Predictive Dynamic Load Balancing of Parallel Hash-Joins over Heterogeneous Processors in the Presence of Data Skew *

Hasanat M. Dewan   Mauricio Hernández†
Kui W. Mok   Salvatore J. Stolfo

Department of Computer Science, Columbia University, New York, NY 10027

CUCS-026-94

(This is an extended version of the paper that appeared in the Proceedings of the 3rd International Conference on Parallel and Distributed Information Systems.)

Abstract

In this paper, we present new algorithms to balance the computation of parallel hash joins over heterogeneous processors in the presence of data skew and external loads. Heterogeneity in our model consists of disparate computing elements, as well as general purpose computing ensembles that are subject to external loading (e.g., a LAN connected workstation cluster). Data skew manifests itself as significant non-uniformities in the distribution of attribute values of underlying relations that are involved in a join.

We develop cost models and predictive dynamic load balancing protocols to detect imbalance during the computation of a single large join. New predictive bucket scheduling algorithms are presented that smooth out the load over the entire ensemble by reallocating buckets whenever imbalance is detected. Our algorithms can account for imbalance due to data skew as well as heterogeneity in the computing environment. Significant performance gains are reported for a wide range of test cases on a prototype implementation of the system.

1 Introduction

In the last decade, many researchers have focused on developing database machine architectures for fast execution of complex select-project-join (S-P-J) queries. Many of these efforts have resulted in the development of efficient parallel join algorithms for multiprocessor environments [13, 6, 8, 7, 16, 12, 1, 11, 15, 9]. These algorithms are parallel versions of sort-merge or hash-based joins previously developed for centralized uniprocessor database machines. While there are many subtle differences among these efforts, they all assume a homogeneous ensemble of processors which do not exhibit performance variations over time. Another class of parallel join algorithms have been described in the literature to specifically address the problems introduced when data skew is present [19, 18, 12, 9]. However, in these cases as in the previous ones, the underlying processing resources are assumed to be homogeneous and time-invariant. Because of these assumptions, work to date on parallel joins have not been concerned with dynamic load balancing, since a good initial allocation of tasks to processors is assumed to suffice under those conditions. (We view a load balancing technique for parallel joins to be dynamic if it attempts to equalize processor utilization during the course of a single S-P-J query.)

In contrast, we are interested in parallel processing of joins against a database fragmented over heterogeneous processing sites (i.e., hardware differences among the sites is typical) where any site of the ensemble may deviate from its nominal rated performance for any period of time due to external loads. In this paper, we show how specialized predictive dynamic load balancing (PDLB) protocols may be utilized to balance the computation of S-P-J queries in a predictive fashion. Moreover, our techniques are applicable to databases that contain skew in the distribution of certain attribute values (e.g., join attributes). We develop our methods in the context of computing the join of two relations, R and S. Our algorithms generalize to multiway joins in a straightforward fashion.

*This work has been supported in part by the New York State Science and Technology Foundation through the Center for Advanced Technology under contract NYSSTFCU01207901, and in part by NSF CISE grant CDA-90-24735.

†This author's work has been supported by an AT&T Cooperative Research Program Fellowship.

We provide a means of balanced parallel processing of general S-P-J queries as well as matching of the LHS of rules in a rule program (the matching of the LHS of rules in a knowledge-base system compile to complex S-P-J queries over a database of facts). Thus, our methods are applicable to the case of general rule processing in expert and active databases, as well as in distributed database transaction processing systems. We develop algorithms to dynamically estimate the cost coefficients for critical phases of a parallel hash join process (e.g., *input* or *inter-site transfer* costs, *join* costs, and *output* costs). The approach we propose is based on a cost model for joining tuples from corresponding hash buckets of two relations, and run-time sampling of performance for the various phases of computing the join of two buckets. We show how the cost coefficients can be used to dynamically reschedule buckets over the sites to achieve predictively balanced parallel join computation over heterogeneous resources, and provide performance data to validate our claims.

2 Previous Work

The major points of difference between our approach and previous work in parallel join algorithms is summarized in table 1.

Table 1 shows the names of researchers and the relevant citations under the "Prior Work" column. The classification of parallel join techniques is based on whether or not the corresponding work supports heterogeneous computing environments ("Hetero."), provides facilities for balanced operation in the presence of temporal variations (e.g., external loads) during the lifetime of a join computation by dynamic load balancing ("Temporal Var/DLB"), handles data skew ("Skew"), utilizes performance feedback from the computing environment to make load balancing decisions dynamically (i.e., during the course of a single complex S-P-J query) and predictively ("Env. Monitoring"), and whether it utilizes statistical meta-data on the database for task allocation among the sites ("Meta Data").

The main contribution of this paper is a set of algorithms and protocols that dynamically and predictively balance the load of a parallel join computation over a set of "unequal" or heterogeneous processing elements. Previous work has not addressed this important dimension, nor the dynamic load balancing issues for very large joins in a predictive fashion.

3 A Parallel Hash Join Algorithm for Fragmented Databases

We describe a general algorithm that assumes no knowledge of the join attribute for joining two relations, say R and S. This is the case when the data placement or initial partitioning of the data over the distributed sites is essentially an off-line activity, and may be conducted independently by a distribution phase that is not cognizant of the particular S-P-J queries that will be processed. The algorithm consists of the following three phases.

1. Data Placement (off-line).
2. Bucket Formation and Batch Size Determination (online or off-line).
3. Parallel Join (PJ) Protocol (online).

The first two phases are obvious. The novel features of the algorithm are contained in the PJ protocol, which will be presented in the next section. The data placement phase consists of distributing the tuples from each relation, R and S, to the P processors of the ensemble in a round-robin fashion. This results in equal numbers of tuples from each relation being allocated to every site.

3.1 Bucket Formation and Batch Size Determination

The second phase, bucket formation, can be either off-line or on-line. An off-line method would be chosen where the same join would be computed many times (as in rule programs which compute rule instances in cycles). In that situation, it is usually more efficient to compute the buckets once and maintain them incrementally as the base relations are modified. We will not discuss the incremental maintenance of buckets and base relations any further. Another option is to compute buckets as an on-line process, in a demand-driven fashion. This approach is more appropriate for database transaction processing systems. The issue of deciding how many buckets to form for a given relation depends on several considerations that we mention below.

Buckets are formed so that the search space for joining a tuple of R is reduced from the entire relation S to the corresponding bucket of S, assuming the same hash function is applied to the join attribute common to both relations. This idea is common to all hash join algorithms. Once buckets are formed, the full join can be computed by joining only the corresponding buckets (i.e., those with the same hash number). If

Prior Work	Hetero.	Temporal Var/DLB	Skew	Env. Monitoring	Meta Data
Kitsuregawa [13, 12]	No	No	No	No	No
Schneider/DeWitt [15, 8]	No	No	Yes	No	No
Wolf/Yu/Turek/Dias [19, 18]	No	No	Yes	No	No
DeWitt and Naughton [9]	No	No	Yes	No	Yes
Present Work [2]	Yes	Yes	Yes	Yes	Yes

Table 1: Comparison of Parallel Join Methods

buckets are small relative to the entire relation, significant savings in computation can result. The joining of corresponding buckets may be carried out either by a nested loop algorithm, or a hash-probe method. In the latter, one of the buckets of R is used to construct a hash table based on the join attribute. The entries of the hash table are then probed by hashing the tuples of the corresponding bucket of S on the join attribute. If the probing results in collisions, the join attribute is explicitly compared for equality and the joined tuple is output if a match is found. We assume a simple hash function exists for the creation of buckets as follows:

$$h(X) = X \bmod N_{bkt}$$

Here, X is drawn from the domain of the join attribute, and N_{bkt} is the number of buckets to be formed. The critical question here is how to determine N_{bkt}. If N_{bkt} is chosen too small, the buckets can be of relatively large size and may not fit into the available physical memory, while too large a value of N_{bkt} may cause some buckets to be empty when the domain size of the join attribute for the smaller relation is small relative to N_{bkt}.

It is possible to chose N_{bkt} in a variety of ways. Here we describe a method for determining a value for N_{bkt} by working backwards and by making some assumptions on the distribution of values for the join attribute in relations R and S. We assume (i) the join attribute is nearly uniformly distributed, and (ii) the domain size of the join attribute for the smaller of the two relations is comparable to the total number of tuples. When these assumptions are not valid, the method we describe is still applicable. We mention that buckets containing skew elements must be detected and handled separately. We will elaborate on this point later.

Before proceeding any further, we introduce a parameter called the "maximum batch size", B. The purpose of this parameter is to provide an upper bound on the number of bucket pairs that may be loaded into physical memory from disk at any one time. The join is computed in a batch-oriented fashion, meaning all the buckets in some batch are processed before the next batch is read into memory. Our load balancing technique attempts to balance the join on a per-batch basis. During actual operation, another parameter, b, is set to a value anywhere between 1 and B. Pairs of buckets are brought into physical memory in batches of size b. Adjusting the value of b allows us to control the frequency of bulk I/O operations at each site in reading tuples from disk or over the network, as well as a means for controlling how available memory is utilized. (The effect of available memory on performance is discussed in Section 8.1.5).

The starting point for computing N_{bkt} is to fix a value for the desired number of tuples, T_{bkt}, in each bucket of the larger relation[1]. This depends on the granularity of buckets we desire, and may depend on the exact join algorithm being used. For instance, desired bucket size would typically be smaller for a nested loop join than for a hash-probe method, since the former has higher computational complexity. Let:

P = number of sites
M = minimum available physical memory over all sites (in bytes)
S_{tuple} = max (size of a tuple of R in bytes, size of a tuple of S in bytes)
T_{bkt} = desired number of tuples in any bucket of R or S
S_{bkt} = expected size of a bucket (in bytes)
S_{frag} = largest local fragment of R or S among all sites (in bytes)
B = maximum possible batch size

Obviously, $S_{bkt} = T_{bkt} \times S_{tuple}$. A first approximation to N_{bkt} is given by the formula shown below.

$$\begin{aligned} N_{bkt} &\approx \frac{MAX(|R|, |S|) \times S_{tuple}}{S_{bkt}} \\ &\approx \frac{P \times S_{frag}}{S_{bkt}} = \frac{P \times S_{frag}}{T_{bkt} \times S_{tuple}} \end{aligned}$$

[1]For this initial implementation, we determined *ad hoc* the value of T_{bkt} to be 1024. More detailed studies are needed to discover the effect this parameter has in the performance of the system.

If this value is non-prime, our method picks the smallest prime number larger than this value, since prime numbers have superior randomizing properties for the purposes of hashing. Once N_{bkt} has been determined, we can determine B, the maximum batch size:

$$B = \left\lfloor \frac{M}{2 \times S_{bkt}} \right\rfloor$$

The factor of 2 in the denominator arises from the fact that a *pair* of buckets, one from each relation R and S, must be loaded into memory at the same time. We assume at least one pair of buckets, one from each relation R and S, fit completely into memory at any site.

4 Parallel Join (PJ) Protocol

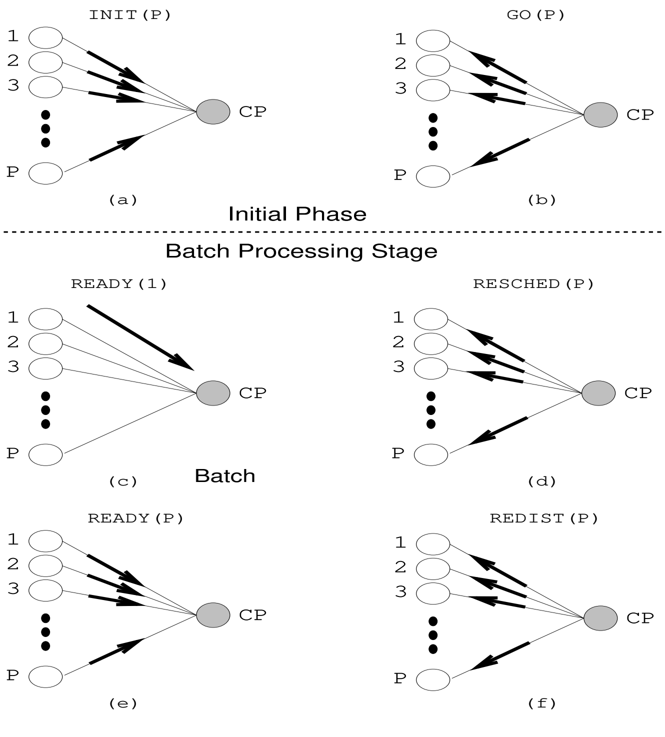


Figure 1: Parallel Join (PJ) Protocol

We assume a shared-nothing hardware architecture consisting of a network of $P+1$ sites, each having its own memory, CPU and disk. The sites are interconnected via a high-speed LAN, and point-to-point as well as broadcast messages are supported. The relations to be joined, R and S, are fragmented in the data placement phase in round-robin fashion over all available disks, where they reside.

From this point on, the PJ protocol controls the processing of a join. We discuss the PJ protocol for an architecture consisting of P processing sites and a single coordinator, referred to as the CP (hence, $P+1$ sites are involved). Variations of the protocol may use more than one coordinator, thus distributing the coordination task [5]. However, in this paper, we discuss the case where there is a single CP. The CP only listens for messages from other sites, and sends system reorganization directives to other sites in the ensemble, but does not participate in the join processing itself. The PJ protocol, shown in Figure 1, consists of an "initial" phase, and a "batch processing" phase. The initial phase is responsible for determining the number of buckets, N_{bkt}, and the maximum batch size, B, using information about the available memory and the size of relation fragments at each site. To achieve this, the protocol carries out the message exchanges between the sites and the CP as shown in steps (a) and (b) of Figure 1. In step (a), all P sites send the local size of relation fragments and physical memory to the CP. The CP uses this information, together with user-specified parameters such as the desired number of tuples, T_{bkt}, in any bucket of R or S, to compute values for N_{bkt} and B. The CP then broadcasts these values to all the sites. Each site generates buckets for each relation fragment and stores these in local bucket files, which may or may not be retained for subsequent joins involving the same two relations. This concludes the initial phase of the PJ protocol.

The bulk of the work is done in the batch processing phase, where batches of buckets are processed until the join is fully computed. This phase begins immediately after the initial phase has been completed and the local bucket files have been formed. Each site reads some number $1 \le b \le B$ buckets from disk, where b is the tuning parameter mentioned earlier. Since a particular bucket may be distributed over all P sites, this may involve each site reading from all the disks. Bucket fragments residing locally are not transferred over the network. A simple *initial* bucket allocation method is to assign bucket numbers $1 \cdots b$ to site 1, $b+1 \cdots 2b$ to site 2, etc. Once a complete set of b buckets are in local memory, each site begins joining corresponding R and S buckets that have been allocated to it. We denote the i-th buckets of R and S by b_R^i and b_S^i, their tuple cardinalities by $\mid b_R^i \mid$ and $\mid b_S^i \mid$, and their sizes (in bytes) by $size(b_R^i)$ and $size(b_S^i)$, respectively.

During operation, disparities in runtimes may arise due to external loads placed on various sites as well as inherent differences in processor speeds and I/O rates among sites. We model these disparities by a 3-tuple $< C_{transfer}, C_{join}, C_{output} >$ for each site (called the "performance vector" henceforth) that is germane to the join computation.

$C_{transfer}$ is the average cost for reading the blocks of buckets into memory, expressed as seconds per block. Blocks are of a standard size of $BLKSIZE$ bytes, where $BLKSIZE$ is a system parameter. A typical value of $BLKSIZE$ is 4096 bytes. This coefficient is initialized to a default value determined by actual measurements of the average inter-site transfer cost for data blocks of size $BLKSIZE$, under normal operating conditions. The value of $C_{transfer}$ is subsequently recomputed dynamically at runtime by measuring actual transfer times as processing progresses and the system undergoes dynamic load variations.

C_{join} is the average cost of comparing a pair of tuples from the two relations, expressed as seconds per tuple per tuple ($\frac{seconds}{tuple \cdot tuple}$). We assume a nested loop join performed in local memory for each pair of corresponding buckets. Hence, the coefficient C_{join} is determined as follows. The total time t_i to compare every pair of tuples in buckets b_R^i, b_S^i is determined. The average time for comparing pairs of tuples when joining the i-th buckets is then $\hat{t_i} = \frac{t_i}{|b_R^i||b_S^i|}$. The value of C_{join} is obtained by taking the average of the $\hat{t_i}$ for the buckets in the current batch processed so far. (If a hash-probe join method is used, the computation of C_{join} must be adjusted appropriately.)

C_{output} is the average cost of writing result tuples back to disk expressed in seconds per block, where each block is of a standard size, $BLKSIZE$. The value of C_{output} is dynamically computed at runtime by measuring actual block output times as processing progresses and the system undergoes dynamic load variations.

Because of disparities among the sites, one site will finish processing the batch of b buckets allocated to it before the other sites. This site immediately informs the CP that it is ready for the next batch by sending a READY(1) message to the CP. This is shown in step (c) of Figure 1. The 1 in parenthesis indicates that this message is received by the CP from only 1 (the fastest) site.

Upon receiving the READY(1) message, the CP broadcasts a RESCHED message, directing all sites to suspend their processing as soon as possible (i.e., immediately after finishing any buckets they are currently working on, but generally prior to completing all b local buckets), and participate in a rescheduling phase. This is depicted in step (d) of Figure 1. The goal of the rescheduling is to reallocate the unprocessed (excess) buckets from the current batch at each site over all the sites in the ensemble so as to minimize the overall completion time for the current batch. The exact method for identifying the excess buckets will be given in section 6.2. In response to the RESCHED message, all P sites send a READY message along with performance data, including 3 coefficients, namely $< C_{transfer}, C_{join}, C_{output} >$. This is shown in step (e) of Figure 1. These values are computed by using local statistics on how much time was spent transferring the buckets from remote sites into local memory (transfer cost), how much time was spent performing the operations for joining corresponding buckets (join cost), and the time spent in writing the result tuples back to disk (output cost), respectively. The PJ protocol we describe does not use the statistics for disk-to-memory transfers in the computation of transfer costs. The algorithms we describe initially load the local memory of each site in the distributed ensemble with the same number of b (the batch size) bucket pairs by collecting fragments of the specific b buckets assigned to the site from remote disks over the network. Runtime imbalances are handled by *reallocation* of excess buckets that cause some sites to become bottlenecks. Thus, the performance statistic that is relevant is the block transfer time between sites, rather than between disk and physical memory.

The CP now has all the information it requires to decide how to reallocate the excess buckets among all the sites and minimize the overall completion time. The PJ protocol accomplishes this with a combination of cost modelling and a modified version of the LPT (longest processing time first) [10] heuristic algorithm reported extensively in the literature. Our version is called WLPT (Weighted LPT). The details are given in the next section. To end the discussion of the PJ protocol, we mention that once the CP has finished computing how the excess buckets should be reallocated, a "rescheduling matrix" is filled in with this information and broadcast to all sites (only one row of this matrix is needed at each site). This matrix has a sparse representation and its size is bounded by a small constant multiple of B. In the current implementation, only 6 bytes of data is needed to encode each row of the matrix. The rescheduling phase is initiated with a REDIST(P) message followed by the rescheduling matrix, and is depicted in step (f) of Figure 1. The sites receive this information and exchange excess buckets as directed, and then resume the sequence of steps beginning at step (c). This process is repeated until the current batch is fully processed.

Upon completion of the current batch, the protocol makes provisions for starting the next batch, until the full join is computed.

5 Cost Models

Here, we derive the cost models to be used in the next section for making dynamic load balancing decisions. Assuming that the i-th bucket pair must first be transferred into the local memory at site s from a remote site, the basic cost formula to join the i-th bucket of R with the corresponding bucket of S ($b^i_R \bowtie b^i_S$) at a given site s is defined as follows.

$$\begin{aligned} COST^i_{basic} &= C^s_{transfer} \times \frac{(size(b^i_R) + size(b^i_S))}{BLKSIZE} \\ &+ C^s_{join} \times (\mid b^i_R \mid \times \mid b^i_S \mid) \\ &+ C^s_{output} \times \left(\left(\frac{\mid b^i_R \mid \times \mid b^i_S \mid}{\mathcal{D}^i} \right) \times \frac{2 \times S_{tuple}}{BLKSIZE} \right) \end{aligned}$$

$\mathcal{D}^i$ denotes the average domain size of the smaller of the domains of b^i_R and b^i_S, estimated by dividing the domain of the corresponding relation by N_{bkt}, the total number of buckets.

The first two terms in the above formula are straightforward. They estimate the time to move the i-th bucket pair into local memory, and the time to compute the join. The third term requires some explanation. Suppose $\mathcal{D}^i_R$ and $\mathcal{D}^i_S$ are the domain sizes for buckets b^i_R, b^i_S respectively. If the join attribute is uniformly distributed for both R and S, then the ratios $\frac{|b^i_R|}{\mathcal{D}^i_R}$ and $\frac{|b^i_S|}{\mathcal{D}^i_S}$ respectively represent the average selection cardinality of join attributes which hash into buckets b^i_R or b^i_S. The product of these two ratios is a measure of the "join output multiplicity", μ, for bucket i. The join output multiplicity is a simple measure of how the join output size deviates from the case when there are only single occurrences of join values in a bucket. Assuming the bucket sizes are approximately equal, the join of the i-th bucket pair is expected to produce $MIN(\mathcal{D}^i_R, \mathcal{D}^i_S)$ result tuples in the absence of multiple occurrences or missing values. However, multiple values and missing values are common (often resulting in mild skew), hence a better approximation is given by:

$$\begin{aligned} \mid b^i_R \bowtie b^i_S \mid_{estimated} &= \frac{\mid b^i_R \mid}{\mathcal{D}^i_R} \times \frac{\mid b^i_S \mid}{\mathcal{D}^i_R} \times MIN(\mathcal{D}^i_R, \mathcal{D}^i_S) \\ &\approx \left(\frac{1}{\mathcal{D}^i} \right) \times (\mid b^i_R \mid \times \mid b^i_S \mid) \end{aligned}$$

The above result is an approximation under the assumption that the domain sizes of the i-th buckets of R, S are approximately equal, and is equal to $\mathcal{D}^i$. Thus, the ratio $\frac{1}{\mathcal{D}^i}$ may be regarded as an estimate of the join selectivity between the i-th buckets of R and S. This agrees with other estimates of join selectivity [17] between two relations, which is often defined to be the inverse of the larger of the domain sizes of the two relations involved in a join. Thus, the third term is an estimate of output cost of the result tuples when the data from both relations exhibit no skew or moderate skew. When the degree of data skew is very high, we must use a more extensive cost model, which is outlined in section 7.

6 Predictive Dynamic Load Balancing by Weighted LPT (WLPT)

The *interrupt point* is defined to be the point in the PJ protocol when the fastest site informs the CP that it has finished processing its current batch. At this point, the CP utilizes performance data received from all sites to compute a reallocation plan for dynamic load balancing. This process, depicted in Figure 2, consists of the following computational steps carried out by the CP.

6.1 Deadline Computation

The deadline, T_D, is an estimate of how much longer it should take to process all of the leftover buckets if the available resources over all sites could be used optimally, i.e., if all the sites could be "collapsed" into a single site encapsulating the processing capabilities of the entire ensemble. The estimation procedure works as follows. The number and sizes of the leftover buckets over all sites are known to the CP. Let the set of leftover buckets be denoted $\mathcal{L}$. The worst case estimate for the amount of data blocks to be transferred, and the total number of join tuples for the leftover buckets, are respectively:

$$\begin{aligned} DATA_{transfer} &= \frac{1}{BLKSIZE} \sum_{i \in \mathcal{L}} (size(b^i_R) + size(b^i_S)) \\ JOIN_{tuple} &= \sum_{i \in \mathcal{L}} \frac{\mid b^i_R \mid \times \mid b^i_S \mid}{\mathcal{D}^i} \end{aligned}$$

Let $\hat{\mathcal{S}}$ denote the set of all sites. The total data transfer capacity of the ensemble (in seconds per block) may be approximated as follows:

$$\begin{aligned} C^{\hat{\mathcal{S}}}_{transfer} &= \frac{1}{\left(\frac{1}{2}\right) \times \left(\sum_{s \in \hat{\mathcal{S}}} \left(\frac{1}{C^s_{transfer}} \right) \right)} \\ &= \frac{2}{\left(\sum_{s \in \hat{\mathcal{S}}} \left(\frac{1}{C^s_{transfer}} \right) \right)} \end{aligned}$$

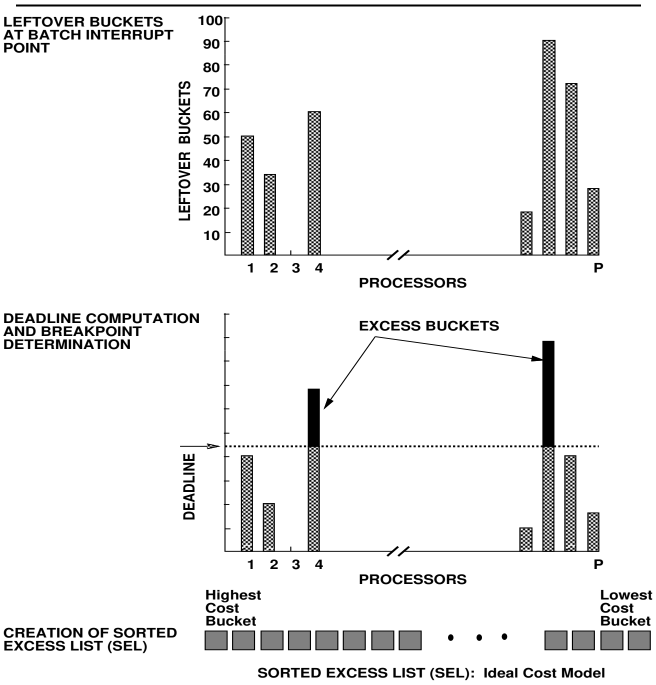


Figure 2: Simplified Diagram of Allocation Steps Using WLPT

The denominator approximates the total number of blocks that may be transferred per second between pairs of sites (assuming the communication network does not reach saturation and does not suffer significant performance degradation). The factor of $\frac{1}{2}$ in the denominator reflects the fact that for every sender, there is a receiver, i.e., the transfer of 1 data block requires the participation of 2 sites, reducing the effective throughput by a factor of 2. Similarly, the total join computation or pair comparison capacity (in seconds per tuple per tuple) is given by $C^{\hat{S}}_{join} = \left(\sum_{s\in\hat{S}}\left(\frac{1}{C^s_{join}}\right)\right)^{-1}$. Finally, the total output capacity for result tuples (in seconds per block) is given by $C^{\hat{S}}_{output} = \left(\sum_{s\in\hat{S}}\left(\frac{1}{C^s_{output}}\right)\right)^{-1}$. Thus, the deadline T_D may be estimated as follows:

$$T_D = C^{\hat{S}}_{trans} \times DATA_{trans} + C^{\hat{S}}_{join} \times \sum_{i\in\mathcal{L}}(\mid b^i_R \mid \times \mid b^i_S \mid) + C^{\hat{S}}_{output} \times \frac{JOIN_{tuple} \times 2 \times S_{tuple}}{BLKSIZE}$$

The factor of 2 in the estimate for output cost appears because the size of the tuples of the join product is expected, in our model, to be twice the size of the original tuples involved in the join.

6.2 Breakpoint Determination

The breakpoints for sets of leftover buckets at each local site s determines the "excess" buckets that should be transferred for processing elsewhere. The breakpoints are determined as follows. A "finish" time is computed based on the performance vector $< C^s_{transfer}, C^s_{join}, C^s_{output} >$ for site s, by adding the join and output costs only for the local leftover buckets. Let the locally resident leftover buckets be numbered $1 \cdots LEFT_s$.

$$TIME^s_{finish} = C^s_{join} \times \sum_{i=1\cdots LEFT_s} (\mid b^i_R \mid \times \mid b^i_S \mid) + C^s_{output} \times \sum_{i=1\cdots LEFT_s} \frac{\mid b^i_R \mid \times \mid b^i_S \mid}{\mathcal{D}^i} \times \frac{2 \times S_{tuple}}{BLKSIZE}$$

If $(TIME^s_{finish} > T_D)$, some buckets must be moved out of site s to be processed elsewhere. These are the excess buckets. The objective in this step is to find the bucket $BREAK_s$ such that $1 \cdots BREAK_s$ is retained at site s for processing locally, and buckets $BREAK_s + 1 \cdots LEFT_s$ are transferred to some remote site. We use a greedy method to find the breakpoint. Buckets are marked for removal starting at the highest numbered bucket, $LEFT_s$, and working down. As a bucket is marked, its join cost is subtracted from the finish time for this site, and an amount of time equal to the transfer time for this bucket is added to the finish time to account for the time the local site must spend in order to transfer this bucket elsewhere. This is repeated until an equilibrium is reached with the T_D (modulo some small threshold). The bucket where equilibrium is reached is the "breakpoint" bucket $(BREAK_s)$ for site s.

6.3 Creation of Sorted Excess List (SEL)

The breakpoints at each site determine the set of excess buckets. The CP logically "collects" them and assigns an estimated "ideal" cost to each bucket using ideal coefficients for join and output, denoted by C^{ideal}_{join} and C^{ideal}_{output}. These coefficients have the usual meaning, but they are constants that are determined apriori by simulations on an ideal system with nominal loads. The cost of some bucket e in the set of excess buckets is given by:

$$COST^e_{ideal} = C^{ideal}_{join} \times \mid b^e_R \mid \times \mid b^e_S \mid + C^{ideal}_{output} \times \left(\frac{\mid b^e_R \mid \times \mid b^e_S \mid}{\mathcal{D}^i}\right) \times \frac{2 \times S_{tuple}}{BLKSIZE}$$

The basic cost formula is used, but the term for data transfer is omitted from this cost computation. The excess buckets are then sorted in descending order according to these costs and placed in a sorted excess list (SEL), which is used by the WLPT algorithm, described next.

6.4 Reallocation Using WLPT

The reallocation of excess buckets over the sites seeks to achieve an even workload for joining the remaining buckets in the current batch. This is the load balancing component of the PJ protocol. The deadline computation discussed previously provides an estimate of the additional time it would take from the current interrupt point to complete the processing of the current batch, provided that resources are utilized in an optimal fashion. While finding a true optimal allocation is NP-complete, a fast heuristic that solves a closely related problem and which has excellent average case performance may be used instead. Such a heuristic is the *Longest Processing Time First* (LPT) algorithm [10]. LPT was designed to be a fast heuristic for solving the *Multiprocessor Scheduling* problem, which is NP-complete. The multiprocessor scheduling problem may be described as follows. Let there be N tasks, with execution times T_i, $\quad i = 1, \cdots, N$, and P processors over which these tasks are to be assigned. Let p be the function that gives the mapping from tasks to processors, i.e., $p(i)$ identifies the processor to which task i is assigned. For each processing site s, define the "busy time" for processor s to be $B(s) = \sum_{i=1,\cdots,N;p(i)=s} T_i$. In other words, $B(s)$ is the sum of the execution times of all the tasks assigned to processor s. The goal of LPT is to find an assignment of tasks to processors such that the maximum busy time over all processors is minimized. A *perfect* assignment, if attainable, would keep each processor equally busy, i.e., $B(s) = \left(\sum_{i=1}^{N} T_i\right)/P, \;\; s = 1, \cdots, P$.

The original LPT heuristic assumes all processors are equal, and do not exhibit performance variations over time. Our goal is to provide load balancing for a heterogeneous and dynamically changing computing environment. When the performance characteristics of the computing ensemble, as reflected in the performance vectors $< C^s_{transfer}, C^s_{join}, C^s_{output} >$ from each site s, remain relatively stable over the intervals between load balancing, our modified version of LPT is designed to produce superior results over straight LPT. The LPT heuristic requires that the largest task yet to be assigned should always be assigned next, and it should be assigned where it fits "best". Our modification works as follows. The SEL is used only for an initial estimate of the relative costs of the excess buckets. However, as the buckets are assigned in turn, the actual performance vector for the target site is used to *re-evaluate* the cost of the bucket. The justification is that the cost of a bucket, from the viewpoint of the target site, changes as the potential target changes. Thus, our modification, WLPT, "weights" the bucket in question with the observed performance of the potential target when applying the assignment heuristic.

WLPT Algorithm

INPUT: number of sites, P
number of excess buckets, N
SEL, a decreasing order sorted list of N excess buckets, sorted according to an ideal cost model
 	
OUTPUT: A heuristic assignment of excess buckets to processing sites such that the maximum busy time at any site is minimized
 	
For $s = 1$ to P DO
initialize $FINISH_s \leftarrow 0$
 	
For $i = 1$ to N DO
Let site $\hat{s}$ be the site that minimizes
$FINISH_s + COST^i_{alloc}[s]$
ASSIGN bucket i to site $\hat{s}$
UPDATE: $FINISH_{\hat{s}} \leftarrow FINISH_{\hat{s}} + COST^i_{alloc}[\hat{s}]$

Figure 3: The WLPT Algorithm

The WLPT algorithm is displayed in Figure 3. For each bucket i on the SEL, largest cost bucket first, the estimated allocation cost is computed for each site s in turn, using coefficients from the most recent performance vector for site s as follows:

$$\begin{aligned} COST^i_{alloc}[s] = {} & C^s_{transfer} \times \frac{size(b^i_R) + size(b^i_S)}{BLKSIZE} \\ & + C^s_{join} \times (\mid b^i_R \mid \times \mid b^i_S \mid) \\ & + C^s_{output} \times \left(\frac{\mid b^i_R \mid \times \mid b^i_S \mid}{\mathcal{D}^i} \right) \times \frac{2 \times S_{tuple}}{BLKSIZE} \end{aligned}$$

The site $\hat{s}$ that gives the lowest finish time when bucket i is allocated to it, obtained by adding the current estimated finish time and the value of $COST^i_{alloc}[\hat{s}]$, is assigned bucket i. The finish time for the recipient site is increased by the amount $COST^i_{alloc}[\hat{s}]$. This process is carried out until all excess buckets are rescheduled.

One point to note is that the target sites for the buckets on the SEL may be computed in parallel by broadcasting the sizes and other pertinent information for each of the bucket pairs on the SEL in turn to all the sites. The sites then compute the local projected finish time if the current bucket were processed locally. These values are then passed up to the CP in a tournament fashion, retaining the identity of the site that minimizes the finish time at every intervening node of the tournament tree. The identity of the target site for each bucket pair is known to the CP in $\log P$ steps. (See [5] for details.)

7 Parallel Join in the Presence of Data Skew

Parallel join algorithms are particularly sensitive to data skew. Most parallel join algorithms largely ignore the issue and assume uniform distribution of values in the domain of the join attributes for the relations to be joined. In many real databases the values tend to exhibit some degree of skew, i.e., some values of the join attribute occur significantly more frequently than others [2]. Such values are called “skew elements”. It has been suggested that the skew in the distribution of the values of interesting attributes in many real databases may be modeled by a Zipf-like distribution [14], where the degree of skew may be controlled with one of the parameters that defines the distribution.

The cost models we have shown so far for the case of uniform distributions do not apply when the base relations are skewed in the distribution of its join column. Since identical values hash to the same join bucket, data skew usually results in some buckets being much larger than the average (it is possible for a bucket to be approximately of average size and still contain skew elements). Moreover, rehashing the bucket using a different hash function does not solve the problem. The main problem with skewed buckets is that the size of the join output can be much larger than for the uniform distribution case. If both buckets being joined are skewed (double skew), the problem can become even more serious, since the output size then tends to grow quadratically with the occurrence cardinality of the skew value or values. The cost models we have developed so far can not produce correct estimates for the join output size when data skew is present, resulting in incorrect load balancing decisions.

The situation can be remedied by developing methods to correctly identify buckets containing skew elements, and then processing such buckets using a different strategy. Even if we could correctly estimate the join cost for a highly skewed bucket, WLPT may not be able to find a good allocation simply because any allocation could completely swamp the one (or few) sites to which the skewed bucket(s) are assigned, i.e., the sites with skew buckets would become hotspots.

[2] For example, compare the frequencies of the last names *Dewan* and *Smith* in a telephone directory.

We now extend the PJ protocol to handle the case when data skew is present. We will refer to this as the “Parallel Skew Join” (PSJ) protocol.

7.1 Outline of the PSJ Protocol

Each bucket is fragmented over all the sites since the base relations are fragmented in round robin fashion. Thus, one requirement is to be able to determine if skew elements are present in a bucket, without having to collect all the fragments of a bucket in one place. Since all occurrences of a specific skew value will be hashed to the same relation fragment on some site, it follows that any particular skew value will occur in a specific bucket fragment. Thus, if different fragments of a hash bucket independently flag a skew element locally, we can conclude that many different skew elements are present in the bucket as a whole. However, to qualify as a skew bucket, it is enough to detect a skew element in only a single fragment of the bucket.

Skew detection and proper handling of skew buckets is crucial to the success of the PSJ protocol. While accurate detection of the presence and magnitude of skew is essential, the protocol must not pay too much overhead for this phase. Ideally, if enough information is gathered at bucket formation time, then flagging buckets as either skewed or non-skewed can be folded into the initial phase of the protocol with minimal additional cost, without exacting a cost specifically for this purpose during actual execution of the join. An example of excessive resource use for computing skew values would be to construct a complete frequency distribution of the join attributes for each bucket fragment. In that situation, skew elements could be detected accurately. However, the penalty in terms of space to maintain the histogram or DFD for each bucket fragment at every site is $O(\mathcal{D})$, where $\mathcal{D}$ is the domain span or range of values contained in the domain. For large databases with many distinct values of the join attribute, $\mathcal{D}$ may be very large so that this approach is not feasible. However, if we make the reasonable assumption that modulo the skew elements, the remaining values in the domain are uniformly distributed, simple thresholding of the size of any bucket relative to an ideal or average bucket may be used for initial flagging of potential skew buckets. Only these buckets are further examined to isolate the skew elements. The PSJ protocol steps are described next.

PSJ Protocol Steps:

Initial Phase

1. CP computes N_{bkt} as in the PJ protocol. It informs local sites of the computed value of N_{bkt}.
2. Local sites form buckets and store them in local bucket files by hashing on the join attributes of the local fragments of relations R and S.
3. Each local site compares the sizes of each local bucket fragment with the ideal bucket fragment sizes for relations R or S ($\frac{|R|}{P \times N_{bkt}}$ or $\frac{|S|}{P \times N_{bkt}}$), as appropriate. If the i-th local bucket fragment exceeds the ideal size by more than a specified threshold, it is further tested for skew as follows.
   - A histogram or DFD is constructed on the join attribute for the fragments of b^i_R and b^i_S. The number of bins is a small integer $\mathcal{K}$, which is specified as a system parameter. Larger values of $\mathcal{K}$ allow a higher resolution of the DFD, with correspondingly greater accuracy in the detection of skew elements. However, the cost of maintaining the DFD also goes up. Thus, $\mathcal{K}$ must be chosen large enough for skew elements to be detected with the desired probability, but small enough so that DFD maintenance is efficient. This requires some experimentation (e.g., statistical probing) with the database in question, and some guidelines may be compiled off-line for different types of databases.
   - Local sites scan the local fragments b^i_R and b^i_S, and update the bin count for each join attribute encountered in either of the bucket fragments by locating of the appropriate bin in constant time.
   - If the frequency of any bin exceeds the average of the (non-zero) bin frequencies by more than a specified factor, then bucket i is considered a skew bucket and the skew elements are removed from b^i_R, b^i_S and placed in two new buckets, $b^{i'}_R, b^{i'}_S$. These latter buckets have a special flag, called the *skew flag*, set to 1.
4. Each local site sends a list of local bucket ids, together with their sizes and the value of the skew flag for each bucket, to the CP.

Main Phase

1. CP uses the per site information obtained from the initial phase to create two pools of buckets, representing normal and skewed buckets. These are called the *NORMAL POOL* and the *SKEW POOL*, respectively.
2. CP initiates processing of the *NORMAL POOL* using the main phase of the PJ protocol detailed in previous sections.
3. After the *NORMAL POOL* has been completely processed, the CP initiates processing of the *SKEW POOL* using allocation algorithms described next.

7.2 Processing the *SKEW POOL*

The bucket pairs on the *SKEW POOL* are processed as follows:

SKEW POOL **Processing Algorithm:**
For each bucket pair $b^{i'}_R, b^{i'}_S$ on the *SKEW POOL* DO:
Without loss of generality, let $| b^{i'}_R | > | b^{i'}_S |$.

1. Let $w^s = C^s_{join} + C^s_{output} \times \frac{2 \times S_{tuple}}{BLKSIZE}$. The w^s represent the average "weight" of each site s in terms of computing the join and writing the join result to disk for any pair of tuples from the two buckets. The values of C^s_{join} and C^s_{output} are those computed in the processing of the *NORMAL POOL* for each site. Let $\hat{w}^s$ denote the normalized values of w^s.
2. Partition the tuples in $b^{i'}_R$ over the P sites in proportion to the normalized weights $\hat{w}^s$ by simple counting.
3. Broadcast the tuples of $b^{i'}_S$ in groups of total size $BLKSIZE$ to all sites. The broadcast tuples are joined with the local partition of the corresponding bucket and the join result written out at each site. The values of C^s_{join} and C^s_{output} are updated locally and propagated to the coordinator after the i-th bucket from the *SKEW POOL* has been completely processed. The most recent values of the coefficients are used for the partitioning of the next bucket in the *SKEW POOL*.

8 Performance Analysis

To test the performance of the parallel join algorithms presented in this paper, we ran several experiments with both uniform and skewed data distributions. The computing environment consists of a cluster of 6 HP9000 workstations with FDDI interconnect. Each of the sites in the cluster is a site with an attached disk and local memory. We present performance of the system for both uniform and skewed data distributions, over varying load conditions among the sites. The data placement and bucket formation are part of the off-line processing. All other times are included in the measured performance.

8.1 Join Performance for Uniform Data Distributions

We assume two relations, R and S, that are to be joined on a common join attribute. The records of each relation are of equal size (32 bytes), with an integer field that serves as the join attribute. Both relations are of equal size. Various experiments were performed by varying the size of each relation from 200,000 to 1,000,000 tuples. The domain size is chosen to be a number smaller than the total number of tuples in each relation (e.g., 50% of the relation cardinality of R or S in these experiments). A data generator generates the relations R and S with uniform distribution of the join attribute over the specified domain, producing relations R and S with specified values for the relation and domain sizes.

8.1.1 Pure Speedup

The first metric we track is "pure speedup". For a range of reasonably large database sizes ($|R|=|S|= 200,000$ to $1,000,000$ tuples), we run the PJ protocol over 6 sites. The sites are "clean", meaning there are no external loads and the processing sites are identical (no heterogeneity). The experiment is run in two modes. In the first mode, all of the predictive dynamic load balancing (PDLB) machinery presented in this paper is disabled, so no dynamic load balancing is attempted. We call this "NO-PDLB mode". The experiment is then run again with the PDLB feature enabled. This is called "PDLB mode". Each experiment is run several times and average performance is computed.

The pure speedup is defined to be the speedup due to parallel processing relative to a clean run (no external load and no data skew) for the same database size on a single site. Thus, we get two sets of values for pure speedup: one for NO-PDLB and the other for PDLB mode. The pure speedup values measured are shown in Figure 4 (a). From the graph, we can see that the pure speedup over the range of database sizes considered remains approximately constant and in the range [5.5-5.96], for both PDLB and NO-PDLB modes. This translates into pure speedups in the range 92-99% of perfect speedup. We note that the pure speedup in NO-PDLB mode is slightly higher than for PDLB mode for each database size. This is due to the fixed "nominal" overhead that PDLB mode has to pay relative to NO-PDLB mode.

8.1.2 Nominal PDLB Overhead

The second metric tracks the "nominal PDLB overhead". The purpose of this metric is to provide a measure of the PDLB overhead when the system is run clean, i.e., there are no external loads and the sites are all homogeneous. We ran the PJ protocol over 6 sites with PDLB enabled and disabled. Each experiment was run several times and average performance is reported. The nominal PDLB overhead is computed by taking the difference in running times between PDLB and NO-PDLB modes for each database size. This is plotted as a function of the database size in Figure 4 (b). Note that if the data distribution as well as the performance of the sites is such that PDLB is actually initiated, then typically there will be data movement across the network between processing sites. Thus, the overhead will be higher than the nominal overhead shown here. It is clear from the figure that the nominal PDLB overhead is a small constant relative to total join time, and is independent of the database size. It is, however, an increasing function of the number of sites, P. In our example, the overhead of PDLB is approximately in the range 1-2% of total join time when the database consists of two relations with 1,000,000 records each.

8.1.3 Join Times Under CPU-bound External Loads

Since the processing sites in our experimental environment are homogeneous, we must devise some means to measure the performance of the algorithms under heterogeneous conditions. We are interested also in measuring the performance when external loads (both CPU and I/O bound) are present. Clearly, introducing external loads in some sites is equivalent to having a parallel ensemble with heterogeneous processors. Thus, we test the effect of both heterogeneity and external loads by systematically varying the CPU and I/O requirements of an external "synthetic load" utility at a small number of sites.

The synthetic load utility is started up at one or a small number of sites when the parallel join begins. The utility runs as a separate process and is an infinite loop computation characterized by two parameters. The `compute` parameter specifies the number of times the utility computes the square root of a floating point number in a tight loop. The `i/o` parameter specifies the number of 4 Kilobyte blocks (of junk data) that the utility writes to disk after finishing the compute intensive loop. Thus, by varying only the value of the `compute` parameter and keeping the `i/o` parameter fixed, it is possible to simulate both CPU-bound and I/O-bound external loads at any given site(s).

When external (or synthetic) loads have a substantial I/O component, the operating system (e.g., UNIX) tends to favor other coexisting processes (e.g., join process) with CPU time slices whenever the external or synthetic load is performing I/O. However, when the external or synthetic load is CPU bound (i.e., does very little I/O), then the CPU resource gets more evenly divided among all processes by the operating system. CPU-bound processes occur frequently in scientific computations. Here, we explore the effect of purely CPU-bound external loads on the performance of the PJ protocol.

The synthetic load utility is configured to run in CPU-bound mode by setting the `i/o` parameter to 0 and the `compute` parameter to a very large number. Thus, the synthetic load process runs a tight loop performing arithmetic operations with no I/O. In PDLB and NO-PDLB modes, respectively, we ran a series of experiments in which the number of external loads running on one of the 6 sites was varied between 1 and 5. The database was kept fixed at 1,000,000 tuples for each of the relations R and S. The overall completion time for the join $R \bowtie S$ in PDLB and NO-PDLB modes are shown in Figure 4 (c). We see that the time in PDLB mode increases very slowly with the external load. This points to the fact that the PJ protocol is very effective in balancing the excess work at the site with CPU-bound external loads of various magnitudes, as long as the I/O channels are free and available. The corresponding overall join times in NO-PDLB mode shows a linear growth with the magnitude of the total external loads, displaying the direct effect of the slow down of the loaded site on the entire parallel ensemble. A reduction of up to 66% in the join time is observed under PDLB relative to NO-PDLB mode operation for the range of external loads tested.

8.1.4 Join Times Under I/O-bound External Loads

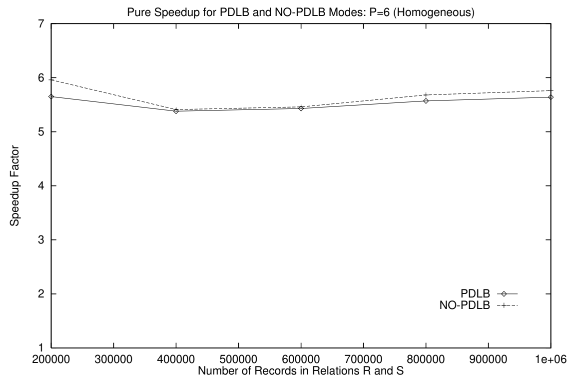
The dual experiment is to test the effect of I/O-bound external loads on the performance of the PJ protocol. In this case, the synthetic load utility is configured to run in I/O-bound mode by setting the `i/o` parameter to 1 and the `compute` parameter to a small value, e.g., 1. Thus, the external load repeatedly performs a trivial amount of CPU activity followed by a write of a 4 Kilobyte buffer to disk, simulating an I/O-bound process.

In PDLB and NO-PDLB modes, respectively, we ran a series of experiments in which the number of I/O-bound external loads running on one of the 6 sites was varied between 4 and 20. The database was kept fixed at 1,000,000 tuples for each of the relations R and S. The overall completion time for the join $R \bowtie S$ in PDLB and NO-PDLB modes are shown in Figure 4 (d). We see that the time in PDLB mode increases very slowly with the external load. This points to the fact that the PJ protocol is very effective in balancing the excess work at the site with I/O-bound external loads of various magnitudes. The corresponding overall join times in NO-PDLB mode shows a linear growth with the magnitude of the total external loads, displaying the direct effect of the slowdown of the loaded site on the entire parallel ensemble. A reduction of up to 45% in the join time is observed under PDLB relative to NO-PDLB mode operation for the range of external loads tested.

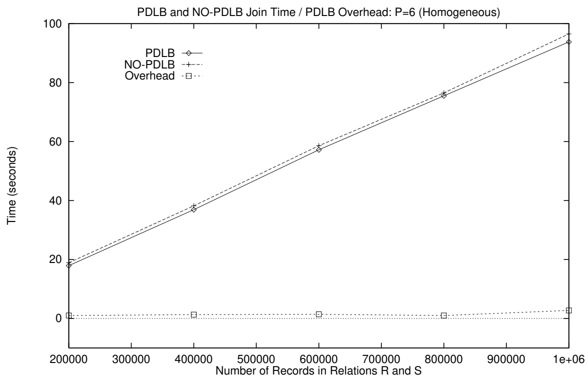
8.1.5 Effect of Available Memory: Varying Batch Size

Recall that the parameter b specifies the number of buckets per batch in the PJ and PSJ protocols. For fixed bucket size, the parameter b reflects the amount of memory available at each site for join processing, as larger physical memories would intuitively call for using larger batch sizes for better memory utilization and superior overall performance. Here we test whether this intuition is sound. We vary the available memory in the range 500 to 4000 Kilobytes, and adjust b to take full advantage of the available memory. The overall join time for $R \bowtie S$ ($\mid R \mid = \mid S \mid = 1,000,000$) in PDLB and NO-PDLB mode are plotted as a function of available memory in Figure 4 (e). In either case, 2 CPU-bound external loads were introduced at one of the six sites.

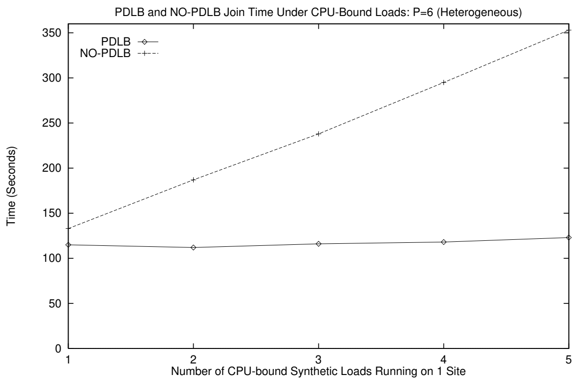
We observe that PDLB mode benefits greatly from larger memory availability by significant reductions in the join time, while no such effect is observed for NO-PDLB mode. This can be explained as follows. In PDLB mode, the effect of larger memory is to allow the fastest site to process correspondingly more buckets (the number of buckets that would fill the memory) before interrupting the CP. At the interrupt point, global rescheduling is invoked by WLPT. All other factors being equal, this results in fewer interrupts and subsequent rescheduling per batch. Combined with the fact that larger memory (and correspondingly larger values of batch size, b) result in fewer batches overall for given database and bucket sizes, the total overall number of interrupts (and rescheduling phases) in PDLB mode decreases as available mem-



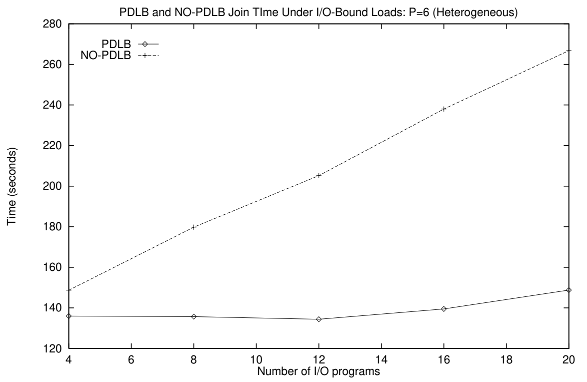
(a) Comparative Pure Speedup: P=6



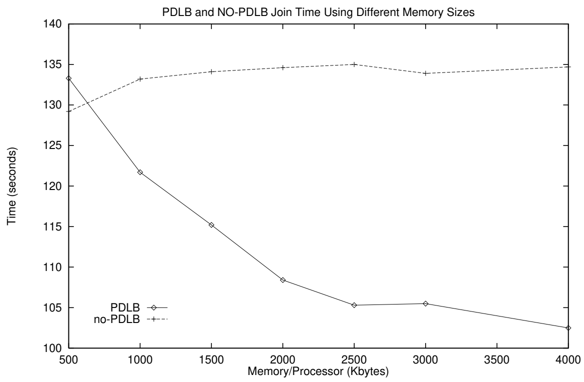
(b) Nominal PDLB Overhead: P=6



(c) Join Times Under CPU-bound Loads: P=6

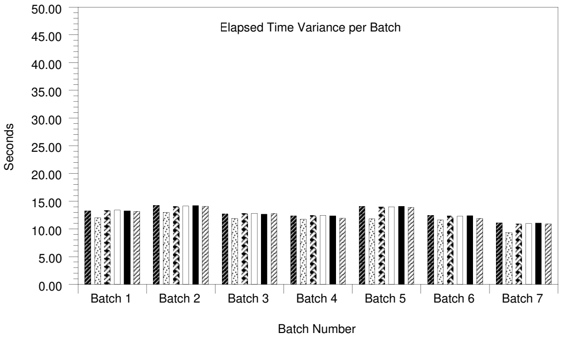


(d) Join Times Under I/O-bound Loads: P=6

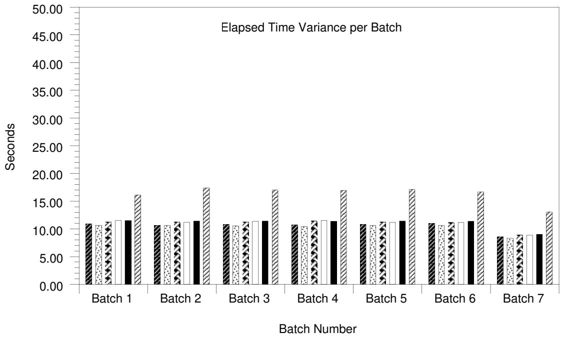


(e) Join Times vs. Memory: P=6

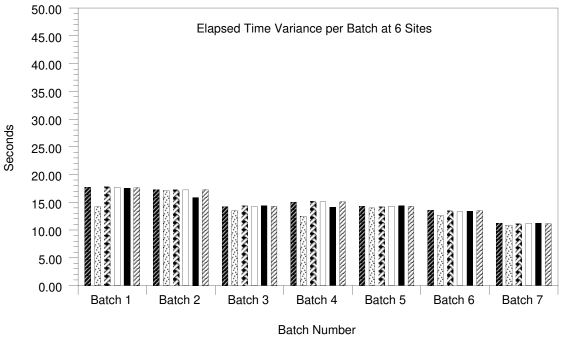
Figure 4: Performance Plate 1



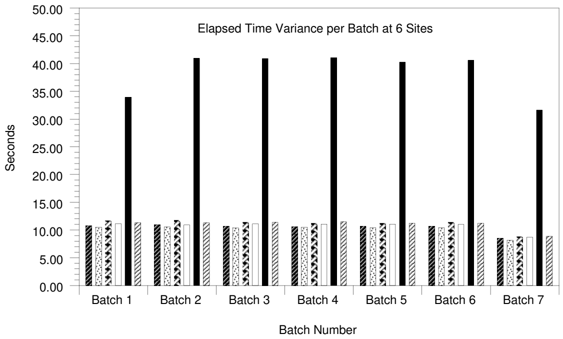
(a) PDLB, 1 External Load, P=6



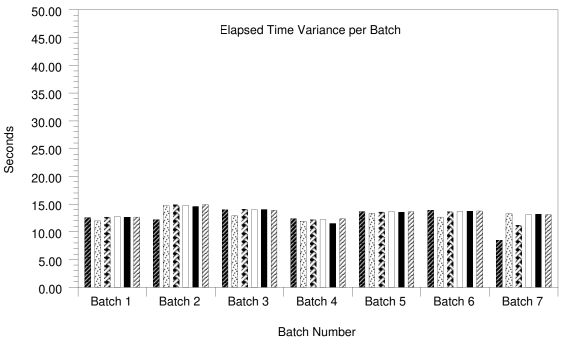
(b) NO-PDLB, 1 External Load, P=6



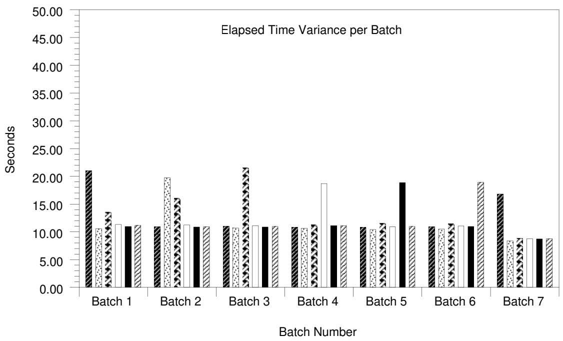
(c) PDLB, 4 External Loads, P=6



(d) NO-PDLB, 4 External Loads, P=6



(e) PDLB, 2 Roving External Loads, P=6



(f) NO-PDLB, 2 Roving External Loads, P=6

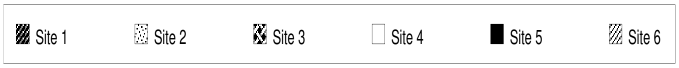
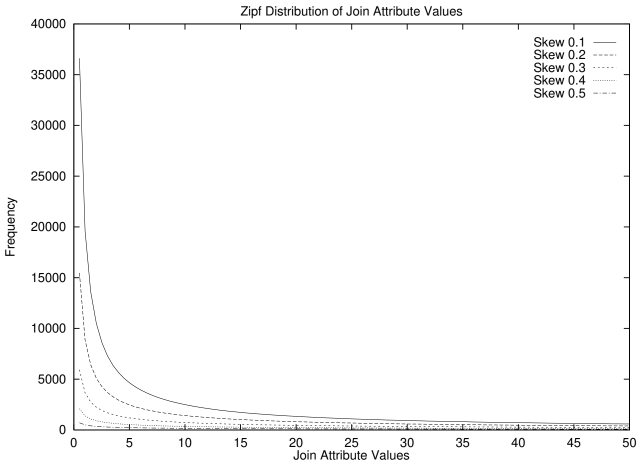
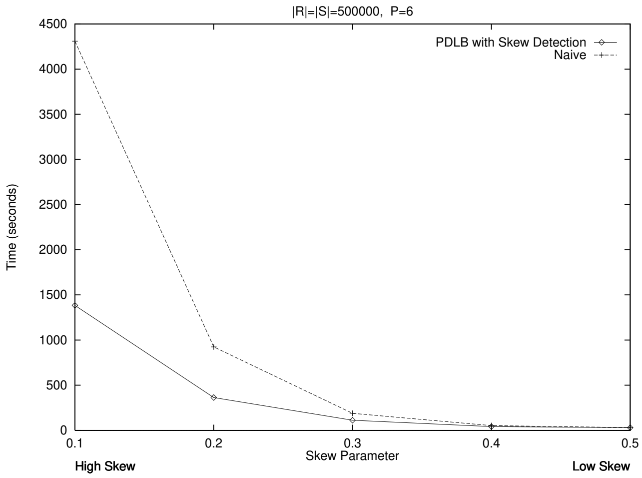


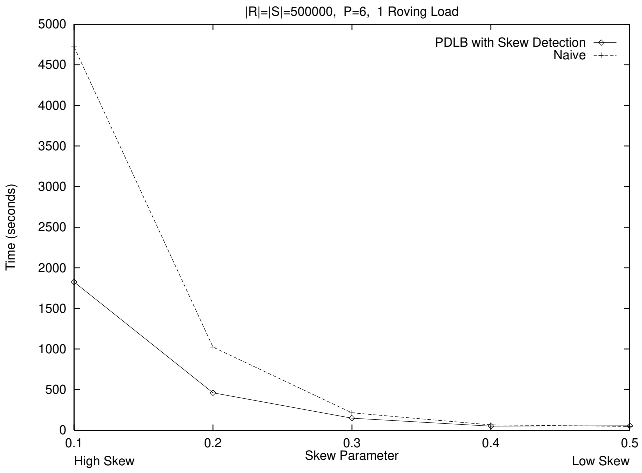
Figure 5: Performance Plate 2



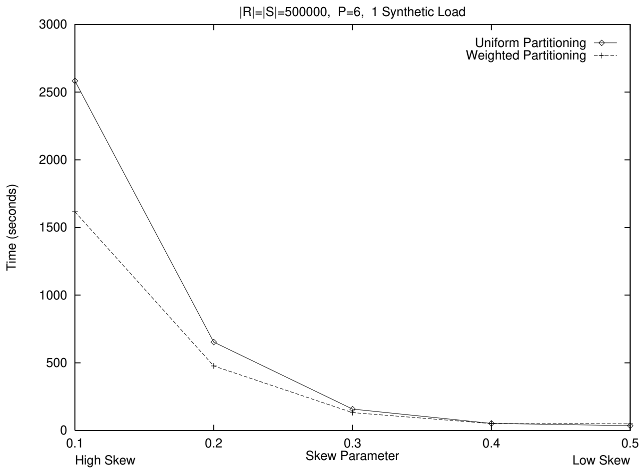
(a) Zipf Distributions for Various Degrees of Skew



(b) Join Time for Skewed Data: P=6



(c) Join Time for Skewed Data with External Loads: P=6



(d) Effect of Weighted vs. Uniform Partitioning of Skew Bins on Join Time: P=6

Figure 6: Performance Plate 3

ory increases, resulting in overall reduction in the join time. In NO-PDLB mode, however, the performance is bounded by the total processing time of the slowest site for the buckets allocated to it at the start of the join computation (since no dynamic reallocation is allowed). Hence, the performance does not change appreciably as more memory is made available. We conclude that predictive dynamic load balancing can take greater advantage of larger memories compared to naive parallel processing.

8.1.6 Join Time Variance under PDLB and NO-PDLB

The improvement in performance (e.g., in the reduction of join time) under PDLB has been demonstrated in the previous graphs. Here we show how PDLB directly affects the load balance of the join computation among all sites. We introduce CPU-bound external loads in one of six processing sites and measure the actual completion time per batch at each site for a computation of $R \bowtie S$ ($\mid R \mid = \mid S \mid = 1,000,000$). In Figures 5 (a) and 5 (b), we show the actual join times at each of 6 sites over 7 batches when 1 CPU-bound external load is running at one of the sites. Under PDLB, all sites complete the processing of their portion of the batch at approximately the same time, while in NO-PDLB mode, significant variances are observed. The overall improvement in the join time under PDLB is 27% for 1 external load running at one of the 6 sites. In Figures 5 (c) and 5 (d), the same effect is studied for 4 CPU-bound external loads running on one of the sites. In this case, the overall improvement is as high as 56%.

In Figures 5 (e) and 5 (f), we show the improved performance per batch in the join time when 1 CPU-bound external load runs in a "roving" pattern over all the sites. In other words, the external load migrates from one site to the next in a round-robin fashion after consuming CPU resources at any given site for a specified duration (e.g., 5 seconds). Once again, PDLB successfully reduces the variance in per batch join times, while NO-PDLB mode suffers from a large variance and reduced performance. The overall improvement under PDLB is observed to be 30% in this case.

8.2 Join Performance for Skewed Data Distributions

We model data skew by generating synthetic data according to a Zipf-like distribution. The Zipf-distribution in our data generator works as follows. Assuming that the domain size of the join attribute is $\mathcal{D}$, we define p_i to be the probability that the value of the join attribute for a particular tuple takes the i-th value in the domain. This probability is given as $p_i = c/i^{(1-\theta)}$, where $c = 1/\sum_{i=1}^{\mathcal{D}}(1/i^{(1-\theta)})$. The data generator chooses the value of the join attribute independently from this distribution. Low values of θ correspond to high skew, while higher values correspond to low skew, with θ ranging between and 0 and 1.

To get a sense of the skewness introduced in the data by varying the parameter θ, in Figure 6 (a) we plot the Zipf-like distributions for $\theta = 0.5, 0.4, 0.3, 0.2, 0.1$ (representing medium to high skew) for a domain size of 250,000, with distinct values in the range [1-250,000]. For clarity, the distribution of only the first 50 values in the domain is shown. The frequency of values greater that 50 is essentially constant (approximately 2). Note, however, that for $\theta = 0.1$ (representing a high degree of skew), the largest skew element has a frequency greater than 3600, which is much larger than the average frequency of domain values. When joins are computed on buckets containing skew elements in both buckets (double skew), the join output size for those bucket pairs can be extremely high. We note that the experiments reported here deal with double skew, since both relations R and S are generated in the same fashion with the same parameter settings in the data generator.

In Figure 6 (b), we show the join time for $R \bowtie S$ ($\mid R \mid = \mid S \mid = 500,000$) tuples for $\theta = 0.1, 0.2, 0.3, 0.4, 0.5$, respectively, covering a range from high to medium skew. The join times displayed correspond to naive operation (NO-PDLB), and PDLB (the PSJ protocol) enabled. We observe that PDLB mode performs 68% better than NO-PDLB mode when $\theta = 0.1$ (high skew). As the severity of skew decreases, the difference in performance between PDLB and NO-PDLB becomes smaller. For medium skew, both modes perform equally well. Our experience is that for uniform distributions, PDLB with *SKEW POOL* processing enabled performs slightly worse (several percentage points) than NO-PDLB mode when no external load is present. This is due to the overhead of skew detection.

In Figure 6 (c), we repeat the experiment of part (b), but introduce external loading in addition to the data skew. PDLB mode still consistently outperforms NO-PDLB for high and medium skew values, and the improvement is as high as 63% when $\theta = 0.1$.

In Figure 6 (d), we plot the performance of PDLB with the PSJ protocol enabled for two different schemes of partitioning skew bins. The *uniform* method divides up the tuples in a skew bin (after skew bins have been isolated) into even size fragments over the sites, and broadcasts blocks of tuples from the corresponding matching bin from the other relation. The *weighted* method, on the other hand, uses the most recently computed values for join and transfer coefficients (as described in Section 7.2) to derive relative weights for the sites and divides skew bins accordingly. We observe that the weighted method performs consistently better relative to the uniform method for moderate to high degrees of skew. An improvement of 38% is observed under weighted partitioning when $\theta = 0.1$.

9 Conclusions

In this paper, we studied a complex problem with numerous dimensions: skewness in the distribution of data, database size, size of hash buckets, batch size, and runtime variations in the characteristics of the processing environment. The results shown in this paper indicate that PDLB indeed does provide better performance in cases when the processing sites are non-homogeneous, and adds only a negligible overhead when the sites are homogeneous. Moreover, the framework we present handles data skew in a natural fashion. We have provided a detailed set of performance data under changing load distributions produced by a synthetic load generator, and also for various degrees of data skew. Our ongoing work aims to explore the efficacy of the PDLB approach under real operating conditions.

The processing potential of each site is modelled by the performance vector $< C_{transfer}, C_{join}, C_{output} >$. The idea borrows from locality principles in paging algorithms. Clearly, if sites vary in processing loads dramatically between rescheduling points, PDLB predictions may not be borne out. Thus, modelling of the processing environment might be a worthwhile pursuit, meaning that future work ought to aim at collecting runtime information at the operating system level about other competing processes at a site. This points to a place where operating system and database research could converge in the pursuit of common standards where each can support the other towards developing improved services and overall improved performance for all applications.

There are several fundamental issues that remain open and require further investigation. Namely, what are the optimal environmental parameters and partitioning method that would maximize the benefits of PDLB at run-time? There is of course a tradeoff between the cost of measuring and maintaining system parameters and the resultant benefit to overall efficiency and performance. For example, in the present work we simply compute the values of $C_{transfer}$, C_{join} and C_{output} based upon runtime performance during the most recently processed batch of buckets. Perhaps a more accurate and useful measure would account for the time evolution of these measures over several consecutive prior batches of buckets. Another crucial issue is the choice of attribute and partitioning function to optimize the distribution of workload at run-time, as well as optimal choice of batch size b for a particular distribution.

The parallel join algorithm is presently being implemented as the underlying rule matching component of the PARADISER [3, 4] expert database system reported elsewhere. PARADISER currently runs in a replicated database configuration on a distributed network of workstations in the processing environment of a "typical" Computer Science department. A future paper will report the results of PDLB and the parallel join algorithm running under PARADISER in fully distributed database configurations.

References

[1] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data Placement in Bubba. In *Proceedings of the ACM SIGMOD 1988, Intl. Conf. on the Management of Data.* ACM Press, 1988.

[2] H. M. Dewan. Runtime Reorganization of Parallel and Distributed Expert Database Systems. Technical report, Dept. of Computer Science, Columbia University, May 1994. Ph.D. Thesis.

[3] H.M. Dewan, M. Hernandez, S. Stolfo, and J. Hwang. Predictive Dynamic Load Balancing of Parallel and Distributed Rule and Query Processing. In *Proc. of the ACM-SIGMOD 1994, Intl. Conf. on the Management of Data*, 1994.

[4] H.M. Dewan and S.J. Stolfo. The Distributed Evaluation of Rules in PARADISER. Technical Report In Preparation, Department of Computer Science, Columbia University, May (expected) 1994.

[5] H.M. Dewan, S.J. Stolfo, and L. Woodbury. Scalable Parallel and Distributed Expert Database Systems with Predictive Load Balancing. *J. Parallel and Distrib. Computing, Special Issue on Scalable Systems*, 1994. To appear.

[6] D. DeWitt and R. Gerber. Multiprocessor Hash-based Join Algorithms. pages 151–164, 1985. In proc. Intl. Conference on Very Large Databases.

[7] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA - A High Performance Dataflow Database Machine. pages 228–237, 1986. In proc. Intl. Conference on Very Large Databases.

[8] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. In *Communications of the ACM*. ACM, June 1992.

[9] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. pages 27–40, 1992. In proc. Intl. Conference on Very Large Databases.

[10] R. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM J. of Applied Mathematics*, 17(2):416–429, 1969.

[11] Tandem Performance Group. A Benchmark of Non-Stop SQL on the Debit Credit Transaction. In *Proceedings of the ACM SIGMOD 1988, Intl. Conf. on the Management of Data*. ACM Press, 1988.

[12] M. Kitsuregawa and Y. Ogawa. Bucket Spreading Parallel Hash: A New, Robust, Parallel hash Join Method for Data Skew in the Super Database Computer (SDC). 1990. In proc. Intl. Conference on Very Large Databases.

[13] M. Kitsuregawa, H. Tanaka, and T. Moto-Oka. Application of Hash to Data Base Machine and Its Architecture. In *New Generation Computing*, volume 1:1, 1983.

[14] D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, Reading, MA, 1973.

[15] D.A. Schneider and D.J. DeWitt. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of the ACM SIGMOD 1989, Intl. Conf. on the Management of Data*. ACM Press, 1989.

[16] M. Stonebraker, editor. *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Weslwy Publishing Company, 1986.

[17] J. D. Ullman. *Principles of Database and Knowledge-Base Systems; Vols. 1 and 2*. Computer Science Press, 1989.

[18] J.L. Wolf, D.M. Dias, and P.S. Yu. A Parallel Sort Merge Join Algorithm for Managing Data Skew. In *IEEE Transactions on Parallel and Distributed Systems*, volume 4:1, pages 70–86, January 1993.

[19] J.L. Wolf, D.M. Dias, P.S. Yu, and J. Turek. Comparative Performance of Parallel Join Algorithms. In *First Intl. Conference on Parallel and Distributed Systems*, pages 78–88. IEEE, 1991.