Revisiting Pipelined Parallelism in Multi-Join Query Processing

Bin Liu Elke A. Rundensteiner

Department of Computer Science, Worcester Polytechnic Institute
Worcester, MA 01609-2280
(binliu | rundenst)@cs.wpi.edu

Abstract

Multi-join queries are the core of any integration service that integrates data from multiple distributed data sources. Due to the large number of data sources and possibly high volumes of data, the evaluation of multi-join queries faces increasing scalability concerns. State-of-the-art parallel multi-join query processing commonly assume that the application of maximal pipelined parallelism leads to superior performance. In this paper, we instead illustrate that this assumption does not generally hold. We investigate how best to combine pipelined parallelism with alternate forms of parallelism to achieve an overall effective processing strategy. A *segmented bushy* processing strategy is proposed. Experimental studies are conducted on an actual software system over a cluster of high-performance PCs. The experimental results confirm that the proposed solution leads to about 50% improvement in terms of total processing time in comparison to existing state-of-the-art solutions.

1 Introduction

Motivation. Many applications such as data integration services, decision support systems, and ETL middleware have their results specified in terms of complex multi-join queries across distributed data sources. Efficient processing of such multi-join queries is thus critical to the success of these applications. The evaluation of multi-join queries can take a prohibitively long time due to the following reasons: (1) the distributed nature of data sources, (2) the possibly large number of data sources, and (3) the large volume of data in each data source. Thus, there is an increasing demand for scalable multi-join query processing solutions.

Parallelizing query processing over a shared-nothing architecture, i.e., a computer cluster, has been shown to have a high degree of scale up and speed up [6]. For simplicity, we use the term *machine* to refer to each computation device in a shared-nothing architecture. Three types of parallelism have been identified in the parallel query processing [12]. First, query operators none of which use data produced by the others may run simultaneously on distinct machines. This is termed *independent parallelism*. Second, query operators may be composed by a producer and consumer relationship such that tuples output by a producer can be fed to a consumer as they get produced. This is termed *pipelined parallelism*. The third, termed *partitioned parallelism*, refers to running several instances of one single operator on different machines concurrently, with each instance only processing a partitioned portion of the complete data.

Two processing strategies at opposite ends of the spectrum, namely, *sequential* processing and *pipelined* processing, have been proposed in the literature [24]. Figure 1 illustrates these two strategies using a four-way join query $R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ on 2 machines. Here, we assume $R_1 \sim R_4$ are not in these 2 machines originally. Figure 1(a) illustrates an example of sequential processing. That is, we first evaluate $R_1 \bowtie R_2$ over 2 machines and get the intermediate result I_1. We then process $I_1 \bowtie R_3$ on the same 2 machines (indicated by the dashed rectangle) and get the intermediate result I_2. This process repeats until we get the final query results. Figure 1(b) shows an example of pipelined processing of this four-way join query. For example, we first distribute R_2, R_3, and R_4 over the 2 machines. Then, tuples read from R_1 probe these relations in a pipelined fashion and generate query results. This pipelined processing of multi-join queries has been shown to be superior to the sequential pro-

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

cessing [24]. As we will discuss shortly, state-of-the-art parallel multi-join query processing solutions tend to maximally apply this pipelined processing as its core execution strategy [3, 24, 31].

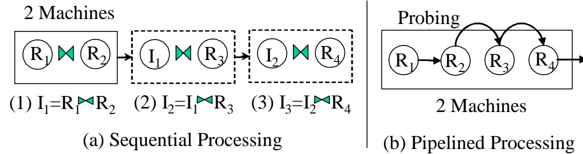

Figure 1: A Motivating Example

However, does this commonly accepted solution of maximally applying pipelined parallelism always perform effectively when evaluating multi-join queries? Or put differently, are there methods that enable us to generate even more efficient parallel execution strategies than this fully pipelined processing?

In this work, we show via a cost analysis as well as real system performance studies (not simulation) that such maximally pipelined processing is not always effective. We propose a *segmented bushy* parallel processing strategy for multi-join queries that outperforms state-of-the-art solutions.

Focus of the Work. We focus on complex multi-join queries, i.e., with 10 or more source relations. We target application scenarios in which all data will be first taken to and then processed in the cluster. This requirement of processing joins outside the data sources is rather common in many applications. For example, in a data warehouse environment (e.g., ETL [22]), operating data sources may be too busy to process such complex join queries or simply may not be willing to give control to outsiders. Data sources may also not have the advanced query processing capabilities to evaluate complex join queries, i.e., web servers.

The key research question that we propose to address is whether maximally pipelined multi-join query processing is indeed a superior solution as commonly assumed in the literature. This pipelined process implies main memory based processing. Hence, we assume that the aggregated memory of all available machines is sufficient to hold the intermediate results of the join relations. In situations when main memory is not enough to hold all join states at the same time, we follow the typical approach to divide the query into several pieces with each piece being processed sequentially. We defer this discussion to Section 5.4. The rationale behind this is that both the main memory of each machine and the number of machines in the cluster are getting increasingly large at affordable cost.

Due to possibly large volumes of data in each source relation, the main memory of one machine may not be enough to hold the full join states of one source relation. Thus, partitioned parallelism is applied to each join operation whenever it is necessary. That is, a partition (exchange) operator [11] will be inserted into the query plan to partition the input data tuples to multiple machines to conduct partitioned join processing.

We focus on hashing join algorithms [19] since they are among the most popular ones in the literature due to their proven superior performance [19, 23]. Hashing joins provide the possibility of a high degree of pipelined parallelism. Other join algorithms such as sort-merge join do not have this natural property of pipelined parallelism [23]. Furthermore, hashing joins also naturally fit partitioned parallelism.

Contributions. To highlight, the main contributions of this work include:

- We question the commonly accepted model of maximally pipelined parallelism in parallel multi-join query processing and support the claim by both an analytical argument as well as experimental observations.
- We propose a *segmented bushy* parallel processing strategy that aims to balance all three forms of parallelism for complex multi-join queries. This integration has not been fully explored in the literature. Optimization algorithms are provided to generate the above segmented bushy strategies.
- We build a distributed query engine to back up our claims. We incorporate our proposed strategies and algorithms into the system. Extensive experimental studies show that the segmented bushy parallel processing has on average a 50% improvement in terms of total processing time in comparison to state-of-the-art solutions.

The remainder of the paper is organized as follows. Section 2 describes the state-of-the-art. Section 3 discusses a multi-phase parallel optimization approach. Section 4 analyzes the cost factors and tradeoffs that affect the parallel processing performance. Section 5 presents the proposed segmented bushy processing and optimization algorithms. Experimental results are provided in Section 6. While Sections 7 and 8 discuss related work and conclusions respectively.

2 State-of-the-Art

Various solutions have been investigated for parallel multi-join query processing in the literature [3, 24, 31]. Here, we use the 10-join query depicted in Figure 2 to explain the core ideas. The query is depicted by its join graph. Each node in the graph ($R_0 \sim R_9$) represents one join relation (data source), while an edge denotes a join between two respective data sources.

2.1 Sequential vs. Pipelined Processing

Two strategies at opposite ends of the spectrum, namely, sequential processing and pipelined process-

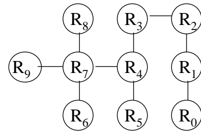

Figure 2: Example Query over 10 Relations

ing, have been proposed [24]. Note that partitioned parallelism is applied by default for each join operator. Sequential processing is based on a left-deep query tree. Figure 3(a) illustrates one example of sequential processing for the query defined in Figure 2. Here B_i represents the building phase of the i-th join operation, while P_i denotes the corresponding probing phase. This processing can be described by the following steps: (1) scan R_0 and build B_1, (2) scan R_1, probe P_1, and build B_2, (3) scan R_2, probe P_2, and build B_3, and so on. This is repeated until all the join operations have been evaluated. As can be seen, it processes joins sequentially and only partial operations, namely, the probing and the successive building operations, are pipelined.

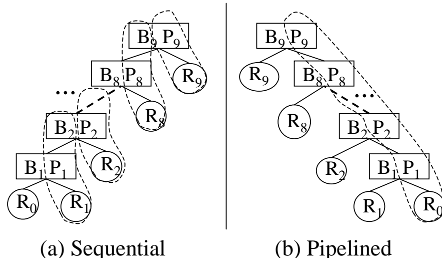

Figure 3: Sequential vs. Pipelined

Pipelined processing is based on a right-deep query tree [24]. Figure 3(b) illustrates an example of pipelined processing for the same query in Figure 2. In this case, all the building operations such as scan R_1 and build B_1, scan R_2 and build B_2, ..., scan R_9 and build B_9 can be run concurrently. After that, the operation of scan R_0 and all the probing operations, probe P_1, probe P_2, ..., probe P_9 can be done in a pipelined fashion. As demonstrated above, it achieves fully pipelined parallelism.

A pipeline process implies main memory based processing [1]. That is, it requires there to be enough main memory to hold all the hash tables of the building relations ($R_1 \sim R_9$ in this case) throughout the duration of processing the query.

As identified in [24], pipelined processing is preferred *whenever main memory is adequate*. This is because (1) intermediate results in pipelined processing exist only as a stream of tuples flowing through the query tree, and (2) even though sequential processing in general may require less memory, this is not always true due to intermediate results have to be stored. A large intermediate result may consume even larger memory than the sum of all building relations.

The simulation results in [24] confirm that pipelined processing (right-deep) is more efficient than sequential one (left-deep) in most of the cases they considered. Without loss of generality, we thus associate *pipelined* processing with a *right-deep* query tree, and *sequential* processing with a *left-deep* query tree in the following discussions.

2.2 Maximally Pipelined Processing

State-of-the-art parallel multi-join query processing solutions maximally pursue the above pipelined parallelism to improve the overall performance [3, 24, 31]. If the main memory is not enough to hold all the hash tables of the building relations, they commonly take the approach of dividing the whole query into "pieces", with the expectation that the building relations of each piece fit into the main memory. That is, pieces are processed one by one with each piece utilizing the entire memory applying fully pipelined parallelism.

For example, zigzag processing [31] takes a right-deep query tree and slices it into pieces based on the memory availability. Figure 4(a) illustrates an example that the right-deep tree (Figure 3(b)) is cut into two pieces, one is $R_0 \sim R_3$, and the other is $I_1, R_4 \sim R_9$ (Figure). I_1 denotes the result of the piece $R_0 \sim R_3$. These two pieces are processed sequentially with fully pipelined parallelism in each piece.

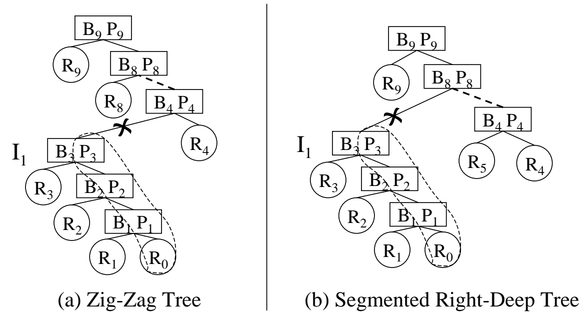

Figure 4: ZigZag and Right-Deep Segment

Segmented right-deep processing [3] uses heuristics, namely, balanced-consideration and minimized-work, to generate pieces directly from the query graph based on the memory constraint. The query tree is similar to the zigzag tree. However, each piece can be attached not only at the first join operation of the next piece, but instead also in the middle of it. For example, Figure 4(b) illustrates one example of segmented right-deep processing. As can be seen, the output (from P_3) is attached as the building relation of B_8.

To summarize, all the above approaches take the common model of pursuing a maximally pipelined pro-

[1] The term main memory henceforth denotes the sum of memory of all machines in the cluster unless otherwise specified.

cessing of multi-joins via a right-deep query tree, with the number of join relations in the right-deep tree primarily being determined by the main memory available in the cluster.

In this work, we question the view of always selecting such a maximally pipelined processing model. This pipeline process implies a main memory based processing. Clearly, more efficient main memory based processing strategies would lead to an improved overall performance. Without loss of generality, we use the term **pipelined segment** to refer a right-deep query tree that can be fully processed in the main memory.

3 A Multi-Phase Optimization

Multi-join query optimization is a complex process [28]. Parallel multi-join query optimization is even harder [9, 14, 25]. We take a *multi-phase* optimization approach to cope with the complexity of parallel multi-join query optimization. That is, we break the optimization task into several phases and then optimize each phase individually. Such multi-phase approach enables us to focus our attention on the research task we are tackling.

Breaking the Optimization Task. We divide the optimization task into the following three phases: (1) generating an optimized query tree, (2) allocating query operators in the query tree to machines, and (3) choosing pipelined execution methods. The complexity of each phase, i.e., phases (1) and (2), still remains exponential in the number of join relations.

As can be seen, the main focus of this work relates to query tree shapes (phase (1)) and different forms of parallelism on the overall performance. To proceed, we first describe the design choices we will assume in the reminder of our work for phases (2) and (3) below.

Allocating Query Operators. Resource allocation itself is a research problem of high complexity that has been extensively investigated in the literature [10, 15, 18]. Like most work in parallel multi-join query processing [3, 24, 31], we focus on main memory in the allocation phase. This is because main memory is the key resource in the above join processing. Other factors such as CPU capabilities of machines are assumed to have less impact on the allocation, i.e., they are often assumed to be sufficient.

The allocation is performed based on pipelined segments to promote the usage of pipelined parallelism [18]. For example, if a right-deep tree is cut into pieces with each piece being processed sequentially due to insufficient memory, then all machines are allocated to each piece. Thus, the whole allocation is performed in a *linear* fashion. As it can be seen, all previous processing strategies described in Section 2 fall into this type of *linear allocation.*

Pipelined Execution Method. The building relations of each pipelined segment can entirely fit into the memory of the machines that have been allocated to it. We apply a *concurrent execution* approach [24] to process a pipelined segment [2]. In this execution method, all scan operations are scheduled concurrently. For example, in Figure 5, we process a 4 way pipelined segment on 3 machines. Each building relation ($R_2 \sim R_4$) is evenly partitioned across all 3 machines. Thus, each machine houses the appropriate partitions from all building relations, denoted as P_i^j. Here, subscript i ($2 \leq i \leq 4$) denotes join relations, while superscript j ($1 \leq j \leq 3$) represents machine ID. The probing relation (R_1) is also partitioned into all 3 machines to probe the appropriate hash tables to generate results.

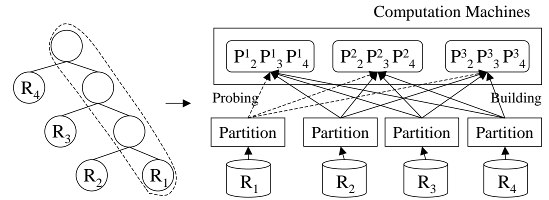

Figure 5: Fully Concurrent Execution

4 Cost Analysis of Pipelined Segment

4.1 Identifying Tradeoffs

The following two factors need to be considered when analyzing the performance of parallel multi-join query processing via a partitioned hashing: (1) redirection costs between join operations, and (2) optimal degree of parallelism.

Redirection Costs. The basic idea behind the partitioned hashing join is that the join operation can be evaluated by a simple union of joins on individual partitions. For example, an equi-join $A \bowtie B$ can be computed via $(A_1 \bowtie B_1) \cup (A_2 \bowtie B_2) \ldots \cup (A_n \bowtie B_n)$ if A and B are divided into n partitions ($A_1 \sim A_n$, $B_1 \sim B_n$) by the same hash function. Assume two partitions in a pair (A_i, B_i) are put in the same machine, while different pairs are spread over distinct machines. This way, all pairs can be evaluated in parallel.

However, for a right-deep segment, it is not possible to always have all the matching partitions reside in the same machine. For example, assume a query tree is defined by "A.A_1 = B.B_1 and B.B_2 = C.C_1". A and B are partitioned based on their common attribute A.A_1 (or B.B_1), while C has to be partitioned based on the common attribute between B and C, namely, B.B_2 (or C.C_1). If we assume A is the probing relation, then the

[2]Other pipelined execution strategies such as *staged partitioning* [3] have also been proposed. The detailed discussion of these strategies and their impact on parallel processing strategies are beyond the scope of this paper.

partition function of B.B_2 has to be re-applied to the intermediate result of $A_i \bowtie B_i$ to find the corresponding partitions C_i. However, this corresponding partition C_i might exist in a machine different from where the current B_i resides. Thus redirection of intermediate results is necessary in this situation. For the special case of a right-deep tree when only one attribute per source relation is involved in the join condition, i.e., "A.A_1 = B.B_1 = C.C_1", the same partition function can be applied to all relations. In that case, all the corresponding partitions can be put into the same machine to avoid such redirections. Such redirection affects the probing cost of the query processing.

Optimal Degree of Parallelism. Startup and coordination overhead among machines will counteract the benefits that could be gained from parallel processing. [21, 29] discuss the basics on how to choose the optimal degree of parallelism for a single partitioned operator, meaning the ideal number of machines that need to be assigned to one operator. As one example, if a relation only has 1,000 tuples, it may not be worthwhile to have it evenly distributed across a large number of machines (i.e., 100) since the startup and coordination costs among these machines might be higher than the actual processing cost. Given the processing of more than one join operator (i.e., a pipelined segment), we expect this factor has a major impact on the overall performance. That is, it affects the building phase cost of the query processing.

4.2 Pipelined Processing Cost Model

We now further provide cost functions to analyze the factors that affect the pipelined processing performance. For pipelined processing of a right-deep segment, the cost in terms of total work versus the overall processing time may not be that closely correlated. We thus derive two separate cost models. To facilitate the description of cost models, we assume R_0 is the probing relation, while $R_1, \ldots, R_n$ are the building relations of the pipelined segment. We also assume k machines are available to process the pipelined segment. These machines are denoted by $M_1, M_2, \ldots, M_k$. Without loss of generality, we use I_i to represent the intermediate result after joining with R_i. For example, I_1 denotes the result of $R_0 \bowtie R_1$, while I_2 represents $I_1 \bowtie R_2$. Thus I_n represents the final output of these joins.

Estimating Total Work. The total work of pipelined processing can be described as the sum of the work in the building phase (W_b) and the work in the probing phase (W_p), as listed below.

$$W_b = (t_{read} + t_{partition} + t_{network} + t_{build}) * \sum_{i=1}^{n} |R_i|$$

$$\begin{aligned} W_p &= (t_{read} + t_{partition} + t_{network} + t_{probe}) * |R_0| \\ &+ \frac{k-1}{k} * \sum_{i=1}^{n-1} |I_i| * t_{network} + (\sum_{i=1}^{n-1} |I_i|) * t_{probe} \end{aligned}$$

t_{read}, $t_{partition}$, $t_{network}$, t_{build}, and t_{probe} in the above formulae represent the unit cost of reading a tuple from a source relation, partitioning, transferring the tuple across the network, inserting the tuple into the hash table, and probing the hash tables respectively. They represent the main steps involved in a partitioned hash join processing. In the probing phase work, $\frac{k-1}{k} * \sum_{i=1}^{n-1} |I_i| * t_{network}$ denotes the redirection cost assuming the redirection occurs after each join operation and the output of each join operation is uniformly distributed across all the machines. The cost of outputting the final results is omitted since it is the same for all processing strategies.

Estimating Processing Time. Similarly, estimation of the processing time can be divided into two parts: one, the hash table building time (T_b) and two, the probing time (T_p). The building time of the pipelined processing T_b can be estimated as follows:

$$T_b = \max_{1 \le i \le n} (t_{read} + t_{partition} + t_{network} + t_{build}) * \frac{f(k)}{k} * |R_i|$$

The processing time of the building phase can be estimated as the maximal building time of each individual relation over k machines. Here, f(k) represents the contention factor of the network since the more machines are involved, the more contention of the network caused by transferring tuples of join relations arises. This is used to reflect the optimal degree of parallelism as discussed in Section 4.1.

The processing time of the probing phase is more difficult to analyze because of the pipelined processing. We use the following formula to estimate the pipeline processing time.

$$T_p = I_{setup} + \frac{W_p}{k} + I_{delete}$$

Here I_{setup} represents the pipeline setup time, while I_{delete} denotes the pipeline depletion time. The steady processing time of the pipeline can be estimated by the average processing time of one tuple ($\frac{W_p}{|R_0|}$) multiplied by the number of tuples ($|R_0|$) that need to be processed over the total of k machines. Clearly, this is a simplified model representing the ideal steady processing time without including for example variations in the network costs.

From above analysis, we can see that both the number of building relations (n) and the number of machines (k) assigned to the pipeline play an important role in the overall processing time. As we will discuss in Section 5, we investigate to break both n and k , and compose smaller pipelined segments to query trees to

improve the query processing performance. Note that above cost model is also applied to find the most efficient pipelined processing strategies for each subgraph (Section 5.3).

5 Breaking Pipelined Parallelism

5.1 Bushy Trees and Independent Parallelism

Query trees of a multi-join query can be classified into two types: sequential trees (i.e., a right-deep tree or a left-deep tree as discussed above), and bushy trees. A right-deep tree has a better performance over a left-deep tree since it has a high potential of pipelined parallelism for a hash-based join algorithm. Thus we now use a right-deep tree as the representative of sequential trees (e.g., Figure 6(a)).

A bushy tree has a height of at least $log_2 n$ (given a binary bushy tree that is balanced) with n being the number of join relations involved in the multi-join query. A bushy tree brings new flexibility to the style of processing, such as having multiple probing relations and composing different pipelined segments. Moreover, a bushy tree has the potential of processing independent subtrees (segments) concurrently. However, such flexibility may also bring dependencies to the execution. This dependency may both affect the allocation of query operators and the corresponding parallel processing performance.

For example, Figure 6(b) illustrates one bushy tree and its possible pipeline segments (each pipeline segment is denoted by one dashed oval). Four segments (P_1~P_4) can be identified. As can been seen, P_1 and P_3 can be processed in parallel by processing them on different machines. While the execution of P_2 depends on P_1, the execution of P_4 depends both on P_2 and P_3.

As can be seen, a right-deep tree has the highest degree of pipelined parallelism without any dependencies because each subtree is a join relation. However, there is no opportunity for independent parallelism except during the initial building phase of the join relations. While a wide bushy tree has many subtrees, it also has up to $log_2 n$ layers of dependencies with n being the number of source relations. These dependencies are likely to impact the overall performance.

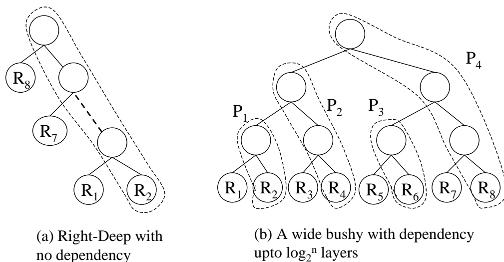

Figure 6: Right-Deep vs. Wide Bushy Tree

5.2 Segmented Bushy Tree

Seen from the cost model, if the results of pipelined segments in a bushy tree are smaller than those of the original join relations, then the bushy tree processing may have less total work ($W_b + W_p$) when compared with the fully right-deep processing. Here we assume all the intermediate results are kept in main memory.

Comparing the overall processing time of fully right-deep and bushy trees is more complicated. As we can see, each pipelined segment in a bushy tree only gets one portion of the total available machines. Thus the network contention ($f(k)$) in the building phase may be less severe than that of the full right-deep case. Moreover, less building relations in the pipeline reduces the variations in the building phase. As a consequence, given the independent processing of these smaller pipelined segments, the processing time of a bushy tree may be better than that of fully pipelined processing. However, as we identified earlier, a bushy tree style processing may be affected by the dependencies among subtrees. Moreover, there may be subtrees (up to $\lceil n/4 \rceil$) that have short pipelined processing. For example, P_1 and P_3 only have a pipeline of one probing followed by the building for the next join. These two factors may eventually counteract the benefits gained by introducing the independent parallelism.

Thus, the key question now is how to balance independent parallelism and pipelined parallelism in parallel multi-join query processing. By reducing each pipelined segment (i.e., identified by dashed oval in Figure 6(b)) into one 'mega-node', we can build a dependency tree out of the original query tree. We note that the dependencies are associated with the height of this dependency tree. Thus reducing the height of the dependency tree should effectively reduce the dependencies. We thus propose to utilize a *segmented bushy* query tree. A segmented bushy tree can be controlled to have a dependency tree with height of 2 as long as we increase the number of subtrees of the root node.

Figure 7 illustrates the example of a segmented bushy tree of the join query in Figure 6. In this example, the whole query is cut into three groups, $R_1 \sim R_3$, $R_4 \sim R_7$, and R_8. Three pipelined segments P_1, P_2, and P_3 can be identified correspondingly. P_1 and P_2 can be processed independently, each with pipelined parallelism. The output from these two segments can be directly fed into P_3. Without loss of generality, the pipelined segment that contains outputs of all other segments is referred to as the *final pipelined segment*. In this case, P_3 is the final pipelined segment. Thus, all pipelined segments except the final one can be executed concurrently without any dependencies. We can see that a segmented bushy tree processing applies independent parallelism with minimal dependencies among subtrees (groups) since it only has one layer of dependencies among pipelines.

Without loss of generality, we always assume the

right-most pipeline of a segmented bushy tree to serve as the probing relation of the final pipelined segment. For example, P_1 is the probing relation of the final segment P_3 in Figure 7.

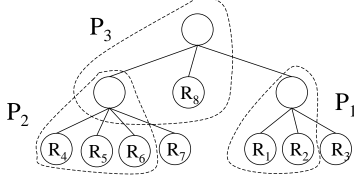

Figure 7: A Segmented Bushy Tree

The cost of above segmented bushy tree processing can be modeled based on the pipeline cost functions discussed in Section 4.2. We omit it here due to the space limitation. Readers can refer to [16] for details.

5.3 Composing Segmented Bushy Trees

Now, we address the question how to generate the segmented bushy tree for a multi-join query. Algorithm 1 sketches our proposed algorithm. It consumes a connected join graph **G** and the maximal number of nodes m per group (we will discuss how to get this m shortly). We would choose the largest join relation as the probing relation of each group since this reduces the time and the memory consumption of the building phase. Once we select the probing relation, we then begin to enumerate all possible groups having a maximum of m join nodes starting from this probing relation. Enumeration is possible since m is usually much smaller than the number of nodes in the join graph. Some of the groups may not contain exactly m nodes due to the nodes in the group being no longer connected by a join edge. Our goal is to avoid Cartesian products given that each data source may be large, thus resulting in huge intermediate results. After that, we choose the best graph, a partition of the original join graph, from these candidates generated from the enumeration based on the cost model we developed in Section 4.2. The selection can also be based on heuristics, i.e., choosing the group in which the join attributes are the same to reduce the possible redirection costs, or selecting the one with the smallest output results.

Figure 8 illustrates how the example join graph depicted in Figure 2 is divided by applying Algorithm 1 when $m = 4$. For example, we start from the relation with largest cardinality, say relation R_7. The enumeration in Step 4 generates all the possible connected groups with 4 nodes starting from R_7, as illustrated in Figure 8(a). In this case, we choose R_7, R_9, R_6, and R_8 as the nodes in the first group (pipelined segment). For simplicity, we call this group G_1. After this, if R_1 is the one with the largest cardinality among the nodes that have not yet been grouped, we then choose

Algorithm 1 ComposeBushyTree(G,m)

Input: *A connected join graph* ***G*** *with n nodes. Number m specifies maximum number of nodes in each graph.* ***Output:*** *Segmented bushy tree with at least* $\lceil n/m \rceil$ *subtrees.*

```
1: completed = false
2: while (!completed) do
3:    Choose a node V with largest cardinality that
      has not yet been grouped as probing relation
4:    Enumerate all subgraphs starting from V with
      at most m nodes
5:    Choose best subgraph, mark the nodes in this
      group have been selected in original join graph
6:    if !(∃K, K is a connected subgraph of G with
      unselected nodes) && (K.size() ≥ 2) then
7:       completed = true
8:    end if
9: end while
10: Compose a segmented bushy tree from all groups
```

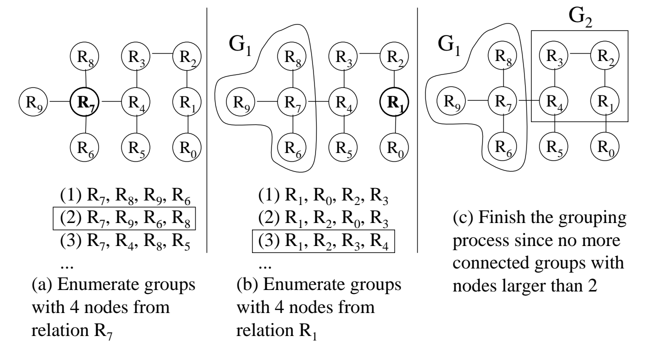

Figure 8: Example of Algorithm 1

R_1 as the probing relation for the second group G_2. We repeat the process as illustrated by Figures 8(b)-(c). After these steps, only R_0 and R_5 are left. They are not connected. We thus end up with 4 groups. An example segmented bushy tree with these 4 groups can be built as shown in Figure 9(a).

Allocating machines to a segmented bushy is based on the number of building relations in each pipelined segment. For example, for the segmented bushy tree shown in Figure 9(a), three pipelined segments can be identified (dashed cycles in Figure 9(b)). The number of machines assigned to each pipelined segment, denoted by k_1, k_2, and k_3, can be computed as follows.

$$
\begin{aligned}
N_b &= \sum_{0 \le i \le 9, i \ne 1,7} |R_i| + |I_1| \\
k_1 &= \lfloor \frac{(|R_6| + |R_8| + |R_9|)}{N_b} \rfloor \\
k_2 &= \lfloor \frac{(|R_2| + |R_3| + |R_4|)}{N_b} \rfloor \\
k_3 &= k - k_1 - k_2
\end{aligned}
$$

Here, I_1 and I_2 denote the outputs of groups G_1 and G_2 respectively. N_b represents the total number of tuples that need to be built assuming R_7, R_1, and I_2 are the probing relations of G_1, G_2, and the final pipelined segment respectively. Note that the selection of the probing relation for the final pipeline segment is not straightforward. We will discuss this in more detail in Section 6.4.

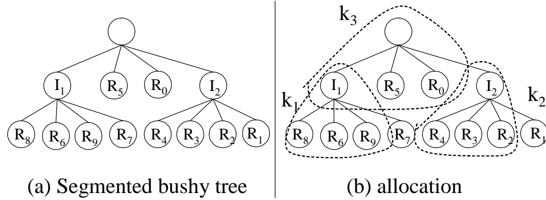

Figure 9: Segmented Bushy and Node Allocation

However, the question remains how to decide the right number of groups given a join graph. Let us now use g to represent this number. Note that the input of Algorithm 1, the maximum number of nodes in each group m can be estimated by $m = \lceil n/g \rceil$ with n being the number of join relations in the query. There are two ways to address this issue. The first is a heuristics-based selection approach. For example, we can choose g as the number of nodes that have cardinality larger than 3/2 of the average cardinality. Here, we assume that g has to be bound within $2 \sim n/2$. The rationale behind this selection criterion is that in the best case, we can choose all these large join relations as the probing relations for the generated groups. The second is a cost-based selection approach. Again we note that the range of the number of groups g is between 2 to $n/2$ [3]. We thus can repeatedly call the function *ComposeBushyTree* (Algorithm 1) with the number m ranging from $n/2$ to 2 (g changes from 2 to $n/2$ correspondingly). We then estimate the cost of processing strategy from *ComposeBushyTree*. The final output will be the one with the best estimated cost. While this may increase the optimization cost, it has the potential to result in a better processing strategy.

5.4 Handling Insufficient Memory

The problem of handling insufficient memory can be addressed using the "cutting" principle as in [3, 24]. That is, we divide the whole query (joins) into pieces such that each piece can be run in the main memory. Note that in the extreme case, the multi-join query processing would have to be sequentialized due to not enough memory being available to hold more than one join. As we mentioned in Section 1, we assume that the aggregated memory can hold at least 2 or more building relations.

Algorithm 2 sketches an incremental approach to address this problem. This incremental approach is based on the static right deep tree [24] or segmented right-deep tree [3] which divides the join query into right-deep segments based on the main memory of the cluster. After that, we further compose each right-deep segment into a segmented bushy tree if it is necessary, i.e., the number of building relations in each piece is larger than a certain threshold. Since each right-deep segment is likely to be more efficiently processed, the performance of the whole query is also expected to be better than the static right-deep or segmented right-deep tree processing.

Algorithm 2 SimpleIncSegTree(G,M)

Input: *A connected join graph* ***G*** *with n nodes, total main memory of cluster* ***M***. ***Output:*** *A sequence of segmented bushy trees, each processable in main memory of cluster.*

```
1: Compose Static or Segmented Right-Deep Tree
2: for each right-deep segment r do
3:    m ← Maximal number of relations per group
4:    t ← ComposeBushyTree(r,m)
5:    Put t into result sequence
6: end for
7: Return result sequence
```

A "top-down cut" approach, dividing the join graph directly such that each group can be processed in the main memory, can also be devised. We then select the groups and process them iteratively. However, as mentioned earlier, the essence of our work is to re-examine the performance of a main memory based maximal pipelined processing. We argue that having a more efficient main memory based processing strategies will also lead to improved overall performance even if we apply a simple incremental optimization algorithm such as Algorithm 2. This claim is confirmed by our experimental studies discussed below.

6 Experiments

6.1 Experimental Setup

We have implemented a distributed query engine to test out our hypothesis. The system is implemented using Java. It is capable of optimizing and executing multi-join queries across a set of shared nothing machines connected by network. Due to space limits, we ask readers to refer our technical report [16] for details about the system. Multi-join queries are processed on a cluster composed of 10 machines. Each machine in the cluster has dual 2.4GHz Xeon CPUs with 2GB RAM. They are connected by a private gigabit ethernet switch. In our experimental setting, all source (join) relations are stored in an Oracle database that

[3]In extreme cases, the actual number of groups may be larger than n/2. However, we assume that we have less interests in these cases when a large number of groups with only one join relation in it.

resides in a different machine outside the cluster having 2 PIII 1G Hz CPUs and 1G main memory. The query results are sent to an application server with one PIII 800M Hz CPU and 256M Memory. This setup follows a typical data warehouse environment where the process has to be performed outside the data sources [5]. This is because the operating data sources may be too busy to process complex join queries or even simply may not be willing to give control to the outsiders.

As done in [3], we use generated data sets and queries in our experiments. This is because benchmark queries such as TPC-H [27] only have a limited number of queries (around 20), and most of them have less than 5 joins. The multi-join queries used in the experiments are randomly generated with the number of join relations ranging from 8, 12, to 16. We actually generate random connect acyclic graphs given a specified number of nodes. Each node represents join relations, while each edge denotes the join condition. The average join ratio of each join is set to 1, which means each probe tuple is expected to produce on average one output tuple. The cardinality of each join relation ranges from 1K $\sim$ 100K tuples, and the average size of each source tuple is about 40 bytes. Each result tuple has about 320 $\sim$ 640 bytes on average, by simply concatenating all tuples from join relations. Thus, the whole data in one test query (including intermediate results) can go up to 600MB. Data size in our experiment is chosen to make sure all the hash tables can fit in the main memory since our main focus of this work is the main memory based processing. Here, each building relation is evenly distributed to all machines that assigned to the segment that this relation belongs to.

6.2 Impact of the Number of Data Servers

Initial experiments have been conducted to evaluate the impact of the number of Oracle data servers in the experimental setup on the overall performance. We compare the performance of multi-join queries using a pure right-deep tree (pipelined) processing given different numbers of data servers. The test queries are generated randomly with 8 $\sim$ 16 join relations. For each query, we vary the number of data servers from 1 to 4. Thus, if we have i data servers with $1 \leq i \leq 4$ and k (either 8, 12, or 16) join relations, then we have each data server hold on average $\lceil k/i \rceil$ join relations. These data servers are deployed on different machines with similar configurations having Oracle 8i installed. Each data point in Figure 10 reflects an average of 50 randomly generated queries for each query type (queries have the same number of join relations). Here, x-axis denotes the number of join relations in the query, while y-axis represents the total processing time. From Figure 10, we see that the number of data servers in the system only has a minor impact on the overall performance. This is because the total time spend on reading the tuples from data servers only represents a small fraction of the total query processing time in our current experimental settings. Thus, the improvement due to shared read by multiple data servers does not play a major role in the overall performance. This indicates that the data server is not the bottleneck in our experimental environment. Without loss of generality, we report our following experimental results with a setup that stores all join relations in one data server.

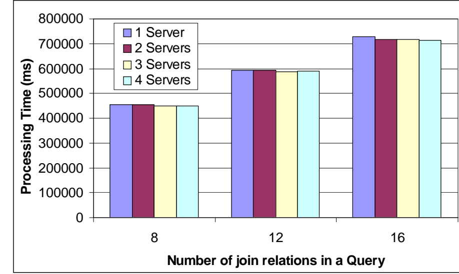

Figure 10: Vary the Number of Data Servers

6.3 Pipelined vs. Segmented Bushy

Experiments have been conducted to compare the total processing time of a right-deep tree having fully pipelined processing to our proposed segmented bushy tree processing that mixes both pipelined and independent parallelism. Figure 11 shows the results of 20 randomly generated queries with 8 join relations. Here, the segmented bushy tree has a maximum of 3 join relations per group. In Figure 11, we see that a segmented bushy tree processing almost consistently outperforms fully pipelined processing.

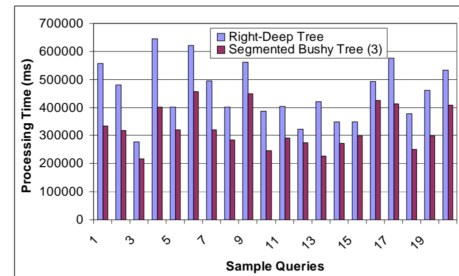

Figure 11: Performance of 20 Example Queries

Figure 12 shows the results of queries with an increasing number of join relations in the query. The number of relations in a query ranges from 8, 12 to 16. The experimental results reflect an average processing time over 50 different randomly generated queries per query type. For example, for queries with 8 join relations, we generate 50 queries randomly. We then produce both the fully pipelined processing and the segmented bushy processing strategies for each gener-

ated query. In this experimental setup, queries with 8 relations are divided into groups having a maximum of 3 relations, while queries with 12 and 16 relations are divided into groups having a maximum of 4 relations.

In Figure 12, we can see that segmented bushy tree processing is consistently better than maximal pipelined parallelism. The performance improvement is around 50% in terms of the total processing time.

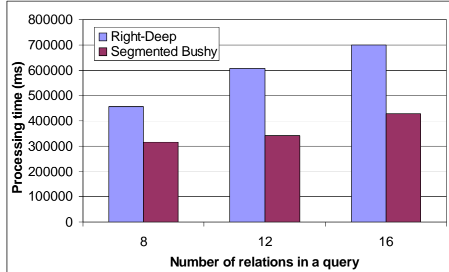

Figure 12: Right-Deep vs. Segmented Bushy

6.4 Probing Relation Selection for the Final Pipelined Segment

Selection of the probing relation of a pipelined segment is usually based on the cardinality of the join relations. This is because choosing a large relation as probing relation can effectively reduce the work and processing time of the building phase. However, for a pipelined segment that involves outputs from other segments (assuming main memory is enough to hold these building relations), the cardinality of the relation alone may no longer be the best choice in general. Changing the probing relation of a pipelined segment that only involves source join relations does not change the number of probes in the probing phase. It only changes the number of probing and building tuples. Here we define the number of probe steps as the maximum number of hash tables that a tuple from the probing relation needs to probe to produce the final output. However, for a pipeline segment having outputs from other segments, changing the probing relation will also change the total number of probes.

For example, if we change the probing relation for the pipeline segment P_1 as shown in Figure 13(a) from R_7 to R_6, no changes in the number of probe steps occur. Both of them are 3 (Figures 13(a)-(b)). However, if we change the probing relation of pipeline P_3 (exchanging P_1 and P_2), then the total number of probe steps changes from 4 to 5 in this case. This is because P_1 itself has 3 probe steps while P_2 only has 2.

Figure 14 shows the experimental results of the impact of the probing relation selection for the final pipelined segment. Here, the number on the x-axis denotes the number of relations in the probing relation of the final pipelined segment. The generated queries have 16 join relations. In Figure 14, we see that in

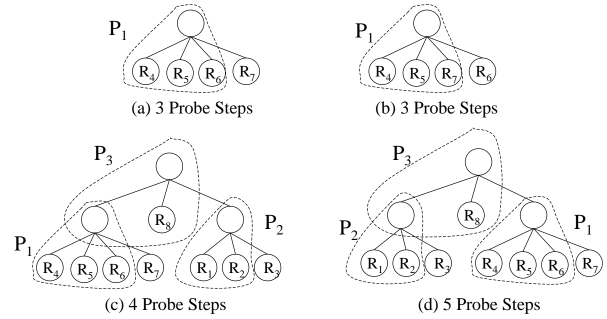

Figure 13: Probing Relation Selection

our current environment, the larger the number of relations in the probing relation of the final pipelined segment, the worse the total processing performance will be. This is because the longer probe steps in the final pipelined segments impair the processing performance. This again confirms our observation that a full pipeline may not be the best performer. Note that the performance degradation for a pipeline that is longer than 8 can be explained by the experiments shown in Figure 12. Hence, in Figure 14, we conveyed the scope of smaller pipeline sizes.

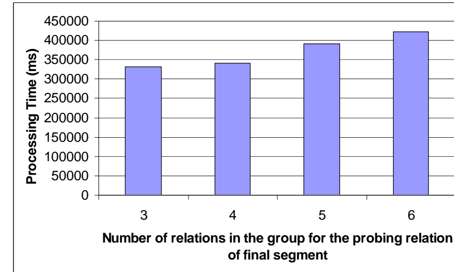

Figure 14: Probing Relation Selection

6.5 Number of Join Relations per Group

Figure 15 illustrates the impact of the maximal number of join relations per group in our environment. Here, all the tested queries have 16 join relations. We vary the number of join relations per group from 3 to 6. As we can see, if the number of join relations per group increases, the total processing time also increases. This is mainly because given our $ComposeBushyTree$ algorithm, the final pipelined segment tends to choose the largest subgraph (the one with the largest number of join relations) as the probing relation since it usually has the largest intermediate results. As shown in Section 6.4, a long pipeline of the final pipelined segment degrades the overall performance. We thus revise our algorithm to choose the subgraph with the smallest number of probing steps as the probing relation of the final pipelined segment. As can be seen, the revised algorithm is less sensitive to the number of

join relations in a group.

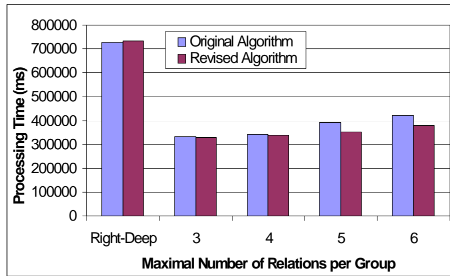

Figure 15: Exchanging the Probing Relation

6.6 Handling Insufficient Memory

Figure 16 shows the experimental results when the aggregated main memory is not sufficient to hold all the hash tables of the building relations. We deploy join queries with 32 join relations. Assume the query will be cut into three pieces with each piece being executed sequentially. Here, the intermediate results of each piece will be first written to the data server, while the next piece will read the intermediate results back into the main memory. We compare the performance of the segmented right-deep tree with our segmented bushy tree generated by Algorithm 2. Note that the segmented right-deep tree has each piece fully pipelined, while the segmented bushy will have the same right-deep segment (piece) further composed into a segmented bushy tree with a maximum of 3 join relations per group. Figure 16 reports the comparison between these two approaches for 10 randomly generated queries. As can be seen, the segmented bushy tree processing consistently outperforms the segmented right-deep processing. This is expected because each piece is processed more efficiently given our segmented bushy tree approach. Thus, the overall performance of the query is correspondingly improved.

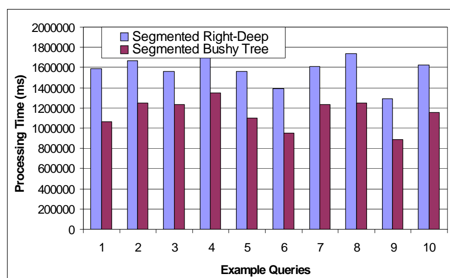

Figure 16: Seg. Bushy vs. Seg. Right-Deep

6.7 Concluding Remarks

As can be seen, these experimental results clearly highlight the main message of our work, namely, the long standing assumption that "maximal pipelining is preferred" is shown to be wrong. Our proposed segmented bushy processing almost consistently beats full pipelined processing. Given the massive application of pipelined processing, especially in growing areas such as continuous query processing, this observation can also shed some new light on how best to optimize distributed pipelined query plans when the optimization function is related to total processing time.

7 Related Work

Parallel query processing has been extensively studied in the literature [4, 6, 11, 13, 14, 20, 21, 24, 29]. Many different research efforts have been conducted in this area. For example, GAMMA [7], Bubba [1], PRISMA/DB [29] are examples of parallel database systems. Many papers were written studying their performance. [13] proposes solutions for scheduling pipelined query operators to minimize the total work. Task scheduling and allocation in general also have been extensively studied [15]. Other focuses such as load balancing [2, 8] and resource allocation [10, 18] are also topics closely related to parallel query processing. As can be seen, these works provide the necessary background for the work presented in this paper. In this work, we instead focus on a specific area of parallel query processing, namely, the parallel multi-join query processing via hashing.

Evaluating a multi-join query via hashing in parallel (applying partitioned and pipelined parallelism) over a shared-nothing environment also has been investigated in the literature before [20, 24, 26]. Different parallel processing strategies such as left-deep and right-deep [24], segmented right-deep [3], and zigzag tree [31] have been proposed, as we have provided an in-depth discussion in Section 2. However, these proposed solutions all share the common approach which is to maximally use pipelined parallelism (i.e., maximally divide a right-deep tree into segments) based on certain objective functions (i.e., memory constraints), and each segment is processed one by one. In this work, we instead consider more tradeoffs in optimizing such parallel multi-join query processing, i.e., other types of query tree shapes, independent parallelism and its dependencies, properties of the join definitions to reduce redirection costs, etc. Moreover, most of the previous works report their results based on simulations, while we report our results based on a working distributed system.

[30] experimentally compares five types of query shapes and various execution strategies based on the PRISMA/DB system [29]. However, it does not explore how to generate optimized parallel processing query plans. In this work, we propose algorithms to generate efficient parallel processing solutions.

8 Conclusion

In this work, we have revisited the common assumption that has been taken by practically all prior work in the literature, namely, to pursue maximal pipelined parallelism when processing multi-join query processing in parallel. We have shown both experimentally and via a cost analysis that the introduction of independent parallelism at the cost of reducing the pipeline can greatly impact the parallel performance. A heuristic-driven optimization algorithm for generating a new class of processing strategies incorporating independent parallelism and yet controlling its dependencies has been proposed in this paper. A working distributed query engine has been implemented. Experimental studies confirm our claim that maximal pipelined parallelism is not always the best choice.

The observation we made in this work also sheds some new light on how best to optimize pipelined query plans in general given the optimization function is related to the total processing time. This optimization is bound to get increasingly attention due to new and growing research areas such as continuous query processing [17].

Acknowledgments. We thank Dr. Paul Larson from Microsoft Research Lab for his feedback on the initial idea of this paper. We thank all WPI DSRG members for their useful comments. We thank Josh Brandt and the WPI CCC Compute Cluster for providing the cluster setup. We also thank TotalETL for the partial research funding.

References

[1] H. Boral, W. Alexander, L. Clay, G. P. Copeland, S. Danforth, M. J. Franklin, B. E. Hart, M. Smith, and P. Valduriez. Prototyping bubba, a highly parallel database system. *IEEE TKDE*, 2(1):4–24, 1990.

[2] L. Bouganim, D. Florescu, and P. Valduriez. Dynamic load balancing in hierarchical parallel database systems. In *The VLDB Journal*, pages 436–447, 1996.

[3] M.-S. Chen, M.-L. Lo, P. S. Yu, and H. C. Young. Using segmented right-deep trees for the execution of pipelined hash joins. In *Proceedings of VLDB*, pages 15–26, 1992.

[4] M.-S. Chen, P. S. Yu, and K.-L. Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *Proceedings of ICDE*, pages 58–67, 1992.

[5] S. Chen, B. Liu, and E. A. Rundensteiner. Multiversion Based View Maintenance over Distributed Data Sources. *ACM Transactions on Database Systems (TODS)*, 29(4):675–709, 2004.

[6] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[7] D. J. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE TKDE*, 2(1):44–62, 1990.

[8] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *Proceedings of VLDB*, pages 27–40, 1992.

[9] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *Proceedings of ACM SIGMOD*, pages 9–18. ACM Press, 1992.

[10] M. N. Garofalakis and Y. E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *Proceedings of ACM SIGMOD*, pages 365–376. ACM Press, 1996.

[11] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proceedings of ACM SIGMOD*, pages 102–111, 1990.

[12] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford University, Dec 1995.

[13] W. Hasan and R. Motwani. Optimization algorithms for exploiting the parallelism-communication tradeoff in pipelined parallelism. In *Proceedings of VLDB*, pages 36–47, 1994.

[14] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Proceedings of PDIS*, pages 218–225, 1991.

[15] Y.-K. Kwok. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4):406–471, 1999.

[16] B. Liu and E. A. Rundensteiner. Revisiting Parallel Multi-Join Query Processing via Hashing . Technical Report WPI-CS-TR-05-05, Worcester Polytechnic Institute, February 2005.

[17] B. Liu, Y. Zhu, and et. al. A Dynamically Adaptive Distributed System for Processing Complex Continuous Queries. In *Proceedings of VLDB Demo Session*, page to appear, 2005.

[18] M.-L. Lo, M.-S. S. Chen, C. V. Ravishankar, and P. S. Yu. On optimal processor allocation to support pipelined hash joins. In *Proceedings of ACM SIGMOD*, pages 69–78, 1993.

[19] H. Lu, K.-L. Tan, and M.-C. Sahn. Hash-based join algorithms for multiprocessor computers with shared memory. In *Proceedings of VLDB*, pages 198–209, 1990.

[20] T. P. Martin, P.-A. Larson, and V. Deshpande. Parallel hash-based join algorithms for a shared-everything. *IEEE TKDE*, 6(5):750–763, 1994.

[21] M. Mehta and D. J. DeWitt. Data placement in shared-nothing parallel database systems. *The VLDB Journal*, 6(1):53–72, 1997.

[22] Sagent Technology. http://www.sagent.com.

[23] 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*, pages 110–121, 1989.

[24] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *Proceedings of VLDB*, pages 469–480, 1990.

[25] J. Srivastava and G. Elsesser. Optimizing multi-join queries in parallel relational databases. In *Proceedings of the 2nd PDIS*, pages 84–92, 1993.

[26] K.-L. Tan and H. Lu. Processing multi-join query in parallel systems. In *Proceedings of ACM Symposium on Applied computing*, pages 283–292, 1992.

[27] TPC. TPC-H Benchmark Standard Specification. *http://www.tpc.org/tpch/*.

[28] C. Wang and M.-S. Chen. On the Complexity of Distributed Query Optimization. *IEEE TKDE*, 8(4):650–662, 1996.

[29] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallelism in a main-memory dbms: The performance of prisma/db. In *Proceedings of VLDB*, pages 521–532, 1992.

[30] A. N. Wilschut, J. Flokstra, and P. M. G. Apers. Parallel evaluation of multi-join queries. In *Proceedings of ACM SIGMOD*, pages 115–126, 1995.

[31] M. Ziane, M. Zat, and P. Borla-Salamet. Parallel query processing with zigzag trees. *The VLDB Journal*, 2(3):277–302, 1993.