Query Optimization for Parallel Execution

Sumit Ganguly* Waqar Hasan Ravi Krishnamurthy

Hewlett-Packard Laboratories
1501, Page Mill Road, Palo Alto, CA 94305
email: lastname@hpl.hp.com

Abstract

The decreasing cost of computing makes it economically viable to reduce the response time of decision support queries by using parallel execution to exploit inexpensive resources. This goal poses the following query optimization problem: *Minimize response time subject to constraints on throughput*, which we motivate as the dual of the traditional DBMS problem. We address this novel problem in the context of Select-Project-Join queries by extending the execution space, cost model and search algorithm that are widely used in commercial DBMSs. We incorporate the sources and deterrents of parallelism in the traditional execution space. We show that a cost model can predict response time while accounting for the new aspects due to parallelism. We observe that the response time optimization metric violates a fundamental assumption in the dynamic programming algorithm that is the linchpin in the optimizers of most commercial DBMSs. We extend dynamic programming and show how optimization metrics which correctly predict response time may be designed.

1 Introduction

The Papyrus project[CHK+91] at HP Labs is investigating ways of integrating highly-tuned, customized data managers while preserving their requirements for performance. One source of high performance is parallel execution of queries. While, in its generality, the project seeks to parallelize queries containing arbitrary data manipulation operators, in this paper we focus on the problem of optimizing Select-Project-Join (SPJ) queries for parallel execution.

*Author's permenant address is Dept. of Comp. Sciences, The University of Texas, Austin, TX 78712. sumit@cs.utexas.edu

The need to reduce response time is evident in decision support applications in which human beings pose complex queries and demand interactive responses. For example, a system for stock portfolio managers is capable of running a non-trivial query at the click of a button and graphing the results by many categories of stocks. This is not an isolated example but typifies the tasks in decision making processes. Achieving a reasonable response time has been an important roadblock for these applications.

The decreasing cost of computing makes it economically viable to throw inexpensive resources to achieve the reduction in response times by exploiting parallel query execution. For example hash partitioned sort-merge[SD89] may be used to reduce response time at the expense of extra work done in comparison to vanilla sort-merge. While extra work may be traded for reduced response time, it is unacceptable to minimize response time at any expense.

Todays' commercial database systems aim at maximizing throughput while imposing some limits on response time. For example, the TPC-A benchmark[Gra91] requires that 90% of the queries have a response time of at most 2 seconds. However, this requirement is typically assured, not by incorporating the requirement into the optimizer, but by setting system parameters such as the maximum degree of multi-programming and the frequency of group commit. Assuring response times for other classes of queries will require the optimizer itself to be modified.

The traditional database problem is to maximize throughput subject to constraints on response time. We claim that decision support applications pose the *dual* optimization problem of minimizing response time subject to constraints on throughput.

Optimizing queries for parallel execution is considered an open problem[DG90]. Previous work on minimization of response time in [AHY83] and XPRS[HS91] made restricting assumptions regarding the parallel machine architecture that restrict the applicability of the proposed solutions. The Gamma project[DGS+90, SD89, Sch90] studied many execution strategies but did not address the problem of query optimization.

Any solution to this optimization problem should build on the decade of experience with commercial optimizers. Taking cue from [SAC+79] we describe our solution to the parallel query optimization problem along the following three dimensions: an *execution space* which defines the syntactic representation of all relevant aspects of an actual execution; a *cost model* which predicts the cost of an execution plan; a *search algorithm* which is used to obtain the minimum cost plan.

The solution described in this paper is applicable to a gamut of parallel machine architectures. Differences across architectures appear as variations in the precise details of the cost model and in the factors considered important in the design of the search metric.

We have extended the traditional execution space to model the sources (such as running different joins in parallel or parallelizing one join) and deterrents of parallel execution (such as data dependence and resource contention). Thus the trade-offs that should be examined by the optimizer have been represented.

We have shown that it is possible to design a cost model that can estimate the above trade-offs in predicting response times. For example, depending on the dataflow dependencies and resource contentions, the response time estimate of a pipelined execution ranges from that of an independent parallel execution to an estimate that is *worse than* that of a sequential execution. This degenerate pipelined execution reflects the penalty for executing operations in a pipelined fashion when there is no available parallelism. Thus, we establish the feasibility of a cost model to be judicious.

We show that response time violates a fundamental property, namely principle of optimality, required by the commercially accepted search algorithm[SAC+79]. While the traditional dynamic programming approach used a total order on work, our extension uses a partial order on a metric that predicts response time. Further we provide an analysis that demonstrates the practicality of this approach.

In summary, we address the problems in designing execution space, cost model, and search algorithm so as to establish the feasibility of a general but practical optimization algorithm for minimizing response time subject to constraints on extra work.

Section 2 presents a precise description of the optimization metric and Section 3 presents an overview of traditional optimizers. In Section 4, we discuss how the execution space models the sources and deterrents of parallelism and Section 5 presents our cost model Section 6 deals with search algorithms and we conclude in Section 7.

2 Parallel Query Optimization : Optimization Metric

The general problem of query optimization may be stated as: Given a query q, a space of execution plans, E, and a cost function $cost(p)$ that assigns a numeric cost to an execution plan $p \in E$, find the minimum cost execution plan that computes q.

We formulate the cost of a query as its response time but place a bound on the amount of "extra" work that can be traded for reducing response time. The bound on extra work is determined by parameters that allow a system administrator control over the work and response-time tradeoff. Our formulation of the query optimization problem is quite novel and we believe, necessary.

Suppose the work and response time costs of the optimal-work plan for query q are W_o and T_o (respectively) and for an arbitrary execution plan p are W_p and T_p. We will consider two ways of limiting the extra work:

- Limit on throughput degradation: The system administrator may specify the factor, k, by which the maximum throughput of the system may be allowed to degrade. The cost of p is T_p if $W_p \leq k * W_o$ and infinite otherwise.
- Cost-benefit ratio: The system administrator may specify a limit, k, on the ratio of the decrease in response time to additional work required. The cost of p is T_p if $\frac{T_o - T_p}{W_p - W_o} \leq k$ and infinite otherwise.

This limit on extra work is incorporated in the search algorithm akin to incorporation of branch and bound techniques.

3 Traditional Optimizer Revisited

Commercial DBMSs are built for optimal throughput and therefore the optimization metric is *work* (total resource consumption). Optimizers in these systems have adopted the framework of the System R optimizer[SAC+79]. The design of such optimizers may be decomposed into three components: Execution Space, Cost Model and Search Algorithm.

Execution Space: Query executions are syntactically represented as *annotated join trees* (also called *plans*). These are *binary* trees in which each internal node is a join operation and each leaf node is a base relation. Annotations of the form *label=value* on nodes permit the important factors of an execution to be modeled. An example annotation for a join node is *join-method=nested-loops*. Other examples of labels are access path, create index, and eliminate duplicates.

The semantics of such trees is intuitive for left-deep trees. We place the semantic constraint that *each tuple of any subtree of a join tree is computed exactly once.* This eliminates some bushy trees that represent the "same" execution as left-deep trees.

Given a query, all trees that compute the query (independent of database state) are considered *legal plans* for the query. Optimizers are designed to only search a finite subset of this infinite space of legal plans. This subset, termed the *search space*, can be syntactically characterized. For example the search space in System R is left-deep join trees with the annotations described in [SAC+79].

Cost Model: The cost model assigns an integer cost to a plan based on some set of assumptions about the statistical distribution of data and the abstract machine that executes plans. A cost function, $cost(plan) \rightarrow descriptor$, is *recursively* defined in terms of the descriptors of the left and right subtrees of the plan. The descriptor contains information such as relation size, interesting orderings, and index availability. The cost function is recursive not only for simplicity but so that it may be used in search algorithms based on dynamic programming.

Search Algorithm: This is the algorithm used to search the search space for the plan with minimum cost. A System R style dynamic programming algorithm is reviewed in Section 6.1.1.

4 A Model for Parallel Execution

In this section we delineate important aspects of parallel execution and then present a macro expansion of the traditional annotated join tree to get a new tree called *operator tree* wherein these new aspects are modeled.

4.1 Aspects of Parallel Execution

In addition to modeling important aspects of sequential execution, a model for parallel execution needs to account for the *sources* and *deterrents* of parallelism.

There are essentially two sources of parallelism. (1) Inter-operator parallelism which includes both the data independent parallel execution of a pair of operators (typically subtrees) and data dependent, (i.e., pipelined) parallel execution of a pair of operators. 2) Intra-operator parallelism which refers to the parallelization of a single operation (e.g., sort), that we term cloning.

Two main deterrents of such parallelism are: 1) *Data Dependencies* between operators which arise due to the logical relationship between the operators (e.g., the probe of a hash join cannot begin until the build is completed). 2) *Resource Contention* arising out of several operators running in parallel and competing for the same resource.

4.2 The Operator Tree

In order to accurately model aspects of parallel execution we refine the execution model of join trees to a finer granularity. Each annotated join tree is expanded to a *unique* operator tree by macro-expanding each join node into an operator subtree in a manner described below. Intuitively, the operator tree has nodes which are atomic in the sense that the run-time scheduler cannot subdivide this operation further except for the partitioning the input for cloning. Note that our intention here is only to provide some exemplary aspects of the operator tree model based on which we can discuss the cost model. Therefore it is not complete.

The node $sort\text{-}merge(R_1, R_2)$[1] is macro expanded with explicit sorting and merging. For example, if $merge(sort(Scan(R_1)), sort(Scan(R_2)))$ is the operator tree then the data dependency of the sort operation and the merge operation as well as the independent parallel execution of the two sort operations are explicitly represented here. Note that if R_2 is already sorted then only one sort operation needs to be stated in the operator tree. Similarly, the node $hash\text{-}join(R_1, R_2)$ can be expanded with explicit reference to the building and then probing of the hash table (e.g., $probe(Scan(R_1), build(Scan(R_2)))$). The *nested-loops* join is expanded with explicit reference to the inflections of this join method such as create-index, duplicate elimination etc. (e.g., $pure\text{-}nested\text{-}loops(Scan(R_1), create\text{-}index(Scan(R_2)))$). Note that a *pure-nested-loops* operator represents an execution of a nested loops without any inflections.

As before, further details of the execution are given as annotations, some of which are given below.

1. The *composition method* annotation for a (child, parent) pair has two possibilities, *pipelined* or *materialized.* It can be set to pipelined if the child can produce partial output, and the parent can consume partial output; otherwise, it is annotated as materialized. Note that this annotation, denoting the method for (child, parent) pair, is associated with the child node which is uniquely defined in an operator tree.

2. The *cloning* annotation is intended to model intra-operator parallelism. This value is a pair (set of resources, the partitioning attribute), representing parallel execution using the attribute partitioned data over the resources. Suppose that a sort-merge node is annotated as being cloned on processes $1, 2, 5, 7$ on the join attribute A. This would represent a hash partitioned sort-merge as described in [SD89].

3. The *data redistribution* between a (child,parent) pair, annotated at the child node, has two possibilities: true or false[2]. This is set to true if it is found necessary to redistribute the output of the child node based on the subsequent cloned operations ; otherwise false.

Example 1 Consider the following join tree $nested\text{-}loops(sort\text{-}merge(R_1, R_2), R_3))$ with the following operator tree for parallel execution:

$$nested\text{-}loops(merge(sort1(scan(R_1)), sort2(scan(R_2))), scan(R_3))$$

[1] Note that strictly speaking, the operation is *join* and the join-method annotation is sort-merge. For brevity, we use this notation whenever there is no ambiguity.

[2] Note that a more expressive annotation would be needed for practical application. As this is sufficient for this paper, we present the simplest case.

The annotations for this operator tree are given below:

Node	cloning	comp. method	redistr.
$scan(R_1)$	$(\{1,2,3,4\}, A1)$	*pipelined*	*no*
$scan(R_2)$	$(\{1,2\}, A2)$	*pipelined*	*yes*
$scan(R_3)$	$(\{1,2,3,4\}, A3)$	*pipelined*	*no*
$sort1$	$(\{1,2,3,4\}, A4)$	*materialized*	*no*
$sort2$	$(\{1,2,3,4\}, A5)$	*materialized*	*no*
$merge$	$(\{1,2,3,4\}, A6)$	*pipelined*	*no*
$n.loops$	$(\{1,2,3,4\}, A7)$	$-$	$-$

■

5 The Cost Model

We present a cost model primarily to show that the tradeoffs — amongst the aspects of parallel execution as it affects the response time — can be represented. We estimate the response time for independent parallel execution (IPE), dependent parallel execution(DPE), parallel execution of cloned operation (CPE) as well as sequential execution (SE) such that the deterrents affect the response time. In particular, the following desiderata for the cost model is put forth:

1. Response time of an IPE degrades to that of a SE depending on the level of resource contention.

2. Response time of a DPE is in the range from that of IPE to a response time worse than that for SE depending on the dataflow dependencies and resource contentions.

3. Response time of a CPE is similar to IPE of the clones.

Note that the DPE may degenerate to an execution that is worse that SE because there is a penalty for executing operations in a pipelined fashion when there is no available parallelism, especially if prohibited by resource contention. This is because, the system necessarily sets up the pipeline and compromises the run time execution of DPE whereas no such penalty occurs for a SE.

The cost model presented below is purely an example that is capable of trading-off the sources and deterrents as per this desiderata. Consequently, this is not intended to be a prescriptive cost model for any parallel execution, and therefore we make simplifying assumptions so long as the above goal of exemplification is not compromised.

We first present the cost model assuming no cloning, or redistribution. Then we will extend it to accommodate these annotations as well. Sans these annotations, the only annotation of interest is the composition method and its effect on parallelism. In short, the execution can be viewed as a traditional execution with parallel access to disks.

The response time (RT) is the metric for the cost model. We assume the usual information in the cost descriptor with the cost estimate in the descriptor representing the response time. In order to account for data dependencies in an operator tree P, especially for pipelined operations, the cost descriptor must have two parts.

1. *First tuple descriptor*, which is a descriptor for those subtrees of P that must be finished before the first tuple of P is output to any subsequent operation. More precisely, given $S1$ the set of subtrees in P, the minimal subset $S2$ of $S1$ that must be finished before the first tuple of $S1$ is output. This is the set of all subtrees of $S1$ that have materialized annotation at the root of the subtree. $S2$ is called the *materialized front* of $S1$. In Example 1 $S2$ is the set of subtrees rooted at $sort1$ and $sort2$. We refer to the *residual query* of $S1$, that is done after the first tuple is computed, as $S1 \ominus S2$.

2. *Last tuple descriptor*, which is a descriptor for the complete execution of P.

The cost portion of the first (last) tuple descriptor is initially represented as an integer tf (resp. tl) denoting the estimated time when the first (resp. last) tuple of the plan is output. We refer to this as the *time descriptor* and is typically denoted by $\mathbf{t} = (tf, tl)$. This will be generalized to include the resource usage in the latter subsection and thus the term *resource descriptor* and this is typically denoted $\vec{r}$. Besides these, we assume the usual statistical information needed for cost estimation, even though we have not explicitly expressed it in the formalism.

We present the cost model in two steps. First, we describe the estimation of response time assuming no resource contention. Therefore, we discuss the estimation of the time descriptor for any query. Then we incorporate the contention on resources into the estimate of the resource descriptor which in turn affects the response time.

5.1 Estimating Resource Contention Free RT

We present an extension to the cost model for traditional DBMS, accounting for data dependencies, intra and inter-operator parallelism assuming there is no resource contention. We introduce three binary operators on time and describe the scenario whose cost each operator is expected to model. We assume that $t1$ and $t2$ represent the completion times of two sets of operators, say $S1$ and $S2$.

1. $t1 \parallel t2$ estimates the response time of IPE of $S1$ and $S2$. Without resource contention, this is $max(t1, t2)$.

2. $t1; t2$ estimates the response time of SE of $S1$ followed by $S2$. Hence, this is $t1 + t2$.

3. $t1 \ominus t2$ estimates the response time for the residual query $S1 \ominus S2$, where $S2$ is assumed to be the materialized front of $S1$ using a DPE of $S2$ piped to $(S1 \ominus S2)$.

This value of $\ominus$ operation depends on the difference between $t1$ and $t2$ and the respective contribution of $S1$ and $S2$ towards the critical path determining the response time. For now we approximate this to be $t1 - t2$, but in the next section when we take the resource usage into account to get a more accurate estimate.

Using these operators we derive the formulae for the two composition methods, namely pipelined and materialized execution as follows:

- Given a pipeline with the producer and consumer time descriptors as $\mathbf{p} = (pf, pl)$ and $\mathbf{c} = (cf, cl)$ respectively, the operator $\mathbf{p} \mid \mathbf{c}$ is the descriptor (tf, tl) for the pipeline given by:
  $tf = (pf; cf) \qquad tl = (pf; cf; ((pl \ominus pf) \parallel (cl \ominus cf)))$
  The tf is computed under the assumption that the first tuple is computed at the earliest possible time. The remainder of producer operations run in parallel with the remainder of the consumer operations as reflected in the formula for tl.

- The materialized execution of a subtree whose time descriptor $\mathbf{t} = (tf, tl)$ is computed using the operation $sync(\mathbf{t})$ that sets tf to tl, i.e., $sync(tf, tl) = (tl, tl)$).

The cost of an operator tree is computed recursively using the above formulae. In the base case, the descriptors of the leaves of the tree as well as the join methods are derived in the traditional manner, wherein the response time is the total work required for that operation. Then recursively, given the descriptors $\mathbf{L}, \mathbf{R}$ and **root** for the left/right operands and the root node respectively, the descriptor for the tree may be computed as follows: The materialized frontier of L may run in parallel with the materialized frontier of R to give a time descriptor of $\mathbf{t_1} = (L_f \parallel R_f, L_f \parallel R_f)$, then the residual query of L may run in a pipelined parallel fashion with the residual query of R to give a descriptor of $\mathbf{t_2} = \mathbf{t_1}; (0, L_l \ominus L_f) \mid (0, R_l \ominus R_f)$, and the result of this pipeline is piped to the root node to give $\mathbf{t} = \mathbf{t_2} \mid \mathbf{root}$. This operator is denoted as $tree(\mathbf{L}, \mathbf{R}, \mathbf{root})$. If the root node has only one subtree, then the formula is simply, $\mathbf{L}|\mathbf{root}$. Note that the above formula is also applicable for subtrees that are materialized as well.

Example 2 We continue with Example 1 and estimate the response time. The time descriptor (tf, tl) for each operation in the tree is estimated using traditional approach. For example, creating an index might incur a tf-component that must be done before using the scan. The following table shows, for hypothetical time descriptors of these operation, the computation of the estimate.

Oper.	(tf, tl)	*formula*	*value*
scan R_1	$(0,1)$		$(0,1)$
scan R_2	$(0,3)$		$(0,3)$
scan R_3	$(0,2)$		$(0,2)$
*sort*1	$(5,5)$	$sync((0,1)\|(5,5))$	$(6,6)$
*sort*2	$(10,10)$	$sync((0,3)\|(10,10))$	$(13,13)$
merge	$(0,2)$	$tree((6,6),(13,13),(0,2))$	$(13,15)$
n.loops	$(0,2)$	$tree((13,15),(0,2),(0,2))$	$(13,15)$

Now let us relax the assumption that disallowed cloning, and redistribution. The ability to clone a join or a select is captured by changing the time descriptor (tf, tl) by an amount k, the degree of cloning. For example, one simple approach is to take the time descriptor to be $(tf/k, tl/k)$. Needless to say that a more ambitious formulae would take into account the overhead associated with the cloning. The redistribution of data involve network and CPU overhead wherein the network is pipelined to the recipient CPU. Currently, the transfer of data from a producer to a consumer (i.e., between any two operation) has been assumed to be of zero cost. In general, this is not true as suggested by the redistribution cost. This transfer cost needs to quantify the inter-process overhead, communication overhead as well as the potential parallelism available in this transfer. This can be incorporated into the above formulae using the time descriptor for the transfer as well as estimating the response time using the above formulae. In short, the above cost model can be extended to handle other annotations as well.

In summary we have presented a cost model that trades-off the sources and deterrents of parallelism in estimating the response time of a query plan. In this, the overlapped/extra-work that is accrued/incurred by various operations are modeled and the response time is appropriately estimated. All this was done assuming that there was no resource contention. This is relaxed in the next section.

5.2 Modeling Resource Contention

We first present a resource usage model, using which we then present the cost calculus.

5.2.1 Resource Usage Model

Resource usage by a task is modeled by two parameters: t and w, where t is the time after which the resource is freed and w, the work measured as the effective time for which the resource is utilized. For example, suppose that a CPU was used for 10 seconds with an effective work of 5 secs. This means the CPU was busy only 50% of the time. However, in this abstraction, we cannot predict exactly when the busy periods were.

In fact, we make a *uniformity* assumption for this (t, w) abstraction of resource usage. The usage of the resource is uniform over the time period t[3]. Further we assume that the resources are *preemptable* in the sense that they can be time shared. Resources like CPU, disk, network are preemptable whereas memory is not[4]. The following property follows from these assumptions.

> *Property of stretching:* If a plan has a resource usage of (t, w) then it can be made (i.e, using an appropriate scheduling strategy) to have a resource usage of (mt, w) for any positive number $m > 1$.

Let us assume that resources $r_1, \ldots, r_n$ are used by a certain set of operations (plan). For each of these resources, the usage is modeled by a pair (t_i, w_i). Since we assume that the property of stretching applies, we assume that all the t_is may be assumed to be the same and equal to t. Thus t denotes the response time of the plan and the total work done by the plan is given by $\sum_{i=1}^{n} w_i$. This pair $(t, \vec{w})$, where the components of the vector denotes the work done on that resource, is called a *resource vector*. It is usually denoted by $\vec{r}$. Analogous to the time descriptor, we define *resource descriptor* to be a pair of resource vectors $(\vec{rf}, \vec{rl})$, where $\vec{rf}$ represents the resource vector until the first tuple is output and $\vec{rl}$ represents the resource vector until the last tuple is output.

We use this resource descriptor to estimate the effect of resource contention on the response time.

5.2.2 Cost Calculus

We now extend the calculus of $\|$,; ,$\ominus$ on integers denoting time to resource vectors. In the following let $\vec{r}_1 = (t_1, \vec{w}_1)$ and $\vec{r}_2 = (t_2, \vec{w}_2)$ be two arbitrary resource vectors for a set of operations $S1$ and $S2$. The scenarios indicated for the applicability of each of the operators is the same as discussed earlier. The operations $+$ and $-$ on vectors denote the usual co-ordinate wise addition and subtraction.

The operation $\vec{r}_1; \vec{r}_2$ is defined as $\vec{r_1} + \vec{r_2}$ and $\vec{r}_1 \ominus \vec{r}_2$ is $\vec{r}_1 - \vec{r}_2$. Note that this accurately estimates the subtraction of the materialized front. Therefore, we use the usual vector minus operator instead of the $\ominus$ operator in the formulae. $\vec{r}_1 \| \vec{r}_2$ is defined as $(t, \vec{w})$ for each resource i where $t = max(t_1, t_2, max_i(w_1^i + w_2^i))$, for each resource i and $\vec{w} = \vec{w}_1 + \vec{w}_2$.

These operations are then used to define *tree* and the pipeline operator $|$ as before, where these take resource descriptors as arguments, rather than time descriptors. The operation of $|$ is slightly altered to take resource contention of the interleaved operations into account so that the resulting estimate is worse than SE if there is high resource contention.

Let $\mathbf{p} = (\vec{pf}, \vec{pl})$ and $\mathbf{c} = (\vec{cf}, \vec{pl})$ represent the resource descriptors for the producer and consumer of a pipeline respectively. Then, according to the previous formulation, the descriptor for $\mathbf{p} \mid \mathbf{c} = (\vec{rf}, \vec{rl})$, where
$\vec{rf} = \vec{pf}; \vec{cf}$ and $\vec{rl} = \vec{pf}; \vec{cf}; ((\vec{pl} - \vec{pf}) \| (\vec{cl} - \vec{cf}))$.
The parallel composition in $\vec{rl}$ is penalized by a (scalar) factor $\delta(k)$ to account for the synchronization overhead of pipelining. Thus $\vec{rl} = \vec{pf}; \vec{cf}; \delta(k) \times ((\vec{pl} - \vec{pf}) \| (\vec{cl} - \vec{cf}))$.

The factor $\delta(k)$ is obtained as follows: Let $(t', \vec{w})$ denote the resource vector for $S1 \| S2$. If t' is close to $t1 + t2$ then it means that the parallel pipelining incurs resource contention. On the other hand, if this t' is close to $max(t1, t2)$ then there is less resource contention. $\delta(k)$ is defined as parametrized linear interpolation of t' between $max(t1, t2)$ and $t1 + t2$; i.e.
$\delta(k) = 1 + k \times (t' - max(t1, t2))/(t1 + t2 - max(t1, t2))$.
In the above formula, k is an adjustable parameter. The linear interpolation may be replaced by other more accurate non linear estimates.

In summary, we have estimated the response time for IPE, DPE and CPE executions wherein the sources and deterrents of parallelism are taken into account.

6 Search

In this section we discuss how, given a query, the search space may be efficiently searched to obtain a query plan with optimal response time.

Starting with a System R style dynamic programming (DP) algorithm, we look at the impact of using response time as the optimization metric. The use of DP requires the problem to satisfy the principle of optimality[Bel57].

We show that response time violates this principle. Further there is no cost metric which correctly predicts response time while satisfying the properties required by DP. We extend DP to solve the problem and discuss how pruning metrics which correctly predict response time may be designed. We show our extended algorithm to have a practical complexity.

We show how limits on the amount of "extra" work may be incorporated and in fact exploited to increase the efficiency of search.

The reader is referred to [GHK92] for proofs of theorems and the extensions of algorithms to the search space of bushy trees.

6.1 Failure of DP

6.1.1 DP Revisited

Figure 1 shows a System R style DP algorithm which searches for an optimal-work plan in the space of left-deep join trees.

Some details of the algorithm are abstracted. The $accessPlan(R)$ routine produces the best plan for the relation R; $joinPlan(p', R)$ *extends* the join plan p' into another plan p in which the result of p' is joined with the relation R in the best possible way. The bindings available from p' may be exploited in joining with R. The

[3] As a consequence of this, we lose some ability to model hot spots.

[4] Assuming that time sharing memory by more than one process using virtual memory is prohibitively expensive

```
Input: An SPJ query q on relations R_1, ..., R_n
Output: A query plan P_opt for q with optimal work.
for i := 1 to n do
   optPlan({R_i}) := accessPlan(R_i)
for i := 2 to n do {
   for all S ⊆ {R_1, ..., R_n} s.t. ||S|| = i do {
      bestPlan := dummy plan with infinite cost
      for all R_j, S_j s.t S = {R_j} ∪ S_j do {
         p := joinPlan(optPlan(S_j), R_j)
         if p ≤_work bestPlan
            bestPlan := p
      }
      optPlan(S) := bestPlan
   }
}
P_opt := optPlan({R_1, ..., R_n})
```

Figure 1: DP Algorithm for Left-Deep Join Trees

predicate, $\leq_{work}$, is supplied by the cost model[5].

The algorithm proceeds by considering increasingly larger subsets of the set of all relations. Plans for a set of cardinality i are constructed as extensions of the best plan for a set of cardinality $i-1$. This is possible since the nature of the optimization problem is such that an optimal plan for a set of relations is an extension of an optimal plan for some subset of the set. This property, which is often called the *principle of optimality*, is exploited by the algorithm. Optimal plans for subsets are stored in the $optPlan()$ array and are reused rather than recomputed .

6.1.2 Fundamental Assumptions in DP

The proof of correctness for the DP algorithm of Figure 1 requires the following assumptions about the cost metric:

• *Principle of Optimality*: If two plans differ only in a subplan then the plan with the better subplan is also the better plan. In particular, for left-deep trees we need the following: if p_1 and p_2 are two plans for the same subquery, then

$$p_1 \leq_{cost} p_2 \Rightarrow (\forall i) joinPlan(p_1, R_i) \leq_{cost} joinPlan(p_2, R_i)$$

• *Total Order*: Plans can be totally ordered based on the cost metric.

$$(\forall p_1 p_2) not(p_1 \leq_{cost} p_2) \Rightarrow (p_2 \leq_{cost} p_1)$$

The cost metric, *work*, is an integer and therefore provides a total order. Whether it satisfies the principle of optimality depends on the choice of the cost model (which in turn depends on the chosen execution space).

For example, consider an execution space that allows nested loops and hash joins as join methods and index scan and relation scan as access methods. In any reasonable cost model for this execution space[6] the cost of the best way of joining a subquery with an additional relation will be dependent only on the subquery and not on the choice of the plan for the subquery. The cost of the join depends only on the logical aspects of the subquery such as available bindings and is independent of the physical plan. More formally, suppose p_1 is a plan for subquery q_1 and $joinwork(q_1, R)$ is the work cost of the best way of joining any relation R with the result of subquery q_1.

Definition (Physical Transparency): A cost model is said to exhibit *physical transparency* iff the following equation holds: $work(joinPlan(p_1, R)) = work(p_1) + joinwork(q_1, R)$

Theorem 1 *The cost metric for work, $\leq_{work}$,*
(1) is a total order and
(2) satisfies the principle of optimality under the physical transparency assumption.

A cost model that exhibits physical transparency is *not* reasonable if the execution space includes sort-merge join. It is now possible that the *physical* ordering of tuples produced by a plan may save work for a subsequent sort-merge join. Suppose plans p_1 and p_2 for subquery q are such that $p_1 \leq_{work} p_2$ but p_2 produces tuples in the order needed by a sort-merge join with relation R. It is possible to have $joinPlan(p_2, R) \leq joinPlan(p_1, R)$ if the interesting order produced by p_2 saves a sort pass in the sort-merge join algorithm. Thus the principle of optimality is violated.

Optimizers, such as in System R, retain some plans for subqueries with different "interesting orders" in the hope that they will prove useful by cutting down the cost of some join at a later stage. This *heuristic* solution to the violation of the principle of optimality is generally accepted to be sufficient in practice.

6.1.3 DP and Response Time

One important question is whether it is possible to use the response time metric, $RT(p)$, as the cost metric in DP.

Does response time satisfy the principle of optimality? The not very surprising answer is that it depends on the choice of the cost model. But we believe that any reasonable cost model for response time will violate the principle of optimality quite violently. The intuitive reason is that data dependencies and resource contention between parts of a plan are crucial factors for response time and any cost model that takes them into account will violate the principle of optimality. The following example shows how resource contention leads to a violation of the principle of

[5] Given two plans p_1 and p_2, we define $p_1 \leq_\alpha p_2$ and $\alpha(p_1) \leq \alpha(p_2)$ to be equivalent notations. Examples of the $\alpha()$ function are $work()$ and the response time function, $RT()$.

[6] The reader is reminded that cost models are built by human beings. Given the quirks of human nature we can only appeal to "reasonableness" when saying that a property of the cost model follows from the choice of the execution space.

optimality.

Example 3 Consider a database consisting of the tables $CTR(course, time, room)$ and $CI(course, instructor)$. Suppose CTR has two indexes, a clustered index $I_{CT}(course, time)$ (stored on disk 1) and an unclustered index $I_{CR}(course, room)$ (stored on disk 2) and CI has one index $I_C(course)$ (stored on disk 1).

The query $\pi_{course}(CTR \bowtie CI)$ may be computed purely by scanning indexes. We consider two nested-loops join plans of the form[7] $NL(p, indexScan(I_C))$ where p is either $p_1 = indexScan(I_{CT})$ or $p_2 = indexScan(I_{CR})$. Suppose the resource vector (considering disk1 and disk2 to be the only significant resources) for p_1 is $\langle(20,20),(0,0)\rangle$; for p_2 is $\langle(0,0),(25,25)\rangle$ and for $NL(*, indexScan(I_C))$ (the join without counting p) is $\langle(40,40),(0,0)\rangle$. Since nested-loops is pipelined, we apply the calculus on resource vectors to obtain the resource usage of $NL(p_1, indexScan(I_C))$ to be $\langle(60,60),(0,0)\rangle$ and that for $NL(p_2, indexScan(I_C))$ to be $\langle(40,40),(25,25)\rangle$.

Applying the definition of response time, $RT(p_1) = 20$ and $RT(p_2) = 25$. However,

$$RT(NL(p_1, indexScan(I_C))) = 60 \text{ and}$$

$$RT(NL(p_2, indexScan(I_C))) = 40$$

showing that response time violates the principle of optimality. ∎

The violation of the principle of optimality due to interesting orderings did pose a problem in the optimization for work. The same principle is violated due to resource contention when optimizing for response time. We expect the problem to be more difficult for response time due to an explosion in the number of causal factors. Resource contention is likely to happen between any two parts of a plan. The resource usage of a plan varies depending on what resources are used for sorting or partitioning intermediate results in addition to variables such as access methods, join methods and join ordering.

A natural question to ask next is whether it is possible to *design a new metric* that correctly predicts response time and satisfies the two properties that we require. Such a metric could then be used for pruning in DP. We will refer to such a metric as a *pruning metric.*

Definition (Correct Prediction): A pruning metric α will be said to *correctly predict* response time iff
$p_1 \leq_\alpha p_2 \Rightarrow p_1 \leq_{RT} p_2$

Theorem 2 *There does not exist any pruning metric that correctly predicts response time, provides a total order and satisfies the principle of optimality.*

The implication of this theorem is that it is not possible to use DP for optimizing response time without relaxing our requirements. We must develop variants of DP based on giving up one of the two fundamental assumptions that we noted. Giving up the principle of optimality essentially amounts to giving up the ability to prune the search space at all and leads us to brute force algorithms for exhaustive search. Therefore the less devastating alternative is the relaxation of requiring a total order.

We show this to be a practical direction by showing
• a generalization of DP that is capable of utilizing a partial order and provides acceptable performance (Section 6.2)
• how pruning metrics which correctly predict response time, provide a partial order and satisfy the principle of optimality may be designed (Section 6.3).

6.2 Generalization of DP for Partial Orders

We may define a less-than relation which provides a partial order in l-dimensions as follows: **Definition (l-Dimensional Less Than):** Given two points, $\langle d_1^1, \ldots, d_l^1\rangle$ and $\langle d_1^2, \ldots, d_l^2\rangle$in l-dimensional space, $\leq_l$ is defined as:
$\langle d_1^1, \ldots d_l^1\rangle \leq_l \langle d_1^2, \ldots d_l^2\rangle$ iff $d_i^1 \leq d_i^2$ for $i = 1 \ldots l$

The next section shows that it is possible to design a pruning metric, α, such that $\alpha(plan)$ is an l-dimensional vector[8]. Such pruning metrics will, by design, correctly predict response time as well as satisfy the principle of optimality.

Figure 2 shows the generalization of DP to use a partial order. The important difference from Figure 1 is highlighted: the actions in the innermost loop(lines 7,8,9 in Figure 1) are replaced by new actions (lines L1 through L8 in Figure 2).

The general idea is that instead of one optimal plan for each subset of relations, we keep a *set* of incomparable but optimal plans. These incomparable plans form a unique cover-set for all possible plans.

Plans for a set S of cardinality i are constructed as extensions of plans in the cover-sets for subsets of S of cardinality $i-1$. The cover-set of these plans is computed and retained as the set of optimal plans for S. By the principle of optimality it follows that a cover-set computed in this manner is indeed the cover-set of all possible plans for S.

The final result of the algorithm is obtained by choosing from the set of plans for $\{R_1, \ldots, R_n\}$ with the best cost.

The time and space complexity of the search for optimal plans depends on the size of the set of best plans for each set of relations. Let us assume that this size is k. The time and space complexity of the algorithm may then be shown to be $kn2^{n-1}$ and $k\binom{n}{\lceil \frac{n}{2} \rceil}$ respectively. Whether this is an acceptable complexity depends on the value of k.

Definition (Cover Set): A *cover-set* C for a set of points P with respect to a relation $\leq_p$ is a subset of P such that all points in C are incomparable and every point

[7] For brevity, we use NL to stand for nested-loops in these expressions.

[8] Given two plans p_1 and p_2, we define $p_1 \leq_{\alpha_l} p_2$ and $\alpha(p_1) \leq_l \alpha(p_2)$ to be equivalent notations.

Input: An SPJ query q on relations $R_1, \ldots, R_n$
Output: A query plan P_{opt} for q with optimal cost.

for $i := 1$ to n **do**
    $optPlans(\{R_i\}) := accessPlans(R_i)$
for $i := 2$ **to** n **do** {
    for all $S \subseteq \{R_1, \ldots, R_n\}$ s.t. $\|S\| = i$ **do** {
        $bestPlans := \emptyset$
        for all R_j, S_j s.t $S = \{R_j\} \cup S_j$ **do** {
            for all $p \in optPlans(S_j)$ **do** {
                $new := joinPlan(p, R_j)$
                if $\nexists p_{better} \in bestPlans$ s.t $p_{better} \leq_{\alpha_l} new$ {
                    for all $p_d \in bestPlans$ s.t $new \leq_{\alpha_l} p_d$
                        delete p_d from $bestPlans$
                    insert new into $bestPlans$
                }
            }
        }
        $optPlans(S) := bestPlans$
    }
}
$P_{opt} := bestCost(optPlans(\{R_1, \ldots, R_n\}))$

Figure 2: Partial Order DP for Left-Deep Join Trees

p' in P is "covered by" some point c' in C (i.e. $c' \leq_p p'$).

Theorem 3 *The expected size of the cover-set (with respect to $\leq_l$) of m randomly chosen points in l-dimensional space is at most $2^l(1-(1-\frac{1}{2^l})^m)$ provided the probability distribution is independent along each dimension.*

The theorem essentially says that 2^l is an upper bound for k. The assumption of independence of dimensions is likely to be optimistic for most pruning metrics. Therefore 2^l is an upper bound under an optimistic assumption.

In summary, if the number of dimensions, l, is kept small then DP with partial orders provides acceptable performance.

6.3 Pruning Metrics and Approximation Techniques

One way of fixing the situation shown in Example 3 is to use the resource vector itself as the pruning metric. The resource vector $rv(p)$, of a plan p, is a l-dimensional vector. Therefore resource vectors may be partially ordered using the $\leq_l$ relation. Obviously, a resource vector correctly predicts response time.

The general idea in designing a pruning metric is to try to make the antecedent in the definition of the principle of optimality false for the cases in which the consequent is false.

For example tuple ordering may be incorporated as an additional dimensions in the pruning metric. This requires a $\leq_{ordering}$ relation over orderings. If an ordering is represented as a sequence of column numbers then the $\leq_{ordering}$ relation may be taken to be "subsequence of". Data dependence and data partitioning may be incorporated in a manner similar to ordering.

However, in order to keep the cost of search acceptable, we must be careful not to have too many dimensions in the pruning metric. If two resources closely track each other, they should be aggregated and modeled as a single resource. For example the RAID system in XPRS[HS91] should be considered to be a single resource since data is hash partitioned on all disks with the intention of keeping each disk equally busy. If a resource is expected to rarely be the bottleneck, it should be ignored. For example resource contention at the disk controller is rarely a problem in a well designed system.

6.4 Discussion

Bushy trees offer more scope for independent parallelism since two subtrees may be executed in parallel. The reader is referred to [GHK92] for the DP algorithms extended to the space of bushy trees.

Search over bushy trees is much more costly than over left-deep trees. The complexity of DP increases for two reasons. Firstly and more significantly the space of plans is much larger. Second and less obviously we are now forced to keep plans for all possible bindings of a subquery. This multiplies the complexity by a factor of 2^b (in the worst case) where b is the number of columns in the output of the subquery.

Table 1 summarizes the time and space complexity of several algorithms. We observe that using the search space of bushy trees instead of left-deep trees takes the time complexity of search from $O(2^n)$ to $O(3^n)$. Using a partial order over l dimensions instead of a total order multiplies the time complexity by 2^l.

Work bounds may be easily incorporated in the search algorithm and in fact cut down the search space.

For example, the limit on throughput degradation may be implemented by first running a work-optimizer (such as Figure 1) to obtain, W_o, the optimal work-cost of a query. The chosen pruning metric for the response-time algorithm is simply extended by the work limit giving a more stringent partial order. A similar approach may be taken for incorporating the cost-benefit ratio.

7 Conclusions

In this paper, we have proposed a novel formulation of the query optimization problem: *Minimize response time subject to constraints on throughput*, which was motivated as the dual of the traditional DBMS problem. We addressed this novel problem in the context of Select-Project-Join queries by extending the execution space, cost model and search algorithm that are widely used in commercial DBMSs. We incorporated the sources and deterrents of parallelism in the traditional execution space. We showed that a cost model can predict response time while account-

Algorithm	size of space (#plans)	time complexity (#plans considered)	space complexity (max #plans stored)
brute force for left-deep	$n!$	$n!$	1
DP for left-deep	$n!$	$n2^{n-1}$	$\binom{n}{\lceil \frac{n}{2} \rceil}$
p.o. DP for left-deep	$n!$	$n2^{n-1}2^l$	$2^l\binom{n}{\lceil \frac{n}{2} \rceil}$
brute force for bushy	$\frac{(2(n-1))!}{(n-1)!}$	$\frac{(2(n-1))!}{(n-1)!}$	1
DP for bushy	$\frac{(2(n-1))!}{(n-1)!}$	$2^b(3^n$-2^{n+1} +n +1)	2^b2^n
p.o. DP for bushy	$\frac{(2(n-1))!}{(n-1)!}$	$2^l2^b(3^n$-2^{n+1} +n+1)	$2^l2^b2^n$

Table 1: Comparison of Search Algorithms

ing for the new aspects due to parallelism. We observed that the response time optimization metric violates a fundamental assumption in dynamic programming algorithm that is the linchpin in the optimizers of most commercial DBMSs. We extended dynamic programming and showed how optimization metrics which correctly predict response time may be designed.

It is a widely held belief that a comprehensive parallel-query optimizer is not available today[DG90]. While we have addressed some of the critical aspects of this problem, there are many open questions. We discuss some of them below.

While we model the sources and deterrents of parallelism in our execution space there are several runtime aspects such as scheduling that are candidates for inclusion. In general, investigating what aspects of the runtime execution should be decided at compile time is an interesting question.

We have addressed the usage of resources that are preemptable (i.e time-sliceable). Incorporating non-preemptable resources such as memory is an open question. Further, our cost model has not quantified the overheads due to pipelining and cloning. This would be necessary in any practical system.

Having observed the drawbacks of the traditional dynamic programming approach, we have presented criteria for designing appropriate pruning metrics. Evaluation of pruning metric alternatives is necessary. We have discussed the search algorithm in the context of left-deep trees. Use of bushy trees is widely believed to provide higher degrees of parallelism. We observe that, even for ten relations, this increases the size of the search space by three orders of magnitude. Consequently use of non-exhaustive search algorithms may be imperative.

Acknowledgements: We thank Marie-Anne Neimat for her advice, encouragement and support from the early stages of this research. We are grateful to Donovan Schneider for sharing his Gamma experience. Thanks are also due to Tim Connors, Curtis Kolovson, Kevin Wilkinson and all other members of the Database Technology Department. We also thank Don Batory, Stefano Ceri, Avi Silberschatz, and Gio Wiederhold for providing useful feedback.

References

[AHY83] P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization Algorithms for Distributed Queries. *IEEE Transaction on Software Engineering*, 9(1), 1983.

[Bel57] R.E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[CHK+91] T. Connors, W. Hasan, C. Kolovson, M.-A. Neimat, D. Schneider, and K. Wilkinson. The Papyrus integrated data server. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991.

[DG90] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of Database Processing or a Passing Fad? *ACM-SIGMOD Record*, 19(4):104–112, December 1990.

[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.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. Technical report, HP Laboratories, 1992. HPL-DTD-92-3.

[Gra91] J. Gray. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann Publishers, Inc., 1991.

[HS91] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, December 1991.

[PMC+90] H. Pirahesh, C. Mohan, J. Cheung, T.S. Liu, and P. Selinger. Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches. Technical report, IBM Research Division, September 1990. IBM Research Report RJ 7724.

[SAC+79] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 1979.

[Sch90] D. A. Schneider. *Complex Query Processing in Multiprocessor Database Machines.* PhD thesis, University of Wisconsin—Madison, September 1990. Computer Sciences Technical Report 965.

[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.