

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

Hasanat M. Dewan  
Kui W. Mok

Mauricio Hernández<sup>†</sup>  
Salvatore J. Stolfo

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

## 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. Data skew appears 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. 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 (e.g., [6, 4, 5, 3]). These algorithms are parallel versions of sort-merge or hash-based joins 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 (e.g., [7, 4, 3]). 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.

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, in addition, 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 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.

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 runtime 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.

\*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.

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

## 2 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 (off-line), and 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  $P$  sites of the ensemble in a round-robin fashion. This results in equal numbers of tuples from each relation being allocated to the disks attached to every site.

In the second phase, 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  $S$ , under the same hash function. 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. If 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. 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  $N_{bkt}$ .

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. However, 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",  $B$ . This parameter sets 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 7.1.5).

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<sup>1</sup>. This depends on the granularity of buckets we desire, and may depend on the exact join algorithm being used. 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.

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

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.

<sup>1</sup>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 on the performance of the system.

### 3 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; point-to-point and broadcast messages are supported.

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 [2]. 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 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,  $B$ .

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  $1 \leq b \leq B$  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  $1 \dots b$  to site 1,  $b + 1 \dots 2b$  to site 2, etc. Once a complete set of  $b$  bucket pairs 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  $|b_R^i|$  and  $|b_S^i|$ , and their sizes (in bytes) by  $size(b_R^i)$  and  $size(b_S^i)$ , respectively.

During operation, disparities in execution times 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  $\langle C_{transfer}, C_{join}, C_{output} \rangle$  for each site (called the “performance vector” henceforth).

$C_{transfer}$  is the average cost for reading the blocks corresponding to the buckets into memory, expressed as seconds per block. Blocks are of a standard size of  $BLKSIZE$  bytes, where  $BLKSIZE$  is a system parameter (typically, 8192 bytes). This coefficient is initialized to a default value determined by actual measurements of the average inter-site transfer cost under normal operating conditions. In the computation of

transfer costs, the performance statistic that is relevant is the block transfer time between sites, rather than between disk and physical memory. 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. The value of  $C_{output}$  is dynamically computed at runtime by measuring actual block output times.

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 message.

Upon receiving the READY 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. 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 5.2. In response to the RESCHED message, all  $P$  sites send a READY message along with local status information, such as the number and sizes of local leftover buckets, as well as the local performance vectors.

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) heuristic algorithm reported extensively in the literature. Our version is

called WLPT (Weighted LPT). Rescheduling is initiated by the CP after the reallocation is computed. This process is repeated until the current batch is fully processed.

## 4 Cost Model

A cost model is used for making 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  $S$  ( $b_R^i \bowtie b_S^i$ ) at a given site  $s$  is defined as follows.

$$\begin{aligned} COST_{basic}^i &= C_{transfer}^s \times \frac{(size(b_R^i) + size(b_S^i))}{BLKSIZE} \\ &+ C_{join}^s \times (|b_R^i| \times |b_S^i|) \\ &+ C_{output}^s \times \left( \left( \frac{|b_R^i| \times |b_S^i|}{D^i} \right) \times \frac{2 \times S_{tuple}}{BLKSIZE} \right) \end{aligned}$$

$D^i$  denotes the average domain size of the smaller of the domains of  $b_R^i$  and  $b_S^i$ , estimated by dividing the domain size 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 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 6.

## 5 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 computes a reallocation plan. This process, depicted in Figure 1, consists of the following computational steps carried out by the CP.

### 5.1 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

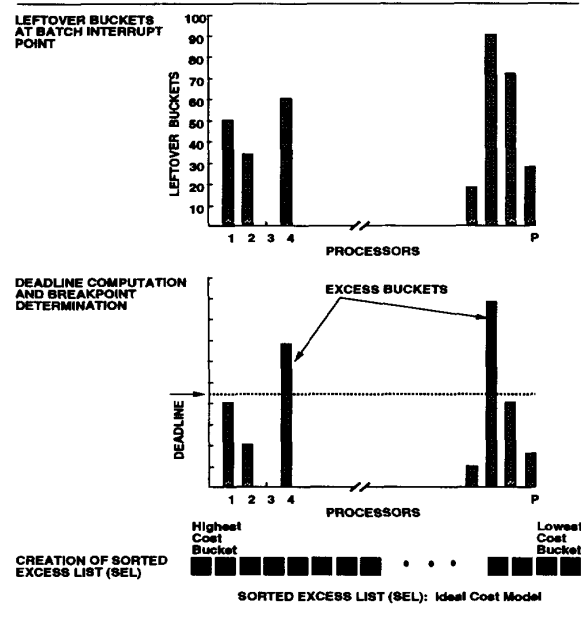


Figure 1: Simplified Diagram of Allocation Steps Using WLPT

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:

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

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:

$$C_{transfer}^{\hat{\mathcal{S}}} = \frac{1}{\left(\frac{1}{2}\right) \times \left(\sum_{s \in \hat{\mathcal{S}}} \left(\frac{1}{C_{transfer}^s}\right)\right)}$$

The denominator approximates the total number of blocks that may be transferred per second between pairs of sites. 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  $C_{join}^s = \left( \sum_{s \in \hat{s}} \left( \frac{1}{C_{join}^s} \right) \right)^{-1}$ . Finally, the total output capacity for result tuples (in seconds per block) is given by  $C_{output}^s = \left( \sum_{s \in \hat{s}} \left( \frac{1}{C_{output}^s} \right) \right)^{-1}$ . Thus, the deadline  $T_D$  may be estimated as follows:

$$T_D = C_{trans}^s \times DATA_{trans} + C_{join}^s \times \sum_{i \in \mathcal{L}} (|b_R^i| \times |b_S^i|) \\ + C_{output}^s \times \frac{JOIN_{tuple} \times 2 \times S_{tuple}}{BLKSIZE}$$

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.

### 5.2 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 by computing a “finish” time based upon the performance vector  $\langle C_{transfer}^s, C_{join}^s, C_{output}^s \rangle$  for site  $s$ , by adding the join and output costs only for the local leftover buckets. The CP logically “collects” the excess buckets and assigns an estimated “ideal” cost to each bucket using ideal coefficients for join and output, denoted by  $C_{join}^{ideal}$  and  $C_{output}^{ideal}$ . These coefficients have the usual meaning, but they are constants that are determined apriori by simulations on an ideal system with nominal loads. 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.

### 5.3 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. 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. LPT was designed to be a fast heuristic for solving the *Multiprocessor Scheduling* problem. Let there be  $N$  tasks, with execution times  $T_i$ ,  $i = 1, \dots, N$ , 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  $B(s) = \sum_{i=1, \dots, N, p(i)=s} T_i$ . 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.,  $B(s) = \left( \frac{1}{P} \right) \left( \sum_{i=1}^N T_i \right)$ ,  $s = 1, \dots, P$ .

The original LPT heuristic assumes all processors are equal, and do not exhibit performance variations over time. Our modification, WLPT, “weights” the bucket in question with the observed performance of the potential target when applying the assignment heuristic.

For each bucket  $i$  on the Sorted Excess List (SEL) of buckets, 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:

$$COST_{alloc}^i[s] = C_{transfer}^s \times \frac{size(b_R^i) + size(b_S^i)}{BLKSIZE} \\ + C_{join}^s \times (|b_R^i| \times |b_S^i|) \\ + C_{output}^s \times \left( \frac{|b_R^i| \times |b_S^i|}{D^i} \right) \times \frac{2 \times S_{tuple}}{BLKSIZE}$$

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  $COST_{alloc}^i[\hat{s}]$ , is assigned bucket  $i$ . The finish time for the recipient site is increased by the amount  $COST_{alloc}^i[\hat{s}]$ . This process is carried out until all excess buckets are rescheduled.

## 6 Parallel Join in the Presence of Data Skew

Parallel join algorithms are particularly sensitive to data skew. In many real databases the values tend to exhibit some degree of skew. 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, where the degree of skew may be controlled with one of the parameters that defines the distribution.

The cost models we have developed so far cannot 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. We will refer to this as the “Parallel Skew Join” (PSJ) protocol.

### 6.1 Outline of the PSJ Protocol.

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. We make the reasonable assumption that modulo the skew elements, the remaining values in the domain are uniformly distributed. Thus, 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. The final determination is made by constructing a histogram on the frequency distribution of the join attribute values for the bucket, and checking for large deviations in frequency of a small number of bins. The PSJ protocol steps are described fully in [1]. The overall algorithm proceeds as follows:

#### Main Phase

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

### 6.2 Processing the *SKEW POOL*.

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

#### *SKEW POOL* Processing Algorithm:

For each bucket pair  $b_R', b_S'$  on the *SKEW POOL* DO:  
Without loss of generality, let  $|b_R'| > |b_S'|$ .

1. Let  $w^s = C_{join}^s + C_{output}^s \times \frac{2 \times \text{Size}}{BLKSIZE}$ . 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  $C_{join}^s$  and  $C_{output}^s$  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  $b_R'$  over the  $P$  sites in proportion to the normalized weights  $\hat{w}^s$  by simple counting.
3. Broadcast the tuples of  $b_S'$  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  $C_{join}^s$  and  $C_{output}^s$  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*.

## 7 Performance Analysis

To test the performance of the parallel join algorithms we ran several experiments with both uniform and skewed data distributions. The computing environment was comprised of a cluster of 6 HP9000 workstations with FDDI interconnect. Each site had an attached disk and 32 Megabytes of local memory. The data placement and bucket formation are part of the off-line processing. All other times are included in the measured performance.

### 7.1 Join Performance for Uniform Data Distributions.

We assume two equal size relations,  $R$  and  $S$ , with tuples 32 bytes long, and with an integer field that serves as the join attribute. We varied 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 produces the relations  $R$  and  $S$  with the specified relation sizes, domain sizes, and distribution of the join attribute over the specified domain.

**7.1.1 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. 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 2(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 due to the fixed “nominal” overhead of PDLB mode.

**7.1.2 Nominal PDLB Overhead.** The second metric tracks the “nominal PDLB overhead”. 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 2(b). 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,  $P$  (this effect is not shown). 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.

**7.1.3 CPU-bound External Loads.** We are interested also in measuring the performance when external loads (both CPU and I/O bound) are present. 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.

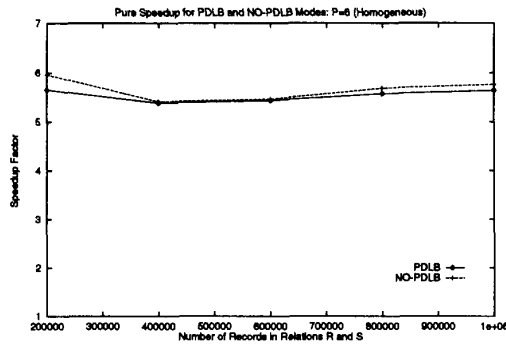
When external (or synthetic) loads have a sub-

stantial I/O component, the operating system (e.g., UNIX) tends to favor other coexisting processes. 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.

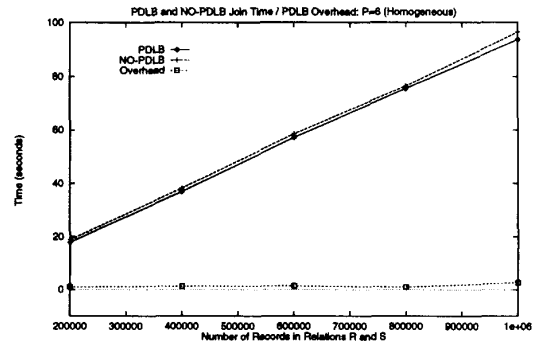
In PDLB and NO-PDLB modes, respectively, we ran a series of experiments in which the number of external loads running on one of the 6 sites was varied between 1 and 5. The database was kept fixed at 1,000,000 tuples for each of the two relations. The overall completion time for the join in PDLB and NO-PDLB modes are shown in Figure 2(c) (lines labelled “CPU-Bound Load”). 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.

**7.1.4 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.

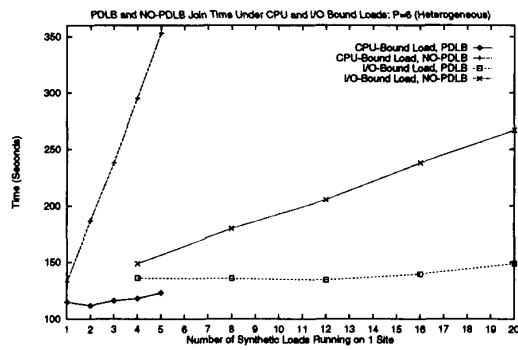
In PDLB and NO-PDLB modes, respectively, we ran a series of experiments in which the number of I/O-bound external loads running on one of the 6 sites was varied between 4 and 20. The database was kept fixed at 1,000,000 tuples for each relation. The overall completion time for the join in PDLB and NO-PDLB modes are shown in Figure 2(c) (lines labelled “I/O-Bound Load”). 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



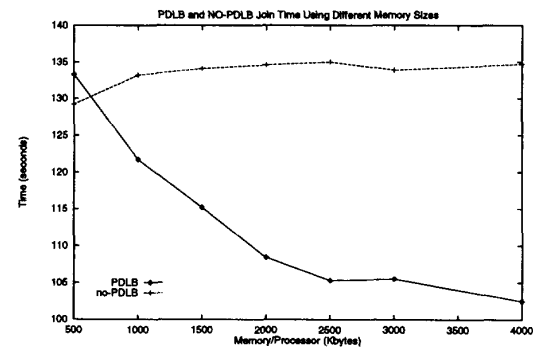
(a) Comparative Pure Speedup: P=6



(b) Nominal PDLB Overhead: P=6



(c) Join Times Under CPU and I/O bound Loads: P=6



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

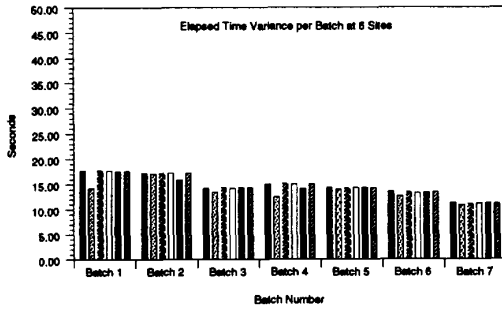
Figure 2: Performance Plate 1

is observed under PDLB relative to NO-PDLB mode operation for the range of external loads tested.

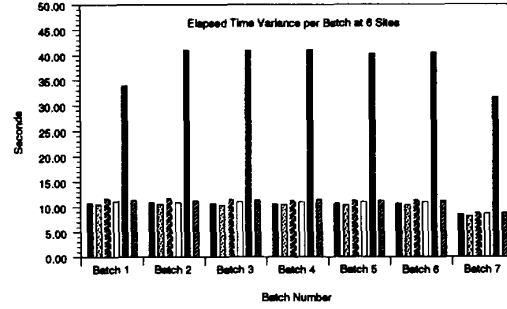
**7.1.5 Effect of Available Memory: Varying Batch Size.** Recall that the parameter  $b$  specifies the number of buckets per batch in the PJ and PSJ protocols. For fixed bucket size, the parameter  $b$  reflects the amount of memory available at each site for join processing, as larger physical memories 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 in PDLB and NO-PDLB mode are plotted as a function of available memory in Figure 2(d). 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. The total overall number of interrupts (and rescheduling phases) in PDLB mode decreases as available memory 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. 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.



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



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

Figure 3: Performance Plate 2

**7.1.6 Join Time Variance.** 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. Due to space limitations, we do not show these results graphically. The reader is referred to [1] for a more complete set of results. 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 were 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 3(a) and 3(b), 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%.

## 7.2 Join Performance for Skewed Data Distributions.

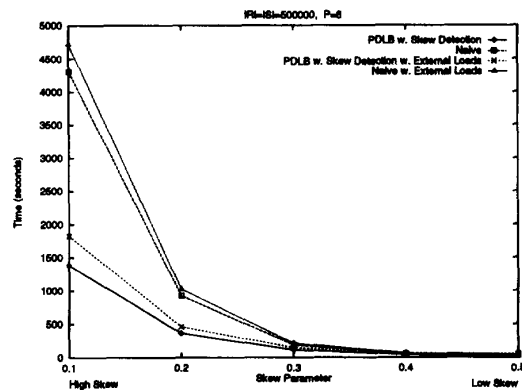
We model data skew by generating synthetic data according to a Zipf-like distribution. Assuming that the domain size of the join attribute is  $\mathcal{D}$ , we define  $p_i$  to be the probability that the value of the join attribute for a particular tuple takes the  $i$ -th value in the domain. This 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  $\theta$  correspond to high skew, while higher values correspond to low skew, with  $\theta$  ranging between 0 and 1.

In Figure 4(a), we show two sets of curves. In the first set, we show the time for  $R \bowtie S$  ( $|R|=|S|=500,000$ ) for  $\theta = 0.1, 0.2, 0.3, 0.4, 0.5$ , re-

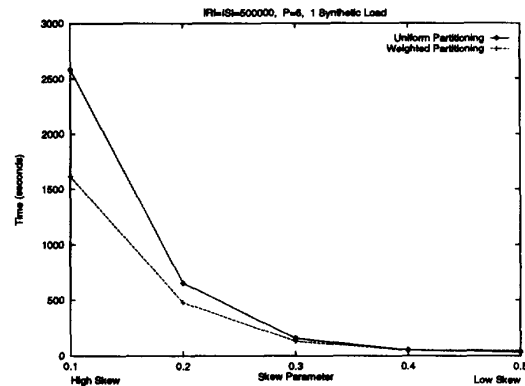
spectively, covering a range from high to medium skew. The join times displayed correspond to naive operation (NO-PDLB), and PDLB (the PSJ protocol) enabled. We observe that PDLB mode performs 68% better than NO-PDLB mode when  $\theta = 0.1$  (high skew). As the severity of skew decreases, the difference 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 the second set of curves in Figure 4(a), we repeat the experiment, 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 4(b), 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 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  $\theta = 0.1$ .



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



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

Figure 4: Performance Plate 3

## 8 Conclusions

In this paper, we studied a complex problem with numerous dimensions. 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.

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.

## References

- [1] H. M. Dewan. Runtime Reorganization of Parallel and Distributed Expert Database Systems. Tech-

nical report, Dept. of Computer Science, Columbia University, May 1994. Ph.D. Thesis.

- [2] 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.
- [3] D. DeWitt, J. Naughton, D. Schneider, and S. Shadri. Practical Skew Handling in Parallel Joins. pages 27–40, 1992. In *proc. Intl. Conference on Very Large Databases*.
- [4] 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*.
- [5] 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.
- [6] M. Stonebraker, editor. *The INGRES Papers: Anatomy of a Relational Database System*. Addison-Wesley Publishing Company, 1986.
- [7] 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.