A Heuristic Approach for Optimization of Path Expressions

Cetin Ozkan, Asuman Dogac and Cem Evrendilek

Software Research and Development Center of TUBITAK
Middle East Technical University
06531, Ankara Turkiye
e-mail: asuman@srdc.metu.edu.tr

Abstract. The object-oriented database management systems store references to objects (implicit joins, precomputed joins), and use path expressions in query languages. One way of executing path expressions is pointer chasing of precomputed joins. However it has been previously shown that converting implicit joins to explicit joins during the optimization phase may yield better execution plans. A path expression is a linear query, therefore, considering all possible join sequences within a path expression is polynomial in the number of classes involved. Yet, when the implicit joins are converted to explicit joins in a query involving multiple path expressions bound to the same bind variable, the query becomes a star query and thus considering all possible joins is exponential in the number of paths involved. This implies that there is a need for improvement by using heuristic in optimizing queries involving multiple path expressions.

A heuristic based approach for optimizing queries involving multiple path expressions is described in this paper. First, given the cost and the selectivities of path expressions by considering a path expression as a unit of processing, we provide an algorithm that gives the optimum execution order of multiple path expressions bound to the same bind variable. For this purpose, we derive the formulas for the selectivities of path expressions. Then by using this ordering as a basis we provide a general heuristic approach for optimizing queries involving multiple path expressions.

Two optimizers are developed to compare the performance of the heuristic based approach suggested in this paper with the performance of an optimizer based on an exhaustive search strategy. The exhaustive optimizer is generated through Volcano Optimizer Generator (VOG). The results of the experiments indicate that the heuristic based optimizer has a superior performance with the increasing number of path expressions.

1 Introduction

The object-oriented database management systems store references to objects (implicit joins, precomputed joins), and use path expressions in query languages. One way of executing path expressions is pointer chasing of precomputed joins. However it has been shown in [Bla 93] that converting implicit joins to explicit

joins during the optimization phase, makes it possible to consider a wider range of join sequences and thus yields better execution plans in most of the cases.

A path expression corresponds to a linear query in relational systems where tables are connected by binary predicates in a straight line. In [Ono 90], it has been shown that the computational complexity (i.e the number of joins that must be considered when using dynamic programming for optimization) of linear queries with composite inners (bushy trees) is (N^3-N)/6. If the composite inners are not considered the complexity reduces to (N-1)^2. Therefore considering all possible join sequences within a path expression is polynomial in the number of classes involved. Yet when the implicit joins are converted to explicit joins, because of multiple (more than one) path expressions bound to the same bind variable, the query corresponds to a star query (in fact a hybrid query, i.e., star of linear queries) in relational systems where a table at the center is connected by binary predicates to each of the other surrounding tables. In [Ono 90], it has also been shown that using dynamic programming to optimize a star query with N quantifiers requires evaluating (N-1)2^{N-2} feasible joins. Thus considering all possible joins is exponential in the number of classes involved.

Many existing relational optimizers use heuristics within dynamic programming to limit the join sequences evaluated. One heuristic employed by System R [Sel 79] and R* [Loh 85] constructs only joins in which a single table is joined at each step with the results of previous joins, in a pipelined way. Thus they produce left deep join trees.

We have developed a heuristic based approach for optimizing queries involving multiple path expressions. First, given the cost and the selectivities of path expressions by considering a path expression as a unit of processing, we provide an algorithm that gives the optimum execution order of path expressions bound to the same bind variable. For this purpose we derive the formulas for the selectivities of path expressions. Then, by using the results obtained as hint a general heuristic approach is developed.

In order to test the effectiveness of the heuristic proposed, two query optimizers are developed. The first optimizer is based on the heuristic suggested in this paper. The second optimizer is based on an exhaustive search strategy and is generated through Volcano Query Optimizer Generator. Since path optimization mainly involves join enumeration, a subset of the transformation rules given in [Bla 93] are used. The results of the experiments indicate that the heuristic based optimizer has a superior performance with the increasing number of path expressions.

It should be noted that in optimizing multiple paths, detecting common subexpressions is an important factor and both of the optimizers developed take care of this situation. However since our focus of attention is not common subexpressions, our sample queries did not contain common subexpressions.

In section 2, the cost model is presented and the formulas for the selectivity of path expressions are derived. In Section 3, an algorithm that gives the optimum execution order of path expressions bound to the same bind variable is provided by considering a path expression as a unit of processing. Section 4 presents the

principles of the heuristic based optimizer. In Section 5, the performance results are given. Finally, Section 6 contains the conclusions.

2 Cost Model

In the object model [Atk 92] used in this paper, complex objects are built from simpler ones by applying constructors to them. The simplest objects are integers, characters, byte strings of any length, Booleans, and floats. The complex object constructors are Tuple, Set, List and Reference. Any constructor can be applied to any object. Each object has a unique Object Identifier (OID). Objects are grouped in the abstraction level of a class, in other words, classes have extensions. Each object is as a member of only one class.

2.1 Cost Model Parameters

In Table 1 the cost model parameters are presented. In this table, C_i is a class, A is either an attribute or a parameterless method of class C_i with an atomic return type which is treated in the same way as an atomic attribute, and C_{i+1} is the class referenced by attribute A of class C_i.

Parameter	Short Hand Notation	Definition
$\|C_i\|$	-	Total number of instances of C_i
nbpages (C)	-	Total number of pages C_i occupies
size(C_i)	-	Size of an instance of class C_i
notnull (A, C_i)	-	The proportion of instances of class C_i where atribute A is not null
fan(A,C_i,C_{i+1})	fan_i	The average number of instances of class C_{i+1} that are referenced by an instance of C_i through attribute A
totref(A,C_i,C_{i+1})	$totref_i$	The total number of objects in class C_{i+1} that are referenced by at least one object in class C_i through attribute A
dist(A, C_i)	dist	The number of distinct values of the atomic attribute A of class C_i
max(A, C_i)	max	The maximum value of the atomic attribute A of class C_i
min(A, C_i)	min	The mimimum value of the atomic attribute A of class C_i

Table 1 . Cost Model Parameters

The parameters calculated by using the above listed cost model parameters are provided in the following. The number of the total references from class C_i to class C_{i+1} through attribute A is denoted by totlinks(A,C_i,C_{i+1}) and given by the following equation :

$$totlinks(A, C_i, C_{i+1}) = fan(A, C_i, C_{i+1}) * |C_i|$$

The probability that an instance of class C_{i+1} is referenced by the instances of class C_i through attribute A is given by the following formula:

$$hitprb(A, C_i, C_{i+1}) = totref(A, C_i, C_{i+1})/|C_{i+1}|$$

The shorthand notation for these parameters are as follows:
$hitprb_i = hitprb(A, C_i, C_{i+1})$ and $totlinks_i = totlinks(A_i, C_i, C_{i+1})$.

2.2 Selectivity

A simple predicate in the system is a triplet of the form $< P_1, \Theta, oprnd >$, where P_1 is a path expression, Θ is a comparison operator $(=, <>, \geq, \leq, >, <)$, and oprnd is either a constant or another path expression.

Selectivity for Atomic Attributes The well-known selectivity calculations assuming the uniform distribution of the atomic values described in [Ozk 90] are used. The selectivity of the expression "s.A = constant", denoted as σ , where s is a bind variable binding to a class C, and A is an atomic attribute, is given by the following formula where dist is distinct values of A in C:

$$\sigma(s.A) = 1/dist(A, C)$$

Selectivity of Path Expressions Assume that there is a path expression involving m classes referenced through attributes, A_1 through A_m, where A_1 through A_{m-1} is constructed using the set or the reference constructors. A_m is an atomic attribute and A_i is an attribute of class C_i. We need to calculate the selectivity, $\sigma_{path}(p.A_1.A_2...A_m, \Theta)$, for a single path expression "$p.A_1.A_2...A_m$ Θ c", where Θ is a comparison operator and c is a constant.

The calculation of the selectivity of "A_m Θ c", (A_m), is clear from the previous section. Therefore the expected number of instances of C_m, denoted by k_m, that satisfies this condition is:

$$k_m = |C_m| * \sigma(A_m)$$

It is clear that when there is no selection on the A_m attribute $k_m = |C_m|$. In forward traversal, assuming that we start with k objects of class C_1 and traverse the path $p.A_1.A_2...A_i$ in forward direction, the expected number of objects of class C_{i+1} , denoted by fref, is given by the following formula:

$$fref(p.A_1.A_2...A_i, k) = \begin{cases} k & , i = 0 \\ c(totlinks_i, totref_i, fref(p.A_1.A_2...A_{i-1}, k) & \\ \quad * fan_i) & , i > 0 \end{cases}$$

where, c(n,m,r) is an approximation to the following statistical problem: Given n objects uniformly distributed over m colors, how many different colors c are selected if we take just r objects? This statistical problem has been solved by using different mathematical approximations. An approximation assumed in [Cer 85] is as follows:

$$c(n,m,r) = \begin{cases} r & ,\mathrm{r} < \mathrm{m}/2 \\ (r+m)/3 & ,\mathrm{m}/2 \leq \mathrm{r} \leq 2\mathrm{m} \\ m & ,\mathrm{r} > 2\mathrm{m} \end{cases}$$

Starting with one instance of class C_1, the number of objects of class C_m obtained at the end of forward path traversal is given by fref(p.A_1...A_{m-1},1). On the other hand, k_m objects have been selected through the predicate A_m Θ c. Then the selectivity of a path expression, p.A_1.A_2...A_m Θ c , which is defined to be the probability of at least one object being in common in two sets with cardinalities fref(p.A_1.A_2...A_{m-1},1) and k_m* hitprb(A_{m-1},C_{m-1},C_m) respectively, is given by

$$\sigma_{path}(\ p.A_1.A_2...A_m, \Theta) = \mathrm{o}(\mathrm{totref}_{m-1}\ ,\ \mathrm{fref}(\mathrm{p}.A_1.A_2...A_{m-1}\ ,1),\ \mathrm{k}_m\ * \mathrm{hitprb}(\mathrm{A}_{m-1}, C_{m-1}, C_m))$$

where o(t,x,y) is the probability that there exists at least one object in common in two sets selected with replacement out of t distinct objects and is defined as follows:

$$o(t,x,y) = 1 - C(t-x,y)/C(t,y)$$

where C stands for combination, and x and y are the cardinalities of the two sets respectively.

3 On the Execution Order of Path Expressions

Consider m path expressions which are bound to the same bind variable, say p, in an AND-term:

p.a_{11}.a_{12}...a_{1n1} Θ_1 c_1
p.a_{21}.a_{22}...a_{2n2} Θ_2 c_2
$\vdots$
p.a_{m1}.a_{m2}...a_{mnm} Θ_m c_m

Assume F_i denotes the cost of executing the path expression i, p.a_{i1}.a_{i2}...a_{ini} Θ_i c_i, and let the selectivity of this path expression be s_i= σ_{path}(p.a_{i1}.a_{i2}...a_{ini} Θ_i c_i).

Given the cost and the selectivity of each of the path expressions, the problem of finding the least costly execution order of these path expressions can be stated as the following minimization problem:

Find a permutation of the integers 1 through m stored in i[1] through i[m] which minimizes

$$f = F_{i[1]} + s_{i[1]} * F_{i[2]} + s_{i[1]} * s_{i[2]} * F_{i[3]} + ... + s_{i[1]} * s_{i[2]} * ...s_{i[m-1]} * F_{i[m]}$$

where F_j and s_j, j $\in$ i[1] through i[m], are the cost of traversing and the selectivity of the j^{th} path expression respectively. In other words, we are trying to minimize the objective function f, denoting the total cost of executing m path expressions in the order induced by the array i.

Theorem 1 : Assume π denotes a permutation of the integers 1 through m such that path expression indices are sorted in ascending order of $F_i/(1 - s_i)$ values, such that $1 \leq i \leq m$. This π minimizes the objective function f.

Sketch of Proof : By induction on the number of path expressions. It is true for 2 path expressions. In this case $f = F_1 + s_1 F_2$ or $f = F_2 + s_2 F_1$.
If $F_1 + s_1 F_2 < F_2 + s_2 F_1$ then by simple manipulation,
$F_1 / (1 - s_1) < F_2 / (1 - s_2)$ is obtained. Assuming that it is true for m path expressions and we will try to show that it is also true for m+1 path expressions. Let us assume that $F_i/(1-s_i) < F_{i+1}/(1-s_{i+1})$ for $1 \leq i \leq m-1$, and assume also that $F_j/(1-s_j) < F_{m+1}/(1-s_{m+1}) < F_{j+1}/(1-s_{j+1})$ for some j where $1 < j < m$.

We claim that $f_1 = F_1 + s_1F_2 + ... + s_1s_2...s_{j-1}F_j + s_1s_2...s_{j-1}s_jF_{m+1} + s_1s_2...s_{j-1}s_js_{m+1}F_{j+1} + ... + s_1s_2...s_{j-1}s_js_{m+1}s_{j+1}...s_{m-1}F_m$ is minimum. Assume on the contrary that,
$f_2 = F_1 + s_1F_2 + ... + s_1s_2...s_{k-1}F_k + s_1s_2...s_{k-1}s_kF_{m+1} + s_1s_2...s_{k-1}s_ks_{m+1}F_{k+1} + ... + s_1s_2...s_{k-1}s_ks_{m+1}s_{k+1}...s_{m-1}F_m$ is minimum with the assumption that $k < j$ without loss of generality.

First observe that by the induction hypothesis, it can be shown that with the addition of the $m+1^{st}$ path expression, the relative order of the previous path expression indices do not change.

Therefore, if we parenthesize f_2 by $s_1s_2...s_{k-1}s_k$ starting from the $k+1^{st}$ term, we observe that the induction hypothesis stating that aforementioned sort order minimizes the objective function for $m+1-k \leq m$ path expressions, is violated. □

The strong assumption underlying this approach is that a path expression is an indivisible unit of processing. However as shown in [Bla 93] by converting implicit joins to explicit joins, wider range of join sequences can be obtained and thus better (i.e. less costly) execution plans can be produced. It is clear that when the implicit joins are converted to explicit joins, because of the path expressions bound to the same bind variable, the query becomes a hybrid query, i.e., star of linear queries. Thus when we allow implicit joins to be converted to explicit joins, by using an exhaustive search strategy it is possible to obtain the optimum execution plan. However the number of join sequences to be considered is exponential in the number of classes involved. This observation indicates that heuristic is necessary to improve the performance of object-oriented queries involving path expressions.

4 A Heuristic based Approach for Object-Oriented Queries Involving Path Expressions

In this section we propose a heuristic based method for optimizing object-oriented queries involving path expressions. In this method, we first order the path expressions by using Theorem 1. Procedure 4.1 which implements Theorem 1 decides on the execution order of the path expressions. In Procedure 4.1 the cost of executing the path expression is taken as its forward traversal cost since we are using this cost as a hint to order the path expressions. Then for the chosen path, heuristic is used again as given in Procedure 4.2, to decide on the execution order of the joins within this path expression.

The heuristic we propose in Procedure 4.2 is to favor the less costly and more selective join at each iteration. Notice that the cost and the selectivity of a join operation directly effects the join order but their effect on the order varies depending upon their values. Therefore, we have tried a number of evaluation functions that all favor less cost and more selectivity but the effect of cost and selectivity on the evaluation function is different in each of them.

Before proceeding further we will provide some definitions to be used in the Procedures 4.1 and 4.2 :

Definition 4.1 Size Selectivity: The size selectivity of a join operation, C=A⋈B where A and B are two classes, is denoted by σ_{size}(A,B), and defined as

$$\sigma_{size}(A,B) = nbpages(C)/(nbpages(A) + nbpages(B))$$

where nbpages(C) is the estimated number of pages of the class produced as a result. □

Definition 4.2 Per-Unit Cost: The per-unit cost of a join operation, C = A⋈B where A and B are two classes and J_{cost} is the minimum of the cost of performing this join operation with different join implementation techniques, is denoted by P_{cost}(A,B), and defined as

$$P_{cost}(A,B) = J_{cost}(A,B)/(|A| + |B|).\square$$

Definition 4.3. Evaluation Function: In defining the evaluation function, we make the following observation: the cost and the selectivity of each join operation in a join sequence directly effect the join order. As an example consider A⋈B⋈C with costs J_{cost}(A,B), J_{cost}(B,C) and selectivities σ_{size}(A,B), σ_{size}(B,C) and assume $J_{cost}(A,B) > J_{cost}(B,C)$ and $\sigma_{size}(A,B) < \sigma_{size}(B,C)$. Here if we only consider cost we will execute B⋈C first, but since $\sigma_{size}(A,B) < \sigma_{size}(B,C)$, executing A⋈(The resulting relation) may be more costly depending upon the cost and selectivity values. Therefore, less costly and more selective join must be favored at the same time. Again depending upon the cost and selectivity values it may be beneficial to increase the effect of cost or selectivity on the join order. With these observations and with some experimentation we have defined four evaluation functions. In each of these functions low cost and high selectivity are favored however from Ψ^1(A,B) to Ψ^4(A,B), the effect of cost in

the evaluation function is reduced while the effect of the selectivity is amplified.

Ψ^1(A,B) = J_{cost}(A,B)/(1- σ_{size}(A,B))
Ψ^2(A,B)= J_{cost}(A,B)*σ_{size}(A,B)
Ψ^3(A,B)= P_{cost}(A,B)*σ_{size}(A,B)
Ψ^4(A,B)= ln J_{cost}(A,B)*$e^{\sigma_{size}(A,B)}$ □

In the following we present the algorithms implementing our heuristics.

Procedure 4.1 The Evaluation Order of Path Expressions

```
double orderPathExp( List ListofPathExpressions ) {
    double totalCost = 0;
    int k=|C_p|;
    PathExpression p,p';
    while( ListofPathExpressions is not empty ) {
        for each p in ListofPathExpressions {
            Calculate the forward traversal cost F_p for each path expression;
            Calculate the selectivity of the each path expression , σ^p_path ;
            M_p=F_p / ( 1- σ^p_path ); }
        p'= min(M_p) where p ∈ ListofPathExpressions;
        totalCost=totalCost+OrderImplicitJoins(p', schemaInfo);
        k=Cardinality from the schemaInfo;
        remove p' from ListofPathExpressions ; }
    return totalCost; }
```

Procedure 4.2 Implicit Join Ordering

Let us assume that there is a path expression $p.a_1.a_2...a_n$ where p is bound to C_0 and a_i references to the instances of the class C_i ($1 \leq i \leq n-1$). $J_{cost}(C_i, C_j)$ and $\sigma_{size}(C_i, C_j,)$denote the individual cost and selectivity of the temporary collection obtained by joining class C_i and class C_j.

```
double OrderImplicitJoins (List PathExpression, structure schemaInfo) {
// the list PathExpression contains the classes  C_0, C_1,,..., C_{n-1}
    List tempPE;
    double totalCost;
    for t=1 to 4; // Ψ^t denotes the Evaluation function in use {
        tempPE=PathExpression;
        totalCost^t=0;
        for each <C_i, C_{i+1}> in tempPE do {
            calculate J_cost(C_i, C_{i+1}) , σ_size (C_i, C_{i+1}), and Ψ^t(C_i, C_{i+1}) ;
            // In evaluating Ψ^t(C_i, C_{i+1}), J_cost(C_i, C_{i+1}) is the minimum of
                the costs of applicable
            // join techniques given in the Appendix. }
        while( tempPE is not empty) do {
            select C_k= <C_i, C_{i+1}> which gives the minimum value
                for Ψ^t(C_i, C_{i+1});
            Generate schemaInfo for C_k;
```

totalCost^t = totalCost^t + Ψ^t(C$_i$, C$_{i+1}$);
Delete ith and i+1st items from tempPE;
Compute J$_{cost}$(C$_{i-1}$, C$_k$), σ_{size}(C$_{i-1}$, C$_k$), J$_{cost}$(C$_k$, C$_{i+2}$),
σ_{size}(C$_k$, C$_{i+2}$), Ψ^t(C$_{i-1}$, C$_k$),
and Ψ^t(C$_k$, C$_{i+2}$);
Insert C$_k$ after C$_{i-1}$ to the list tempPE; } }
totalCost=min$_t$ (totalCost^t) ;
schemaInfo=schemaInfo for C$_k$
return totalCost; }

It should be noted that when a temporary collection C$_{ij}$ is obtained by joining class C$_i$ and class C$_j$, the references from class C$_{i-1}$ can not be used to reach the objects in C$_{ij}$. For such cases, only explicit join techniques can be used.

The time complexity of this algorithm is O(n^2).

5 Performance Evaluation

Two optimizers are developed for optimizing the queries involving path expressions. The first optimizer uses the heuristics described in Section 4. The second optimizer is generated through Volcano Query Optimizer Generator [McK 93], [Gra 93]. The Volcano generated optimizers produce the optimum execution plan when the transformation rules and support functions are provided properly because of its exhaustive search strategy. In this implementation the transformation and implementation rules given in [Bla 93] are used. However, since we are considering only the join and path expression optimizations, in other words, join enumeration, some of the transformation rules given in [Bla 93] are not necessary and therefore they are disabled. The transformation rules used are:

1. The rule implementing the join commutativity.

2. The rule implementing join associativity. Join associativity together with the join commutativity, provides for all possible join sequences.

3. The rule that converts a materialize node into joins. Notice that materialize operator indicates a path expression of length one. By converting a materialize operator into join it becomes possible to apply the transformation rules on join associativity and on join commutativity.

4. The rule that interchanges two successive materialize nodes. The application of this transformation rule may result in other transformations.

With these rules the VOG generated optimizer finds the optimum join ordering for the queries involving path expressions.

5.1 Testbed

Both of the optimizers are run on a Sun Sparc 2 station which has Sun 4/40 CPU, 12 MB of memory and 32 MB swap space. Each of the optimizers were the only active process during the experiments.

A random query generator is used to generate the queries with m path expressions of length n where both m and n ranges between 1 and 9. With our hardware, 12 MB memory, the optimizer generated through Volcano can not run queries when m exceeds 9, or when m*n exceeded 12. Note that, McKenna, was able to go up to 12 joins for star queries with 32 MB of main memory [McK 93]. The reason for this behavior is that the Volcano Optimizer Generator's search engine is highly recursive, and therefore as the number of equivalence classes in the query increases, optimizer rapidly exhausts the memory. The next version of VOG will heuristically reclaim memory to overcome this problem [McK 94].

For each n, m pair, 50 random queries are generated and the average values for optimization time and execution costs are obtained. The size of the classes involved ranged between 1000 and 100,000 objects where object sizes ranges from 100 to 2000 bytes. Exactly the same queries are run on both of the optimizers.

In the cost calculations, the available buffer space for executing the queries is assumed to be 4MB. Furthermore, we have assumed that the results of the join operations are written back to disk. The queries generated do not contain select operator.

The results of the some of the test runs are tabulated in Table 5.1.

		Exhaustive	Exhaustive	Heuristics	Heuristics
Path Length (n)	No of Paths (m)	Optimization time(secs.)	Execution time(secs)	Optimization time(secs.)	Execution time(secs.)
1	1	0.1854	78.9585	0.053	78.9585
1	3	1.203	197.705	0.138	200.806
1	5	11.261	175.962	0.298	176.65
1	9	1881.438	222.342	0.739	226.257
3	1	0.911	168.517	0.098	168.565
3	3	74.124	204.05	0.349	218.255
4	3	385.958	154.577	0.475	171.262
5	1	2.748	420.894	0.149	421.29
6	2	98.233	178.341	0.439	185.088
7	1	7.198	277.837	0.217	277.995
8	1	11.71	435.722	0.261	436.019
9	1	17.082	260.921	0.293	263.913

Table 5.1 Results of some of the test runs

5.2 Query Optimization Time

The Figures 5.1 and 5.2 depict the query optimization times of the optimizers for linear and star queries respectively. From these figures, it is clear that the heuristic based optimizer greatly reduces the optimization time. The explanation for this behavior is two fold:

1. The number of joins enumerated by the heuristic based optimizer for a path expression of length N is 3*(N-1). In procedure 4.2 we first generate N-1

joins, choose one and for the remaining N-1 joins, generate 2 more joins. However, exhaustive optimizer generates (N^3-N)/6 joins for linear queries. Heuristic based optimizer order the path expressions by using Procedure 4.1 and for m path expressions, forward traversal cost is calculated m^2 times. Yet exhaustive optimizer, by converting implicit joins into explicit joins, creates star queries and generates (N-1)2^{N-2} feasible joins.

2. In the heuristic based optimizer, the data structures and the algorithm itself are very simple. Therefore it spends less time in optimization. This fact is clear from the comparison of the optimization times of the two approach for 1 join as depicted in Table 5.1.

Figure 5.1 Query opt. time of linear queries

Figure 5.2 Query opt. time of star queries

Figure 5.3 Total time for linear queries

Figure 5.4 Total time for star queries

Figure 5.5 Time error for linear queries

Figure 5.6 Time error for star queries

5.3 Execution Time and Total Time

The costs are estimated by using the cost model presented in Section 2 after obtaining the query execution plans from both of the optimizers. When we consider the total time, that is, the query optimization time plus the query execution time, heuristic based optimizer outperforms the exhaustive optimizer when number of paths exceed 7 as shown in Figure 5.4. For linear queries, heuristic based optimizer is slightly better than the exhaustive optimizer as shown in Figure 5.3. These results indicate that the heuristic used is an effective one.

In Figure 5.5 and Figure 5.6 we have plotted the execution time error and total time error for linear and star queries respectively, which are defined as follows:

execution time error = execution time of heuristic based optimizer- execution time of exhaustive optimizer/ execution time of exhaustive optimizer

total time error = total time of heuristic based optimizer- total time of exhaustive optimizer/ total time of exhaustive optimizer

From Figure 5.5, it is clear that for linear queries, the plans produced by the heuristic based optimizer deviates from the optimal plans by 1 percent for path length 9, but when the total time is considered for the same path length, the heuristic based optimizer performs 5% better. When star queries are considered, the performance gain in total time is a drastic 90% as shown in Figure 5.6, although slightly worse plans are produced by the heuristic based optimizer.

6 Conclusions and Future Work

Because of the exponential nature of the query optimization for star queries, many existing relational optimizers use heuristics within dynamic programming

to limit the join sequences evaluated. A path expression in an object-oriented query language is a linear query but because of the path expressions bound to the same bind variable, the query becomes a hybrid query when the implicit joins are converted to explicit joins.

Two optimizers are developed to compare the performance of the heuristic based approach suggested in this paper with the performance of an optimizer based on exhaustive optimization. The exhaustive optimizer is generated through Volcano Optimizer Generator. The results of the experiments indicate that the heuristic optimizer greatly reduces the optimization time. The estimated query execution time of the exhaustive optimizer is slightly better. When it comes to total time, the heuristic optimizer has a superior performance with the increasing number of paths. This result is expected because the time spend in query optimization phase by the exhaustive optimizer is uncomparably larger then the execution time. The heuristic optimizer also performs well for linear queries implying that the heuristic suggested is an effective one.

As a future work we plan to generalize the heuristic suggested in this paper to relational systems to involve explicit joins and different bind variables and also to compare its performance with 2 Phase-Optimization technique given in [Ioa 90].

References

[Atk 92] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, D., Maier, D., Zdonik, S., "The Object-Oriented Database System Manifesto", in Building an Object-Oriented Database System, Morgan-Kaufmann, 1992.

[Bla 93] Blakeley, J. A., McKenna, W. J., Graefe, G., "Experiences Building the Open OODB Query Optimizer", Proc. of the ACM SIGMOD Conf., 1993.

[Cer 85] Ceri, S., Pelagatti, G., Distributed Database systems, McGraw Hill, 1985

[Gra 93] Greafe, G., McKenna, J. W., "The Volcano Optimizer Generator:Extensibility and Efficient Search" , Proc. IEEE Conf. on Data Eng., Vienna Austria,1993.

[Ioa 90] Ioannidis, Y., Kang, Y., "Randomized Algorithms for Optimizing Large Join Queries", Proc. of the ACM SIGMOD Conf., 1990.

[Loh 85] Lohman, G.M. et. al., "Query Processing in R*", Query Processing in Database Systems, Kim, Batory, Reiner, eds. Springer-Verlag, 1985.

[McK 93] McKenna, W.J., "Efficient Search in Extensible Database Query Optimization: The Volcano Optimizer Generator", Ph. D. Thesis, Univ. of Colorado, 1993.

[McK 94] McKenna, W.J., Personal communication, 1994.

[Ono 90] Ono, K., Lohman, G. M., "Measuring the Complexity of Join Enumeration in Query Optimization", Proc. of Intl. Conf. on Very Large Databases, 1990.

[Ozk 90] Ozkarahan E., "Database Management Concepts, Design and Practice", Prentice-Hall, 1990.

[Sel 79] Sellinger, P.G.,"Access Path Selection in a Relational Database Management System", Proc. of the ACM SIGMOD Conf., 1979.

This article was processed using the LaTeX macro package with LLNCS style