Coloring Away Communication in Parallel Query Optimization

WAQAR HASAN
Stanford University
and
Hewlett-Packard Laboratories
hasan@cs.stanford.edu

RAJEEV MOTWANI*
Department of Computer Science
Stanford University
Stanford, CA 94305
rajeev@cs.stanford.edu

Abstract

We address the problem of finding parallel plans for SQL queries using the two-phase approach of *join ordering and query rewrite* (JOQR) followed by *parallelization*. We focus on the JOQR phase and develop optimization algorithms that account for communication as well as computation costs. Using a model based on representing the partitioning of data as a color, we devise an efficient algorithm for the problem of choosing the partitioning attributes in a query tree so as to minimize total cost. We extend our model and algorithm to incorporate the interaction of data partitioning with conventional optimization choices such as access methods and strategies for computing operators. Our algorithms apply to queries that include operators such as grouping, aggregation, intersection and set difference in addition to joins.

1 Introduction

An important challenge in parallel database systems [DG92, Val93, BCC$^+$90, DGG$^+$86] is *parallel query optimization*. This is the problem of finding optimal parallel plans for decision-support queries that include operators such as aggregation, grouping, union, intersection, set difference and calls to external functions in addition to joins. Following Hong and Stonebraker [HS91], we break the problem into two phases: *join ordering and query rewrite* (JOQR) followed by *parallelization*. This paper focuses on the JOQR phase and develops optimization algorithms to find query trees that minimize the computation and communication costs of parallel execution.

Partitioned parallelism [DG92] which exploits horizontal partitioning of relations is an important way of reducing the response time of queries. This may require data to be *repartitioned* among sites thus incurring substantial communication costs.

Example 1.1 Assume that the tables `Emp(enum,`<u>`name`</u>`, areaCode, number)` and `Cust(name,`<u>`areaCode`</u>`, number)` are horizontally partitioned on two sites on the underlined attribute. Suppose we want to determine the number of employees who are also customers and group the result by `areaCode`. After deciding it reasonable to guess an employee and a customer to be the same person if they have the same name and phone number, we may write the following query (SQL2 [X3H92] syntax used):

Select areaCode, **Count**(*)
From Cust **Intersect**
    (**Select** name, areaCode, number **From** Emp)
Group by areaCode;

Figure 1 shows two query trees that differ only in how data is repartitioned. Since tuples with the same `areaCode` need to come together, `GroupBy` is partitioned by `areaCode`. However, `Intersect` may be partitioned on *any* attribute. If we choose to partition it by `areaCode`, we will need to repartition the (projected) `Emp` table. If we partition by `name`, we will need to repartition the `Cust` table as well as the output of the intersection. Thus one or the other query tree may be better depending on the relative sizes of the intermediate tables. □

We address the problem of choosing the partitioning attributes in a query tree to minimize the sum total of communication and computation (i.e., disk and cpu costs other than communication) costs. By regarding partitioning attributes as colors, we model it as a *query tree coloring* problem in which repartitioning cost is saved when adjacent operators have the same color. Since the choice of partitioning interacts with decisions such as the choice of join predicates and join methods, we generalize the problem to incorporate such interactions between communication and computation. We generalize color differences to correspond to repartitioning

* Supported by an Alfred P. Sloan Research Fellowship, an IBM Faculty Development Award, an OTL grant, and NSF Young Investigator Award CCR-9357849, with matching funds from IBM, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995

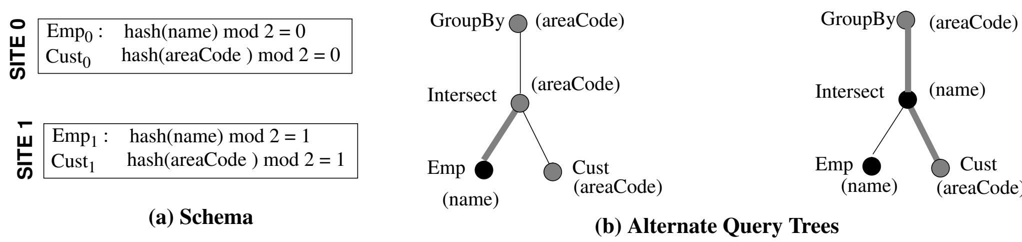

Figure 1: Query Trees: Hatched edges show repartitioning

data, sorting it, and/or building an index on it. This *query tree annotation and coloring* problem requires choosing a strategy for each operator and a color for each input and output so as to minimize total cost.

Communication costs are extremely significant in processing queries in parallel and primarily consist of the cpu cost of sending and receiving messages. Gray [Gra88] estimates communication using inter-processor messages over a LAN to be two orders of magnitude more expensive than intra-processor communication through procedure calls. Pirahesh et al [PMC$^+$90] use a model based on projected path lengths of MVS and DB2 to estimate that the total path length of a specific query doubles when all pipelines are modified to communicate data across processors.

We have performed detailed experiments with NonStop SQL/MP that break down the costs of queries into the costs of individual operators and communication. The interested reader is referred to [EGH95, Has95] for details. These experiments show that repartitioning cost can exceed the cost of common operators such as scans, groupings and joins, and thus establish the need for models and algorithms such as developed in this paper.

The query tree coloring problem is related to the classical problem of MULTIWAY CUTS. Dahlhaus, Johnson, Papadimitriou, Seymour and Yannakakis [DJP$^+$92] show several versions of the problem to have high complexity. However, the restriction of the problem to trees is solvable in polynomial time [CR91, ES94]. Our contribution is to simplify and extend known theory to adapt it for query optimization.

Our work is in contrast to the use of a conventional query optimizer by Hong and Stonebraker [HS91, Hon92] as the JOQR phase in XPRS. However, it should be noted that Hong [Hon92] conjectured the XPRS approach to be inapplicable to architectures such as shared-nothing that have significant communication costs. Other work on parallel query optimization [SE93, LST91, SYT93, CLYY92, HLY93, ZZBS93, GHK92] also ignored modeling communication overheads of parallelism.

Our earlier work [HM94, CHM95] focussed on the parallelization phase and has developed scheduling algorithms that account for the trade-off between parallelism and communication.

Though query processing in parallel and distributed databases [CP84, OV91, YC84] is fundamentally similar, repartitioning intermediate results to reduce response time did not receive much attention until the appearance of parallel machines. Shasha and Wang [SW91] investigated heuristics for join ordering that take repartitioning cost into account. These heuristics apply only to joins. Further, they assume the cost of a join to be proportional to the sum of the sizes of operands thus excluding common join methods that use indexes or sorting.

Section 2 defines the *Query Tree Coloring* and *Query Tree Annotation and Coloring* optimization problems that are solved in this paper. Section 3 develops an efficient algorithm for query tree coloring and shows several extensions. Section 4 reuses the basic ideas in coloring a query tree to develop an efficient algorithm that minimizes the combined communication and computation costs. Section 5 summarizes our contributions and discusses future work.

2 A Model for the Problem

2.1 S/W Architecture of a Parallel Query Optimizer

We adopt a two-phase approach [HS91] to parallel query optimization: *JOQR* followed by *parallelization*. JOQR is similar in functionality to a conventional query optimizer. Given an SQL query, it produces an annotated query tree that fixes the order of operators and other procedural decisions such as the strategy for each join. This phase minimizes the *total cost* for computing the query. The parallelization phase constructs a parallel plan (i.e., a schedule) for the annotated query tree to minimize *response time*. It uses a detailed cost model that incorporates timing constraints between operators and makes decisions about allocation of resources.

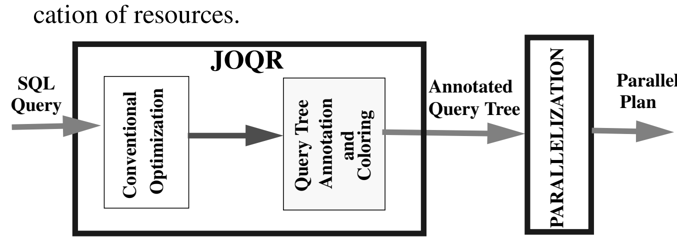

Figure 2: Two-Phase Query Optimization (Algorithms developed in this paper are shaded)

One way of using the algorithms developed in this paper is to incorporate them in the JOQR phase as a post-pass to conventional optimization as shown in Figure 2. Some alternatives are discussed in [Has95].

2.2 Partitioning

We begin with a formal definition of partitioning.

Definition 2.1 A *partitioning* is a pair (a, h) where a is an attribute and h is a function that maps values of a to non-negative integers.

Given a table T, a partitioning produces fragments $T_0, \ldots, T_k$ such that a tuple $t \in T$ occurs in fragment T_i if and only if $h(t.a) = i$. For example, the partitioning of `Emp` in Example 1.1 is represented as `(name, hash(name) mod 2 )`. The function `hash(name) mod 2` is applied to each tuple of `Emp` and the tuple placed in fragment $\texttt{Emp}_0$ or $\texttt{Emp}_1$ depending on whether the function returns 0 or 1.

Partitioning provides a source of parallelism since the semantics of most database operators allows them to be applied in parallel to each fragment. Suppose $S_0, \ldots, S_k$ and $T_0, \ldots, T_k$ are fragments of tables S and T produced by the *same* partitioning $\alpha = (a, h)$.

Definition 2.2 A *unary operator* f is *partitionable* with respect to α if and only if $f(S) = f(S_0) \cup \ldots \cup f(S_k)$. A *binary operator* f is *partitionable* with respect to α if and only if $f(S, T) = f(S_0, T_0) \cup \ldots \cup f(S_k, T_k)$.

Example 2.1 Suppose that the two tables `Emp'(name,` <u>`areaCode`</u>`, number)` and `Cust(name,` <u>`areaCode`</u>`, number)` are each partitioned across two sites using the hash function `hash(areaCode) mod 2`. Since the tables have the *same* partitioning, $\texttt{Emp}' \cap \texttt{Cust} = (\texttt{Emp}'_0 \cap \texttt{Cust}_0) \cup (\texttt{Emp}'_1 \cap \texttt{Cust}_1)$. This permits $\texttt{Emp}' \cap \texttt{Cust}$ to be computed by computing $\texttt{Emp}'_0 \cap \texttt{Cust}_0$ and $\texttt{Emp}'_1 \cap \texttt{Cust}_1$ in parallel. □

Definition 2.3 An *attribute sensitive* operator is partitionable only for partitionings that use a distinguished attribute. An *attribute insensitive* operator is partionable for all partitionings.

The equation $S \bowtie T = \cup_i (S_i \bowtie T_i)$ holds only if both S and T are partitioned on the (equi-)join attribute. Thus join is attribute sensitive. Similarly, *grouping* is attribute sensitive since it requires partitioning by the grouping attribute. `UNION`, `INTERSECT` and `EXCEPT` (set difference), *aggregation*, *selection* and *projection* are attribute insensitive. External functions and predicates may be either sensitive or insensitive.

2.3 Repartitioning Cost

Communicating tuples between operators that use different partitionings requires redistributing tuples among sites. Some percentage of tuples remain at the same site under both partitionings and therefore do not need to be communicated across sites. We believe that the crucial determinant of the extent of communication cost, given a "good" scheduler, is the *attribute* used for partitioning. We argue the following *all or nothing* assumption to be reasonable.

Good Scheduler Assumption: *If two communicating operators use the same partitioning attribute,* no *inter-site communication is incurred. If they use distinct partitioning attributes then* all *tuples need to be communicated across sites.*

Consider the case of two operators with different partitioning attributes. The greatest savings in communication occur if the two operators use the same set of processors. If a table with m tuples equally partitioned across k sites is repartitioned on a different attribute, then assuming independent distribution of attributes, $(1 - \frac{1}{k})m$ tuples may be expected to change sites. Thus it is reasonable to assume all m tuples to be communicated across sites.

Now consider the case of two operators with the same partitioning attribute. We believe that any good scheduler will choose to use the same partitioning function for both operators since it not only saves communication cost but also permits both operators to be placed in a single process at each site.

For example, our assumption is exactly true for *symmetric* schedulers (such as those used in Gamma [DGG+86]) that partition each operator equally over the same set of sites.

2.4 Optimization Problems

We associate colors with nodes as corresponding to the partitioning attribute.

Definition 2.4 The *color* of a node in a query tree is the attribute used for partitioning the node. An edge between nodes i and j is *multi-colored* if and only if i and j are assigned distinct colors.

In a query tree, the nodes for attribute sensitive operators or base tables are *pre-colored* while we have the freedom to assign colors to the remaining *uncolored* nodes.

We will associate a weight c_e with each edge e to represent the cost of repartitioning. Since this cost is incurred only if the edge is multi-colored, the total repartitioning cost is the sum of the weights of all multicolored edges. Thus the optimization problem is:

Query Tree Coloring Problem: *Given a query tree $T = (V, E)$, weight c_e for edge $e \in E$, and colors for some subset of the nodes in V, color the remaining nodes so as to minimize the total weight of multicolored edges.*

Conventional cost models [SAC+79] provide estimates for the size of intermediate results. The weight c_e may therefore be estimated as a function of the size of intermediate results. Our work is applicable regardless of the model used for estimation of intermediate result sizes or the function for estimation of repartitioning cost. We assume some method of estimating c_e to be available.

Query tree coloring models only communication costs. The next problem is to extend the model to capture the interaction between communication and computation costs. We extend the notion of a color to capture physical properties that impact the cost of computation. Now recoloring of data corresponds to repartitioning it, sorting it, and/or building an index on it. We associate a set of strategies with each operator. Each strategy for an operator is an alternate method for computing the operator. The cost of an operator consists of the cost of applying the strategy plus the cost of recoloring the inputs to the colors expected by the strategy. The cost of a tree is the sum of the costs of all operators.

Query Tree Annotation and Coloring Problem: *Given a query tree, a collection of strategies for each operator, and colors for the leaf nodes, find a strategy and input and output colors for each node so as to minimize total tree cost.*

The next section deals with the query tree coloring problem and several extensions. Section 4 deals with the query tree annotation and coloring problem.

3 Query Tree Coloring

In this section we develop an algorithm for coloring a query tree to minimize the cost of repartitioning. The problem of coloring the nodes of a tree may equivalently be viewed as a problem of cutting/collapsing edges. Edges between nodes of different colors may be considered cut while edges between nodes of the same color may be considered collapsed. This view is helpful since it allows us to constrain colors of adjacent nodes to be identical or distinct without fixing the actual colors.

Example 3.1 Figure 3(i) shows the query tree for a query to count parts used in manufacture of aircraft but not of cars or boats. The three base tables are assumed to be partitioned on distinct attributes (colors) A, B, and C. Figures 3(ii) and 3(iii) show two colorings. The cost of a coloring is the sum of the cut edges which are shown hatched. The coloring in Figure 3(ii) is obtained by the simple heuristic of coloring an operator so as to avoid repartitioning the most expensive operand. The minimal coloring is shown in Figure 3(iii); here, `UNION` is not partitioned on the partitioning attributes of any of its operands. □

The query tree coloring problem is related to the classical problem of multiway cuts with the difference that multiway cut restricts pre-colored nodes to have distinct colors. Multiway cut is NP-hard for graphs but solvable in polynomial time for trees [DJP+92]. Chopra and Rao [CR91] developed an $O(n^2)$ algorithm (where n is the number of tree nodes) for multiway cut for trees using linear programming techniques. Our DLC algorithm is substantially simpler and has a running time of $O(n)$. Erdos and Szekely [ES94] provide an $O(nc^2)$ algorithm (where c is number of colors) for the case of repeated colors. Our ColorSplit algorithm is an $O(nc)$ algorithm based on a better implementation of their ideas.

In Section 3.1, we develop an understanding of the problem by presenting some simplifications. In Section 3.2, we develop a simple linear time algorithm for the case when all pre-colored nodes have distinct colors. Section 3.3 uses dynamic programming to develop an $O(nc)$ algorithm for the general case (n is the number of tree nodes and c the number of colors). Section 3.4 discusses extensions to deal with optimization opportunities provided by choices in access methods (due to indexes, replication of tables) and choices in join and grouping attributes.

3.1 Problem Simplification

The problem of coloring a tree can be reduced to coloring a *set* of trees which have the special property that all interior nodes are uncolored and all leaves are pre-colored. This follows from the following observations which imply that colored interior nodes may be split into colored leaves, and uncolored leaves may be deleted.

(Split) A colored interior node of degree d may be split into d nodes of the same color and each incident edge connected to a distinct copy. This decomposes the problem into d sub-problems which can be solved independently.

(Collapse) An uncolored leaf node may be collapsed into its parent. This gives it the same color as its parent which is minimal since it incurs zero cost.

The following procedure achieves the simplified form in time linear in the number of nodes in the original tree. Figure 4 illustrates the simplification process.

Algorithm 3.1 Procedure *Simplify*

while ∃ uncolored leaf l with parent m **do**
  collapse l with m;
while ∃ colored interior node m with degree d **do**
  split m into d copies with each copy
  connected to distinct a edge.

3.2 Algorithm for Distinct Pre-Colorings

We first develop an algorithm for the case when all pre-colored nodes are restricted to have *distinct* colors. By the discussion in the previous section, we need to develop an algorithm for trees in which a node is pre-colored if and only if it is a leaf node.

Definition 3.1 A node is a *mother node* if and only if all adjacent nodes with at most one exception are leaves. The leaf nodes are termed the children of the mother node.

The algorithm repeatedly picks mother nodes and processes them by either cutting or collapsing edges. Each such step creates smaller trees while preserving the invariant that all and only leaf nodes are colored. We are finally

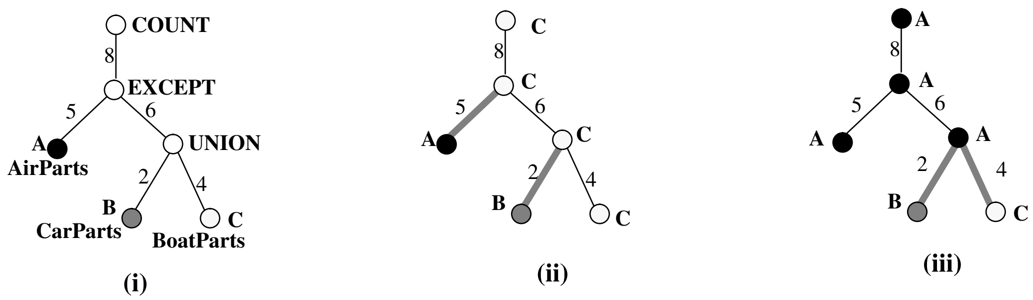

Figure 3: (i) Query Tree; (ii) Coloring of cost 7; (iii) Minimal Coloring of cost 6

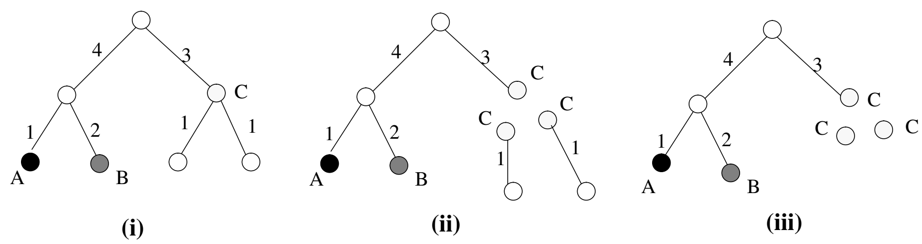

Figure 4: (i) Split colored interior node (ii) Collapse uncolored leaves

left with a set of trivial trees that may be easily colored. Before presenting the algorithm we show two lemmas that make such processing possible.

Suppose m is a mother node with edges $e_1, \ldots, e_d$ to leaf children $v_1, \ldots, v_d$. Assume we have numbered the children in order of non-decreasing edge weight, i.e., $c_{e_1} \leq c_{e_2} \leq \cdots \leq c_{e_d}$.

Lemma 3.1 *There exists a minimal coloring that cuts* $e_1, \ldots, e_{d-1}$.

Proof: The proof uses the fact that all leaves have distinct colors. In any coloring at least $d-1$ leaves have a color different from m. If the optimal colors m differently from all leaves, the lemma is clearly true. If not, then suppose m has the same color as leaf v_i and let this color be A. Let the color of v_d be B. Change all A-colored nodes (other than v_i) to be B-colored nodes. Such a change is possible since no pre-colored node other than v_i may have color A. Since $c_{e_i} \leq c_{e_d}$, the new coloring has no higher cost. □

Notice that after we cut edges using the above lemma, we are left with a mother node with one child. Consider the case in which the mother node has a parent. Then the mother node is of degree 2 and the following lemma shows how we can deal with this case. Let the incident edges be e_1 and e_2 such that $c_{e_1} \leq c_{e_2}$. Since m is not pre-colored, a minimal coloring will always be able to save the cost of the heavier edge.

Lemma 3.2 *There is a minimal coloring that collapses* e_2.

The last case is when the mother node has only one child and no parent. In other words, the tree has only two nodes. Such trees are termed *trivial* and can be optimally colored by giving the child the color of its mother.

Notice that the invariant that exactly leaf nodes are colored remains true after any of the lemmas is used to cut/collapse edges. Thus, for any non-trivial tree, one of the two lemmas is always applicable. Since the application of a lemma reduces the number of edges, repeated application leads to a set of trivial trees. These observations lead to the algorithm given below for find a minimal coloring.

Algorithm 3.2 Algorithm DLC

1. **while** $\exists$ mother node m of degree at least 2 **do**
2. Let m have edges e_1, …, e_d to d children;
3. Let $c_{e_1} \leq \ldots \leq c_{e_d}$;
4. **if** $d > 1$ **then** cut e_1, …, e_{d-1}
5. **else** Let e_p be the edge from m to its parent;
6. **if** $c_{e_p} < c_{e_1}$ **then** collapse e_1
   else collapse e_p.
7. **end while**;
8. color trivial trees.

Since each iteration reduces the number of edges, the running time of the algorithm is linear in the number of edges. The following example should help to clarify this algorithm.

Example 3.2 Figure 5 shows a trace of the algorithm for finding the minimal coloring for the tree of Example 3.1. In Step 1, the mother node is `Union` with degree 3 and its cheapest child is cut away. Step 2 has `Union` as a mother node of degree 2 and collapses the edge from the mother node to its parent. Step 3 and Step 4 are cutting and collapsing steps with `Except` as the mother node. We obtain a set of trivial trees. The coloring for the original tree is extracted by keeping track of node collapses. □

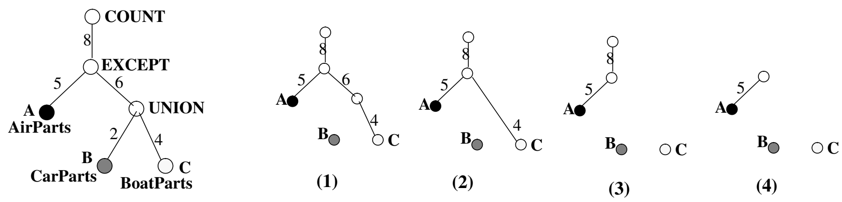

Figure 5: Trace of Algorithm DLC

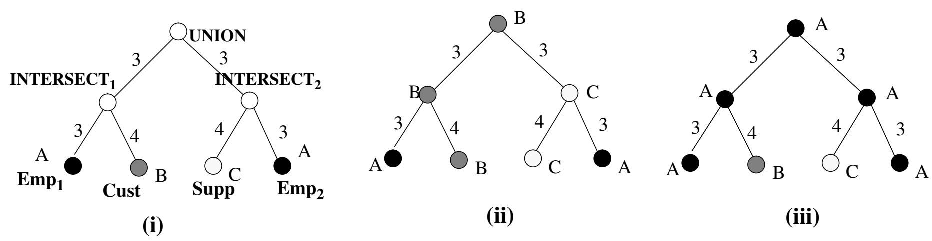

Figure 6: (i) Query Tree (ii) Suboptimal DLC coloring (cost=9) (iii) Optimal coloring (cost=8)

3.3 Algorithm for Repeated Colors

In the last section, we developed the DLC algorithm for the case when no two pre-colored nodes have the same color. The following example shows that DLC may not find the optimal coloring when colors are repeated.

Example 3.3 Figure 6(i) shows a query tree for a query that finds employees who are customers as well as suppliers. Taking the tables `Supp`, `Cust`, and `Emp` to be partitioned on distinct attributes, we pre-color them by colors *A*, *B*, and *C* respectively. We now have repeated colors and two "widely separated" leaves are both pre-colored *A*. The DLC algorithm finds the sub-optimal coloring shown in Figure 6(b) since it makes a *local* choice of cutting away the *A* leaves. The optimal coloring shown in Figure 6(c) exploits the like colored leaves to achieve a lower cost. □

The reason DLC fails to find the optimal coloring in the above example is that repeated colors make it difficult to make local choices of colors. One option is to use the brute force approach of enumerating all possible colorings. Unfortunately the number of colorings for c colors and n nodes is c^n. This exponential complexity makes the brute force approach undesirable.

We now develop an algorithm that exploits one of the observations made in Section 3.1. We observed that a colored interior node may be split to decompose the problem into smaller subproblems that are independently solvable. Since interior nodes are all initially *uncolored*, this observation can only be exploited after coloring an interior node. A further observation that we will make is that the subproblems can be posed in a manner that makes them independent of the color chosen for the interior node. We now develop an efficient algorithm based on dynamic programming that exploits problem decomposition while trying out different colors for each node.

Definition 3.2 $Optc(i, A)$ is defined to be the minimal cost of coloring the subtree rooted at i such that i is colored A. If node i is pre-colored with a color different from A, then $Optc(i, A) = \infty$.

Definition 3.3 $Opt(i)$ is defined as $\min_a Optc(i, a)$, i.e., the minimal cost of coloring the subtree rooted at i irrespective of the color of i.

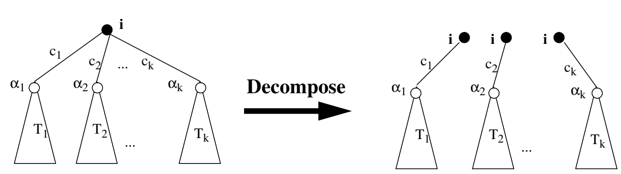

Figure 7: Problem Decomposition after Coloring Node i

Consider a tree (Figure 7) in which root node i has children $\alpha_1, \alpha_2, \ldots, \alpha_k$. Let the edge from i to α_j have weight c_j, and let T_j be the subtree rooted at α_j. If we fix a color for node i, we can decompose the tree into k "new" trees by splitting node i into k copies. Since the only connection between new trees was through i, they may now be colored *independently* of each other. Thus $Optc(i, A)$ is the sum of the minimal colorings for the k new trees.

Consider the jth new tree. The minimal coloring either pays for the edge (i, α_j) or it does not. If it pays for the edge, then it can do no better than using the minimal coloring for T_j, thus incurring a cost of $c_j + Opt(\alpha_j)$. If it does not pay for the edge, it can do no better than the minimal coloring that gives color A to node α_j thus incurring a cost of $Optc(\alpha_j, A)$. The next lemma follows by taking the cost of coloring the jth new tree to be the best of these cases. It provides a way of finding the *cost* of a minimal coloring.

Lemma 3.3 *The minimal cost $Optc(i, A)$ of coloring the subtree rooted at i such that i gets color A is given by*

$Optc(i, A) =$

∞	*i precolored with color other than* A
0	*i a leaf, uncolored or precolored* A
$\sum_{j=1}^{k} \min[Optc(\alpha_j, A), c_j + Opt(\alpha_j)]$	*otherwise*

Example 3.4 We show how $Optc$ and Opt may be computed for the tree of Figure 6. It is useful to think of $Optc$ and Opt as tables as shown in Figure 8. Lemma 3.3 may be applied to fill up columns of these tables in a left to right manner. The first column is for the $\texttt{Emp}_1$ node that is precolored by color A. By the first two cases of the formula of Lemma 3.3, the row for color A in this column is 0 and the other two entries are ∞. The entry in the Opt table is the minimum of the column values.

NODES (POSTFIX ORDER) →

COLORS ↓	Emp$_1$	Cust	Intersect$_1$	Supp	Emp$_2$	Intersect$_2$	Union	
A	0	∞	4	∞	0	4	8	OPTC
B	∞	0	3	∞	∞	7	9	
C	∞	∞	7	0	∞	3	9	

0	0	3	0	0	3	8	OPT

Figure 8: Opt and Optc tables for tree of Figure 7

Consider the last column of the table that represents entries for the `Union` node. This column is computed using the values in the columns for the children of the `Union` node, i.e., columns for $\texttt{Intersect}_1$ and $\texttt{Intersect}_2$. For example, by Lemma 3.3, $Optc(\texttt{Union}, A)$ is the sum: $\min[Optc(\texttt{Intersect}_1, A), 3 + Opt(\texttt{Intersect}_1)] + \min[Optc(\texttt{Intersect}_2, A), 3 + Opt(\texttt{Intersect}_2)]$. □

We now consider how to extract the minimal coloring itself. If the query tree has root i, then $Opt(i)$ is the cost of the any optimal coloring. If A is a color such that $Optc(i, A) = Opt(i)$, then there must be an optimal coloring the gives color A to i. Once we know an optimal color for i, we can pick optimal colors for the children of i by applying Lemma 3.3 in "reverse" as follows:

Lemma 3.4 *If i gets color A in some minimal coloring, there exists a minimal coloring such that child α_j of i has color A if $Optc(\alpha_j, A) \leq c_j + Opt(\alpha_j)$ and any color a for which $Optc(\alpha_j, a) = Opt(\alpha_j)$ otherwise.*

Example 3.5 We now show that the optimal coloring shown in Figure 6 may be obtained from the tables of Example 3.4. Since $Opt(\texttt{Union}) = Optc(\texttt{Union}, A) = 8$, the optimal coloring has a cost of 8 and assigns color A to `Union`. Lemma 3.4 may be applied in a top-down fashion to obtain the colors of the remaining nodes. Now, $Optc(\texttt{Intersect}_2, A) = 4$ which is less $c_{\texttt{Union},\texttt{Intersect}_2} + Opt(\texttt{Intersect}_2) = 3 + 3$. Thus $\texttt{Intersect}_2$ must be of color A. The colors of other nodes may be similarly extracted using Lemma 3.4. □

Lemmas 3.3 and 3.4 lead to the following *ColorSplit* algorithm. Letting C be the set of colors used for precolored nodes, the algorithm has a running time of $O(n|C|)$.

Algorithm 3.3 Algorithm ColorSplit

```
for each node i in postfix order do
   for each color a ∈ C do
         compute Optc(i, a) using Lemma 3.3;
         Opt(i) = min_a Optc(i, a)
   end for
end for;
Let a ∈ C be such that Optc(r, a) = Opt(r);
color(r) = a;
for each non-root node α_j in prefix order do
   Let i be the parent of α_j;
   Let c_j the weight of edge between i and α_j;
   if Optc(α_j, color(i)) ≤ c_j + Opt(α_j)
         then color(α_j) = color(i)
         else color(α_j) = a ∈ C such that
         Optc(α_j, a) = Opt(α_j)
end for.
```

We further observe that *ColorSplit* does not require the input tree be such that all and only the leaf nodes are precolored. It finds the optimal coloring for any tree. In other words, the tree need not be pre-processed by the *Simplify* algorithm of Section 3.1. Having pre-colored interior nodes actually reduces the running time of *ColorSplit* since the first two cases of Lemma 3.3, which are simpler than the third case, may be used.

ColorSplit is a fast algorithm. While pre-processing with *Simplify* offers the possibility of reducing the running time of *ColorSplit* (by reducing the number of colors in each new tree), additional gains may not be worth the implementation effort.

3.4 Extensions

We now focus on extending our results to cover several practical issues. We show that the mechanism of using a *set* of colors rather than a single color to pre-color a node makes several extensions possible. Handling sets of colors does not increase the complexity of *ColorSplit*. The intuitive reason is that any pre-coloring constrains the search space and thus can only reduce the running time of the algorithm.

We first describe modifications to the *ColorSplit* algorithm to allow a pre-coloring to specify a set of colors for each node. We then describe a variety of extensions to exploit optimization opportunities such as choices in access methods (due to indexes, replication of tables), choices in join and grouping attributes and the use of distinct partitioning functions on the same attribute are covered by this mechanism.

3.4.1 Set of Colors: A Swiss Army Knife

Pre-coloring with a set of nodes serves to restrict the choices of colors that the *ColorSplit* algorithm may make for a node. This restriction is implemented by the formula given in Lemma 3.3 which may be modified as shown below.

Lemma 3.5 (Modified Lemma 3.3) *The minimal cost $Optc(i, A)$ of coloring the subtree rooted at i such that i gets color A is given by*

$Optc(i, A) =$

∞ *if A is not in set of pre-colors for i*

0 *if i a leaf, uncolored or has A as a pre-color*

$\sum_{j=1}^{k} \min[Optc(\alpha_j, A), c_j + Opt(\alpha_j)]$ *otherwise*

This is the only modification needed for *ColorSplit* to work with a set of pre-colors. The modified algorithm is guaranteed to find the optimal in $O(n|C|)$ running time. Notice that using a set of pre-colors does not change the worst case running time of the algorithm since any pre-coloring (set or single color) reduces the running time of the algorithm by simplifying the computation of $Optc$.

3.4.2 Access Methods

Typically, the columns needed from a table may be accessed in several alternate ways. For example if a table is replicated then any copy may be accessed. Further, an index provides a copy of the indexing columns as well as permits access to the remaining columns.

Each access method may potentially provide a different partitioning. We may model this situation by associating a set of colors with each base table node, one color per partitioning.

We observe that each access method may have a different cost in addition to delivering a different partitioning. Such interactions between the cost of computation and communication are handled in Section 4.

3.4.3 Compound Attributes

Thus far we have considered attribute sensitive operators such as joins and groupings to have a single color. When such operators are based on compound attributes, additional opportunities for optimization arise that may be expressed by sets of pre-colors.

Example 3.6 Given the tables `Emp(emp#, dep#, city)` and `Dep(dep#, city)`, the following query finds employees who live in the same city as the the location of their department.

Select e **From** Emp e, Dep d
Where e.dep# = d.dep# **and** e.city = d.city

Since a join operator has to be partitioned on the join column, the required partitioning depends on the predicate chosen to be the join predicate. In Figure 9, the first query tree uses the join predicate on `dep#` and requires the `Emp` table to be repartitioned. The second uses the join predicate on `city` and requires `Dep` to be repartitioned.

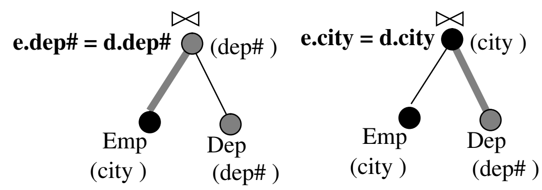

Figure 9: Interaction of Repartitioning with Join Predicates

The optimization opportunities provided by join predicates may be modeled by pre-coloring the join node by a set of two colors $\{$`dep#`, `city`$\}$. We observe that choice of the join predicate may impact the cost of the join-method. Such interactions between the cost of computation and communication are postponed to Section 4. □

Similar observations apply to other attribute sensitive operators. Given a grouping of employees by `department` and `city`, we pre-color the `GROUPBY` operator by $\{$**`dep#`**, **`city`**$\}$.

A partitioning guarantees that tuples that agree on the partitioning attribute(s) are assigned to the same site. Given some set of attributes X, a partitioning on any non-empty subset of X is also a partitioning on X. The most general way of modeling this situation is by pre-coloring an attribute sensitive operator that has compound attribute X by a set colors, one color for each non-empty subset of X.

3.4.4 Partitioning Functions

Suppose two base tables are partitioned on the same attribute A using different partitioning functions (We consider two attributes to be the "same" attribute w.r.t. a query if they are equated by an equality predicate.) For example, one table may be hash partitioned on A and the other range partitioned. We will fix this situation by giving distinct colors (say B_1 and B_2) to the two tables. Any attribute sensitive operator that needs a partitioning on A could use either of the two partitions and will therefore be given the set of colors $\{B_1, B_2\}$.

4 Combining Computation and Communication Costs

We have so far been concerned with the communication costs incurred by repartitioning and have considered the cost of *computing* (i.e. disk and cpu costs other than communication) an operator to be independent of the partitioning attribute. We now extend our model to account for the interaction of these costs and show how the basic ideas of the *ColorSplit* algorithm carry over to the extended model to yield an efficient optimization algorithm.

Several alternate *strategies*, each with a different cost, may be available for an operator. The following example shows the interaction between between computation and communication costs using the standard scenario of having several strategies for computing operators such as joins and grouping.

Example 4.1 Given the schema `Emp(emp#, salary, dep#, city)` and `Dep(dep#, city)`, the following query finds the average salaries of employees grouped by city for those employees who live in the same city as the the location of their department.

Select e.city, **avg**(e.salary)
From Emp e, Dep d
Where e.dep# = d.dep# **and** e.city = d.city
Group by e.city;

Suppose `Emp` is partitioned by `city` and each partition is stored in sorted order by `city`. Suppose `Dep` is partitioned by `dep#` and each partition has an index on `dep#`. Figure 10 shows two query trees. The computation of `Avg` is assumed to be combined with `GroupBy`. The first query tree uses the join predicate on `dep#` and repartitions the `Emp` table. Due to the availability of an index on `Dep`, a nested-loops strategy may be the cheapest for joining each partition of `Emp` (outer) with its corresponding partition of `Dep` (inner). The grouping operator is implemented by a hash-grouping strategy.

The second query tree uses the join predicate on `city` and repartitions the `Dep` table. Since each partition of `Emp` is pre-sorted, it may be cheapest to use a sort-merge join for joining corresponding partitions. Since the output of merge join is pre-sorted in addition to being pre-partitioned on the `city`, the grouping operator uses a sort-grouping strategy. □

The example illustrates several points. Firstly, while partitioning impacts communication costs, other physical properties (sort-order and indexes) impact computation costs. We will generalize the notion of a color to capture *all* physical properties.

Secondly, a strategy expects its inputs to have certain physical properties and guarantees its output to have some other properties. We will specify such input-output constraints using color patterns.

Thirdly, the overall cost is reduced when an input to a strategy happens to have the expected physical property. We will therefore break the cost of computing an operator into the intrinsic cost of the strategy itself and the cost of getting the inputs into the right form. The latter will be modeled as a recoloring cost that may or may not be incurred.

In Section 4.1, we develop the details of the model outlined above. In Section 4.2, we show how the basic algorithmic ideas developed in Section 3.3 carry over to the extended model. Finally, in Section 4.3, we show that coloring also interacts with the order of operators and indicate that it can be incorporated into the traditional System R algorithm.

4.1 Annotated Query Trees and their Cost

We now develop a model in which each interior node of a query tree is annotated by a strategy, an output color, and a color for each input. The leaf nodes have an output color but no strategy.

The cost of a query tree is the sum of the costs of all nodes. The cost of a node consists the cost of recoloring the outputs of its children to have the color of its inputs plus the cost of executing the strategy itself.

Any classical cost model typically consists of two parts: (a) estimation of statistics (such as size, number of unique values in columns) for intermediate results; and, (b) estimation of cost of an operator given statistics and physical properties of operands. The formulas in any such model can be easily extended to account for repartitioning as well. Our goal is not to provide new formulas, but to provide abstractions that make it possible to reason with them in a general context.

We have so far used a color to represent the attribute on which data is partitioned. We now generalize a color to be a *triple* $\langle p : a_1, s : a_2, i : a_3\rangle$ where a_1 is the *p*artitioning attribute, a_2 the *s*ort attribute and a_3 the *i*ndexing attribute.

A strategy specifies a particular algorithm for computing an operator. It requires the inputs to satisfy some constraints and guarantees some properties for its output. We will use *color patterns* to specify such input-output constraints. A constraint has the form $Input_1, \ldots, Input_k \rightarrow Output$, where $Input_j$ and $Output$ are color patterns. A color pattern is similar in syntax to a color but allows the use of variables and wild-cards. Table 1 shows examples of input-output constraints for several strategies.

If some input is not colored in the required manner, a recoloring is needed. Recoloring requires repartitioning, sorting, or building an index.

Example 4.2 The `Emp` table of Example 4.1 (Figure 10) has the output color $\langle p : \texttt{city}, s : \texttt{city}, i : \texttt{none}\rangle$ while `Dep` has $\langle p : \texttt{dep\#}, s : \texttt{none}, i : \texttt{dep\#}\rangle$.

In the first query tree of Figure 10, the join uses the nested-loops strategy and its output has the color $\langle p : \texttt{dep\#}, s : \texttt{city}, i : \texttt{none}\rangle$. From the first row of Table 1, this implies that the color of input1 (`Emp`) should be $\langle p : \texttt{dep\#}, s : \texttt{city}, i : *\rangle$ and that of input2 (`Dep`) should be $\langle p : \texttt{dep\#}, s : *, i : \texttt{dep\#}\rangle$. The color of `Dep` matches the requirements but that of `Emp` does not. □

Definition 4.1 $inpCol(s, A, j)$ is defined to be the color pattern needed by strategy s for input j for the output to be of color pattern A.

Definition 4.2 $recolor(R, c_{old}, c_{new})$ is defined to be the cost of changing the color of table R from c_{old} to c_{new}.

Example 4.3 The color required for the first input of the nested-loops join in the first query tree of Figure 10 is $c_{new} = \langle p : \texttt{dep\#}, s : \texttt{city}, i : *\rangle$. Since the output

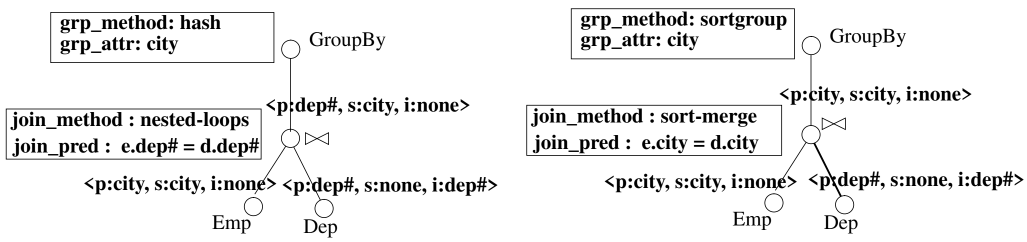

Figure 10: Annotated Query Trees

Strategy	*Output*	*Input1*	*Input2*	*Additional requirements*
Nested-Loops Join	$\langle p : X, s : Y, i : none \rangle$	$\langle p : X, s : Y, i : * \rangle$	$\langle p : X, s : *, i : X \rangle$	Join predicate on X
Sort-Merge Join	$\langle p : X, s : X, i : none \rangle$	$\langle p : X, s : X, i : * \rangle$	$\langle p : X, s : X, i : * \rangle$	Join predicate on X
Hybrid-Hash Join	$\langle p : X, s : Y, i : none \rangle$	$\langle p : X, s : Y, i : * \rangle$	$\langle p : X, s : *, i : * \rangle$	Join predicate on X
Hash Grouping	$\langle p : X, s : none, i : none \rangle$	$\langle p : X, s : *, i : * \rangle$		X is a grouping attribute
Sort Grouping	$\langle p : X, s : X, i : none \rangle$	$\langle p : X, s : X, i : * \rangle$		X is a grouping attribute
Hash Intersect	$\langle p : X, s : none, i : none \rangle$	$\langle p : X, s : *, i : * \rangle$	$\langle p : X, s : *, i : * \rangle$	

Table 1: Examples of Input-Output Constraints

color (call it c_{old}) of *Emp* differs in partitioning attribute, $recolor(R, c_{old}, c_{new})$ is the cost of repartitioning `Emp` on the `city` attribute. □

The cost of an annotated query tree is the sum of the costs of all operators. The cost of an operator consists of recoloring the inputs to have colors needed by the chosen strategy plus the cost of the strategy itself. This is more formally expressed as follows. Suppose we have a tree T in which the root uses strategy s and has output color a, and furthermore that T has k subtrees $T_1, \ldots, T_k$ and that T_j produces table R_j with color c_j.

$$\begin{aligned} Cost(T) &= StrategyCost(s, R_1, \ldots, R_k) \\ &+ \sum_{j=1}^{k} recolor(R_j, c_j, inpCol(s, a, j)) \\ &+ \sum_{j=1}^{k} Cost(T_j) \end{aligned}$$

If T is a leaf, we take its cost as zero since we count the cost of accessing operands as part of the cost of a strategy.

The cost model above has several important properties. Firstly, *no restriction* is placed on the *form* of the *StrategyCost()* or *recolor()* functions. For example, these may have non-linear terms such as logarithms, product and division. Such terms do occur in cost models such as System R [SAC+79]. Secondly, *StrategyCost()* and *recolor()* represent respectively the cost of the strategy and the cost to get the inputs into the physical form assumed by the strategy. This separation of cost into two components is the key to developing the optimization algorithm.

4.2 Optimization Algorithm: Extension of ColorSplit

We will now develop an optimization algorithm that given a tree with colors for the leaf nodes finds a strategy (and input and output colors) for each interior node of the query tree so as to minimize total cost. The algorithm (and its proof) is more complex than the *ColorSplit* algorithm of Section 3.3 but the basic ideas are similar.

Definition 4.3 $Optc(i, A)$ is defined to be the minimal cost of the subtree rooted at node i such that i has output color A. $OptcStrategy(i, A)$ is defined to be the strategy that achieves this minimal value (pick any one strategy if several are minimal).

For a leaf node i, $Optc(i, A) = 0$ if i is pre-colored with a color compatible with A and ∞ otherwise. We will treat $OptcStrategy(i, A)$ as undefined for leaf nodes.

Definition 4.4 $Opt(i)$ for node i is defined to be the minimal cost of the subtree rooted at i. $OptStrategy(i)$ is defined to be the strategy and $OptColor(i)$ the output color for which the minima is achieved.

Definition 4.5 $Strategies(i, A)$ is the set of strategies applicable to the operator represented by node i and whose input-output constraint permits A as an output color.

The following is a generalization of Lemma 3.3. Let node i have children $\alpha_1, \ldots, \alpha_k$. Suppose the subtree rooted at α_j computes table R_j as its output. The minimum cost of the tree rooted at i such that i has output color A is obtained by trying out all strategies capable of producing output color A. The lemma shows that for any such strategy s, the lowest cost is achieved by *individually* minimizing the cost of each input. The minimum cost for the j'th input is the best of two possibilities: (1) the minimum subtree whose output color matches the input color needed by s; and, (2) the minimum subtree plus the cost of recoloring its output.

Lemma 4.1 *For a leaf node i, $Optc(i, A)$ is 0 if i has a color compatible with A and ∞ otherwise. For non-leaf node i, $Optc(i, A)$ obeys the following recurrence.*

$Optc(i, A) = \min_{s \in Q}[StrategyCost(s, R_1, \ldots, R_k)$
$+ \sum_{j=1}^{k} \min[y'_j, y_j]]$
where $Q = Strategies(i, A)$
$y'_j = Optc(\alpha_j, inpCol(s, A, j))$
$y_j = Opt(\alpha_j) + recolor(R_j, OptColor(\alpha_j), inpCol(s, A, j))$

The lemma shows that it is possible to compute $Optc$ and Opt (as well as $OptcStrategy$, $OptStrategy$, $OptColor$) in a bottom-up manner. The following generalization of Lemma 3.4 allows to extract colors and strategies by a top-down pass.

Lemma 4.2 *If i gets color A and strategy s in some minimal solution, then there exists a minimal solution such that the j'th child α_j of i has color A_j and strategy s_j as follows. If $y'_j < y_j$ then $A_j = inpCol(s, A, j)$ and $s_j = OptcStrategy(\alpha_j, A_j)$. Otherwise $A_j = OptColor(\alpha_j)$ and $s_j = OptStrategy(\alpha_j)$.*

The following algorithm applies Lemma 4.1 in a bottom-up pass followed by a top-down pass for Lemma 3.4. C is the set of allowable colors and r is the root of the tree.

Algorithm 4.1 Algorithm ExtendedColorSplit

1. **for each** node i in postfix order
2. Use Lemma 4.1 to compute $Optc(i, a)$
3. and $OptcStrategy(i, a)$ for each color $a \in C$
4. Let $a = A$ be a color for which $Optc(i, a)$ is minimal
5. $Opt(i, a)$ is set to minimal value
6. $OptColor(i) = A$
7. $OptStrategy(i) = OptcStrategy(i, A)$
8. **end for**
9. $strategy(r) = OptStrategy(r)$
10. $color(r) = OptColor(r)$
11. **for each** non-root node α_j in prefix order
12. Use Lemma 4.2 to compute $color(\alpha_j)$ and $strategy(\alpha_j)$
13. **end for**

The algorithm has a worst-case running time of $nS|C|$ where S is the number of strategies, $|C|$ the number of allowable colors and n the number of nodes in the tree.

Since n and $|S|$ are typically small, the running time of the algorithm is dependent on $|C|$. However, $|C|$ can become large when we permit the extensions discussed in Section 3.4. The magnitude of $|C|$ may be kept small by observing (1) no strategy yields an output relation with an index. Thus only 2 components of the triple for colors are relevant for interior nodes (2) only colors that might be useful to subsequent operator need to be considered.

4.3 Interaction with Join Ordering

We now show an example of how repartitioning costs interact with the order of joins. The reader is referred to [Has95] for an extension of the System R style dynamic programming algorithm that integrates repartitioning costs into the join-ordering problem.

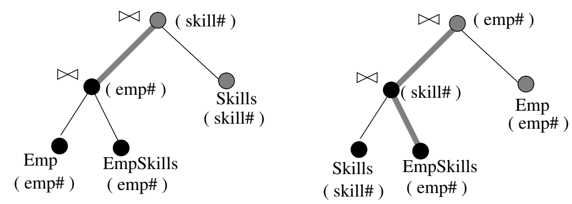

Figure 11: Interaction of Repartitioning with Order of Joins

Example 4.4 Suppose the tables `Emp(emp#, city)`, `EmpSkills(emp#, skill#)`, and `Skills(skill#, skilltype)` are partitioned by the underlined attributes. The following query finds employees who live in Palo Alto and have analytical skills.

Select e **From** Emp e, EmpSkills es, Skills s
Where e.emp# = es.emp# **and** es.skill# = s.skill# **and** s.skilltype = analytical **and** e.city = palo alto

Figure 11 shows two alternate query trees. The trees use different join orders and incur different repartitioning costs. If "`s.skilltype = analytical`" is a highly selective predicate, the second tree may achieve a low cost due to the small size of the intermediate table (`Skills` $\bowtie$ `EmpSkills`). However, the first tree avoids the cost of repartitioning the possibly very large `EmpSkills` table. Thus repartitioning cost needs to be accounted for in join ordering. □

5 Conclusions and Future Work

We have developed models and algorithms for the JOQR phase of a parallel query optimizer. Our work uses a model based on using color as an abstraction for the physical properties of data such as how it is partitioned, sorted, or accessible by indexes. We have proposed and solved two optimization problems: *query tree coloring* that models communication costs and *query tree annotation and coloring* that combines communication and computation. Our algorithms are efficient, guarantee optimality, and apply to queries that include operators such as grouping, aggregation, intersection and set difference in addition to joins.

An interesting direction is to devise optimization algorithms that permit "fragment and replicate" strategies [CP84] in additioned to partitioned strategies. Fragment and replicate is advantageous when, for example, a small table is joined with a large table. It may be cheaper to *replicate* the small table rather than repartition the large table.

As shown in Section 4.3, there is interaction between the cost of repartitioning and join ordering. The standard solution for the join ordering problem is the System R dynamic programming algorithm which has high computational complexity. Integrating coloring further increases the cost of this algorithm [Has95]. It would be interesting to evaluate whether the expensive integrated approach results in significantly better plans as compared to using coloring as a post-pass.

Emerging read-intensive decision-support applications may benefit data placement strategies that use both vertical and horizontal partitioning as well as replication. Such data placement may offer advantages such as reduced communication and IO costs. Development of query processing and optimization techniques for such generalized data placement may be useful.

Acknowledgements

Thanks are due to Surajit Chaudhuri, Umesh Dayal, Hector Garcia-Molina, Susanne Englert, Ray Glasstone, Jim Gray, Ravi Krishnamurthy, Sheralyn Listgarten, Arun Swami, Jeff Ullman and Gio Wiederhold for useful discussions.

References

[BCC+90] H. Boral, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, A Highly Parallel Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.

[CHM95] C. Chekuri, W. Hasan, and R. Motwani. Scheduling Problems in Parallel Query Optimization. In *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1995.

[CLYY92] 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 the Eighteenth International Conference on Very Large Data Bases*, pages 15–26, June 1992.

[CP84] S. Ceri. and G. Pelagatti. *Distributed Database Design: Principles and Systems*. McGraw-Hill, 1984.

[CR91] S. Chopra and M.R. Rao. On the Multiway Cut Polyhedron. *Networks*, 21:51–89, 1991.

[DG92] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Communications of the ACM*, 35(6):85–98, June 1992.

[DGG+86] D. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA: A High Performance Dataflow Database Machine. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, August 1986.

[DJP+92] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. Complexity of Multiway Cuts. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 241–251, 1992.

[EGH95] S. Englert, R. Glasstone, and W. Hasan. Parallelism and its Price: A Case Study of NonStop SQL/MP, 1995. To be submitted for publication.

[ES94] P.L. Erdos and L.A. Szekely. On Weighted Multiway Cuts in Trees. *Mathematical Programming*, 65:93–105, 1994.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 9–18, June 1992.

[Gra88] J. Gray. The Cost of Messages. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 1–7, Toronto, Ontario, Canada, August 1988.

[Has95] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford University, 1995. In preparation.

[HLY93] K.A. Hua, Y. Lo, and H.C. Young. Including the Load Balancing Issue in The Optimization of Multi-way Join Queries for Shared-Nothing Database Computer. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

[HM94] W. Hasan and R. Motwani. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism. In *Proceedings of the Twentieth International Conference on Very Large Data Bases*, pages 36–47, Santiago, Chile, September 1994.

[Hon92] W. Hong. *Parallel Query Processing Using Shared Memory Multiprocessors and Disk Arrays*. PhD thesis, University of California, Berkeley, August 1992.

[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.

[LST91] H. Lu, M-C. Shan, and K-L. Tan. Optimization of Multi-Way Join Queries for Parallel Execution. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, Barcelona, Spain, September 1991.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, 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. In *Second International Symposium on Databases in Parallel and Distributed Systems*, Dublin, Ireland, 1990.

[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.

[SE93] J. Srivastava and G. Elsesser. Optimizing Multi-Join Queries in Parallel Relational Databases. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.

[SW91] D. Shasha and T.L. Wang. Optimizing Equijoin Queries in Distributed Databases where Relations are Hash Partitioned. *Transactions on Database Systems*, 16(2):279–308, June 1991.

[SYT93] E. J. Shekita, H.C. Young, and K-L Tan. Multi-Join Optimization for Symmetric Multiprocessors. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, 1993.

[Val93] P. Valduriez. Parallel Database Systems: Open Problems and New Issues. *Distributed and Parallel Databases: An International Journal*, 1(2):137–165, April 1993.

[X3H92] X3H2. Information technology - database language sql, July 1992. Also available as International Standards Organization document ISO/IEC:9075:1992.

[YC84] C.T. Yu and C.C. Chang. Distributed Query Processing. *ACM Computing Surveys*, 16(4), December 1984.

[ZZBS93] M. Ziane, M. Zait, and P. Borla-Salamet. Parallel Query Processing in DBS3. In *Second International Conference on Parallel and Distributed Information Systems*, San Diego, California, January 1993.