An Adaptive, Load Balancing Parallel Join Algorithm*

Minesh B. Amin   Donovan A. Schneider   Vineet Singh

Abstract

Many parallel join algorithms have been proposed in the last several years. However, most of these algorithms require that the amount of data to be joined is known in advance in order to choose the proper number of join processors. This is an unrealistic assumption because data sizes are typically unknown, and are notoriously hard to estimate. We present an adaptive, load-balancing parallel join algorithm called PJLH to address this problem. PJLH efficiently adapts itself to use additional processors if the amount of data is larger than expected. Furthermore, while adapting, it ensures a good load balancing of data across the processors.

We have implemented and analyzed PJLH on a main memory database system implemented on a cluster of workstations. We show that PJLH is nearly as efficient as an optimal algorithm when the amount of data is known in advance. Furthermore, we show that PJLH efficiently adapts to use additional join processors when necessary, while maintaining a balanced load. This makes PJLH especially well-suited for processing multi-join queries where the cardinalities of intermediate relations are very difficult to estimate.

1 Introduction

The efficient computation of the join operator on multicomputers has continued to be an important research area. However, designers of parallel join algorithms are faced with two major problems: how to determine the proper number of processors to participate in a join (or each join of a multi-join query) and how to achieve a good balance of the load across the processors. We address each of these in turn.

If too few processors are chosen for a join, the processors will be over-loaded and the processing required to resolve the overflow may be very high [SD89]. On the other hand,

*Authors' affiliations: Minesh B. Amin (University of Minnesota, Minneapolis); Donovan A. Schneider and Vineet Singh (HP Labs, Palo Alto). Internet addresses: amin@cs.umn.edu, schneider@hpl.hp.com, and vsingh@hpl.hp.com respectively.

choosing too many processors incurs the overhead of the additional processors [CABK88], their lower utilization, an increased variation in the workload [LY90], and, potentially reserving important resources (memory) at each site.

The problem of accurately determining the number of processors is exacerbated by the difficulty of predicting the number of tuples that are likely to be input to a join, especially that of intermediate joins in multi-join queries. For example, [IC91] shows that, in general, cardinality estimation errors increase exponentially with the number of joins. Most systems simply pre-determine the number of processors to be used for each join operation in a multi-join query and suffer the consequences at runtime when the join sizes are much bigger or smaller than predicted. Since the performance degradation can be high if too few processors are selected, the tendency is to be safe and assign extra processors. An alternative is to fully materialize each intermediate join result in order to obtain exact cardinality information before proceeding to the next join operation in a multi-join query. The disadvantage of this approach is that the pipelining of data between joins is broken and the resulting extra processing is expensive [SD90, ZZBS93, RLM87].

We propose an algorithm called *PJLH*, *P*arallel *J*oin using *L*inear *H*ashing, to address this problem. PJLH is an adaptive, parallel join algorithm. It starts a join operation with a number of processors determined at compile or run time. However, if the capacity of these processors is exceeded, it can efficiently utilize the capacity of additional processors.

In the course of adapting to use an additional processor, some of the tuples on the existing processors are moved to the new server. An advantage of PJLH is that this process is incremental, i.e., tuples from a single server are moved to the new server. Thus, there are no expensive reorganizations in which data at all the processors is accessed. Furthermore, since the growth is one processor at a time, the number of processors is minimized. Another benefit is that while adapting to take advantage of additional resources, a good balance of load across the processors is maintained. This is the only algorithm that we are aware of that adapts itself to use additional processors without requiring any knowledge of data input cardinalities.

In essence, PJLH works by staging one of the joining relations into a distributed file and then probing this file with tuples from the other joining relation. The distributed file has the properties that it is easily addressable through a simple hashing function, it evenly balances the number of tuples on each of the processors used for the join, and it gracefully and efficiently expands to additional processors when the load on the existing ones is too high. Since this algorithm can adapt itself for any number of tuples in a join, it is well-suited for processing pipelined, multi-join queries on systems with large numbers of processors.

We have implemented PJLH and analyzed its performance on a main-memory database on a cluster of workstations. We show that PJLH performs nearly as well as an optimal static hashing algorithm when both use the same number of processors. Furthermore, we demonstrate that PJLH efficiently expands over additional sites when the capacity of the current sites is exceeded.

The remainder of the paper is organized as follows. In Section 2, we describe the PJLH algorithm and in Section 3 the static hash-join algorithm used for the comparison. Section 4 describes the performance results of these algorithms. Related work in parallel join query processing is discussed in Section 5. Section 6 concludes the paper and offers suggestions for future work.

2 PJLH — an adaptive, load-balancing parallel join algorithm

2.1 Overview

PJLH is an adaptive, parallel join algorithm. It is adaptive in that it can increase the number of join sites used for a join while the join is in progress. Furthermore, the adaptation is incremental, i.e., join sites are added one at a time. This "spreads out" the cost of adaptation, and hence minimizes the effects of adding additional join sites. Finally, the load on the join sites is kept balanced while join sites are added.

We consider the join of relations R and S, where each relation is horizontally partitioned (declustered) over a set of sites, called **home sites** [CABK88]. The actual join of R and S takes place over a set of sites, called **join sites**. In general, the set of join sites can intersect with the home sites. The general case of multi-join queries is discussed in Section 2.3.3.

PJLH works in two phases. In the first phase, called the **building phase**, a distributed file is constructed from the tuples of R. Each fragment of this distributed file is kept in the main-memory of a join site. After this phase is done, the **probing phase** is started where tuples from S probe the distributed file to produce the join result. Since the same hashing function is used to redistribute tuples in both phases, the join of R and S is broken up into a number of smaller, independent joins which are executed in parallel. A **scheduler** process controls the entire parallel join operation, including initiating the two phases and managing the evolution of the file.

PJLH was inspired by the LH* algorithm proposed in [LNS93] for the construction of the distributed file. The join sites can be thought of as **servers** for the distributed file, while the sites producing the tuples of R and S are specialized **clients**. Clients maintain a view of the file, although it is important to note that this is a "fuzzy" image for the clients of R. Since clients are not guaranteed to have a correct view of the file, servers have logic to forward tuples to the correct server site. An addressing error triggers a notification message to the client to update its view of the file in order to make fewer errors in the future.

Each fragment of tuples at a join site is called a *bucket*. Each bucket is stored in memory and is organized to support fast access by key value, e.g., a hash file. For the sake of discussion, there is one bucket per site. The file expands one bucket at a time, and only when the capacity of the existing buckets is exceeded. A **split coordinator (SC)** manages the file evolution. In PJLH, the scheduler assumes this responsibility. The distributed file can start with an arbitrary number of buckets. The full details of the algorithm are described below.

2.2 Building phase of the join

2.2.1 File Expansion

A distributed file consists of a collection of servers numbered 0, 1, 2, ... each of which stores a single bucket of the file in RAM. The servers (buckets) are addressable through a directoryless pair of hashing functions h_i and h_{i+1}; $i = 0, 1, 2, ...$ The function h_i hashes join attribute values C onto $N * 2^i$ addresses; N being the initial number of buckets, $N \geq 1$. We use the following family of functions: $h_i\ (C) \rightarrow C \bmod (N * 2^i)$.

Under insertions, the file gracefully expands, through the splitting of one bucket at a time into two buckets. The function h_i is replaced with h_{i+1} when existing bucket capacities are exceeded. A special value n, called the split pointer, is used to determine which function, h_i or h_{i+1}, should apply to a key. The value of n grows one by one with the file expansion (more precisely, it grows from 0 to $N - 1$, then from 0 to $2N - 1$, etc.). It indicates the next bucket to split and it is always the leftmost bucket still using h_i. The record with key C is always stored at the bucket with address a as defined by:

$a \leftarrow h_i(C)$;

```
if  $a < n$ then  $a \leftarrow h_{i+1}(C)$;
```

Buckets maintain the most recent level of the hash function in their header. This is used to determine whether a tuple needs to be forwarded to another bucket.

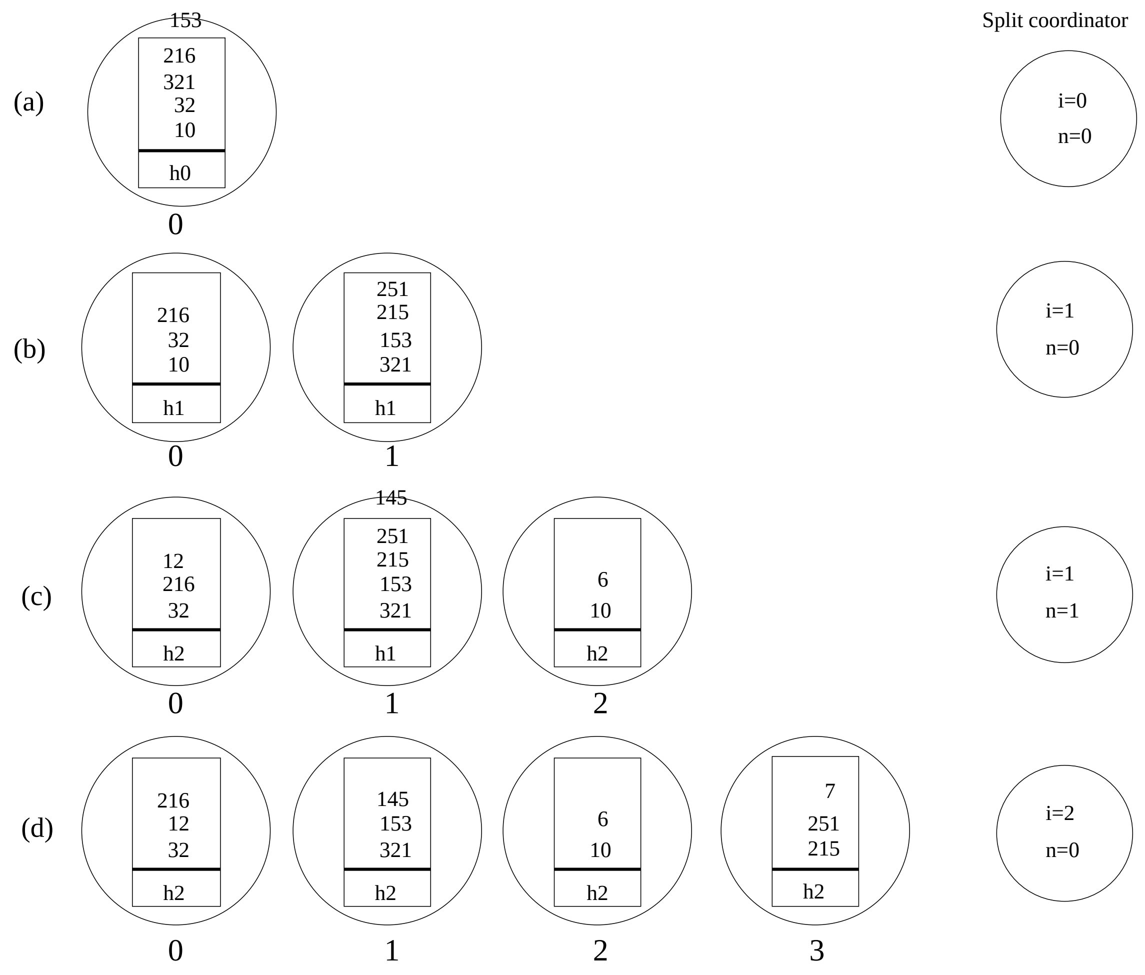

Figure 1: File expansion (bucket capacity=4).

An example of file evolution (adapted from [LNS93]) is shown in Figure 1. For this example, the bucket capacity of each join site is four and $N = 1$. The bucket header is shown at the bottom of each bucket. The state of the split coordinator is shown on the right hand side. In Figure 1(a), the first four values have been inserted into bucket 0 and the insert of 153 causes a collision. The collision triggers a split of bucket 0 into buckets 0 and 1, i.e., the values are rehashed using h_1, with the result that the odd valued records are moved to bucket 1. Both buckets are now using h_1, as their bucket headers show. Figure 1(b), shows the file after the split and the insertions of 251 and 215. Figure 1(c) shows the result of trying to insert 145 into bucket 1. This collision causes bucket 0 to split with half the records moving to bucket 2. Key 145 remains as an overflow record at bucket 1. Values 6 and 12 are then inserted. Finally, Figure 1(d) shows the final file after the insert of 7 causes another collision and thus the split of bucket 1 and the creation of bucket 3.

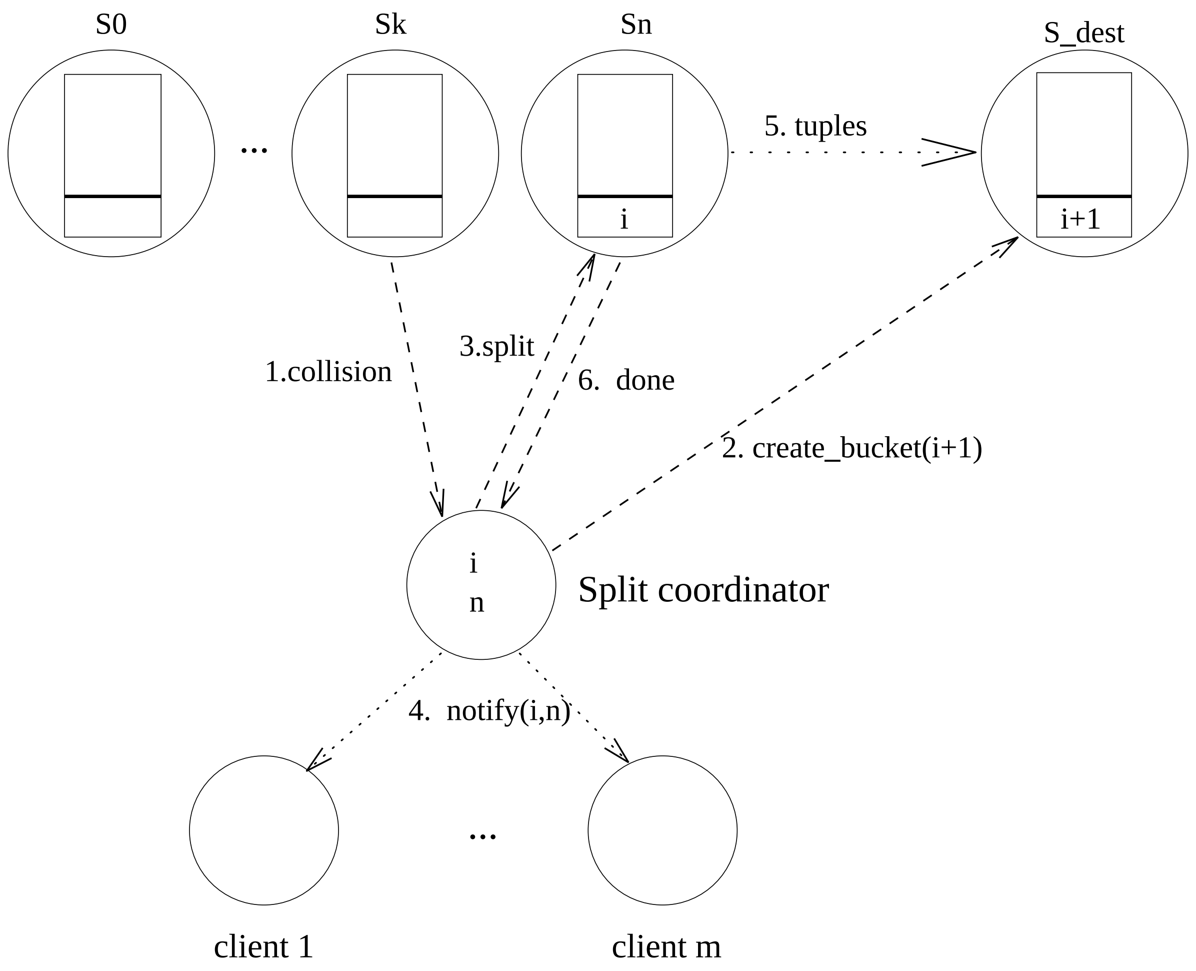

Figure 2: Splitting of bucket n.

Figure 2 shows the algorithm required to split a bucket. When a server k first overflows its capacity, it sends a *collision* message (1) to the split coordinator. The SC computes the address of the new bucket to be added to the file; this is bucket $N * 2^i + n$. The SC sends a *create-bucket* message (2) to the server at this address. This message includes the bucket level of the server — which is equal to $i + 1$. The SC sends a *split* message (3) to server n, informing it to split its bucket. This message includes the address of the new bucket. Upon sending this message, the SC updates its state of the file by:

$n \leftarrow n + 1;$

if $n \geq N * 2^i$ then $i \leftarrow i + 1;$ $n \leftarrow 0;$

Upon receipt of a split message, a server scans all the records in its local bucket and rehashes them using h_{i+1}. On average, about half of the records hash to the new bucket. These records are transferred to the new bucket (5); multiple records are packed into each message to reduce communication costs. The server also updates its header to reflect the hash function $i + 1$. Concurrently with the split, the SC informs each of the clients of the new state of the file through a *notify* message (4). In effect, the split has been "committed". The rationale for this strategy is discussed below. Finally, when bucket n has completed the split operation, it informs the SC through a *split-done* message (6).

The above picture is slightly simplified in that in PJLH buckets are allowed to split concurrently in order to support the high insertion rates required for a parallel join. The split coordinator guarantees that the buckets are split in a linear order, although multiple buckets can be undergoing a split operation simultaneously. The split pointer n is used as before to keep track of the next bucket to split. In addition, the SC has another split pointer called the *barrier split pointer*. The barrier split pointer trails the split pointer. It is advanced only when the SC receives a split-done notification from the bucket that was first told to split, from the list of buckets currently being split. For example, if buckets j, $j+1$ and $j+2$ are in the process of splitting, the barrier split pointer will be j while the split pointer will be $j + 3$. The SC advances the barrier split pointer to $j + 1$ only when bucket j informs it that its split is done. The barrier split pointer is needed to guarantee that a bucket is not told to split while it is currently performing a split. That is, the split pointer cannot loop through the file and "catch up" to the barrier split pointer. This ensures that at most two hash functions are ever active simultaneously, i.e., that a server is using bucket level i or $i + 1$.

The algorithm described above for splitting buckets differs fundamentally from that in LH*.

- In LH*, a bucket split is made visible to a client only after the split coordinator has been informed that the split has finished. This was done to ensure that a client retrieving a record from a bucket undergoing a split would always look for the record in the original bucket, and, if the record was forwarded as part of the split, the query would ultimately be forwarded also. This is not necessary in PJLH since the file is "append-only" during the building phase. Thus, by making splits visible sooner, clients make fewer addressing errors.

- In LH*, notification messages are sent in a lazy manner, i.e., as a result of a client making an addressing error. In PJLH, clients are informed of bucket splits in an eager manner. This was done because each client is inserting large amounts of data into the file, and at a high insertion rate. Thus, there is a need to inform clients of structural changes to the file as soon as possible to prevent forwarding of tuples at the join sites. Also, given the large amounts of data to insert, it is likely that each client would quickly have an addressing error which would generate a notification message and thus this optimization does not add additional messages. Furthermore, these notification messages to the clients do not have to be reliable, i.e., clients can still operate with an outdated view of the file.

- Concurrent splits are important in order to support the high insertion rates needed for parallel joins. The algorithm for concurrent splits in PJLH allows greater concurrency than other algorithms [SPW90, LNS94] by informing clients of new buckets before the transfer of tuples to the new bucket is completed. This is correct because it is guaranteed that no tuples are ever retrieved while splits are on-going, since the semantics of the join are append-only. This is the first design and implementation of concurrent splits for a shared-nothing multiprocessor.[1]

Since the file expands linearly as an LH or LH* file, the load factor is generally between 65% and 70% [Lit80, LNS93].

2.2.2 Clients of the building phase

As stated earlier, the tuples of R are initially horizontally partitioned across a set of home sites. All of these sites are clients of the initially empty distributed file, horizontally partitioned over a set of N initial join sites.

Each client maintains a fuzzy image of the file, although clients are informed of the initial N buckets by the scheduler. The client's image consists of the parameters i' and n', which

[1] [SPW90] implemented concurrent splits but the architecture of the system was a tightly-coupled multiprocessor with distributed shared-memory.

correspond to i and n kept by the split coordinator. The client's values, i' and n', can lag behind i and n. A client's image is updated via a notification message sent by the split coordinator. Upon receipt of this message, a client simply replaces the current values of i' and n' with those enclosed in the message.

The algorithm for a client during the building phase is:

1. Read each tuple r from the local fragment of R
2. Apply any local selection predicates
3. Apply the following hashing function to the join attribute C of r and send r to site a:

```
$a \leftarrow h_{i'}(C);$
if  $a < n'$ then  $a \leftarrow h_{i'+1}(C);$
```

As a performance enhancement, a client batches several inserts that hash to the same bucket and sends them using a single network message. Batching reduces communication costs and allows for a more effective utilization of the network. The tradeoff is that more tuples may be sent to the wrong join site and hence need to be forwarded, because of changes to the underlying file during the buffering process.

2.2.3 Termination of building phase

An interesting problem that arises in PJLH is detecting when the building phase has completed. This can normally be done by having the scheduler wait until all the clients notify it that they have processed their fragment of R. But a complication in PJLH is that buckets may still be undergoing splits or some client inserts may still be queued for processing at a server.

The termination algorithm is as follows. In the first phase, each client piggybacks an end-of-data signal on the last packet to each of the join sites. The client also piggybacks the total number of tuples that it has sent. When each join site receives such a signal from each of the clients, it notifies the scheduler. When the scheduler receives a notification from each of the join sites, it begins phase two. In phase two, the scheduler starts multiple rounds of queries to each of the join sites. In each round, the scheduler asks the join sites for the number of R tuples stored. When the sum of the replies reaches the expected total and no more splits are in progress, the building phase is over. The number of rounds necessary for termination is expected to be low.

The algorithm used in the first phase is called point-to-point in [AC88]. They showed that this algorithm is effective when the number of clients is small. If the number of clients is expected to be large, it would be better to use their control node strategy.

2.3 Probing phase of the join

2.3.1 Clients of the probing phase

The sites storing fragments of relation S constitute the clients of the probing phase of the join. In this phase, tuples of S are routed to a join site where they are used to probe the in-memory hash-table and produce any relevant output tuples. The same hash functions used in the building phase are used on the join attribute values of S in this phase. However, clients of S never make any addressing errors, i.e., they never route a tuple to an incorrect join site since the scheduler informs these clients of the actual state of the file built during the building phase. Hence, i' and n' are guaranteed to equal i and n.

2.3.2 Servers of the probing phase

Because clients never make any mistakes in the probing phase, as an optimization, the servers at the join sites need not re-hash the tuples to see if they should be forwarded. Thus, $|S|$ hash computations are avoided.

2.3.3 A multijoin algorithm

The PJLH algorithm as presented above described the join of two relations. The extension for multi-join queries is straightforward and is adapted from the algorithm used in Gamma [DGS+90]. Consider the k-way join of relations R_1, R_2, …, R_{k+1}, expressed as a left-deep query tree as shown in Figure 3.

The scheduler begins by assigning an initial number of join sites for each join in the query tree. (These values can come from optimizer hints or from more elaborate run-time heuristics.) Next, the join operation J_1 (which represents $R_1 \bowtie R_2$) is started, i.e., the building phase using tuples from R_1 is initiated. The scan of R_1 and the construction of the distributed file for R_1 is then run to completion as described in the previous section for two-way joins. However, instead of immediately starting the scan of R_2 and the resulting probing of the join sites as for two-way joins, the building phase of join J_2, which represents $((R_1 \bowtie R_2) \bowtie R_3)$, is first initiated and

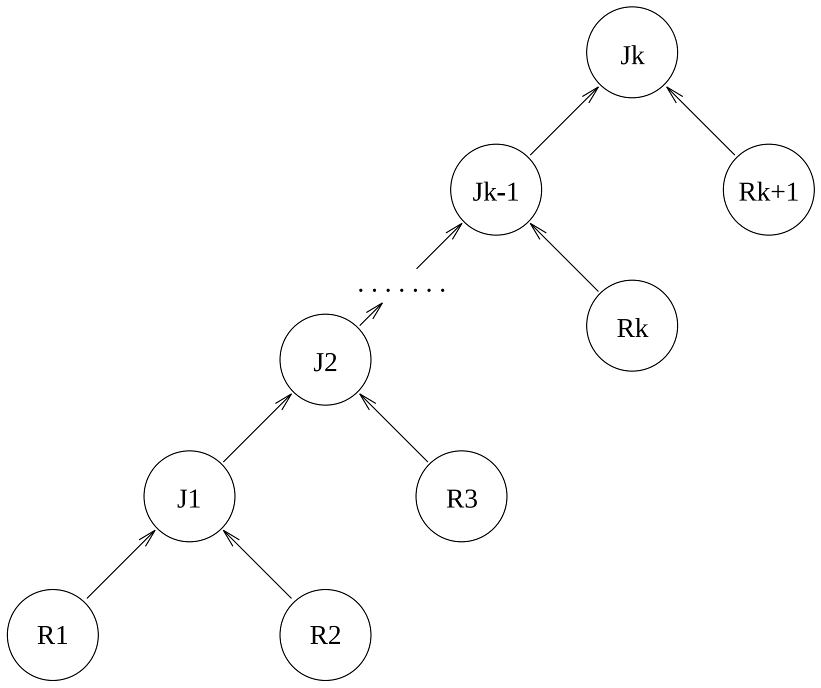

Figure 3: Left-deep tree for k-way join.

then the scan of R_2 is started. This allows an inter-operator pipeline to form between the scanning of R_2, the probing of the file built from R_1, and the building of the distributed file for the tuples resulting from $R_1 \bowtie R_2$. When all of these operations are done, execution continues similarly, i.e., the next level join is initiated, the scan of R_3 is started, the tuples probe the distributed file, and the join result tuples are staged into yet another distributed file. Execution continues in this manner until all the joins in the query are computed. Note that each join in the query can adapt to use as many processors as necessary.

Note that it is trivial to use PJLH as the join method for query plans represented in right-deep trees or bushy trees.

3 Static hashing

In order to evaluate the cost of adding join sites in PJLH, we wanted to compare it with an optimal non-adaptive algorithm. The algorithm we chose is a static-hashing (SH) algorithm where the number of join sites is determined in advance and remains constant during the join operation.

The algorithm works as follows:

1. Each home site of R reads its local fragment of R, applies a hash function, h, to the join attribute C of each tuple, and routes the tuple to join site $h(R.C)$. The result of this operation is that R is **redistributed** over the set of join sites. As tuples arrive at a join site, they are staged into a main-memory hash table. R is referred to as the **building** relation.

2. After step 1 is finished, a similar procedure is applied to the tuples in S. Note that the same hash function from step 1 must be used. As tuples of S arrive at a site, they probe the memory hash table and produce the appropriate join output tuples. S is referred to as the **probing** relation.

This algorithm has the following advantages:

- It is simple to understand and implement.
- Determining the destination of a tuple is very efficient since a single hash computation is all that is required.
- The load on the network is minimized since once a tuple is routed to a join site it is never forwarded.
- The distribution of tuples across the join sites is relatively balanced for many join attribute value distributions [SD89].

The disadvantages are:

- The number of join sites is constant regardless of the number of building tuples. If this number is too small, hash table overflow can result. This can be very expensive to resolve if the overflow is severe [DG85, SD89]. If the number of sites is too large, extra overhead is incurred for the additional sites, utilizations are low, and load imbalance is increased. Furthermore, most algorithms will pre-allocate resources (memory) for the join at each site and hence other concurrent queries may suffer.
- Performance degrades significantly if the hashing function does a poor job of balancing the tuples of R across the join sites resulting in severe hash table overflow [SD89].

Since we want the SH algorithm to be optimal, we ensure in our experiments that there is always sufficient memory such that hash tables built during the building phase never overflow.

4 Experimental results

4.1 Experimental environment

Our test environment was a shared-nothing multicomputer [Sto86] consisting of eight HP 9000 series 735 workstations on a 100 megabits/second FDDI network. Each workstation has a 124 MIPS CPU and 144 megabytes of main memory. The operating system was HP-UX 9.01. The PVM 3.2 message passing library [BDG+91] was used to simplify the programming for the cluster. PVM uses the Internet User Datagram Protocol (UDP) for messages. The experiments were run on lightly loaded workstations and network.

All relations were horizontally partitioned (declustered) over a subset of the workstations such that each site had an equal number of tuples. All relation fragments were stored in memory.

We chose the Wisconsin benchmark relations for our join experiments [DeW91]. The joinABprime query was chosen as the representative join. JoinABprime is a join of relations A and *Bprime*. The A relation consists of X tuples while the *Bprime* relation has 1/10th as many tuples. For our experiments, X is equal to 100,000 or 300,000. The join result consists of the same number of tuples as does *Bprime*. Each tuple is 208 bytes wide, except join result tuples which are 416 bytes each. Result tuples are discarded immediately after being computed. This was done to limit the amount of memory for the experiments. To be consistent with our earlier terminology, the *Bprime* relation can be thought of as relation R while A corresponds to S.

As specified by the benchmark, the join attribute values are uniformly distributed. However, our results are more general. We ran a series of simulations over many normal distributions with a wide variation of standard deviations and found that the load factors of the files were very close to that of the uniform distribution. That is, the hashing algorithms generally did an excellent job.

4.2 Results

The following parameters are used in the results. Let $J_{initial}$ be the set of join sites (servers) that PJLH starts with and J_{max} be the set with the maximum number of join sites. In our experiments, $|J_{initial}|$ varies from 1 to 7 and $|J_{max}|$ varied from 1 to 7. The number of home sites for R and S, shown as $|H|$, varied from 1 to 4, with 2 as the default. R and S were always horizontally partitioned over the same set of home sites. These sites were disjoint from the join sites. Unless stated otherwise, each home site (client) buffered ten requests before sending it to a join site.

4.2.1 Comparison to an optimal algorithm

In this set of experiments, we compared the performance of PJLH to the optimal static hashing (SH) algorithm described in Section 3. The goal was to assess the overheads of PJLH over the optimal algorithm when both algorithms use the same number of join sites. In all the tests, the tuples were evenly distributed across the join sites and the bucket capacity at each join site was sufficient to store all incoming tuples.

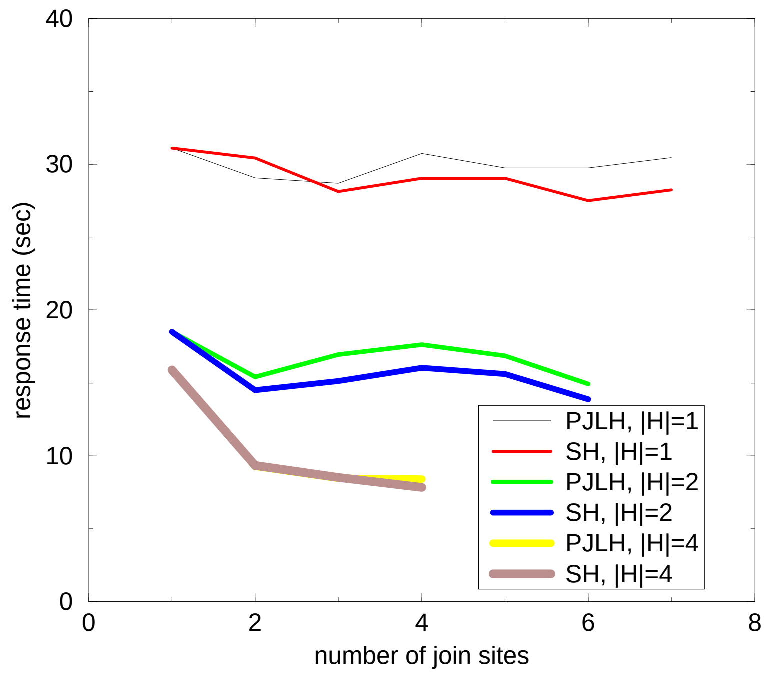

Figure 4: Performance of PJLH and SH.

Figure 4 shows the performance of PJLH and SH when the number of home sites, $|H|$, varies from 1, 2 and 4 and for a range of join sites. The cardinalities of the joining relations were 30,000 and 300,000 tuples. The main conclusion to draw from the curves is that for all values tested, PJLH adds negligible overhead as compared to SH. (For $|H| = 4$, the two curves almost coincide.) PJLH has two potential overheads over SH in these experiments. First, each tuple of R is re-hashed at a join site to see if it should be forwarded. Since there are no forwards in this situation, this adds $|R|$ extra hash computations. However, these computations occur in parallel across the join sites and can be overlapped with network communication. The second source of overhead is in detecting the end of the building phase. As the performance curves show, this time is also negligible, as expected. The one case where PJLH appears to be better than SH is due to the time-sharing nature of UNIX.

Another important conclusion from Figure 4 is that increasing the number of home sites leads to better overall performance. In fact, the speedup is nearly linear. However, increasing the number of join sites does not lead to a comparable performance improvement. This occurs because the bottleneck is in putting the data onto the network. The only exception is for the case with four home sites and a single join site. For this experiment, the four home sites produced data faster then the single join site could consume it. With two join sites, though, the producers and consumers were more evenly balanced.

4.2.2 Cost of dynamic expansion

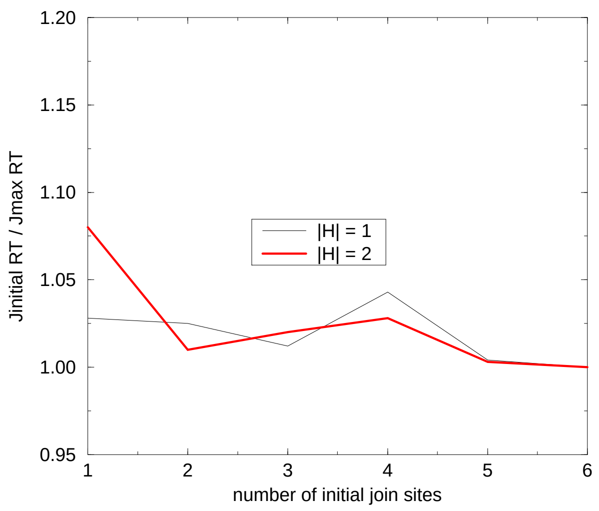

Figure 5: Cost of adaptation in PJLH ($|J_{max} = 6|$).

In this set of experiments, we quantify the cost of adding additional join sites at run-time, i.e., the cost of dynamic adaptation. For these experiments, the number of home sites varied from one to two. Each join started with 1 to 6 join sites and finished with $|J_{max}| = 6$ join sites.

The results are plotted as the time taken compared to the optimal case where $|J_{initial}| = 6$ and $|J_{max}| = 6$, i.e., where no expansion was required. The bucket capacity of each join site was set to that required to ensure no overflow for the optimal case of $|J_{initial}| = 6$.

The results are shown in Figure 5. As the figure shows, the cost of expansion is very reasonable — under 8% in all cases.

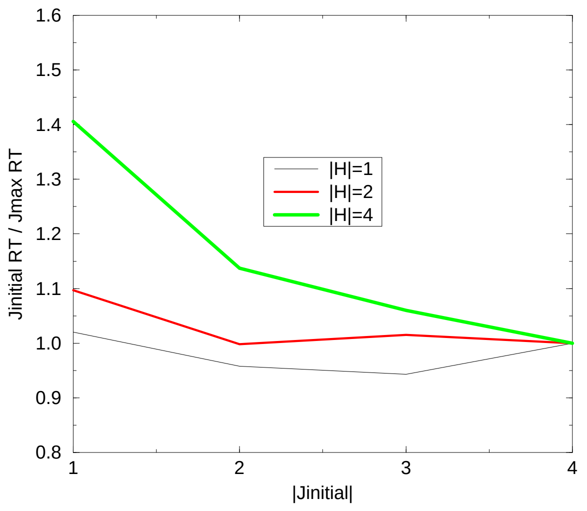

Figure 6: Cost of adaptation in PJLH ($|J_{max} = 4|$).

In Figure 6, $|J_{max}|$ is limited to four and we varied $|H|$ from 1 to 4. The results for $|H|$ equal to one and two are similar to Figure 5. However, the overheads for $|H| = 4$ are more extreme. As the figure shows, when the join starts with a single join site and expands to four join sites, the overhead is approximately 40%. The reason for this is that the tuples are sent to the join site faster than it can process them, as was discussed in the previous section. This is supported by experimental data that shows that 38% of the tuples of R had to be forwarded to a different join site because of an addressing error. With two initial join sites, the overhead of expansion is not nearly as acute. In this case, only 16% of the tuples needed to be forwarded. Less then 1% of the tuples needed to be forwarded when the file started with three join sites. Of course, no tuples were forwarded when the number of initial join sites was four.

4.2.3 Effect of client buffering

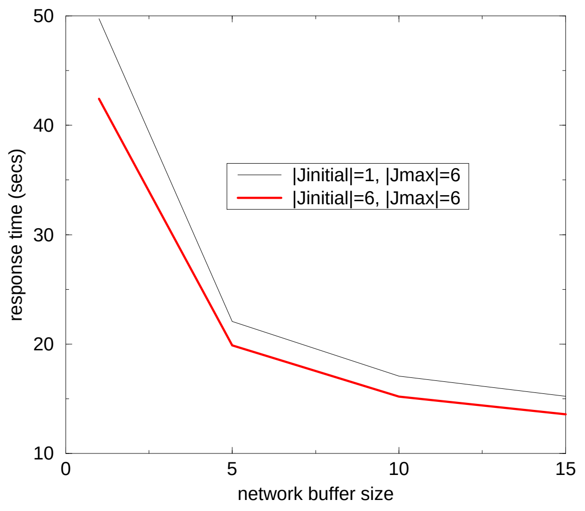

Figure 7: Effect of buffered requests ($|H| = 2$).

In this set of experiments, we measured the effect of varying the number of client requests buffered into a single network message. The degree of buffering was varied from 1 tuple per network packet to 15 tuples per network packet. The query joined relations with 30,000 and 300,000 tuples and $|H| = 2$.

Figure 7 clearly demonstrates the superiority of larger network buffers. The only potential problem with buffering is that some tuples may need to be forwarded if they are directed to an incorrect join site due to file reorganizations during the buffering process. However, as was shown earlier this is a very small cost, and, since no tuples are forwarded during the probing phase, it only occurs for the building relation.

Similar results were obtained for joins of relations with fewer tuples and for different numbers of join sites.

4.2.4 Effect of concurrent splits

In this set of experiments, we forced buckets to split one at a time in order to quantify the effect of allowing concurrent splits. $|H|$ was set to 1 and the number of join sites grew from 1 to 7. The query joined relations of 10,000 and 100,000 tuples.

Figure 8 shows that concurrent splits lead to only modest improvements. This was discouraging since we predicted that concurrent splits would lead to big performance gains. The

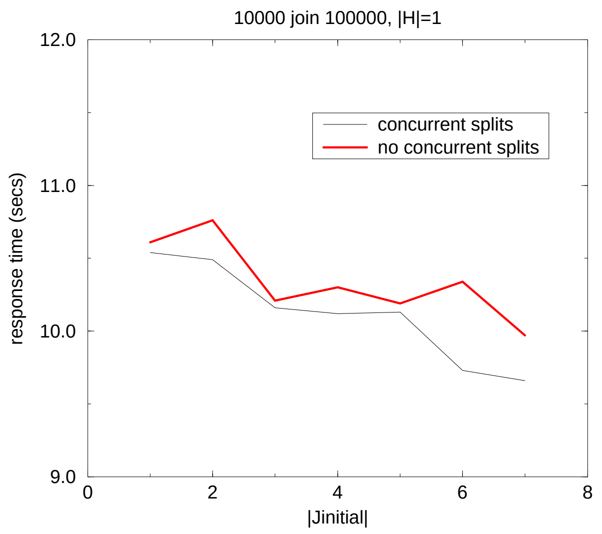

Figure 8: Effect of concurrent splits ($|H| = 1$).

explanation is that the small number of sites in the cluster limited the potential benefits of concurrent splits. That is, if the cluster had more workstations, concurrent splits would have provided a bigger advantage, as long as the bandwidth of the network is not exceeded. This reasoning is supported by some initial experiments we have conducted using a cluster of 14 SUN workstations. In this environment, concurrent splits increased performance by up to 50% for a similar query.

4.2.5 Scalability considerations

We will start with some definitions and consider two cases — one without network congestion and another with network congestion. The analysis is simple and makes some significant assumptions, which are probably valid but need to be justified in future work with more detailed analysis as in [SKAT91, KS91].

Definitions T is the sum of the cardinalities of the relations to be joined. Let H and J be the sets of home sites (clients) and join sites (servers) respectively. k_i, where i is some numeric constant, stands for a constant.

No network congestion case The sequential time to perform the join is k_1T, assuming that the hash-join component itself is linearly proportional to the size of each relation. The time for each client to send data on the network is $\frac{k_2T}{|H|}$. Since we assume that network bandwidth is not a bottleneck, the total time for all clients to send data is the same. Hash-join time is $\frac{k_3T}{|J|}$ assuming that the load is evenly distributed. The time taken for other messages needed for PJLH is $k_4|H| + k_5|J|$ since the number of messages is proportional to the same expression and each message is of constant length. This assumes that the termination algorithm takes a constant number of rounds. The time taken for redirecting data after splits or due to an incorrect hash function being used by a client is some small proportion of the time taken to send the data from the clients; we simply subsume it in the expression given above for redistribution from clients. Therefore,

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

$$= \frac{k_1T}{\frac{k_2T}{|H|} + \frac{k_3T}{|J|} + k_4|H| + k_5|J|}$$

This means linear speedup with increasing $|H|$ or $|J|$ as long as T increases at least as fast as $(|H|)^2$ or $(|J|)^2$.

Network congestion case When the network is congested, redistribution time is proportional to the size of the relations (as opposed to the case without congestion when this time gets divided by $|H|$). In other words, there is no benefit from parallelization due to multiple clients. Since sequential time is also proportional to the size of the relations, one can expect in the best case to have only constant speedup, not linear speedup as is the case without network congestion.

5 Related Work

The majority of research in parallel join query processing algorithms for shared-nothing multicomputers can be broadly classified into two categories: static partitioning schemes and dynamic partitioning schemes [ME92]. In both schemes, the relations to be joined are broken up into a series of k partitions and the original join is computed by joining each of the smaller pairs of partitions, i.e., $R_i \bowtie S_i$ for $i = 1$ to k.

In algorithms based on static partitioning, each pair of partitions is assigned to a join site based on some criteria (e.g., hashing or ranges) to be joined. The join then occurs in parallel across the join sites. Examples of such strategies include [DG85, SD89, NKT88, DNSS92, AOB93, WA93]. The main shortcoming of these techniques is that the size of the partitions may vary across the join sites, and may even exceed the available memory at some sites. This results in an expensive process to resolve the overflow and as a result a longer time to compute the join.

Dynamic partitioning schemes address this problem by balancing the size of partitions at runtime; examples of such algorithms include [KO90, HL91, WDYT91]. However, a disadvantage of both the static and the dynamic partitioning schemes is that they require the set of join sites to be known in advance and they are unable to add additional join sites at run-time. Thus, these schemes require, at a minimum, a good estimate of the cardinality of the building relation. This is a major disadvantage given the known difficulties in predicting the size of intermediate join results [IC91].

The work reported in [KR91] is most closely related to PJLH. Its dynamic partitioning scheme is similar to that of PJLH in that buckets split according to a threshold, and clients and servers maintain images of the file. Although it is not clear from the paper that additional join sites are being added when a bucket splits, it would be easy to extend it to do so. However, the algorithm requires the cardinality of the joining relations in order to distribute partitions. This limits its applicability for pipelined multi-join queries.

Although not specifically related to parallel join processing, the recent load-balancing work done as extensions to LH* is related. In [LNS94], several methods are evaluated for controlling the file load between 80-95%. Several of these do not even require a split coordinator. [VBW94] reports a strategy that achieves file loads of around 90% by allowing multiple buckets per server and by allowing buckets to migrate between servers. PJLH could be extended with similar techniques to achieve higher load factors.

6 Conclusions

We have presented a new, adaptive parallel join algorithm called PJLH. PJLH works by re-distributing one of the joining relations over a set of join sites and then probing these sites with tuples from the other joining relation. The algorithm is unique in that it can efficiently expand to include additional join sites when the capacity of the existing ones is exceeded. The expansion is incremental, thus lessening its impact on performance. Finally, a good balance of tuples is maintained across the join sites, even during the expansion process.

Since PJLH can adapt to handle relations much larger than expected it is particularly well-suited for processing multi-join queries. In such queries, it is not uncommon for join selectivity estimates to be off by several orders of magnitude. This makes it exceedingly difficult for an optimizer to assign the correct number of sites to handle each intermediate join. The advantage of PJLH is that an optimizer can be optimistic and assign relatively few sites for each join with the knowledge that if the size estimates are wrong, the algorithm will efficiently adapt. In contrast, an optimizer doing processor assignment for algorithms that cannot adapt needs to be more pessimistic and assign more processors to protect against the case where the intermediate relations are much larger than expected and the processing to handle the overflow is expensive. The cost of this strategy includes the overhead to manage the additional sites, their lower utilization, and a higher variation in workload.

Because of its dynamic nature, PJLH lends itself nicely to pipelined implementations of multi-join queries. Prior research has demonstrated the performance advantages of a pipelined query execution [CLYY92, RLM87, SD90]

We have implemented and evaluated PJLH on a main-memory database system on a cluster of workstations. In order to assess the overheads of PJLH, we compared it to an optimal static algorithm. The results show that the performance of PJLH is nearly identical to this optimal algorithm when both use the same number of sites for the join. Our experiments also show that the cost of expanding to additional sites is reasonable. It is important to note that the results also hold for disk-based database systems. In fact, the algorithm would perform better in this environment because of the greater cost of accessing the base relations.

Several interesting areas of future work remain. Currently, PJLH balances the number of tuples across the join sites. In the absence of join selectivity skew [WDJ91], the overall work of the join will be balanced. We intend to examine strategies for handling join selectivity skew by building a single distributed file with tuples of both joining relations. We also plan on exploring other strategies for controlling the load across the join sites, including strategies that do not require a centralized coordinator. Finally, it would be interesting to port the algorithm to a scalable multicomputer such as an Intel Paragon or TMC CM-5.

Acknowledgements

Witold Litwin and Marie-Anne Neimat deserve special thanks for their insightful comments and criticisms.

References

[AC88] W. Alexander and G. Copeland. Process and dataflow control in distributed data-intensive systems. In *Proceedings of ACM-SIGMOD*, June 1988.

[AOB93] B. Abali, F. Ozguner, and A. Bataineh. Balanced parallel sort on hypercube multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5), May 1993.

[BDG+91] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. A user's guide to PVM parallel virtual machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, Dec. 1991.

[CABK88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In *Proceedings of ACM-SIGMOD*, June 1988.

[CLYY92] M.-S. Chen, M. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proceedings of VLDB*, Vancouver, Canada, August 1992.

[DeW91] D. J. DeWitt. The Wisconsin benchmark: past, present, and future. In Jim Gray, editor, *The benchmark handbook for database and transaction processing.* Morgan Kaufmann, 1991.

[DG85] D. J. DeWitt and R. Gerber. Multiprocessor hash-based join algorithms. In *Proceedings of VLDB*, Stockholm, Sweden, August 1985.

[DGS+90] D.J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[DNSS92] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of VLDB*, Vancouver, Canada, August 1992.

[HL91] K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In *Proceedings of VLDB*, Barcelona, Spain, 1991.

[IC91] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proceedings of ACM-SIGMOD*, June 1991.

[KO90] 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). In *Proceedings of VLDB*,, Brisbane, Australia, August 1990.

[KR91] A. M. Keller and S. Roy. Adaptive parallel hash join in main-memory databases. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991.

[KS91] V. Kumar and V. Singh. Scalability of Parallel Algorithms for the All-Pairs Shortest Path Problem. *Journal of Parallel and Distributed Processing (special issue on massively parallel computation)*, October 1991. A short version appears in the Proceedings of the International Conference on Parallel Processing, 1990.

[Lit80] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of VLDB*, Montreal, Canada, 1980.

[LNS93] W. Litwin, M. -A. Neimat, and D. A. Schneider. LH*—linear hashing for distributed files. In *Proceedings of ACM-SIGMOD*, May 1993.

[LNS94] W. Litwin, M.-A. Neimat, and D. Schneider. LH*: A scalable distributed data structure. HP Laboratories, 1994. Submitted for journal publication.

[LY90] M. Seetha Lakshmi and Philip S. Yu. Effectiveness of parallel joins. *IEEE Transactions on Knowledge and Data Engineering*, 2(4), December 1990.

[ME92] P. Mishra and M. H. Eich. Join processing in relational databases. *ACM Computing Surveys*, 24(1):63–113, March 1992.

[NKT88] M. Nakayama, M. Kitsuregawa, and M. Takagi. Hash-partitioned join method using dynamic destaging strategy. In *Proc. of VLDB*, 1988.

[RLM87] J. P. Richardson, H. Lu, and K. Mikkilineni. Design and evaluation of parallel pipelined join algorithms. In *ACM SIGMOD*, San Francisco, CA, May 1987.

[SD89] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *ACM SIGMOD*, Portland, Oregon, June 1989.

[SD90] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of VLDB*,, Brisbane, Australia, August 1990.

[SKAT91] V. Singh, V. Kumar, G. Agha, and C. Tomlinson. Efficient Algorithms for Parallel Sorting on Mesh Multicomputers. *International Journal of Parallel Programming*, 20(2), 1991. Shorter version in proceedings of the 1991 International Parallel Processing Symposium.

[SPW90] C. Severance, S. Pramanik, and P. Wolberg. Distributed linear hashing and parallel projection in main memory databases. In *Proceedings of VLDB*, 1990.

[Sto86] M. Stonebraker. The case for shared nothing. *Database Engineering*, 9(1), 1986.

[VBW94] R. Vingralek, Y. Breitbart, and G. Weikum. Distributed file organization with scalable cost/performance. In *Proceedings of ACM-SIGMOD*, May 1994.

[WA93] A. N. Wilschut and P. M.G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1), January 1993.

[WDJ91] 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 VLDB*, Barcelona, Spain, September 1991.

[WDYT91] J. Wolf, D. Dias, P. Yu, and J. Turek. An effective algorithm for parallelizing hash joins in the presence of data skew. In *Seventh International Conference on Data Engineering*, 1991.

[ZZBS93] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel query processing in DBS3. In *Proceedings of the Second International Conference on Parallel and Distributed Information Systems*, San Diego, CA, January 1993.