Centralized, Decentralized and Pipelined Algorithms to Parallelize Joins

Ashish Gupta+ Robert J. T. Morris* Arun Swami* Honesty C. Young*

+ Department of Computer Science, Stanford University, Stanford, CA 94305, USA
* IBM Almaden Research Center, San Jose, CA 95120-6099, USA
+agupta@cs.stanford.edu, *{rjtm,arun,young}@almaden.ibm.com

Abstract: In parallelizing the join operation of database systems, a primary objective is to partition the work in a way that ensures that no processor has a completion time for its part of the join that far exceeds that of any other processor. While one can design a type of bin packing algorithm that achieves this almost exactly, such algorithms can involve a cost (in CPU, I/O and communication resources) that dominates the join activity itself. Therefore the challenge is to create effective, scalable, heuristics which achieve good results with only marginal overhead.

We describe three classes of algorithms: *centralized, decentralized and pipelined hybrid.* We introduce a taxonomy of join types and use this taxonomy to explain how the different algorithms choose a point in the spectrum of complexity, robustness and overhead. We present our results in the form of "tradeoff curves" which make clear that very large response time improvements are obtainable from these algorithms, and at what cost in resource consumption. We describe a spectrum of algorithms ranging from a centralized algorithm that balances the work almost exactly (but at a fairly high cost) to decentralized algorithms which are more heuristic and achieve much the same effect but with much lower overhead. Then we describe a tunable class of pipelined algorithms which parametrically subsume a number of these methods.

This paper shows that the proposed load-balancing algorithms are robust under all types of join loads. In some cases the algorithms yield up to two orders of magnitude better performance in response time at marginally higher resource cost (10-30%) compared to hashing. The taxonomy of join types is intuitive and helpful in categorizing algorithms.

1 Introduction

The join is both one of the most common and most expensive operations in relational database systems and is hence a natural candidate for parallelization. There has been extensive work on parallel database systems [1, 2, 11, 14] and on different parallel join methods [12, 13]. In this paper we consider equijoins involving two relations, R_1 and R_2, over one or more columns. The join operation can be decomposed into multiple subtasks by partitioning on the join attribute(s) values: each subtask consists of joining tuples from the two relations that have the same join column value, i.e., taking the Cartesian product of the corresponding subsets of tuples from each relation. The work involved in any one of these subtasks is sensitive to the *skew*, or multiplicity of values, in the operand relations, and the correlation between their frequent values [10, 15]. When the join work is partitioned among multiple processors in the parallel join, this skew in the operand relations can result in significant imbalance in the work assigned to the different processors. This imbalance can cause significant degradation in the overall response time for the join operation and, as a result, there has been a lot of interest in skew-handling algorithms for parallel joins [8, 9, 16, 17].

One can try to handle this skew by appropriately balancing the distribution of the skewed values among the join processors. But in cases of severe skew, no single join processor can be assigned the joining subtask corresponding to a single value and *splitting*, i.e., the participation of more than one processor in one of these subtasks, is necessary. Such splitting is usually feasible but results in extra overhead. For the parallel join, splitting is achieved using a technique such as *fragment replicate* (FR) [3]. Thus the skew handling problem can be thought of as a form of *load balancing*, although the use of the term here differs from the usage in the operating system context since we have *a priori* knowledge of resource consumptions and are free to split tasks.

We consider joining 2 relations, R_1 and R_2, using a set of p *join sites* ($\mathcal{J}$): $\{J_1,\ldots,J_p\}$. The equijoin is over the common attribute A which takes values from the domain $\mathcal{V}=\{v_1,\ldots,v_k\}$. We often use the term *value of a tuple* to refer to the value of the join attribute A of the tuple. Without loss of generality, we assume R_1 and R_2 are initially fragmented horizontally over s *storage sites* ($\mathcal{S}$): $\{S_1,\ldots,S_s\}$. We assume a message-passing architecture in which nodes are interconnected by a network that lets every pair of nodes communicate. Each node has its own processor(s), memory, disks, operating system, and database software. Each table in the relational model is partitioned into subsets of rows, and each subset is stored at one site. In this paper, we will use *site* to refer to a logical unit and *node* to a physical unit. $\mathcal{S}$ and $\mathcal{J}$ are logical units and are completely connected. C is a central arbitrating site as discussed later. As more than one site is available for the join, the time taken to complete it should be less than that taken by one site, provided the work involved in the join is reasonably well distributed among the sites in $\mathcal{J}$. In order to distribute the work in joining R_1 and R_2, the tuples are divided among the sites in $\mathcal{J}$. Even though this balancing adds to the overall resource cost, it is worthwhile if the increase is not disproportionately high compared to the reduction in the overall response time, also known as the *makespan*. We discuss some of the factors that affect the way the sites should share the load to achieve minimization.

One simple way of distributing the tuples among $\mathcal{J}$ is based on the value of the join attribute. A hash function $h : \mathcal{V} \rightarrow \mathcal{J}$, which is the same at every node, is used to determine the site to which a tuple is sent. All tuples with value v_j are sent to J_i if $h(v_j) = i$. The join can then be materialized on the sites $\mathcal{J}$ in parallel. The work in joining tuples with value v_j depends on the number of tuples in each of R_1 and R_2 with that value. The problem with this simple hash-based strategy is that not all values v_j have the same number of tuples. Hence even though all sites may have approximately the same number of distinct values hashed to them, they may do vastly different amounts of work.

The hash-based scheme attempts to balance work by distributing the distinct values approximately evenly among the sites. It does not account for skew in the data. However hash-based schemes are attractive because they are simple and scale very well. The hash function h can be used by all the storage sites independently with no coordination needed. It is also easy and efficient to implement and compute. Though hashing is more for *assigning* work than balancing, a good hash function would achieve the goal of balancing the work if the data is well distributed between the possible distinct values.

The following classification of relations and joins is useful in discussing the results. A type **D** relation is one with a large number of duplicates (i.e., tuples with the same join attribute value). Relations with very few duplicates are said to be of type **I**. Three possible join types are discussed and approximately classified as follows:

- **D**x**D** Each relation has multiple tuples with the same join attribute value and a high degree of correlation, i.e., the frequent values in both relations are the same.
- **D**x**I** One of the relations has multiple tuples with the same join attribute value but the other has very few tuples per distinct value.
- **I**x**I** Each relation has a single or small number of tuples for each distinct value.

The main observation is that type **D** relations open up a join to the possibility of skew due to the product of a large number of duplicates. Such skew may not necessarily occur (for example, if the duplicates are well distributed), but could be particularly acute in the **D**x**D** case. We will keep referring to the above classification in this paper. We will show that in load balancing, hash algorithms are perfectly adequate to handle **I**x**I** situations, and sometimes **D**x**I** situations. The most important role of load balancing algorithms is in handling **D**x**D** and providing robust behavior in the presence of **D**x**I** situations. The results also show that load balancing algorithms can sometime provide *orders of magnitude* improvement in response time over hash algorithms (the reader may want to preview the spreads between algorithms on Figure 1). But this response time improvement comes at the cost of doing the balancing. Furthermore if one is not careful and uses a centralized load balancing algorithm, this central site can become the bottleneck. This leads us to define decentralized algorithms, of which 1SLB is the simplest. We then go on to complete our repertoire of algorithms by defining a tunable algorithm, PHLB, which can be configured to be a robust decentralized load balancing algorithm, but has a tuning parameter so that may be used to control its overhead and degree of pipelining.

1.1 Previous Work

Load balancing algorithms operating under different assumptions have been studied for a long time [4, 6]. The problem of distributing (or scheduling) independent tasks so that a task is restricted to run on a single processor, i.e., no splitting is allowed, is called the *independent task scheduling problem*. An algorithm for this problem, called the *LPT* (*L*argest *P*rocessing *T*ime first) algorithm, was presented in [5]. The problem can be generalized to allow any task to be split into multiple parts. In our work, we do not impose any constraints on the size or the total number of fragments. This is a realistic scenario for the join load balancing problem.

The work on load balancing algorithms by Wolf et al. [16, 17] is relevant to our work. Their method uses only part of the histogram for the input relations initially but requires the complete histogram to make progressive refinements. Their algorithms are centralized in nature. Load balancing for the hash-based join operation is addressed by Roy and Keller [9]. Their algorithms use sophisticated logic to perform dynamic load balancing while the join is being evaluated. It is not clear how their algorithm can be generalized for local join algorithms other than the hash join algorithms. In our algorithms the primary distribution of tuples is done using hashing. Incremental modifications to this distribution are made using the histograms. We also describe a family of algorithms that span the range from centralized to distributed. The algorithms do not depend on the local join algorithm used.

Hua and Lee [8] describe two skew avoidance algorithms and one skew resolution algorithm. Their skew resolution algorithm (ABJ) and one of our decentralized algorithm (1SLB) share a similar idea. Both algorithms adopt an "optimistic" approach of sending the tuples to the join sites based on a hash function and then redistribute tuples when significant skew is detected. There are several major differences between ABJ and 1SLB. First, ABJ balances the number of tuples of each relation, but 1SLB balances the join work. When severe skew is present, a balanced number of tuples does not guarantee balanced join work. Second, ABJ does not allow a bucket to be split further, and the join work of a bucket consisting of a very frequent value must be executed on a single site. The 1SLB algorithm permits bucket splitting to evenly distribute the join work.

In the paper, Section 2 discusses centralized balancing schemes. Section 3 discusses a join site based scheduling scheme. In Section 4 we describe a tunable algorithm. Section 5 describes the experimental methodology designed to compare the different load balancing algorithms, and to study the tradeoff between response time and resource costs. In Section 6, we present our results. We conclude with Section 7. In this paper we will concentrate on describing basic versions of each algorithm and not attempt to analyze the effects of various conceptually minor simple variations which can potentially improve their performance. In a later paper [7], we will include the variations of the algorithms discussed in this paper, the cost of splitting join tasks, and additional experimental results.

2 Centralized, Decentralized and Pipelined Algorithms

Load-balancing algorithms need an estimate of the join work to be able to distribute the work in a balanced way. Because the join work is divided on the basis of the values of the join attribute, the amount of work corresponding to each distinct value is required. *Histograms* are used to store the information about the distribution of tuples among the attribute values and among source sites $\mathcal{S}$. A histogram $H_i(R)$ for the partition of relation R at site S_i is a set of $< value, frequency >$ pairs:

$$H_i(R)=\{< v_1, n_1^i >, \ldots, < v_k, n_k^i >\}$$

where n_j^i is the number of tuples of relation R on site S_i with value v_j for the join attribute. If there are no tuples with attribute value v_j on S_i, then $< v_j, n_j^i >$ is omitted. The histograms of relation R can be obtained by combining the $H_i(R)$'s from all the storage sites. The histograms of R_1 and R_2 can then be used to compute the estimated work in joining the relations:

$$H\!:\!\{< v_1, w_1 >, \ldots, < v_k, w_k >\}$$

where w_j is the work corresponding to joining tuples with value v_j.

This section describes a centralized algorithm (CLBCH), introduces an initial decentralized algorithm (2SLB), and then discusses the advantages of more refined versions (1SLB and PHLB) which will be explained in detail in the subsequent sections.

CLBCH (Centralized Load Balancing with Complete Histograms) requires a coordinator at the central site C. Each source site sends its full local histogram to C which computes the complete profiles of the relations being joined and the total join work. C then computes the tuple distribution needed to achieve balanced join work for each join site. A variant of CLBCH uses only part of the histograms to compute the tuple distribution needed for approximate load balancing. We omit the details of how to compute the required distribution of attribute values among join sites, or how to determine the relevant part of the histogram.

2SLB (2 Stage Load Balancing) is a join site based or decentralized scheduling algorithm. In the first stage in 2SLB, the source sites send their histograms to the join sites. The histogram entries are distributed among the join sites based on a simple hash function. Each join site can compute the work it needs to do if the tuples were sent according to the initial hashing function. The join sites can then use a central coordinator to compute the work redistribution needed for a balance and inform the source sites of the decision. In the second phase, the source sites send the tuples that need to be joined.

In join site based schemes, join sites can incorporate local information about other work loads while doing the load-balancing, and share the computational and communication load. Although a central site is proposed in a simple description of these algorithms, it does not play a synchronization or co-ordination function. The data sent to this site is very small, as is the processing performed. Thus, although for simplicity we include only one such site in our description, its function is easily replicated at multiple sites. These algorithms are thus argued to be "decentralized".

In 2SLB, histogram entries and tuples are distributed among the join sites in two separate stages. Particularly, the tuples are distributed after the synchronization point of running the load balancing algorithm. An improvement is to distribute the tuples among the join sites before running the load balancing algorithm. With the tuples, each join site can compute its share of the histogram entries. If severe skew is detected, some tuples are re-distributed to the appropriate join sites to balance the load. This leads to the *1 Stage Load Balancing* (1SLB) algorithm, which will be explained in detail in Section 3.

Both 2SLB and 1SLB perform the join after running the load balancing algorithm and pipelined execution is broken. Moreover in a parallel architecture, a significant portion of the inner relation may fit in the aggregate memory. Breaking the pipelined execution could have a big performance penalty in terms of both the resource cost and makespan. In this paper, we propose a *Pipelined Hybrid Load Balancing* (PHLB) algorithm to perform load balancing for hybrid hash join. We will use the abbreviation HASH to mean the hybrid hash join with no load balancing. Recall that the inner of the first partition in HASH fits in memory and the corresponding tuples from the outer relation are joined directly without being written to disk. PHLB joins most of the tuples in the first partition in a pipelined fashion just like HASH does. Then PHLB uses a decentralized load balancing algorithm (say 1SLB) to balance the join work for the remaining tuples in the first partition and all other partitions. The details of PHLB are in Section 4.

3 Decentralized One Stage Load Balancing (1SLB) Algorithm

In the 1SLB algorithm, the hash function h is used to assign the work to the J_i's initially. Each site can then estimate its real load using this work assignment and its background load. This load estimate can be communicated to a central site to arrive at an acceptable assignment. In case the hash function is sufficient given an acceptable imbalance, no further balancing need be done. In general, little communication overhead is incurred because the central site only needs estimates of the work and not the details of the data on the join sites.

1. **Hash Phase**: Each S_i uses h to decide the site to which each tuple is sent.
2. **Histogram Building Phase**: J_i builds $H_i(R_1)$ and $H_i(R_2)$ which are the histograms of the parts of relations R_1 (similarly R_2) that J_i gets. $H_i(R_1) = \cup_{j:h(v_j)=i}\{< v_j, n_j >\}$, $n_j = \sum_{l=1}^{s} n_j^l$ Because each distinct value is hashed to only one site, n_j is the number of tuples in R_1 with value v_j. $H_i(R_1)$ and $H_i(R_2)$ are used to compute W_i the join work on J_i.
3. **Table Building Phase**: Each join site J_i builds a 3-tuple $< J_i, F_i, W_i >$ where F_i is a "stretch factor" (described below) which conveys the background load and speed of the site, and W_i an estimate of the work due to the current join. Each join site J_i sends such a 3-tuple to the central load balancing site (C) which puts the information in a load table $\mathcal{T}$. C therefore has an estimate of the true load and join work on each J_i.
4. **Load Balancing Phase**: C uses $\mathcal{T}$ to compute the finishing time and excess load on each site. Because C has only the work estimates, it cannot determine the tuple distribution needed to achieve load balancing. C can however decide the fraction of its work (f_{iq}) overloaded site J_i needs to offload to the more lightly loaded site J_q. It broadcasts this information to $\mathcal{J}$ as a *load transfer table.*
5. **Data Re-transmission Phase**: The LTT tells J_i that fraction f_{iq} of its total work has to be sent to J_q. J_i decides which tuples should be sent to J_q to transfer that much work and sends those tuples.

The advantage of sending the tuples to $\mathcal{J}$ is that if little load balancing is needed, then two rounds of communication are avoided. As before, if the number of distinct values is large, this will be a worthwhile saving. The decomposition of H across multiple nodes also increases the chance that it will be able to be kept entirely in memory while it is analysed.

To account for the characteristics of the join sites, the load table $\mathcal{T}$ also contains a *stretch factor.* The stretch factor characterizes the relative speed of sites, and is defined as the number of elapsed time units expected to be taken to complete one time unit of processing. It can consist of a load index based on the backward averaged utilization of the processor together with an estimate of future work, which can be obtained from examination of queue lengths and knowledge of work in the pipeline. It can also take into account heterogeneity of processor configurations. Work can then be assigned to the processors in inverse proportionality to their stretch factor.

4 Pipelined Hybrid Load Balancing (PHLB) Algorithm

In this section, we explain the Pipelined Hybrid Load Balancing (PHLB) algorithm and its variations.

PHLB has 3 phases – pipelined execution phase, load balancing phase, and balanced execution phase. The join work corresponding to fraction X of the inner tuples in the first partition is performed in the pipelined execution phase. As the name suggests, the operation carried out in the pipelined execution phase is performed in a pipelined fashion and the corresponding outer tuples are not written to disk. In the load balancing phase, an offline load balancing algorithm is used to balance the remaining join work. The remaining join work includes the join work corresponding to the $1 - X$ fraction of the inner tuples in the first partition and the join work of all other partitions. Algorithm PHLB does not rely on any particular offline load balancing algorithm being used during the load balancing phase. In this paper, we use 1SLB as the load balancing algorithm within PHLB. The remaining join work is balanced during the load balancing phase and is carried out in the balanced execution phase. We first explain how the basic algorithm works and then we describe possible variations.

4.1 Basic PHLB

In this section, we describe the basic PHLB algorithm. With a given X, PHLB determines a threshold count T for the inner relation such that the sum of the number of tuples with replication factor less than or equal to T is at most X fraction of all inner tuples and T is the maximum such value. The join work for the inner tuples with replication factor T or less is performed during the pipelined execution phase and the remaining join work is done in the balanced execution phase.

In PHLB, first the inner is shipped from the source sites to the join sites where its histogram is computed as the tuples are received. Then, the T value corresponding to the parameter X is computed. Next, as the outer tuples come in, it is determined whether they should be joined immediately and forwarded to the destination node, or stored on disk to await learning their final join site destination, as determined by the load balancing phase.

Here is the pseudo code for hybrid hash join with the basic Algorithm PHLB (with emphasis on the pipelined execution phase).

```
// pipelined execution phase
for each inner tuple {
    insert this tuple into an in-memory
      hash table;
    increment the count corresponding to
      this join value;
    // a (value,count) pair is maintained
      for each distinct value
}
determine a threshold T such that the sum
  of counts less than or equal to T is
  at most X fraction of the total count
  and T is the maximum such threshold;
for each outer tuple {
    if the count field in the (value,count)
      pair of the corresponding inner is
      greater than T {
        write this tuple to disk;
        // join of this tuple will be done
        // in the balanced execution phase
    }
    else {
        do the join now
        // in the pipelined phase
    }
}

// load balancing phase
run steps 2 to 5 of 1SLB for the
  values/tuples that have not been joined;

// balanced execution phase
perform join for the remaining values/tuples;
```

The above decription assumes that the inner can fit into main memory. When this is not the case, the above algorithm is modified by replacing the inner with its first partition. The partition is chosen in such a way that it contains either all or none of the tuples with a particular join value. All arriving outer tuples are then subjected to the test for possible pielining if and only if they match a tuple in the first partition; if not, they are written to disk to await the balanced execution phase. Clearly the amount of pipelining will decrease substantially when there are many partitions needed for the inner.

4.2 Discussion

As stated above, the 1SLB algorithm already exploits the single most important mechanism in parallel join load balancing by breaking up the "**D**x**D** effect", i.e., the multiplicative effect of many coinciding duplicates in both the relations being joined. The new twists in PHLB are: (1) its ability to do more selective load balancing when the risk of imbalance is low, (2) its ability to avoid storing both inner and outer tuples to disk at the join site when memory is large enough to accommodate a significant part of one of the relations, and, (3) the ability to pipeline the join results so that the first join tuple emerges very early in the join phase.

In terms of the choice of parameters for PHLB, if one chooses a value of T that is fairly small, say $T < 5$, then the **DxD** effect is strongly limited, although not totally eliminated. But T should not be chosen too small (e.g., $T = 0$) or else it is clearly cheaper for the join site to do the join of unique tuples "on-the fly", i.e., without storing them to disk or considering them for balancing, even in the presence of some redistribution skew. Thus a conservative choice of T may be 2 or 3. Nevertheless a higher value of T could be profitably employed when it is known *a priori* that that there is little risk of significant skew, and one wishes to exploit the potential of squeezing out another 5-20% of overhead resource cost from the join algorithms (see results below), or wishes to enhance the pipelining capability to reduce overall response time of a complex query. Thus either the value of T, or the value of X can be externalized for tuning purposes, with a small default value. It allows the algorithm to be adjusted to encompass the entire spectrum of algorithms between HASH and 1SLB. This adjustment can be carried out manually by an administrator, or by the optimizer if the latter has access to some statistics concerning the relations being joined.

5 Experimental Comparison of Algorithms

To compare the performance of the different algorithms, both the resource cost of balancing and the response time are important. In our analysis we do count all communication times and costs, but omit the final step of shipping the materialized join tuples. Our justification for this omission is that this time and resource cost is common to all schemes. Given this, there are various response times that are important in these algorithms, for example the time until the first join tuple is materialized, the time until the last tuple is materialized, etc. Although some other measures are intersting, in this paper we concentrate on examining the response time from when the join is initiated until the last tuple is materialized, and to emphasize this we refer to it using the corresponding term *makespan* from scheduling theory.

In our analysis we present the performance of algorithms in the form of "trade-off" or pareto charts which simultaneously convey both performance and cost. The reader is urged to consider this rather unusual method of plotting the data. While they contain all the information in more commonplace plots, these charts are particularly revealing, since while on occasion we observe the "purchase" of better performance at a cost, in other cases we can observe that the comparison between two algorithms is not a tradeoff at all, but a *dominating* solution in the sense that one algorithm is superior both in performance and cost. Such a property would be hard to discern on separate plots.

5.1 Experimental Methodology

A common experimental methodology is to build analytical cost models for each algorithm. However parallelizing a join involves many parameters that are determined by the relations being joined and cannot be modeled without actually incorporating all the characteristics of the relations being joined.

To illustrate this, suppose relations R_1 and R_2 are being joined and the join attribute value v_1 contributes 80% of the join work. Tuples with attribute value v_1 will then have to be joined at multiple sites. The number of tuples that need to be replicated cannot be computed without actually joining the two relations. Thus the algorithms need to be implemented in order to get the numbers needed to do valid comparisons. The cost models require values like the number of tuples shipped from one site to another, tuple redistribution etc. In our study, these numbers are generated from executions of all the algorithms on the actual test relations being joined. On top of complete implementation of the algorithms, and the use of real data, we then use the detailed analytical models for makespan and resource costs [7]. We have not gone as far as doing discrete event simulations of the parallel architecture, but we do believe that our methodolgy is very effective and accurate for comparative purposes.

5.2 Models

A lot of factors affect the performance and cost of joining two relations with load balancing. They include the number of storage sites, the kind of data partitioning across $\mathcal{S}$, the number of join sites, I/O bandwidth, communication bandwidth, join method used, available memory, cost of comparisons, buffering costs, synchronization costs when pipelines are broken, and many others. The models for our two main metrics, makespan and resource cost, were obtained by detailed models of CPU, I/O and communication time taken for each operation. For the typical parameter values, we have taken estimates based on measurements of currently available workstation class processors and LANs. Note that we are not requiring the use of LAN and workstation technolgy for such an application, these are just the parameter values used to obtain these modeling results. The numbers were varied about an order of magnitude both ways to test for sensitivities. The conclusions reported here are representative of this range of values. Summaries of the cost models can be found in [7].

5.3 Data Used for the Experiments

Section 6 discusses the performance of the different algorithms given data from a variety of sources. The data used for the experiments is described briefly in this section. Load balancing algorithms perform differently depending on the kind of relations being joined. The following characteristics of the relations being joined become important:

- Number of tuples with the same join attribute value, i.e., number of duplicates.
- Correlation between the frequent values of R_1 and R_2.
- Size of the final join result which is affected by both of the above.

We used two sources of data for our experiments, a large commercial database and synthetically generated data. The former, from which we obtained both tables and queries, allowed us insight into the degree of skew that really does occur in practice. Having this experience, we then used synthetic data which allowed us a wide control over the amount of skew. In the commercial database, while we did find type **D** relations, we did not find joins between 2 highly skewed relations together with a high degree of correlation (the **D**x**D** type described above). Furthermore, analysis of a range of queries leads us to believe that such joins are relatively rare. However since we want our algorithms to be robust to and perform well in this situation, we created such joins in this database by eliding some of the join columns in the queries.

The synthetic data was generated using Zipf-like distributions that have been observed when the distribution of column values is skewed. We vary the skew in each relation by using different zipfian distributions for generating the relations. In addition the correlation between the frequent values in the relations is also varied. By varying the zipfian coefficient and the correlation, a variety of join types are simulated and the algorithms compared on all these scenarios.

6 Experimental Results

The behavior of the algorithms varies with the type of the relations being joined. As described in Section 5.3, the factors that affect the performance of the algorithms vary with the relation types.

Section 6.1 compares the three basic schemes discussed earlier: hybrid hash join algorithm with no load balancing (HASH), Centralized Load Balancing with Complete Histograms (CLBCH) and the One Stage Load Balancing algorithm (1SLB). The data used for this analysis is the real workload described above. These three algorithms represent "pure and simple" representatives of the main ideas in this paper. While many minor variations and improvements on these schemes suggest themselves, the results discussed below illustrate the key comparative effects and dependencies on the type of workload. In Section 6.2, we then repeat the analysis with synthetic data that we have created to contain a mix of the three join types. We also then add the results for the PHLB algorithm, and show how PHLB spans the space between HASH and 1SLB.

6.1 Results Using Commercial Data

The graphs (refer to Figures 1, 2, 3) are plotted for the three algorithms applied to each join type **D**x**D**, **D**x**I** and **I**x**I**. Each curve corresponds to a fixed number of storage sites (s) and traces increasing number of join sites (p). The initial partition of data across storage sites was found to affect the relative behaviour of the curves only marginally. The change in the curves with varying I/O or communication costs is not significant though noticeable. The curves therefore indicate the relative performance of the algorithms and are indicative within wide variations of other parameters like degree of imbalance (the tolerance within which the load is required to be balanced), relative cost of I/O, communication, partitioning of data at storage sites, CPU speed, etc.

6.1.1 Results for the DxD join

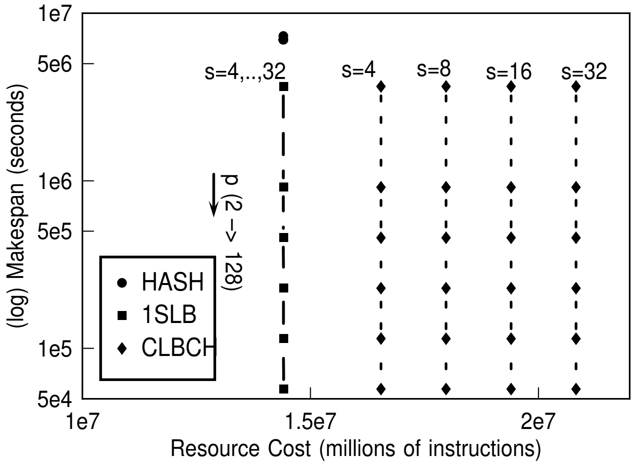

Figure 1: Results of all three algorithms for the **D**x**D** join

For this type of join, the relations have correlated duplicates and so the join result has a "quadratically large" number of duplicates. In the hashing scheme the join site that receives the join attribute value with many duplicates dominates the makespan, as it handles a large fraction of the work. Even if p is increased, this cost remains the same and that explains why the "curves" for hashing cluster together with a high makespan.

Makespan: As p increases, makespan falls for both 1SLB and CLBCH because the result is then distributed over multiple sites. Note that the result size determines makespan because it tends to be proportional to the square of the relation size. That also explains why 1SLB and CLBCH have similar makespans. All their other expenses are overshadowed.

Resource Costs: As s increases the resource costs of CLBCH increase because the cost of shipping load balancing information (histograms) is proportional to the number of storage sites. However the percentage increase is fairly small.

6.1.2 Results for the DxI join

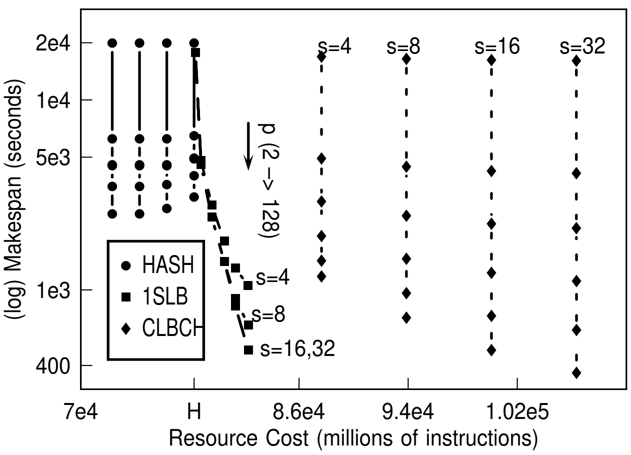

Figure 2: Results of all three algorithms for the **DxI** join

In this case, the **D** relation has some values with a large number of duplicates. The result of the join is proportional to the size of the larger relation. These observations are useful in understanding the results. Note that the curves for hashing are split for visibility even though they all appear at the X coordinate marked 'H' in the graph.

Makespan: Hashing reaches a bound quickly as p increases because the sites with maximum load also receive/join the maximum number of tuples. Hence they dominate the costs. 1SLB does better. Because it distributes tuples before doing balancing, some sites still get a disproportionate number of tuples. This effect shows up as an diminishing improvement of the 1SLB curve with increasing p. CLBCH does a little better and the makespan decreases continuously. This is because both the distribution of tuples as well as the final result is balanced. The gain however is small (note that the vertical axis is on a log scale). The decreasing incremental gain in makespan is also noticeable.

Resource Costs: The resource costs for CLBCH rise with increasing s because although the number of histograms shipped from $\mathcal{S}$ to C increases but the size of each histogram does not decrease proportionately. This is because in these experiments the tuples of R_1 and R_2 are randomly partitioned on the join attribute over $\mathcal{S}$ and so a distinct value can occur at more than one site and hence contribute to more than one histogram.

6.1.3 Results for the IxI join

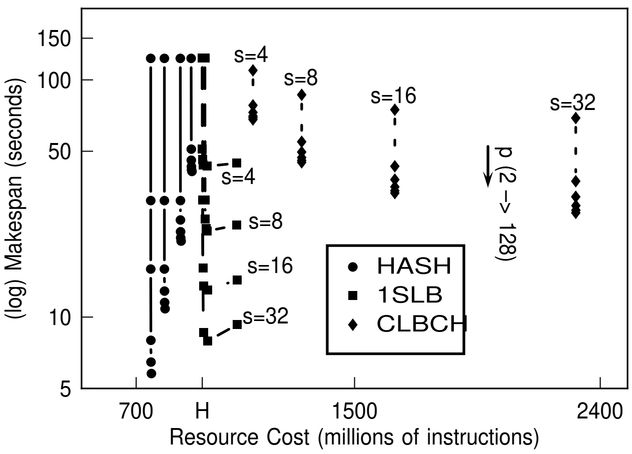

Figure 3: Results of all three algorithms for the **IxI** join

This is an example where hashing achieves a good enough distribution of work that no load balancing in needed. Consequently all load balancing costs are essentially overheads. Note that the curves for hashing are split for visibility even though they all appear at the X coordinate marked 'H' in the graph.

Makespan: The hash scheme does better than all the others because it does not incur any communication costs which add to the makespan. CLBCH does significantly worse because the size of the histograms being shipped are proportional to the size of the relations. This adds significantly to the makespan. However, as s increases, there are fewer distinct values and tuples on each storage site. The time for creating the histograms decreases. As a result the makespan decreases with increasing s. 1SLB performs just a little worse than hashing because it achieves the ideal load distribution without any tuple redistribution, and a little unnecessary coordination in the shipping of the work estimates from the join sites to C.

Resource Costs: The increase in resource costs (with s) for CLBCH is somewhat larger than 1SLB, again due to unneeded coordination at C.

Note also that the anomaly in that the makespan for 1SLB increases slightly between $p = 64$ and $p = 128$ because at that point the overhead of communication between the join sites becomes greater than the cost of the useful join work. Because increased number of join sites entail increased overhead costs, this effect shows that beyond a point it is harmful to increase the degree of parallelism. An alternative experimental methodology, in which the amount of join work is increased in proportion to p, would not be expected to exhibit this anomalous effect. A similar effect is noticeable in the hash and CLBCH curves where the points cluster close together with increasing p. As the size of the relations being joined is increased, this effect would be deferred until larger values of p.

6.1.4 Discussion

In summarizing the results, the first thing to note is that the resource cost for the hashing scheme is the lowest because it involves the least decision making. It does the bare minimum: read, ship, and join the tuples. Load balancing algorithms such as CLBCH are guaranteed to balance the load within some prescribed tolerance but these come at a resource cost of 30-50%. An algorithm such as 1SLB is more heuristic in nature, and operates with a fairly small overhead in comparison with HASH (10-30%). It also has properties such as being able to deal dynamically with processor congestion, by factoring the load on processors while doing load balancing.

6.2 Results Using Synthetic Data

We now continue our experimentation using synthetically generated data containing a mix of all three join types, and include results for the PLBH algorithm. In Figure 4 we observe that, because the workload contains joins of type **D**x**D**, HASH can be two orders of magnitude worse in makespan than a load balanced algorithm. Given the extreme penalty to be paid in makespan when this skew does arise, the balancing algorithms seem well worth the additional resource cost. The curves for PHLB lie between Hash and 1SLB as predicted earlier. This is further illustrated by Figure 5 where X is varied in PHLB for one configuration ($s = 8, p = 32$).

The appropriate choice of X depends on the query and database. As discussed above, the best value of X for a given query and database should be determined by tuning. We observe that it is possible to adjust PHLB to squeeze out most of the overhead of 1SLB: this comes about partially because of its savings in I/O during the pipelining phase. Note that all makespans are for completion of the join, and do not reflect the fact that the time until first tuple is delivered is shorter in PHLB than other algorithms. While this is an important property of this algorithm, its advantage is mainly felt in complex query and multi-join applications, and we have decided not to present these results here since that would divert us from the focus of this paper.

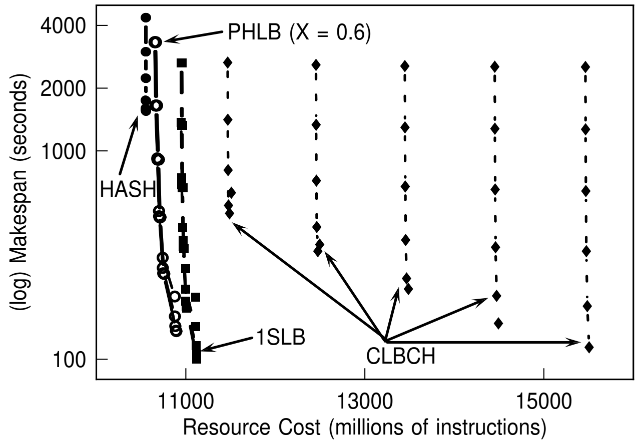

Figure 4: Results of all three algorithms for the mix of all join types

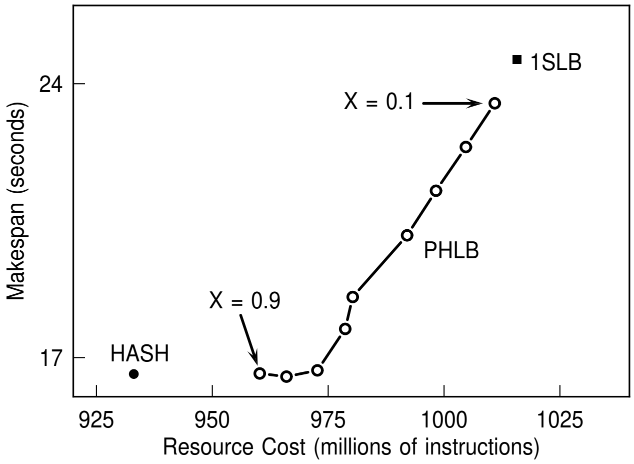

Figure 5: Varying X for the mix of all join types

7 Conclusions

In general, the simplest hash-based algorithm costs little over and above the necessary cost of reading and shipping the tuples from the storage sites. If it can be safely assumed that few duplicate values will occur, it is the algorithm of choice. However, for a marginally greater resource cost (10-30% in our experiments), robust performance can be achieved without relying on this assumption. The more involved load balancing algorithms differ in their resource costs depending on the nature of the data and the number of storage and/or join sites. Algorithm 1SLB is an attractive algorithm since it does a good job at handling skew while not incurring as much resource cost as the centralized (CLBCH) and two stage (2SLB) algorithms.

The taxonomy of joins into the types **D**x**D**, **D**x**I**, and **I**x**I** is illuminating. When there is a significant **D**x**D** effect, the complex algorithms can be dramatically better than simple hashing: two orders of magnitude in makespan at negligible cost.

We have shown that PHLB spans the spectrum from HASH to 1SLB and is easily set up to be a robust load balancing algorithm. Moreover it offers the ability to be fine-tuned for resource costs and exploited for its pipelining capabilities. This pipelining which can be important if the join operation is an intermediate operation in a more complex query. It is a simple algorithm to implement, is highly robust to the type of join workload, and is not significantly outperformed by any other algorithm considered for any of the join types.

References

[1] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):4–24, March 1990.

[2] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma Database Machine Project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):44–62, March 1990.

[3] R. Epstein, M. Stonebraker, and E. Wong. Distributed Query Processing in a Relational Database System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 169–180, 1978.

[4] M. R. Garey and D. S. Johnson. *Computers and Intractability, A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, NY, 1979.

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

[6] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. R. Kan. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[7] A. Gupta, R. J. T. Morris, A. Swami, and H. C. Young. Enhanced Hash Based Algorithms for Handling Skew in Parallel Joins. Technical report, IBM Research Division, Mar. 1992. IBM Research Report to appear.

[8] K. A. Hua and C. Lee. Handling data skew in multicomputer database systems using partition tuning. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 525–536, Barcelona, Spain, 1991. Morgan Kaufman.

[9] A. Keller and S. Roy. Adaptive Parallel Hash Join in Main-Memory Databases. In *Proceedings of the 1st International Conference on Parallel and Distributed Information Systems*, pages 58–67. IEEE Computer Society, Dec. 1991.

[10] M. S. Lakshmi and P. S. Yu. Effect of Skew on Join Performance in Parallel Architectures. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems, Austin, Texas*, pages 23–34. IEEE Computer Society, 1988.

[11] R. A. Lorie, J.-J. Daudenarde, J. W. Stamos, and H. C. Young. Exploiting Database Parallelism In A Message-Passing Multiprocessor. *IBM Journal of Research and Development*, 1992. To appear.

[12] J. P. Richardson, H. Lu, and K. Mikkilineni. Design and Evaluation of Parallel Pipelined Join Algorithms. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 399–409, San Fransisco, California, 1987.

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

[14] M. Stonebraker, P. Aoki, and M. Seltzer. Parallelism in XPRS. UCB/ERL M89/16, UC Berkeley, February 1989.

[15] C. B. Walton, A. G. Dale, and R. M. Jenevein. A Taxonomy and Performance Model of

Data Skew Effects in Parallel Joins. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 537–548, Barcelona, Spain, 1991. Morgan Kaufman.

[16] J. Wolf, D. Dias, and P. Yu. An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew. In *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 103–115. IEEE Computer Society, 1990.

[17] J. Wolf, D. Dias, P. Yu, and J. Turek. An Effective Algorithm for Parallelizing Hash Joins in the Presence of Data Skew. In *Proceedings of IEEE Data Engineering Conference*, pages 200–209. IEEE Computer Society, 1991.