I/O pattern of bucket partitioning and multi-way merge is almost the same, except for the I/O direction.

In order to facilitate the sequential I/O advantage during bucket partitioning and multi-way merge, we need to allocate buffer space efficiently. For example, if we increase the value of N (see Figure 3), the number of merge passes will decrease. On the contrary, the cost of merging at each step can increase with increasing N since we have fewer pages for each buffer slot with a larger N. The optimal choice of N will depend on many factors, such as available memory size, disk capacity (access time and data transfer time) and file size. Additionally, we see that the size of the input/output at each merging step is different. During the merging, N sorted runs are read into the input buffer and merged into one sorted run through the output buffer. Thus, the size of the output is the total size of N input runs. This means that the same buffer allocation for input/output will not give the optimal performance. We must also consider the difference of input/output size during the buffer allocation. In the following section, we will examine in detail how to determine the optimal N and the buffer allocation for input/output.

5.1 Choosing the optimal N

In order to choose the optimal N, we need to analyze the cost of the N-way merge sort under the given buffer allocation. Here, we consider the case of sorting very large files that are several orders of magnitude larger than main memory. In this case, we can assume that the CPU time is negligible and can evaluate the sorting cost in terms of the disk I/O time. The I/O cost for the external N-way merge sort consists of the cost for generating initial runs and the cost for merging. Also, the time for the run generation is independent of N, the merge order. Since our goal is to choose the optimal N, we will ignore the initial run generation cost in our analysis.

Given the buffer allocation as in Figure 3 (M buffer pages are divided into N slots which have W pages each; $N-1$ slots are used for reading input runs, and 1 slot is used for writing merged output), we need $\lceil \log_{N-1} R \rceil$ merge steps and at each step we need to read/write the whole file in blocks of W pages. The size of an initial run is M pages (M = available memory size) as previously assumed. The whole file size would be $M \cdot R$ (R = the number of initial runs). Thus, the I/O cost

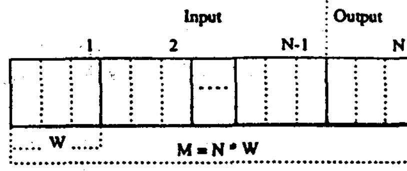


Figure 3: Buffer Allocation for N-way Merge Sort

during merging will be as follows (T_{access} = disk access time, $T_{transfer}$ = data transfer time for one page):

$$I/O_{merge} = 2 \cdot \lceil \log_{N-1} R \rceil \cdot \lceil \frac{M \cdot R}{W} \rceil \cdot (T_{access} + W \cdot T_{transfer}) \tag{1}$$

Claim 1 (Optimal N): If there are M pages of available buffer space for the external N-way merge sort, then the external N-way merge sort whose N value satisfies the following condition yields minimum sequential I/O cost.

$$N \cdot (\ln N - 1) = \frac{T_{transfer}}{T_{access}} \cdot M \tag{2}$$

Proof: The sequential I/O cost for the N-way merging is represented as Equation 1 under the given buffer allocation (Figure 3). For simplicity, we can eliminate the ceiling function ($\lceil\,\rceil$) without losing the accuracy since we consider large input files. Also, we can replace $\log_{N-1} R$ by $\log_N R$ if we assume that N is reasonably large (e.g., $\log_{19} 1000 \approx 2.346$ and $\log_{20} 1000 \approx 2.305$). The simplified equation becomes the following:

$$\begin{aligned} I/O_{merge} &= 2 \cdot \log_N R \cdot \frac{M \cdot R}{W} \cdot (T_{access} + W \cdot T_{transfer}) \\ &= 2 \cdot \log_N R \cdot (N \cdot R) \cdot (T_{access} + \frac{M}{N} \cdot T_{transfer}) \quad \Longleftarrow (M = N \cdot W) \\ &= 2 \cdot R \cdot \log_N R \cdot (N \cdot T_{access} + M \cdot T_{transfer}) \\ &= 2 \cdot R \cdot \frac{\ln R}{\ln N} \cdot (N \cdot T_{access} + M \cdot T_{transfer}) \end{aligned} \tag{3}$$

With the assumption that the input file is much larger than the memory buffer and the memory buffer is reasonably large (i.e., more than 100 pages), we can approximate the discrete case by computing the continuous one. After differentiating Equation 3 by N, setting the result to zero to

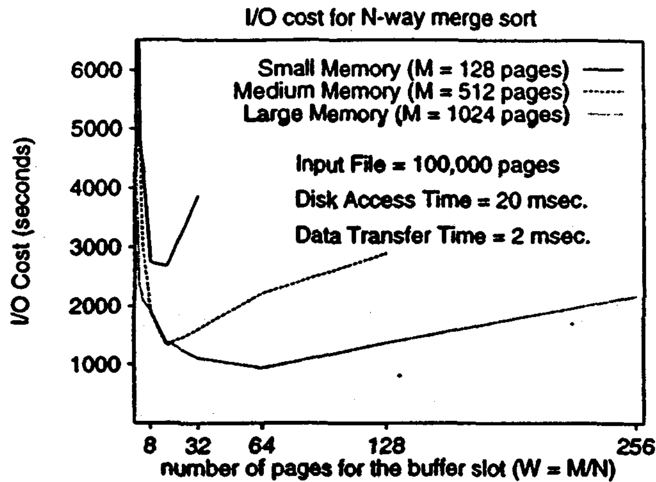


Figure 4: I/O Cost with Different W Values

find the minimum gives:

$$0 = \frac{2 \cdot R \cdot \ln R}{N \cdot (\ln N)^2}(N \cdot \ln N \cdot T_{access} - N \cdot T_{access} - M \cdot T_{transfer}) \tag{4}$$

Since $R \neq 0$, $N \neq 0$, and $M \neq 0$,

$$0 = (N \cdot \ln N \cdot T_{access} - N \cdot T_{access} - M \cdot T_{transfer})$$
$$\Longrightarrow \quad N \cdot (\ln N - 1) = \frac{T_{transfer}}{T_{access}} \cdot M$$

Thus, Equation 2 satisfies the condition which suffices to yield minimum sequential I/O cost for N-way external sort.

Our analysis shows that the optimal N depends on the available memory size and disk capacity (disk access time and data transfer time) but is independent of the input file size. The available memory size determines the optimal N because the disk capacity can be treated as constant. Even though our analysis was obtained by approximating the discrete case with the continuous one, we can find a nearly optimal N from Equation 2. Figure 4 represents the variance of I/O cost with the different $W(W = M/N)$ values.

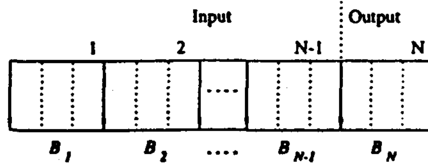


Figure 5: Buffer Allocation for Input/Output

5.2 Buffer Allocation for Input/Output

Another thing to be considered in the buffer allocation for the external N-way merge sort is the fact that the size of the input/output at each merge step is different. As we see from the previous example, the I/O cost can be reduced if we consider each operation's file size. In this section, we will look into the relationship between the buffer allocation and file sizes.

Claim 2: Consider the case where we need K buffer slots for input/output at one time, the file size which is read or written through the i-th buffer slot is F_i, and the number of blocks allocated to the i-th buffer slot is B_i (Figure 5). If we allocate B_i as follows, we can achieve the minimum sequential I/O cost.

$$\frac{F_1}{B_1^2} = \frac{F_2}{B_2^2} = \frac{F_3}{B_3^2} = \cdots = \frac{F_K}{B_K^2} \tag{5}$$

Proof: If we allocate B_i blocks to the i-th buffer slot, then the I/O cost will be:

$$\begin{aligned} I/O_{cost} &= \lceil \frac{F_1}{B_1} \rceil \cdot (T_{access} + B_1 \cdot T_{transfer}) \\ &+ \lceil \frac{F_2}{B_2} \rceil \cdot (T_{access} + B_2 \cdot T_{transfer}) \\ &+ \cdots \\ &+ \lceil \frac{F_K}{B_K} \rceil \cdot (T_{access} + B_K \cdot T_{transfer}) \\ &= \sum_{i=1}^{K} \lceil \frac{F_i}{B_i} \rceil \cdot (T_{access} + B_i \cdot T_{transfer}) \end{aligned} \tag{6}$$

For large F_i, we can eliminate the ceiling function ($\lceil\ \rceil$) without loss of accuracy.

$$I/O_{cost} = \sum_{i=1}^{K} \frac{F_i}{B_i} \cdot (T_{access} + B_i \cdot T_{transfer}) \tag{7}$$

Also, if M is the total size of available memory, there is a condition such that

$$M = B_1 + B_2 + \cdots + B_K. \tag{8}$$

Now we can minimize the I/O cost (Equation 7). subject to Equation 8. For convenience, we assume that the B_i's do not have to be integers. If we increase a B_i by ϵ, then some other B_j might decrease by ϵ according to Equation 8. The difference in the I/O cost after this change is shown below.

$$\begin{aligned} I/O_{cost}(before) &= \sum_{k=1}^{K} \frac{F_k}{B_k} \cdot (T_{access} + B_k \cdot T_{transfer}) \\ I/O_{cost}(after) &= \sum_{k=1, k\neq i, k\neq j}^{K} \frac{F_k}{B_k} \cdot (T_{access} + B_k \cdot T_{transfer}) \\ &+ \frac{F_i}{(B_i + \epsilon)} \cdot (T_{access} + (B_i + \epsilon) \cdot T_{transfer}) \\ &+ \frac{F_j}{(B_j - \epsilon)} \cdot (T_{access} + (B_j - \epsilon) \cdot T_{transfer}) \end{aligned}$$

$$\begin{aligned} I/O_{cost}(difference) &= I/O_{cost}(after) - I/O_{cost}(before) \\ &= T_{access} \cdot (\frac{F_i}{B_i + \epsilon} - \frac{F_i}{B_i} + \frac{F_j}{B_j - \epsilon} - \frac{F_j}{B_j}) \\ &= T_{access} \cdot (\frac{(F_j B_i^2 - F_i B_j^2)\epsilon + (F_i B_j + F_j B_i)\epsilon^2}{B_i(B_i + \epsilon)B_j(B_j - \epsilon)} \\ & (\epsilon^2 \approx 0 \ \ since\ \epsilon\ is\ very\ small) \\ &= T_{access} \cdot (\frac{(F_j B_i^2 - F_i B_j^2)\epsilon}{B_i(B_i + \epsilon)B_j(B_j - \epsilon)} \end{aligned}$$

Thus, if $F_j \cdot B_i^2 \neq F_i \cdot B_j^2$, there can be an I/O cost improvement. Therefore, the minimum I/O cost occurs when Equation 5 holds.

Now, we apply this condition to our problem of input/output buffer allocation for the N-way external sort (Figure 5). The assumption that we are doing an N-way merge where the size of N input runs are all the same gives (B_i = # of blocks for i-th input buffer, B_{out} = # of blocks for output buffer, M = total available memory size, F_i = input run size, and F_{out} = merged output size):

$$B_1 + B_2 + \cdots + B_{N-1} + B_N + B_{out} = M$$

Table 2: Values of the System Parameters

Parameters	Values	Description
T_{access}	20 msec.	Average disk access time
$T_{transfer}$	2 msec.	One page data transfer time
comm	0.1 msec.	Time in sending/receiving a packet
comp	0.005 msec.	Time in comparing two tuples and moving a tuple
hash	0.008 msec.	Time in hashing a tuple and moving a tuple
probe	0.01 msec.	Time in probing hash table

$$F_1 + F_2 + \cdots + F_{N-1} + F_N = F_{out}$$

$$F_1 = F_2 = \cdots = F_{N-1} = F_N, \quad F_{out} = N \cdot F_i$$

Since all F_i's are the same, we know that all B_i's will be the same for the optimum condition. By Equation 5, we know

$$\frac{F_i}{B_i^2} = \frac{F_{out}}{B_{out}^2} \implies \frac{F_i}{B_i^2} = \frac{N \cdot F_i}{B_{out}^2}. \quad Thus, \quad B_{out} = \sqrt{N} \cdot B_i.$$

Also,

$$M = B_{out} + N \cdot B_i = B_{out} + N \cdot \frac{B_{out}}{\sqrt{N}} = B_{out} \cdot (\sqrt{N} + 1).$$

Therefore,

$$B_{out} = (\frac{\sqrt{N} - 1}{N - 1}) \cdot M \quad and \quad B_i = (\frac{N - \sqrt{N}}{N(N-1)}) \cdot M. \tag{9}$$

As previously mentioned, we can get a nearly optimal B_i and B_{out} even though it approximates the discrete case with the continuous one. For example, if $N = 9$ and $M = 48$ pages, then $B_{out} =$ 12 pages and $B_i = 4$ pages will be the optimal buffer allocation for input/output.

6 Experimental Result

With the values of the system parameters shown in Table 2, we examine the performance effect of other parameters. Figure 6 shows the effect of varying memory size. As previously mentioned, sequential I/O is used for reading/writing the contiguously stored data pages. As we see from Figure 6, the execution cost of the parallel hash join algorithm is significantly reduced with the

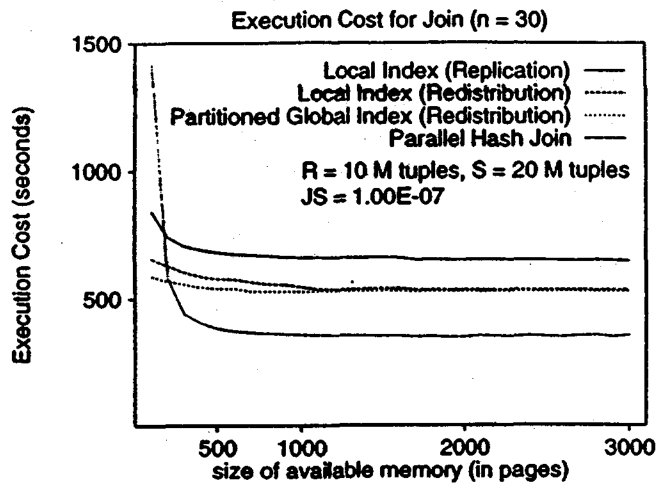


Figure 6: Effect of Available Memory Size

increase of available memory. In the parallel hash join algorithm, a small memory size causes hash table overflow more frequently. Thus, a memory increase can reduce the I/O cost of overflow data significantly. Another reason is that this algorithm requires several table scans for data redistribution and bucket partitioning and that a large memory allows the reading of many data pages at one disk access. We also see that the cost reduction is negligible after 500 memory pages. This is because hash table overflow does not occur after this point.

As for the parallel index-based join, the effect of a memory increase is not noticeable compared to the parallel hash join. This is because the hash table overflow does not occur during the local join with index leaf pages. Furthermore, when we retrieve the joining tuples, we should read data pages one by one because the sequence of data pages might not be contiguously stored in the disk. So, it can not utilize large memory very well.

Among the index-based algorithms, index redistribution methods perform better. The reason for this is that, as indicated in section 3, unnecessary index fragments can be avoided in the index redistribution method. The small gap between the local index and the partitioned global index comes from the cost of index fragment redistribution. But this is not too great compared to the cost of retrieving the joining tuples from the data pages of R and S.

Figure 7 shows how the performance improves with a greater number of nodes. Since the size of local data fragments (indexes and tables) are reduced with the increase of nodes (the increase of nodes also implies the increase of total memory size in a system), the execution cost of each algorithm is reduced as much. As we see from Figure 7, the parallel hash join algorithm shows a

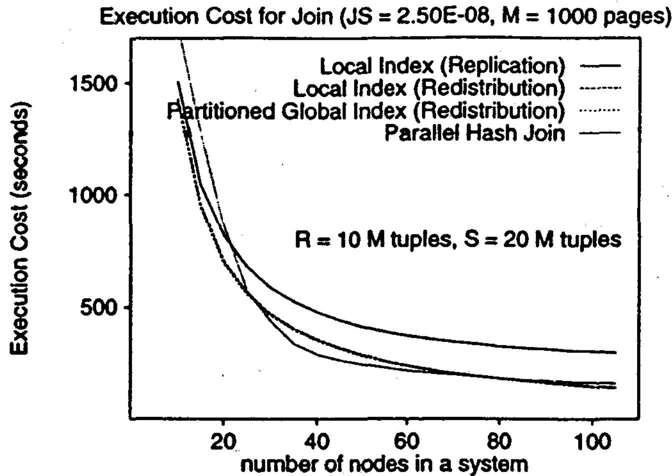


Figure 7: Effect of Varying Nodes in a System

noticeable performance improvement between 10 and 40 nodes compared to the other algorithms. The reason for this is that hash table overflow is significantly reduced in that range.

Figure 8 shows the effect of the join selectivity factor. As expected, the execution cost is increasing with the increase of the join selectivity factor in the index-based join. This increase is not exactly proportional to the increase of the join selectivity factor due to the optimized access of pages during the retrieval of the joining tuples. We also see that if the join selectivity factor is smaller than 5.0×10^{-8}, the index-based join will be the the best choice of the join algorithms.

7 Conclusion

In this paper, we analyzed the execution cost of the index-based parallel join algorithm and the ordinary parallel hash join algorithm. Our results show that the parallel hash join algorithm is more sensitive to the available memory size than the index-based join algorithm. Our investigation also shows that if the join selectivity factor is relatively small and the available memory is not large, then the index-based parallel join method is the best method for joining large input relations.

References

[1] D. DeWitt, J. Naughton, and J. Burger. Nested Loops Revisted. In *Proceedings of the International Symposium on Parallel and Distributed Information Systems*, 1993.

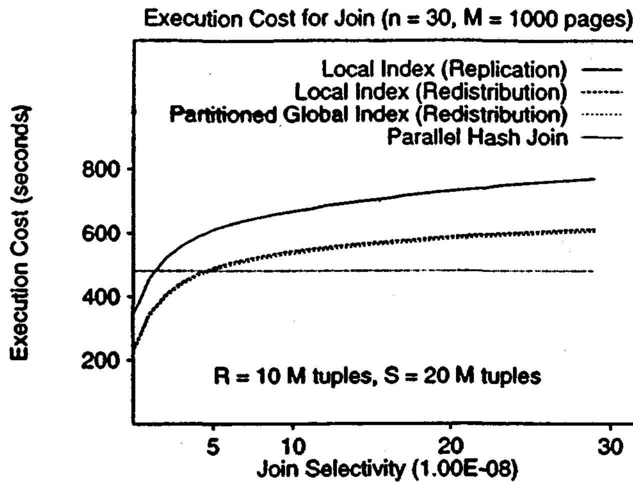


Figure 8: Effect of Join Selectivity Factor

[2] Byeong-Soo Jeong and Edward Omiecinski. Index File Partitioning in Parallel Database Systems. In *Submitted to International Conference on Parallel and Distributed Systems*, 1995.

[3] H. Lu, M. Shan, and K. Tan. Optimization of Multi-Way Join Queries for Parallel Execution. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 549–560, 1991.

[4] H. Lu, K. Tan, and M. Shan. Hash-Based Join Algorithms for Multiprocessor Computers with Shared Memory. In *Proceedings of the 16th International Conference on Very Large Data Bases*, pages 198–209, 1990.

[5] E. Omiecinski. Performance Analysis of a Load Balancing Relational Hash Join Algorithm for a Shared-Memory Multiprocessor. In *Proceedings of the 17th VLDB Conference*, pages 375–386, August 1991.

[6] E. Omiecinski and E. T. Lin. Hash-Based and Index-Based Join Algorithms for Cube and Ring Connected Multicomputers. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):329–343, September 1989.

[7] Edward Omiecinski and Ron Shonkwiler. Parallel Join Processing using Nonclustered Indexes for a Shared Memory Multiprocessor. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 144–151, 1990.

[8] D. A. Schneider and D. J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of ACM SIGMOD - International Conference on Management of Data*, pages 110–121, 1989.

[9] J. L. Wolf, D. M. Dias, and P. S. Yu. An Effective Algorithm for Parallelizing Sort-Merge Joins in the Presence of Data Skew. In *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems*, pages 103–115, 1990.

[10] J. L. Wolf, D. M. Dias, P. S. Yu, and J. Turek. An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 200–209, 1991.