VLDB Journal, 2, 303-330 (1993), Michael Carey and Patrick Valduriez, Editors

Considering Data Skew Factor in Multi-Way Join Query Optimization for Parallel Execution

Kien A. Hua, Yo-Lung Lo, and Honesty C. Young

Received December 1, 1992; revised version received February 1, 1993; accepted March 15, 1993.

Abstract. A consensus on parallel architecture for very large database management has emerged. This architecture is based on a shared-nothing hardware organization. The computation model is very sensitive to skew in tuple distribution, however. Recently, several parallel join algorithms with dynamic load balancing capabilities have been proposed to address this issue, but none of them consider multi-way join problems. In this article we propose a dynamic load balancing technique for multi-way joins, and investigate the effect of load balancing on query optimization. In particular, we present a join-ordering strategy that takes load-balancing issues into consideration. Our performance study indicates that the proposed query optimization technique can provide very impressive performance improvement over conventional approaches.

Key Words. Parallel-database computer, load balancing, multi-way join, query optimization.

1. Introduction

Parallelism has been recognized as the only way to handle projected increases in data size and query complexity in future database applications. Various multiprocessor architectures can be used for database management. However, a consensus has emerged that shared-nothing structure (Stonebraker, 1986) is most scalable to support very large databases (Su, 1983; Teradata Corp., 1988; Boral et al., 1990; DeWitt et al., 1990; Englert et al., 1990; Hua and Young, 1990; Lorie et al., 1991). This hardware organization consists of a set of *processing nodes* (PNs) interconnected

---

An earlier version of this article was presented at the 1993 International Conference on Parallel and Distributed Information Systems in San Diego, California, U.S.A.

Kien A. Hua, Ph.D., is Assistant Professor, and Yu-Lung Lo is Ph.D. candidate, Department of Computer Science, University of Central Florida, Orlando, FL 32816-2362. Honesty C. Young, Ph.D., is Research Staff Member, IBM Research Division, Almaden Research Center, San Jose, CA 95120-6099.

through a communication network. Each PN consists of its own memory and local disk drives. Communication among PNs is carried out by message passing. In this computation model, each relation is partitioned into disjoint fragments and distributed across some number of PNs. Since each PN can independently process a portion of the database on its disks, a high degree of parallelism is achievable. This strategy, however, is very sensitive to skew in tuple distribution. When skewed tuple distribution occurs, load balancing is necessary to ensure good system performance. Since it has not been demonstrated that severe join product skew happens in practice, this article emphasizes redistribution skew (Walton et al., 1991).

The join operation has been the most intensively studied among relational operations for shared-nothing architecture. In recent years, several parallel join algorithms with dynamic load balancing capabilities have been proposed to address the skewed tuple distribution problem (Hua and Lee, 1990, 1991; Kitsuregawa and Ogawa, 1990; Wolf et al., 1990, 1991*a*, 1991*b*; DeWitt et al., 1992; Swami et al., 1992). Performance studies indicate that these techniques can provide very significant performance improvement over conventional parallel join strategies. However, none of these works considered the multi-way join problem. Although these join algorithms can be used in a system in which a multi-way join query is executed as a sequence of 2-way joins, a good query optimizer for the shared-nothing environment should exploit *inter-join* parallelism where several joins are performed concurrently (Lu et al., 1991). Depending on the sizes of the operand relations and join selectivity factors, the execution of each join operation can be centralized to a few PNs or to as many PNs as is appropriate to minimize the response time of the query. Furthermore, in this article, we introduce data skew as a new factor for query optimization. We will show that load balancing is more difficult for a shared-nothing system with many PNs. When skew is severe, one can employ inter-join parallelism to reduce the effect. With inter-join parallelism, a smaller subset of PNs is used to perform a join; hence a balanced load is easier to achieve.

Recently, several interesting techniques have been proposed to optimize queries for parallel execution. Schneider and DeWitt (1990) studied the behavior of query plans with different type of structures (*left-deep, right-deep,* and *bushy*) in processing multi-way join queries. Their study demonstrated that right-deep scheduling strategies can provide significant performance advantages in large shared-nothing systems under many circumstances. Chen et al. (1992) proposed a technique that uses a segmented right-deep tree (a bushy tree of right-deep subtrees) for query execution tree selection. Two heuristics were developed to determine the query execution plan in this scheme. The objective of the *minimal work* heuristic is to select relations in the current segment to minimize the total amount of work. This is a greedy heuristic and tends to include small relations in the first few pipeline segments. Another heuristic, *balanced consideration*, considers both the penalty (join work) and the benefit (join size reduction) to determine which relation to include in the current segment in order to avoid the tendency to select small relations. Lu et al. (1991) modeled the execution of a multi-way join query by dividing the join

operations into synchronized iterations. For each iteration, a number of joins are executed concurrently, and the joins of the next iteration will not start to execute until all joins in the current iteration have been completed. This scheme tries to join as many pairs of relations as possible in parallel for each execution step. A greedy algorithm is used to decide the optimal degree of inter-join parallelism to exploit for each iteration. Stonebraker (1988) and Hong and Stonebraker (1991) proposed a 2-step optimization technique for shared memory systems, in which a set of good sequential plans is first generated based on the possible buffer pool sizes, and the parallelization of these plans are done in step 2. The outcome is a collection of plans and a memory range over which each should be run. Since inter-query parallelism is also allowed at the time of execution (Stonebraker, 1988), the query executor must call on the buffer pool manager to determine space availability and to choose one of the optimized execution plans.

In this article, we propose a new multi-way join optimization algorithm, called Load Balancing Optimization (LBO), based on the parallel join strategy presented in Hua and Lee (1991). Our technique is different from the existing ones in that load balancing also is considered in determining the degree of inter-join parallelism for each execution iteration. To investigate the efficiency of the proposed scheme, we developed a performance model and compared its performance to the following three strategies:

1. *Linear Tree With Load Balancing* (L_LB) Strategy: In this scheme, a multi-way join query is treated as a sequential order of two-way or single joins. At least one operand of a join operation is a base relation. We ordered the join operations by the increasing sizes of their intermediate relations, and dynamic load balancing was performed for each join operation of the query without exploiting inter-join parallelism.

2. *Bushy Tree Without Load Balancing* (B_NLB) Strategy: This scheme is similar to the technique presented by Lu et al. (1991). We exploited inter-join parallelism, but did not perform load balancing.

3. *No Load Balancing Optimization* (NLBO) Strategy: In this scheme, the query is optimized using B_NLB, and dynamic load balancing is performed at runtime for each join operation. Although this strategy exploits inter-join parallelism and also performs load balancing, it is different from LBO in that the load balancing issue is not considered during query optimization.

The performance comparison of NLBO and L_LB demonstrates the advantages of exploiting inter-join parallelism, whereas the comparison of NLBO and B_NLB shows the importance of load balancing in multi-way join execution. The comparison of LBO and NLBO confirms the need to consider load balancing in query optimization to achieve the best performance. We will discuss these query processing strategies in more detail in Section 4.

The rest of this article is organized as follows. In Section 2, the effect of skewed tuple distribution is discussed, and a load balancing strategy based on the *partition tuning* concept is described. The proposed LBO algorithm, along with L_LB, B_NLB and NLBO, is presented in Section 3. In Section 4, we introduce a simulation model and discuss the performance comparison of the proposed scheme with the other three techniques. We offer our conclusions in Section 5.

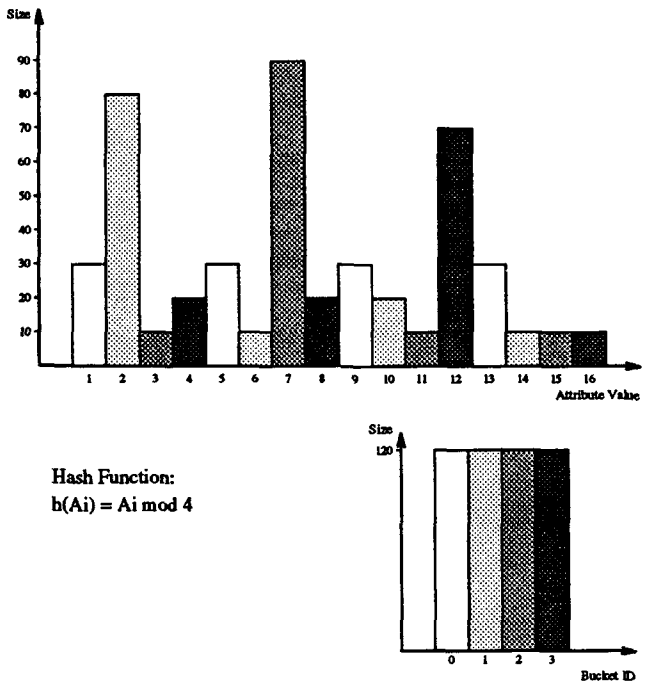
2. Skew Effect and Dynamic Load Balancing

The performance of conventional parallel hash join algorithms relies on the randomizing hash function to redistribute the tuples of the join relations evenly across all PNs in the system. Their performance degrades when the join attribute values of the relations are non-uniformly distributed (Kitsuregawa et al., 1983; Lakshmi and Yu, 1988, 1989). That is, some PNs have more tuples to process than the remaining PNs in the system. The concept of *data skew* was described by Lakshmi and Yu (1988) as the phenomenon in which certain values for a given attribute occur more frequently than other values. The effect of data skew, however, needs to be clarified.

A *partition* is a set of hash buckets assigned to a PN. We use the term *bucket* to mean the tuples hashed to the same bucket for distribution purposes, which should not be confused with the bucket chain of a hash table. If we know the distribution of the relation, we can always design a hash function to minimize fluctuation in the size of the hash buckets, provided that the skew is not too severe. In this case, if every PN is allocated the same number of hash buckets, then the data load is balanced for all PNs (i.e., the sizes of all the partitions are the same). Unfortunately, we usually do not know the distributions of the relations. The general approach is to assume that they are uniform, and employ a randomizing hash function to hash the relations into matching join buckets. When the uniformity assumption is violated, imbalanced buckets occur and consequently the size of the partitions will not be uniform.

It is important to distinguish among data skew, bucket skew, and partition skew. From the above discussion, we define *bucket skew* as the phenomenon in which some hash buckets have more tuples than other buckets due to non-uniformity in the distribution of the join attribute. Similarly, *partition skew* can be defined as the phenomenon in which some partitions have more tuples than others due to the non-uniformity in the size of the join buckets. We note that data skew may not cause any negative effect. One such example is given in Figure 1, where the values 2, 7, and 12 occur significantly more frequently than the other values (i.e., data skew). Nevertheless, the randomizing hash function produces perfectly balanced hash buckets. Similarly, depending on how the imbalanced buckets are mapped to the PNs, bucket skew may not cause any negative effect either. *Partition skew,* however, implies load imbalance and the problem must be rectified for good system performance. We determined the size of a hash bucket by the Zipf-like distribution (Zipf, 1949; Sacco, 1986; Turbyfill, 1987). This will be discussed in more detail in

Figure 1. Data skew example



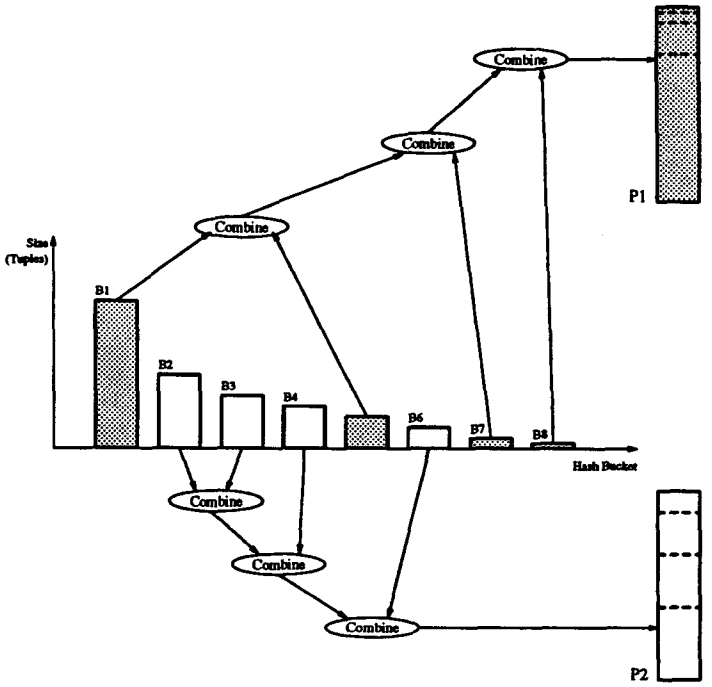
Section 4.

When the hashing phase of a join operation results in skewed partitions, one can rehash the relations using a better hash function design based on the distribution information collected during the first hashing process. The redistribution of the relations, however, is very costly. Alternatively, one can use a finer grain hash function to decluster the relations into smaller buckets so that these uneven hash buckets can be combined to form balanced partitions for the PNs. This process is referred to as *partition tuning* (Hua and Lee, 1990, 1991). A Best Fit Decreasing strategy for partition tuning is illustrated in Figure 2. In this scheme, the hash buckets are first sorted by size in decreasing order. In each iteration, the currently largest bucket is assigned to the currently smallest partition (or PN). This process is repeated until all the buckets have been allocated. Similar strategies for load balancing have been proposed (Wolf et al., 1990, 1991*a*, 1991*b*; Swami et al., 1992).

A parallel join algorithm, Extended Adaptive Load Balancing Parallel Hash Join (ABJ^+), based on the partition tuning concept, was presented by Hua and Lee (1991). A simplified version of that algorithm is given below:

1. *Split Phase:* Each PN partitions its portion of each relation into small sub-buckets and stores them back to its own disks.

Figure 2. Best Fit Decreasing partition tuning strategy



2. *Partition Tuning Phase:* Each PN reports the sizes of its subbuckets to a designated coordinating PN. The coordinator adds up the sizes of the matching subbuckets distributed across the PNs to derive the sizes of the corresponding buckets. The coordinator then allocates the buckets to the PNs using the Best Fit Decreasing strategy.

3. *Bucket Tuning Phase:* Each PN combines the small buckets to form optimally sized join buckets that fit the memory capacity.

4. *Join Phase:* Each PN performs the local joins of respectively matching buckets.

3. Multi-Way Join Optimization Techniques

In this section, we discuss the multi-way join execution strategies in more detail. In this article, we consider queries of the form of conjunctions of equi-join predicates. This class of queries covers most joins. A join query graph is denoted by G = (V, E), where V is the set of vertices and E is the set of edges. Each vertex in a join query graph represents a relation. Two vertices are connected by an edge if

there is a join predicate on some attribute of the two corresponding relations. The execution of a query is denoted by a query execution tree. In a query execution tree, a leaf vertex represents an input (or base) relation and an internal vertex represents the relation that results from joining the two relations of its two child vertices. The query execution tree is executed in a bottom up (from leaf to root) manner and can be one of three forms: left-deep tree, right-deep tree, or bushy tree. For both the left-deep tree and the right-deep tree, the execution order is uniquely determined by the query execution tree and inter-join parallelism cannot be achieved. Pipelining is possible for query execution and is an important factor to be considered by the query optimizer. However, to avoid mixing the advantages of pipelining and those of the proposed scheme, we do not consider pipelining in our study. Regardless of the shape of the query execution tree, the intermediate result of each join is written back to disk before next execution iteration.

For either the left-deep tree or the right-deep tree, the number of iterations to complete the join is equal to the depth of the query execution tree. A single join is carried out in an iteration by all PNs. For a bushy tree, it is possible to perform several joins within the same iteration and a subset of PNs is used by each join.

We implemented four optimizers for multi-way join queries. They accept a query graph as input and generate an optimized query execution tree as output. When the query execution tree is a bushy tree, they also determine which joins are executed in the same iteration (i.e., degree of inter-join parallelism) and the degree of parallelism for each join operation (i.e., degree of intra-join parallelism). The details of the optimizers are described in the following subsections.

3.1 Linear Tree With Load Balancing (L_LB) Strategy

In this strategy, a multi-way join query is treated as a sequential order of two-way joins. A simple static query optimization algorithm is used to order the join operations by the increasing sizes of their intermediate relations. At execution time, ABJ^+ is used to execute each join operation according to the optimized join order.

Although partition tuning is very effective in balancing skewed tuple distribution, it becomes increasingly difficult to balance the workload as the number of PNs increases (or as the sizes of the relations decrease). Let *average size* be the number of tuples each PN should have if the workload is evenly distributed across the PNs.

$$average\ size = \frac{size\ of\ two\ relations}{number\ of\ PNs}.$$

Obviously, if there is a pair of matching join buckets that is larger than the average size, then partition tuning using Best Fit Decreasing strategy will fail to balance the workload perfectly. When this happens, one can increase the average size by decreasing the number of PNs used for the join operation. In other words, inter-join parallelism can be used to alleviate skew. Because L_LB strategy always uses all the PNs for each join, its performance suffers when severe skew is encountered.

3.2 Bushy Tree Without Load Balancing (B_NLB) Strategy

This scheme was proposed by Lu et al. (1991). In this strategy, the execution of a multi-way join query is divided into synchronized iterations. For each iteration, a number of joins are executed concurrently, and the joins of the next iteration cannot start executing until all joins in the current iteration have finished. This scheme tries to join as many pairs of relations as possible in parallel for each execution iteration. The detail of the algorithm is given in the following.

Algorithm B_NLB

Input: A join graph $G = (V, E)$
where vertex set V is a set of relations and edge set E represents the join predicates.
Output: S, the query execution tree.

```
begin
    S ← ∅
    while Size(V) > 3 do
        R ← Select_rel_pairs(G);
        S ← S ⋃ R;
        G ← G with each pair of relations in R replaced by their join results;
    end-while
        R ← Two_way_seq(G);
        S ← S ⋃ R;
end
```

Algorithm Select_rel_pairs
selects a set of relation pairs to be joined in the current iteration.

Input: G, a join graph
Output: R, a set of relation pairs to be joined concurrently in the same iteration

```
begin
    k ← 0;
    repeat
        k ← k + 1;
        C_k ← Minimum_cost(G,k,R_k);
        if (R_k does not contain all relations in G) then
        C_{k+1} ← Minimum_cost(G,k+1,R_{k+1});
    until C_{k+1} > C_k or R_{k+1} contains all pairs in G;
    if C_{k+1} > C_k then
        return R_k
    else
        return R_{k+1}
end
```

Function *Minimum_cost* is the core part of B_NLB. It takes a join graph G, and the number of joins to be performed concurrently, k, as input, and returns the minimum cost of those plans that join k pairs first. In addition, it determines those k pairs of relations and returns them in R_k. We note in B_NLB that when the number of relations in the join graph is less than four, we cannot perform two or more joins in parallel, and function *Two_way_seq* is called to determine the sequence of joining those relations. To perform the individual join operations, one can consider using either GRACE or Hybrid Hash Join. Brief descriptions of these algorithms are given in the following paragraphs.

GRACE Hash Join was presented by Kitsuregawa et al. (1983). This join method consists of two distinct phases: split phase and join phase. In the Split Phase, each of the operand relations is partitioned in parallel. Each PN independently declusters its local portion of the relations into buckets by hashing on the join attribute of each tuple in the relations. If a tuple belongs to a bucket allocated to the local PN, it is written back to the local disk. Otherwise, it is transferred to the PN that corresponds to the bucket ID, and is stored in that PN's disk. In the join phase, each PN performs joins of the allocated buckets in its local disk. Because tuples of a relation in one bucket join only with tuples of a respective matching bucket from the other relation, there is no communication among the PNs during the join phase. The whole join operation completes when all the PNs have finished their local joins.

Hybrid Hash Join (Schneider and DeWitt, 1989), is another popular parallel hash-based join algorithm. This method combines the GRACE Hash Join technique with the simple hash join algorithm (DeWitt et al., 1984). This method can also be viewed as having two phases: split phase and join phase. In the split phase, the operand relations are partitioned into buckets in parallel as in the GRACE Hash Join. However, as a bucket of relation R is being formed in a PN, another hash function is used to further decluster the bucket into subbuckets so that each subbucket can fit individually in the memory. As the local subbuckets are being formed in a PN, a hash table is built for one of the subbuckets while tuples belonging to the remaining subbuckets are written to the local disk. Buckets of relation S are also further declustered into subbuckets in the same way. As these subbuckets are being formed, the tuples that belong to the subbuckets corresponding to the in-memory hash table are used immediately to probe the hash table for matches. During the join phase, each PN reads the next subbucket from relation R to build an in-memory hash table. The respective matching subbucket from relation S is then read and its tuples are used to probe the hash table. This process continues until all matching subbuckets are joined in all PNs.

In the GRACE Hash Join, the split phase is completely separated from the join phase. Both operand relations must be written back to disks before beginning the join phase of the algorithm. The Hybrid Hash Join algorithm overlaps the split phase with the join phase. The first two subbuckets are immediately joined during the split phase to save disk I/O's. The savings can be significant when the

memory capacity is large. Nevertheless, without loss of generality, we assume that the GRACE algorithm is used because it has a simpler cost function.

3.3 No Load Balancing Optimization (NLBO) Strategy

In this scheme, although we perform dynamic load balancing for each join operation at run time, the load balancing issue is not considered in query optimization. The query processing strategy consists of the following steps:

1. *Compilation*: We optimize the multi-way join query using B_NLB.

2. *Execution*: We execute the join operations in accordance with the schedule generated by B_NLB. We used ABJ^+ to perform each join operation.

Due to the dynamic load balancing feature, we can expect NLBO to outperform B_NLB, particularly when skew is severe. However, the advantage of load balancing is not fully exploited here because the degree of inter-join parallelism is fixed by B_NLB, which does not consider the load balancing issue. When severe skew is encountered, it is advantageous to reduce the degree of intra-join parallelism to minimize the effect of skewed tuple distribution. Unfortunately, this dynamic reconfiguration of the PN allocation is not allowed in this scheme. In the next subsection, we will present a query optimization algorithm that addresses this deficiency.

3.4 Load Balancing Optimization (LBO) Strategy

In general, a query may be optimized at different times relative to the actual time of query execution. Optimization can be done *statically* before executing the query (e.g., System R, Selinger et al., 1979) or *dynamically* as the query is executed (e.g., Ingres, Wong and Youssefi, 1976). Without loss of generality, we choose to present LBO as a dynamic optimization algorithm for the sake of clarity. The proposed strategy can easily be adapted for static optimization. LBO consists of the following phases:

1. *Hash Phase:* Each PN partitions its portion of each operand relation into considerably smaller subbuckets. Each subbucket is stored back in the local disks.

2. *Optimization Phase:* The algorithm *Select_relation_pairs* is called to determine the pairs of relations to be joined at the current iteration. Function *Select_relation_pairs* includes as many joins in the current iteration as is beneficial.

Algorithm Select_relation_pairs
Input: G, a join graph
Output: S, a set of relation pairs to be joined concurrently
begin

```
  R ← ∅;
  S ← ∅;
  C2 ← very large value;
  repeat
   C1 ← C2;/* save previous cost */
   S ← S ⋃ R;
   R ← Search_smallest(G);/* select relation pair with smallest join result */
   if R ≠ ∅ then
    T ← G with the relations in (S⋃R) replaced by their join results;
    C2 ← Par_join_cost_skew(S⋃R) + Seq_join_cost(T); /* compute new cost */
   end-if
  until C2 > C1 or R = ∅;
  return S
end
```

The following functions are used in the algorithm *Select_relation_pairs*:

- *Search_smallest*: This procedure accepts a join graph as input. It determines, from the unmarked relations, the relation pair whose join result is smallest. It then marks the selected pair, and returns it to *Select_relation_pairs*.
- *Par_join_cost*$_{skew}$: This procedure accepts a set of relation pairs as input, and returns the cost of joining those relation pairs in parallel. It also performs the processor allocation function. Since the information on tuple distribution is available for these concurrent join operations, this procedure is able to take skew into consideration in the estimation of the execution cost. That is, it simulates the partition tuning process (without actually moving the tuples) to determine the most heavily loaded PN, and the estimated join costs are dictated by the execution times at this bottleneck PN. We noted that, because the joins performed concurrently in the same iteration are (data) independent, we can view them collectively as a single larger join operation. Therefore, partition tuning can be applied to this set of joins as if they are a single join operation. In other words, a PN can participate in more than one join operation during an execution step. This strategy allows a maximum flexibility for load balancing. It also minimizes the communication cost because we do not have to migrate the tuples of the operand relations to concentrate the concurrent join operations to different disjoint sets of the PNs. A similar technique was independently proposed for shared-disk architectures (Tan and Lu, 1992).
- *Seq_join_cost*: This procedure accepts a join graph as input. It orders the join operations by the increasing sizes of their join results, and returns the total cost of the execution plan. In other words, this function returns the cost to execute the remaining joins sequentially.

3 *Execution Phase:*

Stage 1 : ABJ$^+$ is used to execute the join operations selected by the *optimization phase* for concurrent processing. As we have explained in the optimization phase, ABJ$^+$ can treat all these joins as a single larger join operation. Furthermore, because the relations have already been hashed into join buckets, the split phase of ABJ$^+$ can be omitted here. In addition, because the partition tuning process has been simulated during optimization phase, the PNs only need to collect the join buckets in accordance with the simulation results during the partition tuning phase of the ABJ$^+$ algorithm.

Stage 2 : The join graph G is updated by replacing the relations joined in Stage 1 by their result relations and merging the join edges accordingly. If there are remaining join operations, they go to the optimization phase.

We observe that load balancing in LBO is integrated into the query optimization process. When severe skew occurs in some k-th join during the optimization phase, and because skew is considered in the cost estimation, adding the $(k+1)$-st join to the current iteration (i.e. trading some intra-join parallelism for additional inter-join parallelism) is likely to result in a better execution plan. Thus, unlike conventional techniques (e.g., Lu et al., 1991), we introduce skew effect as a new parameter for query optimization in addition to the traditional factors (e.g., relation sizes, selectivity factors, etc.). In other words, severe skew in tuple distribution plays a decisive role in determining the optimal level for inter-join parallelism.

4. Performance Analysis

In general, the cost of a parallel join method is a function of the relation sizes and the number of processors participating in the join operation. In this section, we develop a simulation model for the performance analysis of the parallel multi-way join query processing strategies presented in Section 3.

4.1 Simulation Model

In our model, the parallel execution of the multi-way join queries is simulated on a Sun SPARCstation 1+ to obtain the size information of the work load, disk accesses, and data communication at each PN. The cost functions presented in the next subsection are then used to calculate the response times of the queries according to the simulation results.

We assume that each operand relation of the multi-way join is initially partitioned horizontally, and distributed evenly across all PNs. During the split phase of the join algorithms, a relation, say R, is hashed into b buckets, B_1, B_2,...,B_b where b

is several times that of the number of PNs. Assuming that the relations are not uniformly distributed, the sizes of these buckets are determined by the Zipf-like distribution (Zipf, 1949; Sacco, 1986; Turbyfill, 1987) as follows:

$$|B_i| = \frac{|R|}{i^{Z_b} \sum_{j=1}^{b} \frac{1}{Z_b}}$$

In this article, bucket skew is Z_b . When $Z_b = 1$, the equation becomes a Zipf distribution, and when $Z_b = 0$, it is a uniform distribution. Similarly, the sizes of the base relations are also assumed to follow the Zipf-like distribution.

Also note that our simulator is a bucket-level simulator. That is, we do not actually compare the tuples of the operand relations. Instead, the time for joining a pair of matching join buckets is estimated using a cost function. We assume that this cost is proportional to the sum of the sizes of the two buckets involved. This cost function is based on the assumption that joining two hash buckets is done in two sequential steps (DeWitt et al., 1984):

Step 1: One bucket is used to build a hash table in the memory.

Step 2: The other bucket then is scanned and its tuples are used to probe the in-memory hash table.

Because the building of an in-memory hash table and the succeeding probing process are typically I/O bound for today's processor technology, we assume that the join cost is proportional to the size of the sum of the two buckets in terms of disk accesses. In any case, should the task time be large, the savings due to the balanced workload will be even larger. That is, our assumption is without bias in favor of load balancing schemes. We avoid using a tuple-level simulator here because it is extremely slow and does not provide us any additional information.

The following parameters are designed for cost evaluation. They are similar to those used by Hua and Lee (1991) and Lakshmi and Yu (1988).

- *Workload Parameters:*

  N_r : Total number of relations to be joined. In other words, it is an N_r-way join.

  $|R|$: Total number of tuples in all the relations of a multi-way join.

  J_s : Join selectivity factor.

  Z_b : The degree of bucket skew according to Zipf-like function.

  Z_r : The degree of variation in relation sizes according to Zipf-like function. That is, the relation sizes of a multi-way join follow a Zipf-like distribution.

t : Size in bytes of each tuple.

- *System Parameters*

  N : Number of PNs in the system.

  μ : CPU processing rate in million-instructions-per-second (MIPS).

  ω_{io} : I/O bandwidth in Mbytes/sec between a PN and its secondary storage.

  ω_{comm} : Effective communication channel bandwidth in Mbytes/sec per PN.

  I_{cpu} : CPU pathlength for processing a tuple in any step of a join operation.

- *Measurement Parameters Determined by Simulation.* We have explained that we can treat joins of the same execution iteration of a multi-way join as a single larger join operation when applying the GRACE or ABJ$^+$ algorithm. In discussing cost functions, we will refer to the set of tuples associated with the relations relevant to a particular execution iteration and reside at the same PN as a *partition.*

  $|P_{max\_init}|$: The largest partition before data redistribution.

  $|P_{max\_io}|$: Number of tuples being loaded and stored by the busiest PN (in terms of disk accesses) during a particular execution iteration of a join strategy.

  $|P_{max\_comm}|$: Number of tuples being transmitted and received by the busiest PN (in terms of communication) during a particular execution iteration of a multi-way join operation.

  $|P_{max\_final}|$: The largest partition after data redistribution.

  $|P_{max\_result}|$: Size in tuples of the largest partition after a particular execution iteration of a multi-way join operation.

- *Computed Parameters Determined by Cost Functions:*

  T_{split} : Time cost in seconds due to a split phase.

  T_{parti} : Time cost in seconds due to a partition tuning phase.

  T_{bucket} : Time cost in seconds due to a bucket tuning phase.

  T_{join} : Time cost in seconds due to a join phase.

  T_{split_io} : Time cost in seconds for disk accesses during a split phase.

  T_{split_cpu} : Time cost in seconds for processing tuples during a split phase.

  T_{parti_io} : Time cost in seconds for disk accesses during a partition tuning phase.

T_{parti_comm} : Time cost in seconds for transferring data among PNs during a partition tuning phase.

T_{join_cpu} : Time cost in seconds for building hash table and probing the hash table during the join phase of GRACE or ABJ$^+$ algorithm.

T_{join_io} : Time cost in seconds due to disk accesses during a join phase of GRACE or ABJ$^+$ algorithm.

T_{GRACE} : Time cost for performing one iteration of the parallel multi-way join operation using GRACE algorithm (Kitsuregawa et al., 1983).

T_{ABJ+} : Time cost for performing one iteration of the parallel multi-way join operation using ABJ$^+$ algorithm.

T_{L_LB} : Time cost for the execution plan generated by L_LB.

T_{B_NLB} : Time cost for the execution plan generated by B_NLB.

T_{NLBO} : Time cost for the execution plan generated by NLBO.

T_{LBO} : Time cost for the execution plan generated by LBO.

4.2 Cost Functions

In this subsection, cost functions are presented for the multi-way join query processing strategies based on the SN architecture and the described workload. Since partial overlap between the phases of the join algorithms is possible, the total join cost T_{total} is bounded by:

$$\max(T_{phase\_1}, T_{phase\_2}, \cdots, T_{phase\_m}) \leq$$
$$T_{total} \leq T_{phase\_1} + T_{phase\_2} + \cdots + T_{phase\_m}$$

where max represents the maximum function. Similarly, each phase consists of several steps (disk accesses, tuple processing, and communication). Overlap of those steps also is achievable. A performance upper bound and lower bound for the phases can be derived accordingly. In our study, we made the following assumptions:

- The overlap within each phase is perfect. The system is assumed to include a separate I/O processor and a separate communication processor which allow the overlap among disk I/O, CPU computation, and data communication (Hua and Young, 1990).

- The overlap between two phases is not allowed, i.e., a simple barrier-type synchronization (Jordan, 1978) is used between the join phases to guarantee the correct parallel execution.

Therefore the total join time can be computed as:

$$T_{total} = T_{phase\_1} + T_{phase\_2} + \cdots + T_{phase\_m}$$

Similarly, we assume that the overlap between any two execution iterations during the execution of a multi-way join is not allowed. Thus, the cost functions for the query processing strategies presented in Section 3 can be computed as follows:

$$T_{L\_LB} = \sum_{i=1}^{n} T_{ABJ+}(i) \quad T_{B\_NLB} = \sum_{i=1}^{n} T_{GRACE}(i)$$
$$T_{NLBO} = \sum_{i=1}^{n} T_{ABJ+}(i) \quad T_{LBO} = \sum_{i=1}^{n} T_{ABJ+}(i)$$

where n's are the numbers of execution iterations for the respective query processing schemes. The derivation of T_{GRACE} and T_{ABJ+} is presented in the following subsection. The cost equations for algorithms L_LB, NLBO, and LBO are the same, but the total execution times may differ because they generally generate different query execution trees.

4.2.1 Time Cost of One Execution Iteration Using GRACE. In this subsection, we derive the cost function for one execution iteration of a parallel multi-way join using the GRACE algorithm. As we have described briefly in Section 3.2, this algorithm has two distinct phases: split phase and join phase. Its cost function, therefore, can be written as follows:

$$T_{GRACE} = T_{split} + T_{join}$$

T_{split} and T_{join} are computed below:

$$T_{split} = \max(T_{split\_io}, T_{split\_cpu}, T_{split\_comm})$$
$$T_{split\_io} = |P_{max\_io}| \cdot \frac{t}{\omega_{io}} \qquad T_{split\_cpu} = |P_{max\_init}| \cdot \frac{I_{cpu}}{\mu}$$
$$T_{split\_comm} = |P_{max\_comm}| \cdot \frac{t}{\omega_{comm}}$$

In the above equations $|P_{max\_io}|$ is the measurement of the number of tuples being loaded from disk and stored to disk by the busiest PN (in terms of disk accesses) during the split phase of the algorithm. This parameter is determined by the simulator (i.e., a monitor was included in our simulator to measure $|P_{max\_io}|$). To compute the I/O cost due to split phase (i.e., T_{split_io}), we convert $|P_{max\_io}|$ into bytes, and divide it by the I/O bandwidth. T_{split_cpu} is derived similarly. T_{split_comm} is obtained in the same way, in which $|P_{max\_comm}|$ is the total number of tuples being transmitted and received by the busiest PN in terms of data communication during split phase of the algorithm. Likewise, $|P_{max\_comm}|$ is determined by the simulator. Thus, the effect of skew on T_{split_io} and T_{split_comm} depends on the initial distribution of the tuples among the PNs.

During the join phase, the busiest PN must process $|P_{max\_final}|$ tuples (i.e., building in-memory hash tables and probing the hash structures). We assume that this PN, which processes the largest number of tuples, also will generate the largest intermediate result of $|P_{max\_result}|$ tuples. Thus, the I/O cost and the computation cost of the join phase are determined by this bottleneck PN, and can be computed as follows:

$$T_{join} = \max(T_{join\_io}, T_{join\_cpu})$$
$$T_{join\_io} = \frac{|P_{max\_final}|t}{\omega_{io}} + \frac{|P_{max\_result}|t}{\omega_{io}} \qquad T_{join\_cpu} = \frac{|P_{max\_final}| \cdot I_{cpu}}{\mu}$$

4.2.2 Time Cost of One Execution Iteration Using ABJ$^+$. As shown in Section 2, ABJ$^+$ consists of four phases. The time cost for computing one execution iteration of the multi-way join operation using ABJ$^+$ (i.e., T_{ABJ+}) can be derived as follows:

$$T_{ABJ+} = T_{split} + T_{parti} + T_{bucket} + T_{join}$$
$$T_{split} = \max(T_{split\_io}, T_{split\_cpu}, T_{split\_comm}) \qquad T_{split\_io} = 2 \cdot |P_{max\_init}| \cdot \frac{t}{\omega_{io}}$$
$$T_{split\_cpu} = |P_{max\_init}| \cdot \frac{I_{cpu}}{\mu} \qquad T_{split\_comm} = 0$$

Because PNs do not exchange data during the split phase, the partition sizes remain unchanged at each PN after hashing. The "largest" PN, therefore, has to read and write $|P_{max\_init}|$ tuples. The number 2 in the expression for T_{split_io} indicates that each tuple must be loaded from disk for hashing, and then stored back to the appropriate subbucket on the same disk system.

Because the bucket tuning phase manipulates only the directory information, it does not involve tuple processing and its time cost is negligible compared to those of the other phases. Therefore we make the following approximation: $T_{bucket} \approx 0$.

At the beginning of the partition tuning phase, the coordinating PN consults the directory and allocates the buckets to the PNs. Again, since this process does not involve tuple manipulation, its time cost is negligible (i.e., the CPU time is negligible and only the disk I/O time and communication time are shown in T_{parti}.

$$T_{parti} = \max(T_{parti\_io}, T_{parti\_comm})$$
$$T_{parti\_io} = |P_{max\_io}| \cdot \frac{t}{\omega_{io}} \qquad T_{parti\_comm} = |P_{max\_comm}| \cdot \frac{t}{\omega_{comm}}$$

The cost function for the join phase is similar to that derived for the GRACE algorithm, and is given below:

$$T_{join} = \max(T_{join\_io}, T_{join\_cpu})$$
$$T_{join\_io} = \frac{|P_{max\_final}|t}{\omega_{io}} + \frac{|P_{max\_result}|t}{\omega_{io}} \qquad T_{join\_cpu} = \frac{|P_{max\_final}| \cdot I_{cpu}}{\mu}$$

4.3 Sensitivity Analysis

With our model we are able to do the performance sensitivity analyses with respect to different system and workload parameters. We have run a large number of experiments, but we are able to show only the representative and non-obvious results here. The values of the parameters used in those experiements are listed in the following:

1. Workload Parameters:
   - Total number of relations (N_r): 8.
   - Total relation size ($|R|$): 8,000,000 tuples. The size of each relation is determined by the Zipf-like function

- Tuple size (t): 200 bytes per tuple.
- Join selectivity (J_s): varied from 0.0000002 to 0.000002.
- Degree of bucket skew (Z_b): varied from 0 to 1.
- Degree of relation skew (Z_r): varied from 0 to 1.

2. System Parameters:

- Number of PNs (N): varied from 32 to 256.
- CPU processing rate (μ): 20 MIPS.
- I/O bandwidth (ω_{io}): varied from 0.8 to 4.8 Mbytes/Second per PN.
- Effective communication channel bandwidth (ω_{comm}): varied from 0.8 to 4.8 Mbytes/second per PN.
- Instruction pathlength (I_{cpu}): 1,000 instructions.

Among these parameters, we select degree of bucket skew, degree of relation skew, join selectivity, number of processors, disks, I/O bandwidth, and communication bandwidth for the sensitivity analyses.

When a parameter is not under investigation, its value is fixed as follows. The I/O bandwidth is set to 4 Mbytes/second which is typical for the industry standard SCSI bus. The communication bandwidth for each port of the communication network also is set to 4 Mbytes/second to match the data transfer rate of the disk controller. To prevent the processor from becoming a bottleneck, the processing rate of each PN is set to 20 MIPS which is derived as $\mu = I_{cpu} \cdot \frac{\omega_{io}}{t}$. In addition, the number of PNs is 256. In the following subsections, we present the results of the sensitivity analyses. In our study, we used 8-way join queries with the following characteristic. Their join graphs form a chain (i.e., each relation can be joined with exactly two "neighboring" relations, except that the two relations at the two ends can only be joined with a single "neighboring" relation).

4.3.1 Effect of Bucket Skew. The 8-way join queries selected for the study of bucket skew effects can be grouped into three types (Figure 3) which are optimized using B_NLB (i.e., skew is not considered). We studied a much larger number of queries, but we present only representative cases to illustrate the effect various system and workload parameters have on the behavior of the query processing strategies.

The effect of bucket skew on the query processing strategies is depicted in Figure 4. We explain the behavior of the performance curves as follows:

- *Figure 4*(a): Because B_NLB generates a right-deep query tree in this case and it does not perform dynamic load balancing, its performance is worse than L_LB. Because NLBO exploits inter-join parallelism in addition to performing load balancing at run time, we would expect it to outperform L_LB. However, both NLBO and L_LB generate the same right-deep tree for this particular

query, and they perform identically in this study. LBO is the best performer for a wide range of bucket skews. When $Z_b = 0$, LBO generates the same right-deep tree as those generated by L_LB and NLBO. Therefore, all three approaches have the same performance. However, as the degree of bucket skew increases beyond .35, LBO trades intra-join parallelism for inter-join parallelism in an attempt to reduce skew (i.e., LBO generates bushier trees as the bucket skew increases. As a result, it exhibits a better performance than any of the other approaches). For very mild skew, none of the load balancing strategies perform as well as B_NLB due to the load balancing overhead. Nevertheless, the slight degradation in performance due to load balancing when skew is mild is within the acceptable level for most parallel database environments; whereas, the serious degradation in performance when skew is severe and load balancing is *not* performed is likely to be intolerable for most applications.

- *Figure 4*(b): In this case, we observe a distinct performance curve for each of the processing strategies. Again, we see that B_NLB is the worst performer for $Z_b > 0.2$. Unlike Figure 4(a), NLBO performs better than L_LB because it generates a bushy tree for this particular query. Unintentionally, the inter-join parallelism helps to reduce some degree of skew effect on NLBO. Nevertheless, since LBO is able to generate a bushier tree as the degree of bucket skew increases, it is the overall winner in handling the skewed tuple distribution problem.

- *Figure 4*(c): The performance curves behave similarly in this case, except that NLBO and LBO perform identically. This is due to the fact that NLBO generates a very bushy tree for these particular queries, and there is no more inter-join parallelism that can be exploited by LBO.

In the following subsections, we will not show the query execution trees. However, we will refer to them as right-deep trees, bushier trees, and bushiest trees, etc., when we explain the behavior of the corresponding performance curves.

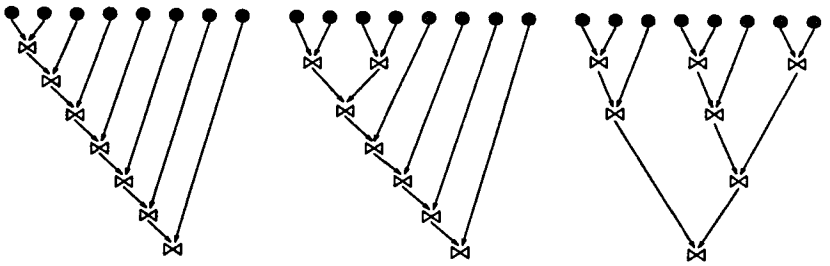
4.3.2 Effect of Relation Skew. As we mentioned, although the total number of base tuples involved in the multi-way join is fixed at 8,000,000, the size of each of the eight operand relations is randomly assigned one of the eight numbers computed by the Zipf-like function as follows:

$$|R_i| = \frac{|R|}{i^{Z_r} \sum_{j=1}^{N_r} \frac{1}{j^{Z_r}}}$$

In this article, relation skew is Z_r.

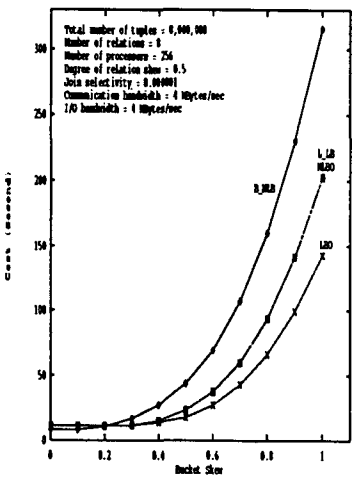
The results of the study on relation skew are shown in Figure 5. When the relation skew is zero, all the operand relations have the same size, and each has

Figure 3. Structures of query execution trees generated by B_NLB

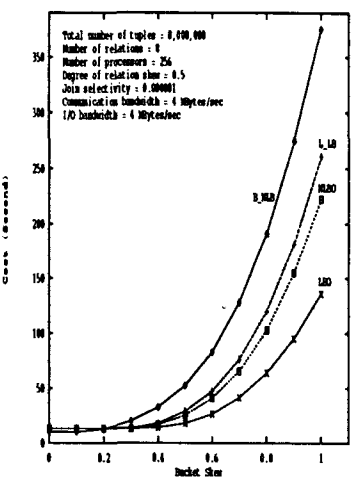


(a) *Type 1*: Right-deep tree (b) *Type 2*: Bushy tree (c) *Type 3*: Bushier tree

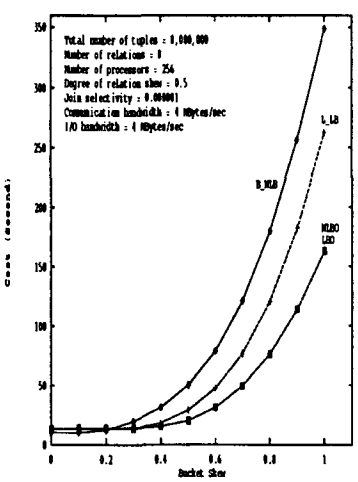
Figure 4. Effect of bucket skew



(a) Type 1 queries

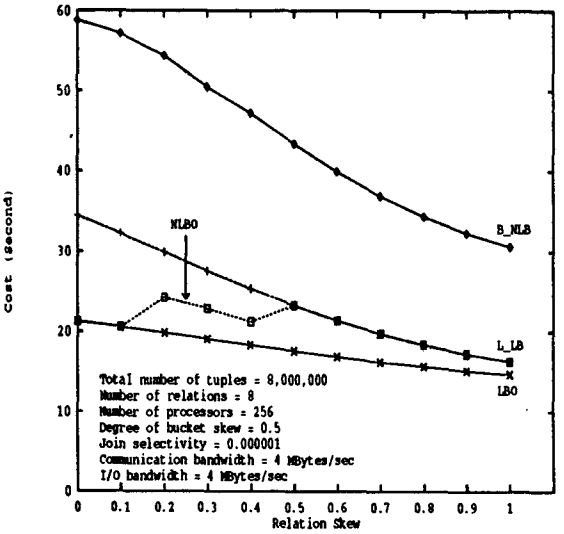


(b) Type 2 queries



(c) Type 3 queries

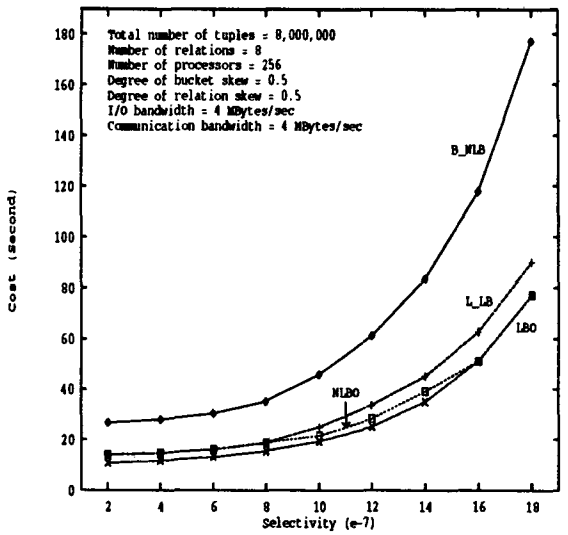
Figure 5. Effect of relation skew



1,000,000 tuples. Because the join selectivity factor was fixed at 0.000001, the size of the join results of any relation pair remains at 1,000,000 tuples for any execution iteration. Under this condition, any parallel execution plan will result in the same response time assuming bucket skew is zero. Since B_NLB employs a greedy algorithm to gradually increase the number of concurrent joins for each execution step (as long as the increase in inter-join parallelism does not worsen performance), the query tree generated by B_NLB is a bushiest tree. For this reason, NLBO performs as well as the plan generated by LBO when $Z_r = 0$. It does not perform as well for L_LB as the other load balancing schemes because inter-join parallelism is not exploited. As we increase the relation skew, the deviation in the sizes of the relations increases. As a result, the query generated by B_NLB becomes less bushy as the relation skew increases. A less bushy tree results in poorer performance under the bucket skew effect (i.e., $Z_b = 0.5$). Consequently, the performance curves of NLBO rise for $0.1 \leq Z_r \leq 0.2$. However, they fall for $0.2 < Z_r \leq 0.3$ since the relation skew becomes severe and the rate of data reduction becomes more rapid. This increase in data reduction rate also forces NLBO to generate a right-deep tree for $Z_r > 0.5$. Therefore, NLBO and L_LB perform identically for larger relation skews.

4.3.3 Effect of Join Selectivity. Join selectivity factors have a strong influence on multi-way join query optimization. In general, query execution trees tend to be less bushy for smaller join selectivity factors in order to speed up the data reduction

Figure 6. Effect of join selectivity



rate during query execution. However, if we consider the data skew issue in query optimization as in LBO, using a bushier tree (i.e., increasing the degree of inter-join parallelism) might be necessary to achieve best performance. This phenomenon can be observed in Figure 6.

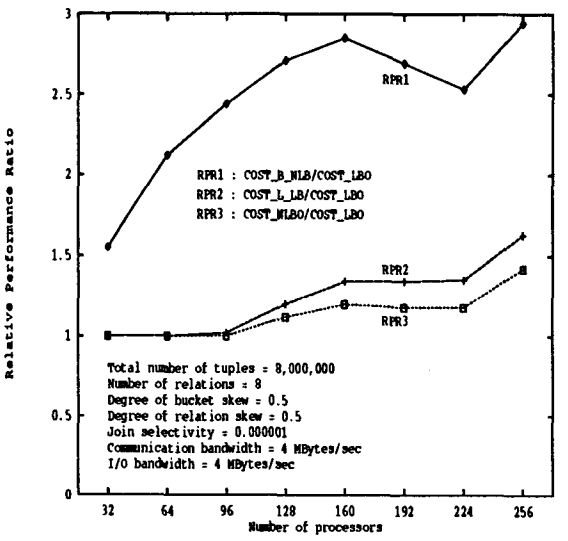
In this study, we assumed that all the join operations have the same selectivity factor. When the selectivity factor is small, NLBO generates a right-deep tree, and its performance is identical to that of L_LB. For selectivity factors greater than 8×10^{-7}, the trees generated by NLBO become bushier as the selectivity factor increases. Consequently, NLBO outperforms L_LB for larger selectivity. When this factor becomes sufficiently large, the query tree generated by NLBO becomes very bushy, and its performance matches that of LBO. Again, we observe that LBO exhibits the best performance, and B_NLB shows the worst performance for this workload because it does not perform load balancing.

4.3.4 Effect of Number of PNs. To compare the performance of LBO to the other schemes under various number of PNs, we introduce the following *relative performance ratios* (RPRs):

$$RPR1 = \frac{COST_{B\_NLB}}{COST_{LBO}} \qquad RPR2 = \frac{COST_{L\_LB}}{COST_{LBO}} \qquad RPR3 = \frac{COST_{NLBO}}{COST_{LBO}}$$

The effect of number of PNs on the multi-way join strategies is shown in Figure 7. For the same degree of bucket skew, we observe that *RPR*1 increases with the increase in the number of PNs. This again exemplifies the importance of load

Figure 7. Effect of number of PNs

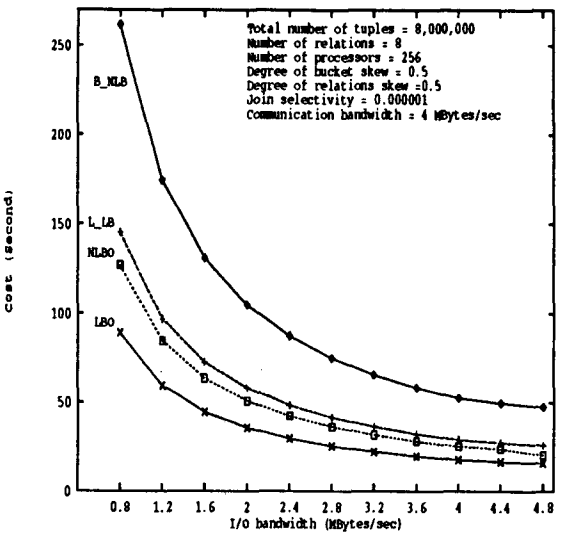


balancing in larger shared-nothing systems. The curve for *RPR*1, however, does not increase monotonously. At $N = 160$, the curve falls because LBO generates a bushier tree that has little inter-join parallelism to exploit after the second execution iteration. Thus, LBO suffers from the increase in the number of PNs. We observed that when $N > 224$, LBO generates the bushiest tree and it regains its advantage from the increase in the degree of inter-join parallelism. In general, the curve strictly increases if the number of operand relations is sufficiently large relative to the number of PNs in the system (e.g., when the number of relations is eight, *RPR*1 strictly increases for $32 < N < 160$) (Figure 7).

The benefit of LBO (compared to other load balancing schemes) increases with the increase in system size. However, it does not provide any advantage in smaller systems for this particular workload. For other workloads with smaller relations or more severe skew, we observed that LBO provided savings for smaller system configurations.

4.3.5 Effect of I/O Bandwidth. In the previous studies, we assumed that the hardware design was "balanced"—the processors, the I/O subsystems, and the communication processors were tuned for the join operations. In this and the following subsections, we are interested in a system environment that is less than ideal. Here we consider how the performance of the I/O subsystems affects the algorithms; in particular, how the overhead due to dynamic load balancing is related to the I/O bandwidth (Figure 8).

Figure 8. Effect of I/O bandwidth



LBO provides very significant savings compared to B_NLB for any I/O bandwidth. The cost of B_NLB is three times that of LBO. L_LB and NLBO also can provide very impressive savings. However, LBO remains the overall winner.

4.3.6 Effect of Communication Bandwidth. We also studied the effect of the communication bandwidth on the multi-way join algorithms. The results are plotted in Figure 9. These performance curves behave similarly to those in Figure 8. However, dynamic load balancing is more critical in a system with inadequate communication capability than in a system with limited I/O performance, because the relative performance ratio between B_NLB and LBO is higher for small communication bandwidth than for small I/O bandwidth.

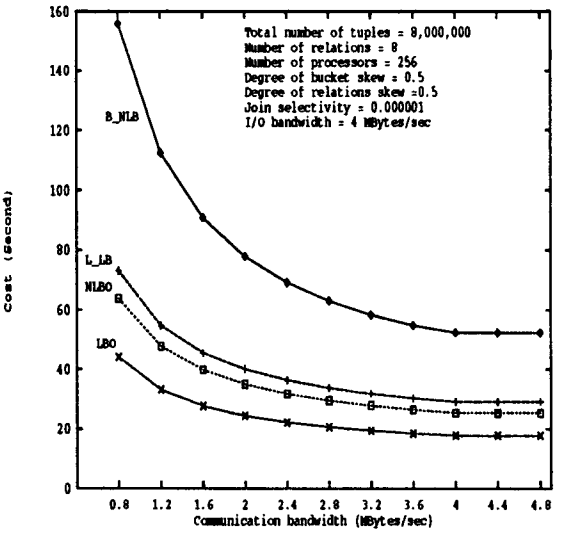
5. Conclusion

We have discussed dynamic load balancing issues in multi-way join operations. In particular, we implemented four multi-way join query optimizers, and developed a simulator to investigate the effect of skewed tuple distribution on these techniques. From our study, we can draw the following conclusions:

- Dynamic load balancing is very critical to the performance of shared-nothing systems, particularly when the system is large.

- Because load balancing becomes more difficult for larger systems, one should

Figure 9. Effect of communication bandwidth



exploit inter-join parallelism to limit the number of PNs used for each join operation within manageable size.

- Although dynamic load balancing provides very impressive savings, considering the skew issue during query optimization is necessary to achieve the best performance.

Although we limited our discussion to intra-query parallelism, the load balancing techniques proposed for multi-way join can be extended easily to support inter-query parallelism in a multiuser environment.

Finally, we note that an interesting technique was recently proposed by DeWitt et al. (1992) to estimate skew in tuple distribution. In this scheme, a sample of the relations being joined is used to estimate the skew. Therefore, an appropriate join algorithm can be determined for a particular degree of skew. This technique can be adapted to improve the hash phase of the LBO algorithm. In the modified scheme, instead of hashing all the relations and writing them back to the local disks, we can hash the relation samples to save disk I/Os. Partition tuning then can be performed based on the statistics obtained from those samples. This modification should provide some performance improvement over the original LBO algorithm. Furthermore, we are currently enhancing the LBO optimizer to allow a pair of oversized matching buckets to be joined by more than one PN (i.e., broadcast-based join.) The new optimizer is capable of trading between the degrees of multicast and inter-join parallelism in determining the best join strategy for a query.

References

Boral, H., Alexander, W., Clay, L., Copeland, G., Danforth, S., Franklin, M., Hart, B., Smith, M., and Valduriez, P. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering,* 2(1):4-24, 1990.

Chen, M.-S., Lo, M., Yu, P.S., and Young, H.C. Using segmented right-deep trees for the execution of pipelined hash joins. *Proceedings of the Eighteenth International Conference on VLDB,* Vancouver, British Columbia, 1992.

DeWitt, D.J., Ghandeharizadeh, S., Schneider, D.A., Bricker, A., Hsiao, H.-I., and Rasmussen, R. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering,* 2(1):44-62, 1990.

DeWitt, D.J., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D. Implementation techniques for main memory database systems. *Proceedings of the 1984 SIGMOD Conference,* Boston, Mass., USA, 1984.

DeWitt, D.J., Naughton, J.F., Schneider, D.A., and Seshadri, S. Practical skew handling in parallel joins. *Proceedings of the Eighteenth International Conference on VLDB,* Vancouver, British Columbia, 1992.

Englert, S., Gray, J., Kocher, T., and Shah, P. A benchmark of NonStop SQL release 2 demonstrating near-linear speedup and scaleup on large databases. *ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems,* Boulder, Colorado, 1990.

Hong, W. and Stonebraker, M. Optimization of parallel query execution plans in XPRS. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* Miami Beach, Florida, 1991.

Hua, K.A., and Lee, C. An adaptive data placement scheme for parallel database computer systems. *Proceedings of the Sixteenth International Conference on VLDB,* Brisbane, Australia, 1990.

Hua, K.A., and Lee, C. Handling data skew in multiprocessor database computers using partition tuning. *Proceedings of the International Conference on VLDB,* Barcelona, Spain, 1991.

Hua, K.A., Lo, Y.-L., and Young, H.C. Including the load balancing issue in the optimization of multi-way join queries for shared-nothing database computers. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* San Diego, California, 1993.

Hua, K.A. and Young, H.C. Designing a highly parallel database server using off-the-shelf components. *Proceedings of the International Computer Symposium,* Hsinchu, Taiwan, 1990.

Jordan, H.F. A special purpose architecture for finite element analysis. *Proceedings of the International Conference on Parallel Processing,* Bellaire, Michigan, 1978.

Kitsuregawa, M. and Ogawa, Y. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC). *Proceedings of the Sixteenth International Conference on VLDB,* Brisbane, Australia, 1990.

Kitsuregawa, M., Tanaka, H., and Moto-oka, T. Application of hash to database machine and its architecture. *New Generation Computing,* 1(1):66-74, 1983.

Lakshmi, S. and Yu, P.S. Effect of skew on join performance in parallel architectures. *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems,* Austin, Tex., USA, 1988.

Lakshmi, S. and Yu, P.S. Limiting factors of join performance on parallel processors. *Proceedings of the Fifth International Conference on Data Engineering,* ??, 1989.

Lorie, R.A., Daudenarde, J.-J., Stamos, J.W., and Young, H.C. Exploiting database parallelism in a message-passing multiprocessor. *IBM Journal of Research and Development,* 35(5/6):681-695, 1991.

Lu, H., Shan, M.-C., and Tan, K.-L. Optimization of multi-way join queries for parallel execution. *Proceedings of the Seventeenth International Conference on VLDB,* Barcelona, 1991.

Sacco, G.M. Fragmentation: A technique for efficient query processing. *ACM Transactions on Database Systems,* 11(2):113-133, 1986.

Schneider, D.A. and DeWitt, D.J. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. *Proceedings of the SIGMOD Conference,* Portland, Oregon, 1989.

Schneider, D.A. and DeWitt, D.J. Tradeoffs in processing complex join queries via hashing in multiprocessor database machine. *Proceedings of the Sixteenth VLDB Conference,* Brisbane, Australia, 1990.

Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., and Price, T.G. Access path selection in a relational database management system. *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Boston, Mass., USA, 1979.

Stonebraker, M. The case for shared nothing. *IEEE Database Engineering,* 9(1):4-9, 1986.

Stonebraker, M. The design of XPRS. *Proceedings of the Fourteenth International Conference on VLDB,* Los Angeles, Calif., USA, 1988.

Su, S.Y.W. A microcomputer network system for distributed relational databases: Design, implementation, and analysis. *Journal of Telecommunication Networks,* 2(3):307-334, 1983.

Swami, A., Young, H.C., and Gupta, A. Algorithms for handling skew in parallel task scheduling. *Journal of Parallel and Distributed Computing,* 16(4):363-377, 1992.

Tan, K.-L. and Lu, H. Processing multi-join query in parallel systems. *Proceedings of the 1992 Symposium on Applied Computing,* Kansas City, Okla., USA, 1992.

Teradata Corporation, Los Angeles, California. *Teradata DBC/1012 Data Base Computer Concepts and Facilities,* release 3.1 edition, 1988.

Turbyfill, C. Comparative Benchmark of Relational Database Systems., Ph.D. Thesis, Cornell University, 1987.

Walton, C.B., Dale, A.G., and Jenevein, R.M. A taxonomy and performance model of data skew effects in parallel joins. *Proceedings of the International Conference on VLDB,* Barcelona, Spain, 1991.

Wolf, J.L., Dias, D.M., and Yu, P.S. An efficient algorithm for parallelizing sort-merge joins in presence of data skew. *Proceedings of the Second International Symposium on Databases in Parallel and Distributed Systems,* Dublin, Ireland, 1990.

Wolf, J.L., Dias, D.M., Yu, P.S., and Turek, J. An efficient algorithm for parallelizing hash joins in presence of data skew. *Proceedings of the International Conference on Data Engineering,* Kobe, Japan, 1991*a*.

Wolf, J.L., Dias, D.M., Yu, P.S., and Turek, J. Comparative performance of parallel join algorithms. *Proceedings of the International Conference on Parallel and Distributed Information Systems,* Miami, Flor., USA, 1991*b*.

Wong, E. and Youssefi, K. Decomposition: A strategy for query processing. *ACM Transactions Database Systems,* 1(3):223-241, 1976.

Zipf, G.K. *Human Behavior and the Principle of Lease Effort: An Introduction to Human Ecology.* Reading, MA: Addison-Wesley, 1949.