Tree Query Optimization in Distributed Object-Oriented Databases

Hyeokman Kim, Sukho Lee
Dept. of Computer Eng.
Seoul Nat'l Univ., Seoul, 151-742, Korea
E-mail: {hmkim, shlee}@snucom.snu.ac.kr

Abstract

Much of the work in query optimization in object-oriented database is devoted to finding efficient ways of path traversals expressed in a query. In the environment where a query involves classes distributed across several sites, communication cost must be taken into consideration. In this paper we describe a tree query optimization algorithm using dynamic programming technique for a distributed object-oriented database. The primary objective of the optimization is to minimize the total processing time including local processing and communication costs. In contrast to optimizers which consider a restricted search space, the proposed optimizer searches the execution plans which may be deep or bushy trees. Especially the plans can start traversal anywhere in a query graph and not only from either end of the query graph.

1. Introduction

In a distributed object-oriented database system, processing a query involves transmission of classes and intermediate results among the different sites of the computer network. To process the distributed query efficiently, query optimization which minimizes total processing time for a system or response time for a user is required. In particular, many works have been done on the optimization for join operations in distributed relational database[1,3,9,14,15,16].

Most of the research efforts have been done on generating an optimal or efficient query execution plan that could answer the query with minimum data transmission, based on the assumption that the communication cost largely dominates local processing cost. This assumption is based on very slow communication networks such as wide area networks with a bandwidth of a few kilobytes per second. As recent advances on network communications have drastically increased the bandwidth, distributed database environments now exist where the communication network is much faster, thus the cost of local processing is no longer negligible.

In object-oriented databases, the attribute of an object can have the *object identifier(oid)* of another object as a value. Through the oids, objects can refer to other objects, that is, each class can refer to another class by defining the domain of its attribute as the referred class, and the attribute is called a *complex attribute*. In an object-oriented query, these reference relationships are expressed by path expressions. Thus, path expressions imply traversals from one object, through others, to a result.

Much of the work in object-oriented query optimization is devoted to finding efficient ways of path traversals expressed by path expressions. These works may be classified into indexing techniques to speed up the access through a path, clustering techniques used to reduce storage accesses to retrieve portions of paths, and manipulation of the path expression itself to find alternatives to the traversal[13]. The last approach is to transform the path traversal into join operations. This approach is based on the observation that optimization techniques and results in relational databases form the basis for research in object-oriented query optimization[10,12]. However, there are only a few research works to process an object-oriented query in distributed database environment[2,8].

In COMANDOS system[2], the optimizer employs top-down and bottom-up execution of implicit joins as distributed execution for a path expression. The top-down strategy starts the execution of the implicit join from the first and second classes in the path and then proceeds consecutive joins until the last class is reached, while the bottom-up strategy proceeds the consecutive join executions from the last(n-th) and (n-1)-th classes until the first class is reached. Thus, the optimizer considers only the plans which start traversal from either end of a path. In Distributed Orion system[8], the query graph transformation technique based on the cluster concept is proposed to determine the traversal orderings of classes for a tree query. Because the clusters are formed successively from the leaf nodes of the transformed query graph, the execution of the obtained traversal orderings is a bottom-up strategy as a whole.

To obtain an optimal query execution plan, the optimizer must consider all feasible execution plans. However, the optimizers of COMANDOS and Distributed Orion system consider only the restricted portions of all feasible plans, thus producing only suboptimal plans.

In this paper, we propose a query optimization algorithm for a tree query in distributed object-oriented databases. The primary objective of the optimization is to minimize the total processing time including local processing and communication costs. The results of the optimization include the order for implicit joins and

0-8186-6430-4/94 $4.00 © 1994 IEEE

transmissions to be performed with minimum cost, the join method to apply and the execution sites. In contrast to the optimizers explained above, our optimizer considers plans which start traversal not only from either end but also from anywhere of a query graph.

The paper is organized as follows. In Section 2, we describe the implicit join and path expression. The model for the distributed query optimization is described in Section 3. In Section 4, we propose an algorithm which produces an optimal plan with its cost and then computes the complexity of the algorithm. Finally, in Section 5, we draw some conclusions and describe future research directions.

2. Path Expressions and Implicit Joins

In an object-oriented query, the reference relationships between classes involved in a query can be represented in the form of a directed graph called a *query graph*. Each node in the query graph corresponds to a class involved in the query and an edge from node A to node B means that the domain of the attribute of class A is class B. Each node has a label specifying the site where the objects of the class are stored and each edge also has a label representing the complex attribute of the reference relationship. The class to which a query is directed is called the *target class*. A query is said to be cyclic if the query has at least one cyclic path in the graph. Otherwise, it is acyclic. A *tree query* is an acyclic query in which each node has at most one parent node[8]. In this paper, we focus on tree queries whose target class is the root node of the graph.

Example: The following is an example query which is used throughout the paper.

SELECT Person
WHERE Person.own.manufacturer.location.name = "Detroit"
AND Person.own.drivetrain.engine.cylinder ≥ 6
AND Person.own.color = "blue"

Suppose that the classes Person and DriveTrain are stored at site 1, the classes Company and Engine at site 2, and the classes Vehicle and City at site 3. The corresponding query graph is represented in Figure 1. The target class of the query is the class Person.□

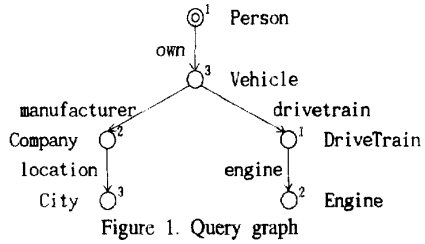


Figure 1. Query graph

Fetching the objects of the target class requires that all objects it refers through its complex attributes must be recursively fetched. The process of recursive fetching of objects is called *path traversal*. The path traversal can be transformed into join operations. This join is called as an *implicit join*[11]. In the query of Figure 1, the reference relationship between the classes Person and Vehicle may be represented as a join predicate, that is, Person.own=oid(Vehicle). The values of the complex attribute own is the oids for the class Vehicle, and this predicate means the join operation between the classes Person and Vehicle uses the oids for the class Vehicle as the joining attribute values.

Let *join identifiers* be the oids used as the values of the joining attribute of an implicit join. As each class has at most one parent class in a tree query, the oids for a class which is not a root class are used only to perform the implicit join between the class and its parent class. These oids are the join identifers of the implicit join. Clearly, the join identifiers can be eliminated after the corresponding implicit join so as to reduce the data transmission required for subsequent implicit joins to be performed at any other site. Thus, the intermediate result of an implicit join includes only the oids for subsequent implicit joins and the oids for the target class if the target class was implicitly joined into the result.

To perform an implicit join, the classes or the intermediate results involved must be at the same site. If they are not, one of them may be transferred to the site of the other or both of them may be transferred to a third site. In the last case, the subsequent implicit join between the intermediate result of the implicit join performed at this third site and the class stored here may then be performed without transmissions. When a class is transferred, only the oids for the class and the values of its complex attributes if they exist are transferred because they will be used as join identifiers of the subsequent implicit joins.

As explained before, path expressions imply the consecutive implicit joins and transmissions if they are required to perform the implicit joins. Thus, to process a query efficiently, we must determine the best traversal ordering specifying the execution sequence of implicit joins and transmissions to be performed with *minimum cost*.

3. Distributed Query Optimization Model

Distributed query optimizer can be defined as an algorithm to choose an optimal global processing strategy for a given query. The design of such optimizers may be divided into three components: execution space which is the set of the execution plans to be searched by the optimizer, cost model which predicts the cost of an execution plan, and search strategy to obtain the minimum cost plan. We describe our model to the distributed query optimization problem along these components.

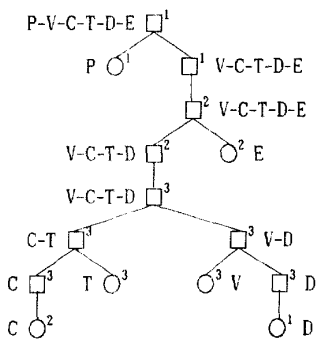
3.1 Execution Space

Query executions are represented as execution plans which transform a nonprocedural query into a sequence of operations. An execution plan can be syntactically represented as a join processing tree[7,12] or a dataflow graph[1,8]. We extend the representations into a *global execution tree*(GET) to express the execution plans processed in distributed database environment. Our optimizer considers all feasible GETs for a given query and choose the lowest cost plan as the optimal plan.

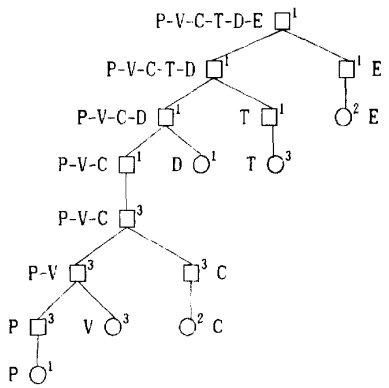
GET is a labelled tree where the outdegree of each node must not exceed two. The leaf nodes are original classes and each non-leaf node is an intermediate result from the implicit join or

transmission: a node with outdegree 2(*join node*) is an intermediate result from the implicit join of its children and a node with outdegree 1(*transmission node*) is an intermediate result from the transmission of its child. The intermediate result associated with each non-leaf node is stored in a temporary file. Each node has a label of the form *label=value* and the label specifies the important factors of an execution. The label for a join node specifies the method of implicit join and its execution site. An example label for a join node is *site=2, method=nested-loop.* Classes or intermediate results must be at the same site to be joined implicitly. Thus, the site label for a join node and those for its children must be the same. The site label for a transmission node represents the site to which its child is transferred. If the site labels for a transmission node and its child are the same, no transmission occurs. The site label for a leaf node specifies the site where its corresponding class is stored. Leaf and non-leaf nodes are graphically represented by a circular and square node respectively.

Clearly, GET is a labelled tree and can be distinguished into *deep* or *bushy trees*[7]. If all join nodes of a GET have at least one leaf or transmission node as a child, the tree is called deep. Otherwise, it is called bushy. As defined, the execution space of bushy GETs includes deep GETs. Figure 2 gives examples of deep and bushy GETs generated from the query of Figure 1. In the figure, the number annotated on each node is the site label, and P,V,C,T,D,E represent the classes involved in the query. For example, P represents the class Person.



(a) bushy GET



(b) deep GET

Figure 2. Global Execution Tree(GET)

Execution plans represented by GET specify the order for implicit joins and transmissions to be performed, the join method to apply, and its execution site. The order can be obtained by traversing GET in a postorder sequence. The GET of Figure 2.a shows the execution strategy where the execution sequence of implicit joins starts from the middle(class V and D) of the query graph, while the GET of Figure 2.b expresses the strategy where the implicit joins are performed from the root to leaves in top-down fashion.

The execution plan represented by a GET can be expressed as a sequence of operators. Let a, b be subplans of GET and $IJ_t(a,b)$ be an operation to perform the implicit join between the results of the subplans a and b at site t, $TR_{xt}(a)$ be an operation to transfer the result of the subplan a from site x to site t. The join node with site label t is transformed to IJ_t operator and its children to operands. Also, the transmission node whose site label is t and has a child with site label x is to TR_{xt} operator and its child to operand. For example, the operators for the GET in Figure 2.a can be represented as

$IJ_1(P,TR_{2,1}(IJ_2(TR_{3,2}(IJ_3(IJ_3(TR_{2,3}(C),T),IJ_3(V,TR_{1,3}(D)))),E)))$.

The set of all execution plans searched by an optimizer is the *execution space* for the optimization. If the space does not include all feasible plans for a given query, the optimizer will produce a suboptimal plan not optimal one. For example, R* optimizer always produces a suboptimal plan because it has a restricted execution space that includes only deep trees[14]. The plans produced in the COMANDOS system and the Distributed Orion system are also suboptimal because they only consider top-down and/or bottom-up strategies. Our optimizer searches execution plans which may be deep or bushy trees. Also, it considers plans which start traversal anywhere in a query graph as well as either end of a query graph, therefore, alway producing an optimal plan.

3.2 Cost Model

Based on the cost of the individual implicit join and transmission specified in a GET, we can estimate the cost of the corresponding execution of the GET. In this subsection, we compute the cost of the individual implicit join and transmission for a given GET.

Let C,D be original classes or intermediate results and x,t be sites. We use $trans_{xt}(C)$ to denote the communication cost for transferring C from site x to site t. We assume that the communication cost is $trans_{xt}(C)=c_1+c_2*|C|$, where c_1 is the start-up cost of initiating transmission, c_2 is a proportionality constant and |C| is the size of C expressed in bytes[16]. If x=t, $trans_{tt}(C)=0$. Also, we use $join_t(C,D)$ to denote the local processing cost for performing implicit join between C and D at site t.

For a given GET, the right-hand child of a join node can be a previously materialized join node or a transmission node. When the result of an implicit join is materialized, the low cost access to files through oids may be lost. For example, the implicit join between Person and Vehicle is on the oid of Vehicle. If the oid is physically implemented, as in most object-oriented databases, only one access is needed to get the Vehicle of a Person. In the GET of Figure 2.a, this is no longer true because the oid of Vehicle becomes a stored value in the temporary file that contains the materialized result of implicit joins performed previously. Similarly, if the oid is a physical pointer, it is only valid within

its resident site where it was created. When the oid of an object is transferred to other site, the physical meaning of the oid may be lost. For example, the oid of DriveTrain at site 1 is transferred to site 3 to perform the implicit join between Vehicle and DriveTrain at site 3. At this site, the oid of DriveTrain has no longer a physical meaning and is just another stored value in a temporary file too. In these cases, the costs of implicit joins are similar to those of joins in relational model.

There are many algorithms for implicit joins. In this paper, we consider the well known nested loop and hash join algorithm. Let $nljoin_i(C,D)$, $hsjoin_i(C,D)$ be the cost for performing implicit join by the nested loop and hash join algorithms respectively. The implicit join will then be performed by the algorithm with minimum cost, that is, $join_i(C,D)=min\{nljoin_i(C,D), hsjoin_i(C,D)\}$. The cost functions for $nljoin_i(C,D)$ and $hsjoin_i(C,D)$ will not be expressed here in detail because it has been studied in many local query optimizations. Once the algorithm is determined, the name of this algorithm is written into the method label of the corresponding join node in GET.

It is assumed that there is no fast access path such as index for all non-leaf nodes and some leaf nodes in GET. This is justified by the observations that in distributed query processing,

- The join node is an intermediate result generated by an implicit join. The intermediate result is not supported by fast access paths unless some are created dynamically[9].
- A fast access path is not valid outside the site where it was established. When a class or intermediate result is sent from one site to another, its fast access path existing in the sending site, will not be sent. Thus, the transmission node is not supported by fast access paths.
- The operations like selection and projection are performed before implicit joins. Thus, the leaf nodes which are qualified by these operations are also intermediate results[9].

Besides these, we assume that the usual statistical information for cost estimation is available, even though we have not explicitly expressed it in formalism. In Section 4, we will give a cost model to estimate the cost of a given GET by integrating the costs of all its implicit joins and transmissions.

3.3 Search Strategy

A search strategy is a strategy used to search the execution space for the plan with minimum cost. *Dynamic programming technique* has been used as a search strategy by many researchers in query optimization[3,5,8,9,12,14,15]. Dynamic programming is a bottom-up technique. A dynamic programming algorithm divides the problem into dependent subproblems, solves the subproblems recursively, and then combines their solutions to solve the original problem. When each subproblem is solved, the answer is saved, thereby avoiding the work of recomputing the answer every time the subproblem is encountered[4].

If dynamic programming is used to solve optimization problems, the problems must satisfy the *principle of optimality*. In the following section, we will show that the distributed tree query optimization problem obeys the principle of optimality and propose a dynamic programming algorithm to solve the problem.

4. Distributed Tree Query Optimization Algorithm

4.1 Basic Ideas

The principle of optimality states that an optimal execution plan(optimal GET) for a query is composed of the optimal plans for its subqueries as subplans. Therefore, in order to produce an optimal plan for a query by dynamic programming, the optimal plans for its subqueries must be produced in advance.

Let an *i-class query* be a subquery of a given query with n classes such that the query graph of an i-class query is a connected subgraph with i nodes constructed from that of a given query(to be exact, this subgraph is an induced subgraph. We will explain it in the following subsection). By the definition, the query with n classes becomes an n-class query. To produce an optimal plan for an i-class query by dynamic programming, the optimal plans for its subqueries from 1-class to (i-1)-class queries must be produced in advance. If this procedure is repeatedly applied to all i-class queries for i=1,2,...,n, an optimal plan for an n-class query will be obtained.

An i-class query can be divided into a pair of subqueries. Let the query graph of an i-class query be split into two connected subgraphs with r and i-r nodes respectively. By the definition of an i-class query, these subgraphs become the query graphs of r-class and (i-r)-class subqueries. In a tree, the removal of an edge yields two subgraphs which are also trees. The removed edge implies an implicit join because an edge of a query graph means object references. The result of an i-class query is generated by performing implicit join between the results of its r-class and (i-r)-class subqueries. Before performing the implicit join, data transmission must be performed if the results of subqueries are not at the same site. Thus, the cost of an i-class query can be calculated by summing the costs of its r-class and (i-r)-class subqueries, the cost for performing the implicit join between the results of these subqueries, and the cost for transmissions if they occur.

There are i-1 ways of dividing an i-class query into a pair of subqueries because i-1 edges exist in the corresponding query graph. One must compute the costs of an i-class query for each pair of subqueries and choose the lowest cost as the cost of the optimal plan for an i-class query. If these computations are applied to each i-class query at all related sites with varying i from 2 to n by 1, the minimum cost plan of a given n-class query at a query site can be finally obtained. Whenever the optimal plans and its costs of each i-class query at all related sites are obtained, they are saved for reuse when this query is employed as a subquery of another larger query at a certain site.

If r is always restricted to be 1, every time an i-class query is divided into r-class and (i-r)-class subqueries, the plan produced by these pairs of subqueries becomes a deep GET. Our optimizer must produce all feasible plans which may be deep or bushy GETs because we do not restrict the execution space. We consider all i-1 pairs of subqueries for each i-class query.

In this section, we describe an algorithm to generate all possible subqueries of a given query and a cost model to compute the cost of execution plan of a query. We then show that this cost model obeys the principle of optimality. Based on these, we propose a dynamic programming algorithm to produce the optimal

execution plan of a query. Finally, the complexity of the algorithm is characterized.

4.2 Generation of Subqueries

As explained before, in order to produce an optimal plan for a query by dynamic programming, the optimal plans for its subqueries must be found in advance. In this section, we describe the algorithm to generate all possible subqueries for a given query.

The subquery of a tree query can be represented as an *induced subgraph*. Let the query graph of a tree query be denoted by a graph T=(V,E) where V is the set of nodes and E is the set of edges. For any set V' of nodes of T, the induced subgraph of T, denoted as T'=(V',E'), is the maximal subgraph of T[6]. Thus, E' contains all the edges of E whose endpoints are in V' and we say that T' is induced by V'. In contrast to a subgraph, all nodes in an induced subgraph are connected. According to the definition, any connected subgraph of a query graph is an induced subgraph, and is also a tree. The reason why we represent a subquery as an induced subgraph rather than a subgraph is that, in the subgraph, there may be nodes which are not connected with other nodes. This corresponds to the heuristic which avoids unnecessary Cartesian products[14].

The algorithm *gensub* in Figure 3 generates all induced subgraphs for a given tree T with n nodes numbered in breadth-first-search order. The generated induced subgraph with i nodes is represented as a node set T_{ij} by which the subgraph is induced, where i represents the number of nodes in T_{ij} and j represents the j-th generated subgraph with i nodes. The induced subgraph with i nodes, T_{ij}, of a tree T is the i-class subquery T_{ij} of the n-class query T. The number of generated induced subgraphs with i nodes is represented as m_i. Thus, the total number of generated induced subgraphs is $\sum m_i$.

Procedure gensub(T,T_{ij},m_i)
Input: A tree T whose nodes are numbered in BFS order
Output: All induced subgraphs with i nodes (i-class queries) where m_i is the number of generated subgraphs with i nodes.

```
1   for j:=1 to n do T1j := {Cj};
2   m1 := n;
3   for i:=2 to n do
4       j := 1;
5       for k:=1 to mi-1 do
6           let max be the largest node number in Ti-1,k
7           if max < n then
8               for p:=max+1 to n do
9                   if there is an edge between Cp and
                    any node in Ti-1,k then
10                      Tij := Ti-1,k ∪ {Cp};
11                      j := j + 1;
12      mi := j - 1;
```

Figure 3. Induced subgraph generation algorithm

The algorithm first generates n induced subgraphs with 1 node, $T_{1j}(1 \le j \le n)$, each of which contains only one node in the input tree. Based on these subgraphs, it then generates the induced subgraphs with i nodes, $T_{ij}(1 \le j \le m_i)$, using the generated induced subgraphs with i-1 nodes, $T_{i-1,k}(1 \le k \le m_{i-1})$, for i=2,3,...,n. The induced subgraph with n nodes, T_{n1}, which is finally generated is the input tree itself. Thus, it contains all the nodes in the tree.

4.3 Principle of Optimality

In this subsection, we show that the distributed tree query optimization problem obeys the principle of optimality. It is assumed that an i-class query T_{ij} is a subquery of a tree query T with n classes and is divided into a r-class subquery T_{rp} and an (i-r)-class subquery $T_{i-r,q}$. Then the queries T_{rp},$T_{i-r,q}$ and T_{ij} are the induced subgraphs of T such as $T_{ij}=T_{rp} \cup T_{i-r,q}$. That is, dividing a query T_{ij} into subqueries T_{rp} and $T_{i-r,q}$ implies that the node set T_{ij} is partitioned into two node sets T_{rp} and $T_{i-r,q}$. It is also assumed that $a_{u(x)}$ is the u-th execution plan among all feasible plans which will give a result of T_{rp} at site x, $b_{v(x)}$ is the v-th execution plan among all feasible plans which will give a result of $T_{i-r,q}$ at site x, and Cost(p) is the cost of the plan p. The execution plans for implicit joins and transmissions are produced by the following functions. The function $\text{buildJPlan}_x(a_{u(x)},b_{v(x)})$ combines the plans $a_{u(x)}$ and $b_{v(x)}$ into the plan in which the results of T_{rp} and $T_{i-r,q}$ at site x are joined implicitly at this site, that is, $\text{buildJPlan}_x(a_{u(x)},b_{v(x)}) = IJ_x(a_{u(x)},b_{v(x)})$. We denote $c_{w(x)}$ to be the plan produced by the function. By executing this plan, the result of T_{ij} is materialized at site x. The function $\text{buildTPlan}_{xt}(c_{w(x)})$ extends the plan $c_{w(x)}$ into another plan in which the result of T_{ij} at site x is transferred to site t, namely, $\text{buildTPlan}_{xt}(c_{w(x)})=TR_{xt}(c_{w(x)})$. By executing this plan, we can get the result of T_{ij} at site t.

As the results of T_{rp} and $T_{i-r,q}$ must be at the same site to be joined implicitly, the cost of the plan produced by the $\text{buildJPlan}_x(a_{u(x)},b_{v(x)})$ is the sum of the costs for getting the results of the subqueries T_{rp} and $T_{i-r,q}$ at site x and the cost for the implicit join at this site. Similarly, the cost of the plan produced by the $\text{buildTPlan}_{xt}(c_{w(x)})$ is the sum of the cost for materializing the result of the query at site x and the cost for transferring the result of the query T_{ij} from site x to site t. Thus, we have

$$\text{Cost}(\text{buildJPlan}_x(a_{u(x)},b_{v(x)})) = \text{Cost}(a_{u(x)}) + \text{Cost}(b_{v(x)}) + \text{join}_x(T_{rp},T_{i-r,q}) \quad (1)$$

$$\text{Cost}(\text{buildTPlan}_{xt}(c_{w(x)})) = \text{Cost}(c_{w(x)}) + \text{trans}_{xt}(T_{ij}) \quad (2)$$

where $\text{join}_x(T_{rp},T_{i-r,q})$ is the cost for the implicit join between the results of T_{rp} and $T_{i-r,q}$ at site x and $\text{trans}_{xt}(T_{ij})$ is the cost for transferring the result of T_{ij} from site x to site t by slightly modifying the definition for implicit join and transmission costs mentioned in section 3.2.

To choose the optimal plans among the plans produced by the buildJPlan and buildTPlan respectively, we use the principle of optimality: If plans differ only in subplans, the plan with optimal subplans is also optimal. This principle can be expressed formally as follows

$$\text{Cost}(a_{l(x)}) \le (\forall u)\text{Cost}(a_{u(x)}) \text{ and } \text{Cost}(b_{l(x)}) \le (\forall v)\text{Cost}(b_{v(x)}) \Rightarrow \text{Cost}(\text{buildJPlan}_x(a_{l(x)},b_{l(x)})) \le (\forall u,v)\text{Cost}(\text{buildJPlan}_x(a_{u(x)},b_{v(x)})) \quad (3)$$

$$\text{Cost}(c_{l(x)}) \le (\forall w)\text{Cost}(c_{w(x)}) \Rightarrow \text{Cost}(\text{buildTPlan}_{xt}(c_{l(x)})) \le (\forall w)\text{Cost}(\text{buildTPlan}_{xt}(c_{w(x)})) \quad (4)$$

where the subplans $a_{l(x)}$, $b_{l(x)}$ and $c_{l(x)}$ are the lowest cost plans among all feasible $a_{u(x)}$, $b_{v(x)}$ and $c_{w(x)}$ respectively.

Theorem 1: The cost model for implicit join and transmission plans satisfies the principle of optimality.

Proof: To show the theorem, we have to prove Equation (3) and (4). The proof for Equation (3) is as follows, where steps ① and ④ are according to the cost definition of buildJPlan and steps ② and ③ are by the assumption that $Cost(a_{1(x)}) \leq Cost(a_{u(x)})$ and $Cost(b_{1(x)}) \leq Cost(b_{v(x)})$ for an arbitrary u and v respectively.

$Cost(buildJPlan_x(a_{1(x)},b_{1(x)}))$
$= Cost(a_{1(x)}) + Cost(b_{1(x)}) + join_x(T_{rp},T_{i-r,q})$ ①
$\leq Cost(a_{u(x)}) + Cost(b_{1(x)}) + join_x(T_{rp},T_{i-r,q})$ ②
$\leq Cost(a_{u(x)}) + Cost(b_{v(x)}) + join_x(T_{rp},T_{i-r,q})$ ③
$= Cost(buildJPlan_x(a_{u(x)},b_{v(x)}))$ ④

The proof for Equation (4) is as follows, where steps ⑤ and ⑦ are according to the cost definition of buildTPlan and step ⑥ is by the assumption that $Cost(c_{1(x)}) \leq Cost(c_{w(x)})$ for an arbitrary w.

$Cost(buildTPlan_{xt}(c_{1(x)}))$
$= Cost(c_{1(x)}) + trans_{xt}(T_{ij})$ ⑤
$\leq Cost(c_{w(x)}) + trans_{xt}(T_{ij})$ ⑥
$= Cost(buildTPlan_{xt}(c_{w(x)}))$ ⑦ □

4.4 Algorithm for Distributed Query Optimization

In this subsection, we describe the dynamic programming algorithm to determine the minimum total cost to get the result of a query T_{n1} at a query site. This cost is obtained by recursively computing the following costs. Let $CostJ_x(T_{ij})$ be the minimum cost for materializing the result of T_{ij} at site x and $CostT_t(T_{ij})$ be the minimum cost for getting the result of T_{ij} at site t. In order to materialize the result of T_{ij} at site x, the implicit join between the results of subqueries T_{rp} and $T_{i-r,q}$ must be performed at this site. As there are i-1 pairs of subqueries T_{rp} and $T_{i-r,q}$ for a query T_{ij}, $CostJ_x(T_{ij})$ must be minimal among i-1 costs for performing the implicit join between the results of T_{rp} and $T_{i-r,q}$. The results of T_{rp} and $T_{i-r,q}$ must be at site x for the implicit join at this site. Therefore, $CostJ_x(T_{ij})$ includes the costs for getting the results of T_{rp} and $T_{i-r,q}$ at site x. By the principle of optimality, these costs must be minimal, that is, $CostT_x(T_{rp})$ and $CostT_x(T_{i-r,q})$. Thus, we have

$$
\begin{aligned}
CostJ_x(T_{ij}) &= \min_{all\ pair\ T_{rp},T_{i-r,q}\ s.t.\ T_{ij}=T_{rp}\cup T_{i-r,q}} \{ \min_{all\ u,v} \{Cost(buildJPlan_t(a_{u(x)},b_{v(x)}))\}\} \\
&= \min_{all\ pair\ T_{rp},T_{i-r,q}\ s.t.\ T_{ij}=T_{rp}\cup T_{i-r,q}} \{Cost(buildJPlan_t(a_{1(x)},b_{1(x)}))\} \\
&= \min_{all\ pair\ T_{rp},T_{i-r,q}\ s.t.\ T_{ij}=T_{rp}\cup T_{i-r,q}} \{Cost(a_{1(x)}) + Cost(b_{1(x)}) + join_x(T_{rp},T_{i-r,q})\} \\
&= \min_{all\ pair\ T_{rp},T_{i-r,q}\ s.t.\ T_{ij}=T_{rp}\cup T_{i-r,q}} \{CostT_x(T_{rp}) + CostT_x(T_{i-r,q}) + join_x(T_{rp},T_{i-r,q})\} \quad (5)
\end{aligned}
$$

In order to get the result of T_{ij} at site t, the result may be directly materialized by performing implicit join at site t or materialized at another site and then transferred to site t. Therefore, the costs for materializing the result of T_{ij} at site x must be computed, varying x over all related sites where the result of T_{ij} may be materialized. By the principle of optimality, these costs must be minimal, that is, $CostJ_x(T_{ij})$. Thus, we have

$$
\begin{aligned}
CostT_t(T_{ij}) &= \min_{all\ x\ s.t.\ 1\leq x\leq s} \{ \min_{all\ w} \{Cost(buildTPlan_{xt}(c_{w(x)}))\}\} \\
&= \min_{all\ x\ s.t.\ 1\leq x\leq s} \{Cost(buildTPlan_{xt}(c_{1(x)}))\} \\
&= \min_{all\ x\ s.t.\ 1\leq x\leq s} \{Cost(c_{1(x)}) + trans_{xt}(T_{ij})\} \\
&= \min_{all\ x\ s.t.\ 1\leq x\leq s} \{CostJ_x(T_{ij}) + trans_{xt}(T_{ij})\} \quad (6)
\end{aligned}
$$

In this equation, if x=t, the result of T_{ij} is directly materialized at site t because of $trans_{tt}(T_{ij})=0$. Otherwise, it is materialized at the site x and then transferred to site t.

The initial conditions for Equation (5) and (6) are

- when C_j is at site t: $CostT_t(T_{1j})=0$
- when C_j is at other site than t:

$$CostT_t(T_{1j}) = \min_{x\in DUP(C_j)} \{trans_{xt}(C_j)\}$$

These conditions allow duplications of a class. Let $DUP(C_j)$ be the set of sites at which duplicates of a class C_j exist. The first condition means that, for all $t \in DUP(C_j)$, $CostT_t(T_{1j})$ is zero. The second also means that, for all $t \notin DUP(C_j)$, $CostT_t(T_{1j})$ is the minimum cost among the costs for transferring a duplicate of C_j from a site $x \in DUP(C_j)$ to site t. If transmission rates between sites are equal, it need not choose the minimum cost in the second condition because all $trans_{xt}(C_j)$s are the same. In such a network, $CostT_t(T_{1j})$ may be any $trans_{xt}(C_j)$ where the site x is any available site in $DUP(C_j)$.

If $CostJ_x(T_{ij})$ and $CostT_t(T_{ij})$ are computed for each i-class query T_{ij} varying x and t over all related sites and these computations are repeatedly applied in a bottom-up fashion, $CostT_{query\_site}(T_{n1})$ for n-class query T_{n1} can be eventually obtained. When computing $CostJ_x(T_{ij})$, Equation (5) prunes all feasible implicit join plans except the optimal one. Similarly when computing $CostT_t(T_{ij})$, all feasible transmission plans except the optimal one are pruned by Equation (6). Therefore, applying Equation (5) and (6) implies a *two-step pruning* mechanism for each i-class query. The obtained $CostJ_x(T_{ij})$ and $CostT_t(T_{ij})$ are saved and reused rather than recomputed when the query T_{ij} is employed as a subquery of another larger query.

Generating an optimal plan for each i-class subquery is the process of obtaining feasible subtrees(subplans) of the optimal GET. For each i-class subquery T_{ij}, computing the costs $CostJ_x(T_{ij})$ and $CostT_t(T_{ij})$ means the process of computing the costs of the subtrees whose roots are the corresponding join and transmission nodes in the optimal GET. That is, the cost of an optimal subplan whose root is the join node with site label x is represented by $CostJ_x(T_{ij})$ and the cost of an optimal subplan whose root is the transmission node with site label t is represented by $CostT_t(T_{ij})$. In Figure 4, we represent the generating process of the GET in Figure 2.a with the optimal cost of each subplan. The subqueries for the optimal GET and the optimal cost of the given query are as follows

T_{11}={P}, T_{12}={V}, T_{13}={C}, T_{14}={D}, T_{15}={T}, T_{16}={E}, T_{23}={V,D}
T_{24}={C,T}, T_{44}={V,C,T,D}, T_{53}={V,C,T,D,E}, T_{61}={P,V,C,T,D,E}

$$
\begin{aligned}
CostT_1(T_{61}) = {} & trans_{23}(T_{13}) + join_3(T_{13},T_{15}) + trans_{13}(T_{14}) \\
& + join_3(T_{12},T_{14}) + join_3(T_{24},T_{23}) + trans_{32}(T_{44}) \\
& + join_2(T_{44},T_{16}) + trans_{21}(T_{53}) + join_1(T_{11},T_{53})
\end{aligned}
$$

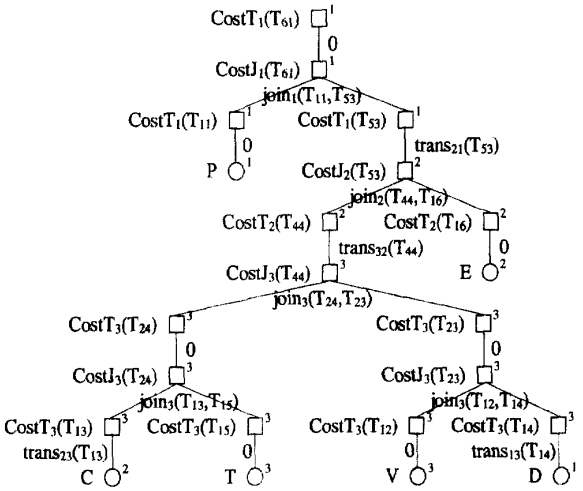


Figure 4. Generation of the optimal GET

Algorithm TQOD

Input: A tree query with classes $C_1,...,C_n$ distributed at s sites as query graph T=(V,E).

Output: An optimal GET with its cost.

```
1   gensub(T,Tij,mi);
2   for t:=1 to s do
3      for j:=1 to n do
4         if Cj is located at site t then CostTt(T1j) := 0
5         else
6            find site x at which a duplicate of Cj exists;
7            CostTt(T1j) := transxt(T1j);
8   for i:=2 to n do
9      for x:=1 to s do
10        for j:=1 to mi do
11           CostJx(Tij) := ∞;
12           for each pair Trp,Ti-r,q s.t. Tij=Trp ∪ Ti-r,q do
13              jc := CostTx(Trp)+CostTx(Ti-r,q)+joinx(Trp,Ti-r,q);
14              if CostJx(Tij) > jc then CostJx(Tij) := jc;
15           save p,q,r which minimize CostJx(Tij)
             into OptPlanx(Tij);
16     for t:=1 to s do
17        for j:=1 to mi do
18           CostTt(Tij) := ∞;
19           for x:=1 to s do
20              tc := CostJx(Tij) + transxt(Tij);
21              if CostTt(Tij) > tc then CostTt(Tij) := tc;
22           save x which minimizes CostTt(Tij)
             into OptPlant(Tij);
23  Plan := OptPlanquery_site(Tn1) with the cost CostTquery_site(Tn1);
```
Figure 5. Distirbuted tree query optimization algorithm

The dynamic programming equations are coded in the algorithm *TQOD* of Figure 5. The algorithm first generates all i-class subqueries T_{ij} of a given query T in line 1. It then initializes $CostT_t(T_{1j})$ for each 1-class query T_{1j} in lines 2-7. In lines 8-22, it uses Equation (5) and (6) to compute $CostJ_x(T_{ij})$ and $CostT_t(T_{ij})$ of each i-class query T_{ij} varying x and t over all related sites. This process is repeatedly applied in a bottom-up fashion, that is, for i=2,3,...,n.

4.5 Complexity of Algorithm

The algorithm *TQOD* generates all i-class subqueries for a given query and considers all feasible plans for each generated subquery at each related site. It then chooses the plan with minimum cost as the optimal plan of the subquery at each related site, and saves the optimal plan at a site with its cost for future use. Thus, the time complexity of the algorithm depends on the number of plans considered and the space complexity depends on the number of plans saved.

In order to compute the complexities, the number of i-class subqueries generated by the algorithm *gensub* should be known. This is determined by the shape of the query graph and the number of classes in the graph. Let's consider the special cases of a tree query, namely, a *chain query* and a *star query* whose corresponding query graphs are chain and star respectively. For a chain query with n classes, an i-class subquery is composed of i consecutive classes. Thus, the number of i-class subqueries generated by the algorithm *gensub*, m_i, is n-i+1. Also, for a star query with n classes, an i-class subquery is constructed by choosing i-1 classes from n-1 classes around the root class. Thus, we have $m_i = \binom{n-1}{i-1}$.

For the tree query with n classes, the number of subqueries generated by the *gensub*, $\sum m_i$, is the largest with a star query and the smallest with a chain query. This is easily verified by the following observations: If any edge between any class i and the root class of a star query is moved to be connected to any non-root class j instead, there is only one way to perform the implicit join between class i and the rest of the altered query. Thus, the number of generated subqueries of the altered query is always smaller than that of the star query. Also, if this procedure is repeatedly applied to the star query, any shape of a tree query can be obtained. Conversely, if any edge between the leaf class(n-th) and the (n-1)-th class of a chain query is moved to be connected to any class j instead, the number of generated subqueries of the altered query is always larger than that of the chain query. Similarly, if this procedure is repeatedly applied to the chain query, any shape of a tree query can be obtained.

The complexity of the algorithm *TQOD* depends on the number of subqueries generated and this number is maximal with a star query and minimal with a chain query respectively. Thus, the algorithm has the worst case complexity with a star query and the best case complexity with a chain query.

Theorem 2: The algorithm *TQOD* has the time complexity $O(sn^3)$ at the best case, and $O(sn2^{n-1})$ at the worst case.

Proof: The time complexity of the algorithm is determined by the number of plans which are considered in line 13 and 20. A chain query has the best case time complexity, and the number of generated i-class subqueries T_{ij} of the chain query with n classes is n-i+1. There are i-1 pairs of subqueries for each T_{ij}. Thus, the total number of implicit join plans considered in line 13 is

$$\sum_{i=2}^{n}\sum_{x=1}^{s}\sum_{j=1}^{n-i+1}\sum_{p=1}^{i-1}1 \;=\; s\sum_{i=1}^{n-1}(ni-i^2) \;=\; s\frac{(n-1)n(n+1)}{6}$$

and the total number of transmission plans considered in line 20 is

$$\sum_{i=2}^{n}\sum_{t=1}^{s}\sum_{j=1}^{n-i+1}\sum_{x=1}^{s}1 = s^2\sum_{i=1}^{n-1}(n-i) = s^2\frac{(n-1)n}{2}$$

Namely, considering all implicit join and transmission plans take time $O(sn^3)$ and $O(s^2n^2)$ respectively. Because the number of sites related with the query, s, is less than or equal to n, the total best case time complexity is less than $O(sn^3)$.

Similarly, a star query has the worst case time complexity, and the number of generated i-class subqueries T_{ij} of the star query with n classes is $\binom{n-1}{i-1}$. There are i-1 pairs of subqueris for each T_{ij}. Thus, the total numbers of implicit join and transmission plans considered in line 13 and 20 respectively are

$$\sum_{i=2}^{n}\sum_{x=1}^{s}\sum_{j=1}^{\binom{n-1}{i-1}}\sum_{p=1}^{i-1}1 = s\sum_{i=1}^{n-1}i\binom{n-1}{i} = s(n-1)2^{n-2}$$

$$\sum_{i=2}^{n}\sum_{t=1}^{s}\sum_{j=1}^{\binom{n-1}{i-1}}\sum_{x=1}^{s}1 = s^2\sum_{i=1}^{n-1}\binom{n-1}{i} = s^2(2^{n-1}-1)$$

Therefore, the total worst case time complexity is $O(sn2^{n-1})$ according to the same reason.□

Corollary: The algorithm *TQOD* has the space complexity $O(sn^2)$ at the best case, and $O(s2^{n-1})$ at the worst case.

Proof: The space complexity of the algorithm is determined by the number of plans which are saved in line 15 and 22. Similar to Theorem 2, the best and worst case time complexities are obtained by computing the number of plans saved for chain and star queries respectively. Thus, we have

$$\sum_{i=2}^{n}\sum_{t=1}^{s}\sum_{j=1}^{n-i+1}1 = s\sum_{i=1}^{n-1}(n-i) = s\frac{(n-1)n}{2}$$

$$\sum_{i=2}^{n}\sum_{t=1}^{s}\sum_{j=1}^{\binom{n-1}{i-1}}1 = s\sum_{i=1}^{n-1}\binom{n-1}{i} = s(2^{n-1}-1)$$ □

5. Conclusions

In this paper, we describe a model for distributed query optimization and propose an algorithm which determines the optimal traversal orderings for a tree query. The objective function of optimization is to minimize the total processing time which includes not only the communication costs but also the local processing costs. The proposed optimizer considers the execution plans which may be deep or bushy trees. It also considers the plans which start traversal anywhere in a query graph. Thus, our optimizer considers all feasible plans for a query and always produces the optimal one.

The search work done by the optimizer is reduced by applying dynamic programming technique. We devise the two-step pruning mechanism to select an optimal plan among all feasible plans and prove that this pruning mechanism satisfies the principle of optimality. While building an optimal plan of a query, we apply the mechanism successively to each subquery so that the plans except the optimal one are pruned efficiently. The algorithm has the polynomial time complexity $O(sn^3)$ at the best case, but the exponential $O(sn2^{n-1})$ at the worst case.

Future work includes: extending the algorithm to minimize the response time of processing a query, elaborating the cost model to predict the cost of query processing exactly, and developing the heuristics to reduce the optimization overhead. In particular, we are developing the heuristics based on the fact that the proposed algorithm has a polynomial complexity with a chain query. We plan to validate our heuristics through experimentation.

References

[1] Bernstein, P.A., Goodman, N., Wong, E., Reeve, C.L., and Rothnie, Jr., J.B., "Query Processing in a System for Distributed Databases (SDD-1)", ACM Trans. Database Systems, Vol.6, No.4, Dec. 1981, pp.602-625.

[2] Bertino, E., and Rosati, S., "Query processing in an object-oriented distributed DBMS in a LAN environment", Rivista di Informatica, Vol.20, No.3, July 1990, pp.221-249.

[3] Chiu, D-M, Bernstein, P. and Ho, Y-C, "Optimizing chain queries in a distributed database system", SIAM Journal of Computing, Vol.13, No.1, Feb., 1984, pp.116-134

[4] Cormen, T. H., Leiserson, C.E., Rivest, R.L., *Introduction to Algorithms*, 1992, MIT Press

[5] Ganguly, S., Hasan, W., and Krishnamurthy, R., "Query optimization for parallel execution", Proc. ACM SIGMOD, June 1992, pp.9-18

[6] Harary, F., *Graph theory*, p.11, Addison-Wesley Publishing Company, Inc., 1972

[7] Ioannidis, Y.E., Kang, Y.C., "Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization", Proc. ACM SIGMOD, May 1991, pp.168-177.

[8] Jenq, B.P., Woelk, D., Kim, W. and Lee, W-L, "Query Processing in Distributed Orion", Int'l Conf. Extending Databse Technology(EDBT), Mar. 1990, pp.169-187

[9] Kang, H., Roussopoulos, N., "On cost-effectiveness of a semijoin in distributed query processing", Proc. 11th Int'l Computer Software and Applications Conference, Oct. 1987, pp.531-537

[10] Kemper, A., Moerkotte, G., "Access support in object bases", Proc. ACM SIGMOD, June 1990, pp.364-374

[11] Kim, W., "A model of queries for object-oriented databases", Proc. Conf. Very Large Data Bases, Aug. 1989, pp.423-432

[12] Lanzelotte, R.S.G., Valduriez, P., Ziane, M., Cheiney, J.P., "Optimization of nonrecursive queries in OODBs", Proc. Conf. Deductive and Object-Oriented Databases, Dec. 1991, pp.1-21

[13] Mitchell, G.A., Extensible query processing in an object-oriented database, Ph.D thesis, Brown University, May 1993

[14] Selinger, P.G., Adiba, M., "Access path selection in distributed data base", 1st Int'l Conf. Data Bases, 1980, pp.204-215

[15] Yu, C.T., "Optimization of distributed tree query", Journal of Computer and System Science, Academic Press, Inc., Vol.29, 1984, pp.409-445

[16] Yu, C.T. and Chang C.C., "Distributed query processing", ACM Computing Surveys, Vol.16, No.4, Dec., 1984, pp.399-432