Join Query Optimization in Parallel Database Systems

Anant Jhingran
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Sriram Padmanabhan
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Ambuj Shatdal
Computer Sciences Department
University of Wisconsin-Madison

Abstract

In this paper we present a new framework for studying parallel query optimization. We first note that scheduling and optimization must go together in a parallel environment. We introduce the concept of response time envelopes which integrates scheduling and optimization. We show that it can be used effectively to develop parallel query optimization algorithms which have same order of complexity as the traditional sequential query optimization algorithms and produce *provably* optimal or near optimal join plans.

1 Introduction

Query optimization in single-site Relational Database Management Systems (RDBMS) is a well studied problem [8, 3]. Given a query on several relations, say $R_1 \ldots R_n$, a query optimizer must find a plan that optimizes a given cost metric considering the following factors: the join order of the relations, the join method for each join, and the access methods for each relation. The join order can be represented by a binary join tree, which may be of several types such as *left-deep*, *bushy*, or *right-deep*. Since there are an exponential number of binary join trees for n relations, finding the optimal join order is a hard problem. Selinger *et al* [8] have described a Dynamic Programming (DP) algorithm for finding the optimal left-deep join order plan based on the *total cost* (I/O and CPU utilization costs) of the query. The DP approach is based on the observation that once i relations have been joined, *how* they are joined is irrelevant for any future joins, and hence only the best way of joining these i relations needs to be stored.

Query optimization in a Shared-Nothing parallel RDBMS introduces a new set of problems. For example, the optimization cost metric that is best suited for parallel query optimization is the *response time* of the query, which accounts for the benefit of parallel processing unlike total system cost. In a parallel RDBMS, one cannot, in general, restrict the search space to left-deep trees but must consider bushy join trees, as bushy trees can be naturally exploited in a parallel RDBMS. Finally, since scheduling of the various operators affects response time, the new performance metric, it must be considered along with optimization. It can be easily shown that divorcing the two will not produce a response time optimal join plan. Thus, the complexity of the query optimization problem is harder in a parallel RDBMS than in a single-site RDBMS, which is already an NP-Hard problem. Recognizing this complexity, past work on this topic has focused on finding heuristic algorithms [2, 7], or on outlining a general algorithm with high complexity [3]. Also, there has been work on scheduling and processor allocation of query tasks assuming a optimal query plan has already been generated [1]. Yu et al [9] review some aspects of parallel query processing but have not studied the optimization problem in detail. Lanzelotte et al [5] have studied some trade-offs in cost of optimization and the quality of plans obtained in a parallel environment.

In this paper, the parallel query optimization problem is studied using a general framework that considers several factors such as variably declustered relations, different join algorithms including *Nested-Loop*, *Sort-Merge*, and *Hash*; left-deep and bushy join orders, flexible scheduling of the join operation across nodes, and pipelining or materialization relationships at various stages of the query plans. We employ a novel concept of maintaining a *cost envelope* of a sub-plan to aid in scheduling of operations and in optimizing the query for response time. The cost envelope approach enables us to develop a DP algorithm that optimizes the response time of queries with a time-complexity that is only polynomially larger than the single-site DP algorithm. We show that this algorithm finds the optimal join plan in a left-deep search space when considering specific types of join algorithms and pipelining/materialization relationships. We relax some of the restrictions and show that similar DP algorithms can find near optimal (which are within a specified bound

0-8186-5250-0/93 $3.00

of the optimal plan) plans.

The rest of the paper is organized as follows. Section 2 outlines the framework of our approach. Then we discuss the left-deep optimization in detail and briefly touch upon bushy plans in section 3. Conclusions and plans of future work are presented in section 4.

2 Framework

Throughout this paper, we assume an arbitrary partitioning of each base relation, uniquely specified by its partitioning function and the set of nodes the relation is declustered on. For example, table T1 might be partitioned across nodes 1,2,3 and 7 by hashing on the T1.A. We make no assumptions about the partitioning method (hash/range/round-robin) or about skew in either the initial data distribution, or after the selection predicates have been applied. Furthermore, two relations could be partitioned in totally different ways. Just like each base relation, each intermediate relation (result of join/aggregation/sort etc.) also has an associated partitioning.

In serial query optimization, the main task of optimization is determining the optimal join sequence and the join method (mergesort, nested loop, hash) for each join. In parallel environments, however, this must be augmented by (a) determining the partitioning of each join and (b) determining the schedule of various operations (base table scans/sorts and allocated joins) per node. Scheduling is further complicated by the fact that some operations can be *pipelined*, whereas others must fully complete before the next one can begin (we call the latter as *materialized* operations). In this paper we study the following precedence constraints:

- for outer relation: pipelined or materialized
- for inner relation: scan followed by sort/build or scan in parallel with sort/build

The cost model used in serial query optimization is generally a weighted sum of the CPU and I/O costs and can be taken as a measure of *both* the resource consumption, as well as of the response time of a query.[1] In the parallel environment, the two measures are not directly related— one can, by increasing the degree of parallelism, possibly decrease the response-time while increasing the resource consumption. We believe that response-time is the right metric and that the maximum of the response times across all the nodes is the response time of the query. We make a simplifying assumption here that by increasing the degree of parallelism, we only decrease the response time *and do not change the resource consumption.* This assumption is easy to relax (see [4] for further details).

Thus a single step in our query optimization step is the following: When we decide to schedule a join J, we figure out what the resource consumption of that join is. We also figure out *when is the earliest that the join can be scheduled.* We then schedule that join across a set of nodes *such that the final response time is minimized among all schedules which follow the same partial join order.* In other words, we build in the scheduling into our optimization phase to get *optimal response time solutions.* This is in contrast with a two-step approach (optimization followed by scheduling the optimal plan) [1, 7] which unfortunately does not yield to optimal solutions.

We make several assumptions that simplify our presentation somewhat. Most of these are either not very restrictive, or relatively easy to relax. We do not consider *compatible partitioning* which says that two (intermediate) relations are partitioned on their joining attributes across the same set of nodes so that the join can be done locally. This can be relaxed by keeping a set of plans for every plan that we keep. In the exposition below, we also ignore the other interesting orders [3] that even a serial optimizer has to keep. Both these multiply the complexity by only a constant number, and hence are not pertinent to the discussion below.

Furthermore, we assume that the join work is finely partitionable, in that for a total join work of J units, for an N node system, we can arbitrarily partition it into $j_1, ...j_N$ units such that $\sum j_i = J$. This is possible with finer hash partitioning and allocation of buckets to processors. We refer to this as *fine-grained join scheduling.*

The only simplifying assumption we make is that memory allocation and limitations can somehow be modeled as a cost per node. But we believe that this will not make a difference in relative performance of competing plans. In case there is sufficient memory available to hold all intermediate relations this assumption is not required.

2.1 Definitions

In order to evaluate a partial (or total) join sequence with respect to a similar plan, a query optimizer needs to have a measure of the cost of the partial plan and an ordering of costs so that it can prune suboptimal sub-plans.

In our framework, a sub-plan over a subset of relations S is defined to be the join of the relations in S and the scan (and possible sorts) of all the relations in S. The fixed costs of the query involve the scans which must be done on which the relation is declustered and the sorting costs if sorts are done on the nodes where

[1] In some distributed environments [6], weighted message costs are added to the overall costs in order to evaluate a plan.

the relation is scanned.
Let **scansort**(R_i) be the vector $(c_1, c_2, \ldots, c_N)$ of scan (and sort when relevant) cost of the relation R_i.

Definition 2.1 (Fixed Cost Envelope) *It is a vector* $\mathbf{FC}(S) = (f_1, f_2, \ldots, f_N)$ *of fixed costs per node of a query sub-plan for the relations in* S.

$$\mathbf{FC}(S) = \sum_{i \in S} \mathbf{scansort}(R_i)$$

In our scheme, we schedule a join *around* $\mathbf{FC}(R)$—the fixed cost envelope of all the relations. Intuitively, at each node we try and perform the scan and sort of the base relations *first* before any join work. Based on this scheduling, each subplan produces a *response-time envelope*—$\mathbf{RT}(S)$—which is simply the union of the scheduled joins in S **and** $\mathbf{FC}(R)$. Furthermore, we spread the join costs using fine-grained scheduling such that the join finishes across all the nodes at the same time.

Definition 2.2 (Water Line) *It is the current level (time) to which joins of the sub-plan on relations,* $S \subseteq R$, *set of all relations, have been scheduled (around the fixed cost envelope). Formally:*

$$\mathcal{W}(S) = \max(t_i), i \text{ s.t. } t_i > f_i$$

where $\mathbf{RT}(S) = (t_1, t_2, \ldots, t_N)$ *and* $\mathbf{FC}(R) = (f_1, f_2, \ldots, f_N)$.

Subplans are judged on the basis of their water lines—if a subplan on S has a lower waterline than another subplan on S, then the latter is discarded.
We need one more definition here:

Definition 2.3 (height) *Height of an envelope* $\mathbf{Env}(S) = (e_1, e_2, \ldots, e_N)$ *is:*

$$height(\mathbf{Env}(S)) = \max_{i=1}^{N}(e_i)$$

Figure 1 shows various components of a response time envelope.

3 Query Optimization

Since there are exponential number of possible join orders and each join can be done in several ways the search space is inherently very large. This search space must be pruned in order to make efficient optimization feasible. The approach presented in System R [8], prunes this search space by limiting the search space to left deep trees and by using DP with the observation that once i relations have been joined, how they were joined is irrelevant to any future join. In parallel environments, conventional wisdom says that DP is too expensive [3] and hence a lot of researchers have gone straight onto heuristic algorithms, mostly based on some kind of greedy approach [2].

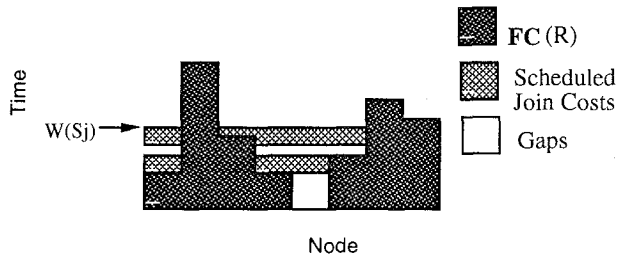

Figure 1: A Response Time Envelope

The question we attempt to answer is: Is it possible to extend the DP approach to optimize parallel queries? And if so, then how? Ganguly et al [3] show that, in general, the ability to prune the search space decreases in the parallel environment. We show that in some cases, that is not necessarily so and that we can use efficient, single-site like, DP algorithm. In general, however, we show that there is a trade off between achieving optimality and efficiency. In particular, approximate optimal (with bounded error) plans can be obtained efficiently but to obtain the optimal solution one has to keep more state in the DP algorithm i.e. the ability to prune decreases.

3.1 Exact Optimization for Materialized Joins

We illustrate the parallel query optimization for left-deep materialized sort-merge joins (i.e. both relations must be materialized before join can proceed). We consider only equi-joins, where the inner exists on nodes N_i, the outer of nodes N_o, and the join is performed on nodes N_j. Both the inner and the outer are sorted on their respective nodes, and each tuple of the inner and outer are then routed according to the fine-grained join scheduling decided for that join. On each node of N_j, the inner stream is merged, the outer stream is merged, and a merge-sort join between the two streams is then performed. For other join methods and strategies, see [4] for details.

The algorithm 3.1 shows the dynamic programming outline. Except for initialization in steps 1–4 and step 10 which involves the scheduling of the join, the steps are identical to the single-site left deep query optimization [3].

Algorithm 3.1 *Optimize*
Let $R = \{R_1, \ldots, R_n\}$ *be the set of relations in the query.*

```
1    FC(R) := 0
2    for i := 1 to n do {
3      optPlan(R_i) := scansort(R_i)
4      FC(R) := FC(R) + scansort(R_i)
5    }
6    for i := 2 to n do {
7      forall S ⊆ R s.t. ||S|| = i do {
8        bestPlan := dummy plan with W(bestPlan) = ∞
9        forall R_j, S_j s.t. S_j = S − {R_j} do {
10         p := plan obtained by Schedule
11         if W(p) < W(bestPlan)
12           bestPlan := p
13       }
14       optPlan(S) := bestPlan
15     }
16   }
17   Popt := optPlan(R)
```

Algorithm 3.2 *Schedule*

```
1    FC(S) := FC(S_j) + scansort(R_j)
2    startlevel(S) := max(height(FC(S)), W(optPlan(S_j)))
3    perform scheduling of optPlan(S_j) ⋈ R_j around
     FC(R) starting at startlevel(S)
```

In step 10 of *Optimize*, the *Schedule* procedure is invoked to compute the response time envelope of the new plan resulting from the join of S_j and relation R_j. Algorithm *Schedule* first finds the start level at which this join can be scheduled (steps 1–2). Due to precedence constraint, the join of the $optPlan(S_j)$ and R_j can only be scheduled after the scan and sort of R_j is completed and the intermediate relation resulting from $optPlan(S_j)$ is materialized (step 2). Then the scheduling of the computed join work of $optPlan(S_j) \bowtie R_j$ is done around the fixed cost envelope, $\mathbf{FC}(R)$. This anticipation of expected future work (which is fixed) enables this scheduling scheme to produce optimal schedule. Intuitively, since no future work can be scheduled below the current envelope, the optimal envelope of relations S satisfies the principle of optimality (theorem 3.1). The scheduling is done such that the water line of plan p is kept to the minimum. This is done by flexibly scheduling the cost of the join on the nodes to retain a flat water line (figure 1). That is, we ensure that all nodes participating in this join finish at the same time by allocating possibly different amount of work to each of them. This can be done in $O(N)$ time by solving the following equation where J is the join cost of $optPlan(S_j) \bowtie R_j$:

$$\sum_{i \text{ s.t. } \mathcal{W}(p) > \mathbf{low}_i} \mathcal{W}(p) - \mathbf{low}_i = J$$

where $\mathbf{low}_i = max(\mathbf{FC}(R)_i, startlevel(S))$ indicating the point where scheduling of the join cost can start on a particular node.

If there are n relations in the join query and N nodes in the system, the time complexity of this algorithm is $O(n.2^n.N)$. Note that complexity of the algorithm increases only by a *polynomial* coefficient when compared to the single-node dynamic programming algorithm whose complexity is $O(n.2^n)$.

Figure 2 illustrates this scheduling. First we find how soon can this join be scheduled. Then the join cost of $optPlan(S_j) \bowtie R_j$ is scheduled above that such that all nodes participating in the join finish at the same time. The nodes which are still doing a scan are ignored.

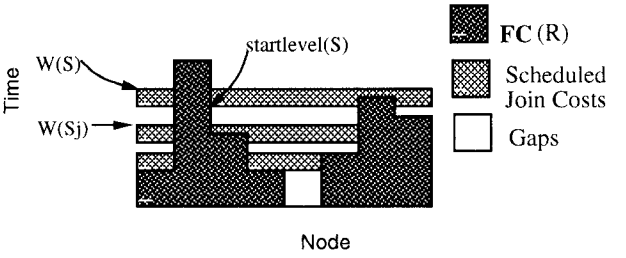

Figure 2: Scheduling $optPlan(S_j) \bowtie R_j$ on $\mathbf{RT}(S_j)$ of figure 1

We briefly sketch why this algorithm achieves optimality.

Theorem 3.1 *For left deep trees, the algorithm 3.1 gives a response time optimal solution.*

Proof: First, we show that for any join order, the above scheme produces the minimum response time schedule. At each stage we choose the plan with the least waterline satisfying all precedence constraints and *including the fixed costs.* At the final stage, the optimal plan **must** have the least waterline and must include the fixed costs. Thus we produce the plan that has the least waterline in the end *for a given join order.* (Note that our subplans could be suboptimal, but the nice thing is that the final plan is not.)

This, combined by the dynamic programming strategy which examines the various possible join orders produces the optimal schedule across all join orders. □

However, we find that this exact optimization does not hold in the more general cases without increasing the complexity of the DP.

3.2 Approximate Optimization of Semi-Pipelined Joins

The algorithm 3.1 can be generalized to consider left deep semi-pipelined joins like nested loop and hash join. The characteristic of semi-pipelined joins is that one of the relations (typically the outer) can be pipelined. The inner goes through a build (for a hash join) or a sort (if sorting is done the join nodes instead of the scan nodes). In nested loop joins the build phase is absent. As mentioned earlier, there can be two precedence cases for the inner. These are whether the build/sort is done in parallel with the scan of the relation or after the scan is complete. It appears that optimal solution in the former (parallel) case is not obtainable without keeping significantly more state i.e. all the incomparable plans, thus making the DP approach inefficient. But near optimal, with bounded error, solutions can be efficiently generated. The pipelined outer case is similar to the materialized outer case because pipelining the outer relation only affects one relation. Formally, we propose the following:

Proposition 3.1 *If some part of the work of a future join (that is a join not yet considered) can be scheduled below the current water line, then, in general, more than one plan for each subset of relations must be kept to guarantee optimality. In other words, efficient DP solution is not possible.*

Proof: Suppose there are plans p_1 and p_2 where $\mathcal{W}(p_1) = \mathcal{W}(p_2) + \epsilon$ but p_1 has a lot of gaps which can be used. It is easy to see that if gaps in p_1 can be utilized by a future join with a relation R_j then $\mathcal{W}(p_1 \bowtie R_j) < \mathcal{W}(p_2 \bowtie R_j)$ i.e. now p_1 is the optimal plan instead of p_2. Hence if we discard p_1 then we do not achieve optimality. We have to keep all incomparable plans to guarantee optimality. □

However, we can obtain a near optimal solution in several ways. The simplest one is by restricting the allocation of future work to only above the current water line.

Theorem 3.2 *Algorithm 3.1 when extended to considering semi-pipelined joins with above restriction gives near optimal solution with the error bounded by the amount of work which could potentially have been done below the water line.*

Proof: Since, at best, all the builds/sorts can be done in parallel with the scan of the relevant relations, doing them after the scan (which the above restriction implies) can, at worst (if there were enough "gaps" in the schedule), increase the water line by the time it takes to do the builds/sorts. Assuming builds/sorts represent a fixed percentage of cost of the join, the plan produced could be no worse than the percent of cost represented by the builds/sorts. □

In practice, however, we can come close to optimal by estimating the gaps which could be use for builds/sorts and considering that while eliminating plans. For example, pessimistic estimation of the useful gaps can be done as follows. First we order each relation by a measure of *build cost per tuple*, b_r, and consider them in that order in the following summation. A gap is characterized by an area and its upper bound, g^u (the location of the gap being then fixed by $\mathbf{FC}(R)$). Then pessimistic estimate the amount of useful gaps is equal to:

$$\sum_{\text{all gaps } g} \sum_{i=1}^{N} \sum_{r \in (R - S_j) \text{ s.t. } \sum r \leq g^u} \|r\| . b_r$$

which is the least amount of gaps which could be utilized by future joins. Then in the scheduling algorithm the amount of work to be scheduled is decreased by the amount of useful gaps. Such an estimate will more accurately reflect the real possible schedule and can also be used in reducing the number of plans required to keep for an exact solution.

3.3 Optimization for Bushy Plans

The basic single site left deep DP algorithm can be extended to consider bushy trees with accompanying increase in complexity. In the basic bushy DP algorithm, the computation of the new plan from two subplans is more involved than the left deep case. Since both the schedules are developed independent of the other, they can not be merged trivially. Since operations of the two plans are (in several cases) independent of each other, in the worst case it could involve considering all possible merging of the two sub-plans to obtain the best merged plan.

However, bushy trees do not lend themselves to exact optimization for the reasons similar to those mentioned in previous section. There is an added difficulty that now two sub-plans which form the bushy tree interact in in unpredictable ways as each was developed ignoring the other. Therefore, keeping a single sub-plan for a set of relations in not sufficient even for materialized joins. It can be shown that even with the best possible merging of sub-plans, a bushy DP will obtain a solution which is worse than the optimal. But this error is bounded.

Theorem 3.3 *The solution obtained by basic bushy DP will be at most $\frac{n-1}{2}$ times as expensive as the optimal (bushy) solution where n is the number of relation.*

Proof: In full paper [4]. □

4 Conclusion and Future Work

Query optimization in a Shared-Nothing parallel Relational DBMS is a hard problem. Unlike past work, which have mostly resorted to heuristics, we describe a novel algorithm that generates the optimal response time plan for a multi-join query in several specific instances, and near optimal plans for general instances of the problem. The algorithm is applicable across a broad and comprehensive framework that includes different join algorithms, scheduling of the operators, left-deep and bushy tree search spaces, and pipelined or materialized relationships between join stages. The algorithm is based on dynamic programming and we show that its complexity increases by only a polynomial factor when compared with the dynamic programming algorithm used for query optimization in a single-site DBMS.

There are many interesting problems which remain, some of which are currently under investigation. For example, for left deep semi-pipelined joins, using simulation we can find the error in the plans obtained in practice. Another interesting problem is to find how many plans are required in general to be kept around to get the optimal solution for both semi-pipelined and the bushy tree case. Can there be better scheduling techniques? Explicitly modeling of memory constraints would be another challenging task.

References

[1] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Scheduling and processor allocation for parallel execution of multi-join queries. In *8th Int'l Conference on Data Engineering*, pages 58–67, February 1992.

[2] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Optimization of parallel execution for multi-join queries. Internal Report, 1993.

[3] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In *1992 ACM SIGMOD Conference*, pages 9–18, May 1992.

[4] Anant Jhingran, Sriram Padmanabhan, and Ambuj Shatdal. Query optimization for parallel database systems. In Preparation, 1993.

[5] Rosana S.G. Lanzelotte, Patrick Valduriez, and Mohamed Zait. On the effectiveness of optimization search strategies for parallel execution spaces. In *19th VLDB Conference*, 1993.

[6] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *12th VLDB Conference*, 1986.

[7] Donovan A. Schneider. *Complex Query Processing in Multiprocessor Database Machines.* PhD thesis, University of Wisconsin-Madison, 1990.

[8] Pat Selinger, M. M. Astrhan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *1979 ACM SIGMOD Conference*, 1979.

[9] Philip S. Yu, Ming-Syan Chen, Joel L. Wolf, and John J. Turek. Parallel query processing. Research Report RC 18903, IBM T. J. Watson Research Center, 1993.