Proc. of 14th Int'l Conf. on Database and Expert Applications (DEXA'03)
LNCS, Springer Verlag, Vol. 2736, pp 747-759, Sept 2003

Exploiting Similarity of Subqueries for Complex Query Optimization*

Yingying Tao[1], Qiang Zhu[1], and Calisto Zuzarte[2]

[1] Department of Computer and Information Science
The University of Michigan, Dearborn, MI 48128, USA.
[2] IBM Toronto Laboratory **
Markham, Ontario, Canada L6G 1C7

Abstract. Query optimizers in current database management systems (DBMS) often face problems such as intolerably long optimization time and/or poor optimization results when optimizing complex queries. To tackle this challenge, we present a new similarity-based optimization technique in this paper. The key idea is to identify groups of similar subqueries that often appear in a complex query and share the optimization result within each group in the query. An efficient algorithm to identify similar queries in a given query and optimize the query based on similarity is presented. Our experimental results demonstrate that the proposed technique is quite promising in optimizing complex queries in a DBMS.

1 Introduction

Query optimization is vital to the performance of a database management system (DBMS). The main task of a query optimizer in a DBMS is to seek an efficient query execution plan (QEP) for a given user query. Extensive study on query optimization has been conducted in the last three decades [2, 3, 6]. However, as database technology is applied to more and more application areas, user queries become increasingly complex in terms of the number of operations (e.g., more than 50 joins) and the query structure (e.g., star-schema queries with snowflakes). The query optimization techniques adopted in the existing DBMSs cannot cope with the new challenges.

As we know, the most important operation for a query is the join operation. A query typically involves a sequence of joins. To determine a good join order for the query, which is an important decision in a QEP, two types of algorithms are adopted in the current DBMSs. The first type of algorithms is based on dynamic programming or other exhaustive search techniques. Although such an algorithm can guarantee to find an optimal solution, its worst-case time complexity is exponential. The second type of algorithms is based on heuristics. Although such

* Research supported by the IBM Toronto Lab and The University of Michigan.
** IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

algorithms have a polynomial time complexity, they often yield a suboptimal solution.

Hence, the above techniques cannot handle complex queries with a large number of joins, which occur more and more often in the real world. For an algorithm with exponential complexity, it may take months or years to optimize such a complex query. On the other hand, although a heuristic-based algorithm takes less time to find a join order for a complex query, the efficiency difference between a good solution and a bad one can be tremendous for a complex query. Unfortunately, the heuristics employed by current systems do not take the characteristics of a complex query into consideration, which often leads to a bad solution.

Therefore, we need a technique with a polynomial time complexity to find an efficient plan for a complex query. Note that the general query optimization problem has been proven to be NP-complete [4]. Hence, it is generally impossible to find an optimal plan within a polynomial time. Several studies have been reported to find a good plan for a large query (i.e., involving many joins) within a polynomial time in the literature, including the iterative improvement (II), simulated annealing (SA) [5], Tabu Search (TS) [7], AB algorithm (AB) [9], and genetic algorithm (GA) [1]. These algorithms represent a compromise between the time the optimizer takes for optimizing a query and the quality of the optimization plan. However, these techniques are mostly based on the randomization method. One advantage of this approach is that it is applicable to general queries, no matter simple or complex. On the other hand, a method only based on randomization has little "intelligence". It does not make use of some special characteristics of the underlying queries.

In our recent work [11], we introduced a technique to optimize a complex query by exploiting common subqueries that often appear in such a query. This technique is shown to be more effective than a pure randomization based method since it takes the structural characteristics of a complex query into consideration when optimizing the query. However, since this technique requires existence of common subqueries in a query, its application is limited.

We notice that many complex queries often contain similar substructures (subqueries) although they may not be exactly the same (i.e., common). This phenomenon appears more often when predefined views are used in the query, the underlying database system is distributed, or some parts of the query are automatically generated by programs. The more complex the query is, the more similar subqueries there will possibly be. As an example, consider a UNION ALL view defined as a fact table split up by years. Assume that dimension tables are to be joined with the view. To avoid materializing the view, a DBMS usually pushes down the dimension tables into the view, resulting in similar UNION ALL branches (subqueries).

Based on the above observation, we introduce a new similarity-based technique to reduce optimization time for a complex query by exploiting its similar subqueries in this paper. It is a two-phase optimization procedure. In the first phase, it identifies groups of similar subqueries in a given query. For each group

of subqueries, it performs optimization for one representative subquery and share the optimization result with other member(s) within the group. After each similar subquery is replaced by its (estimated) result table obtained by using the corresponding execution plan in the original query, the modified query is then optimized in the second phase. Since the complexities of similar subqueries and the modified final query are reduced, they have a better chance to be optimized by a conventional approach (e.g., dynamic programming) or a randomization approach (e.g., AB) efficiently and effectively.

The rest of this paper is organized as follows. Section 2 introduces the concepts of a query graph and its similar subqueries. Section 3 gives the details of an algorithm to identify similar subqueries in a given query and optimize the query based on similarity. Section 4 shows some experimental results. Section 5 summarizes the conclusions.

2 Query Graph and Similar Subquery

In this section, we introduce the definitions of a query graph and its similar subquery graphs, which are the key concepts for our query optimization technique.

2.1 Query Set Considered

Most practical queries consist of a set of selection (σ), join ($\bowtie$), and projection (π) operations. These are the most common operations studied for query optimization in the literature, which are also the ones to be considered in this paper. Note that π is usually processed together with other operations (σ or $\bowtie$) it follows in a pipelined fashion. For simplicity, we assume that there is always a projection operation following each $\sigma/\bowtie$ operation to filter out the columns that are not needed for processing the rest of the given query. We also assume that the query condition is a conjunction of simple predicates of the following forms: $R.a\ \theta\ C$ and $R_1.a_1\ \theta\ R_2.a_2$, where R, R_1 and R_2 are tables (relations); a, a_1 and a_2 are columns (attributes) in the relevant tables; C is a constant in the domain of $R.a$; and $\theta \in \{=, \neq, <, \leq, >, \geq\}$. Another assumption is that joins are executed by the nested-loop method. Under this assumption, we can simply consider one cost model. However, our technique can be extended to include other join methods by adopting multiple cost models.

2.2 Query Graph

Let Q be a query, $T = \{R_1, R_2, ..., R_m\}$ be the set of (base) tables referenced in Q, and $P = \{p_1, p_2, ..., p_n\}$ be the set of predicates referenced in Q. We call each table reference in Q a table instance. The logical structure of query Q can be represented by a query graph, based on the following rules:

- Each table instance in Q is represented by a vertex in the query graph G for Q. Let V be the set of vertices in G. Since several table instances may reference the same base table, there exists a many-to-one mapping $\delta : x \mapsto R$, where $x \in V$ and $R \in T$. In G, each $x \in V$ is labeled with $\delta(x) = R$.

– For any table instances (vertices) x and y, if there is at least one predicate in P involving x and y, then query graph G has an edge between x and y, with the set of all predicates involving x and y labeled on the edge. If x and y are the same table instance, the edge is a self-loop for the vertex in query graph G. Let E be the set of all edges in G. There exists a mapping $\varphi : e \mapsto c$, where $e \in E$ and $c \in 2^P$. $\varphi(e)$ gives the set of all simple predicates involving the (both) vertices of edge e. In G, each $e \in E$ is labeled with $\varphi(e) = c$.

Therefore, a query graph for query Q is a 6-tuple $G(V, E, T, P, \delta, \varphi)$, with each component defined as above. For the rest of the paper, we use the following notation: $edge(x, y)$ denotes the edge between vertices x and y in a query graph, $vertices(e)$ denotes the set of (one or two) vertices connected by edge e in a query graph. $sizeof(x)$ denotes the size of the table represented by x, $sel(e)$ denotes the selectivity of $\varphi(e)$, i.e., the selectivity of the conjunction of all the simple predicates in $\varphi(e)$.

Without loss of generality, we assume our query graph is connected in this paper. Otherwise, each isolated component graph can be optimized first and a set of Cartesian products are then performed to combine the results of the component graphs.

For our query optimization technique, we adopt a data structure called the ring network representation to represent a query graph. In such a representation, every vertex x in query graph G has a node x. Assume that node x has a set of adjacent nodes $y_1, y_2, ..., y_n$. We use a closed link list (i.e., a ring): $x \rightarrow y_1 \rightarrow y_2 \rightarrow ... \rightarrow y_n \rightarrow x$ to represent such an adjacency relationship. Node x is called the owner (node) of the ring, while nodes $y_1, y_2, ..., y_n$ are called the members of the ring. Each node in the ring network for G is the owner of one ring, and it also participates in other rings as a member.

2.3 Similar Subquery Graphs

Let $V' \subseteq V$ be a subset of vertices in the query graph $G(V, E, T, P, \delta, \varphi)$ for query Q. Let $E|_{V'} = \{\ e \mid e \in E \text{ and } vertices(e) \subseteq V'\ \}$; $T|_{V'} = \{\ R \mid R \in T \text{ and there exists } x \in V' \text{such that } x \text{ is an instance of } R\ \}$; $P|_{V'} = \{\ p \mid p \in P \text{ and } p \in w \text{ and } w \text{ is the set of predicates labeled on } e \in E|_{V'}\ \}$; $\delta|_{V'} : x \mapsto R$, *where* $x \in V'$, $R \in T|_{V'}$ *and* $\delta(x) = R$; and $\varphi|_{V'} : e \mapsto c$, *where* $e \in E|_{V'}$, $c \in 2^{P|_{V'}}$ *and* $\varphi(e) = c$. Clearly, $G'(V', E|_{V'}, T|_{V'}, P|_{V'}, \delta|_{V'}, \varphi|_{V'})$ is a subgraph of G. If G' is connected, we call it a subquery graph in G. The corresponding query is called a subquery of Q.

Let $G'(V', E|_{V'}, T|_{V'}, P|_{V'}, \delta|_{V'}, \varphi|_{V'})$ and $G''(V'',\ E|_{V''},\ T|_{V''},\ P|_{V''},\ \delta|_{V''},\ \varphi|_{V''})$ be two subquery graphs in G. r_t and r_s are two given error bounds for table sizes and condition selectivities, respectively. Assume that every table size and selectivity are non-zero. Otherwise, the result of the entire query will be empty, in which case no optimization is needed. If G' and G'' satisfy the following conditions, we regard them as a pair of similar subquery graphs with respect to error bounds r_t and r_s, denoted as $G' \approx |_{(r_t, r_s)} G''$:

– There exists a one-to-one mapping f between V' and V'', such that for any $x \in V'$ and $f(x) \in V''$, we have $\varepsilon_t(x, f(x)) < r_t$, where $\varepsilon_t(x, f(x)) = max\{\frac{|sizeof(x)-sizeof(f(x))|}{sizeof(x)}, \frac{|sizeof(x)-sizeof(f(x))|}{sizeof(f(x))}\}$.
– There exists a one-to-one mapping g between E' and E'', such that, for any $e \in E'$ and $g(e) \in E''$, if $vertices(e) = \{x, y\}$, then $vertices(g(e)) = \{f(x), f(y)\}$, and $\varepsilon_s(e, g(e)) < r_s$, where $\varepsilon_s(e, g(e)) = max\{\frac{|sel(e)-sel(g(e))|}{sel(e)}, \frac{|sel(e)-sel(g(e))|}{sel(g(e))}\}$.

Let us consider an example. Given a query graph G in Fig. 1 (the size of the table represented by each node and the selectivity of the predicates on each edge are marked in parentheses), using error bounds $r_t = r_s = 0.3$, the similar subqueries are shown in the dotted circles marked as G' and G''. Intuitively, a pair of similar subquery graphs have the same inner graph structure, and the table sizes for the corresponding vertices and the selectivities for the corresponding edges in the pair are within error bounds r_t and r_s, respectively.

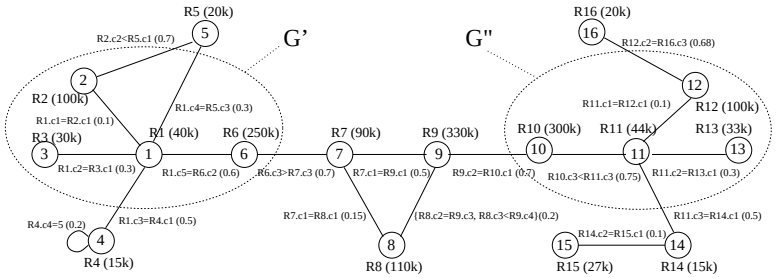

Fig. 1. Example of a query graph with similar subqueries

3 Query Optimization Via Exploiting Similarity of Subqueries

In this section, we will introduce a technique to perform query optimization via exploiting similarity of subqueries in a given complex query. Due to the paper length limitation, we only discuss the technique making use of pairs of similar queries. However, the technique can be generalized to utilize more than two similar subqueries in each group (with one representative subquery)[12].

3.1 Basic Idea

The basic idea of the technique is as follows: (1) identify pairs of similar subqueries with respect to the given error bounds r_t and r_s; (2) optimize one subquery in a similar pair, and map and apply the resulting execution plan to the other subquery, which would reduce the optimization time by sharing the same (mapped) plan between two similar subqueries; and (3) replace similar subqueries with their (estimated) result tables in the query graph and optimize the resulting modified query. As the number of tables/vertices in the query graph is reduced, less overhead is needed to optimize the modified query.

3.2 Algorithm Description

The description of the algorithm for our technique is given below. Note that if there is a self-loop on a node/vertex, it is usually a good strategy to perform such a unary subquery first to reduce the operand table size. Hence we assume that all self-loops are removed in such a way for a given query graph.

ALGORITHM 1 Query Optimization Based on Similar Subqueries
Input: query graph $G(V, E, T, P, \delta, \varphi)$ for user query Q, and error bounds r_t and r_s.
Output: Execution plan for query Q.
Method:

1. Initialize flag selected(x)=0 for each node x;
2. Initialize the sets of identified pairs of similar subquery graphs $S_{accept} = S_{hold} = \emptyset$;
3. **while** there are unselected node pairs with similar table sizes within r_t **do**
4. Pick up nodes x and y following the rules for choosing starting nodes in Sect. 3.3;
5. Let V_1={x}; V_2={y};
6. Record one-to-one mapping f : $x \mapsto y$; /* i.e., $y = f(x)$ */
7. Let selected(x)=selected(y)=1;
8. **while** V_1 and V_2 have unexpanded nodes **do**
9. Pick the next pair $x \in V_1$ and $y \in V_2$ based on FIFO order for expanding;
10. **for** each member x_1 with selected(x_1)=0 in the ring owned by x **do**
11. **for** each member y_1 with selected(y_1)=0 & $y_1 \neq x_1$ in the ring owned by y **do**
12. **if** $\varepsilon_t(x_1, y_1) < r_t$ and [$edge(x_1, x')$ exists for any $x' \in V_1$ if and only if
13. $edge(y_1, y')$ exists for $y' = f(x') \in V_2$ & $\varepsilon_s(edge(x_1, x'), edge(y_1, y')) < r_s$]
14. **then** $V_1 = V_1 \cup \{x_1\}$; $V_2 = V_2 \cup \{y_1\}$; selected(x_1)=selected(y_1)=1;
15. Record one-to-one mapping f : $x_1 \mapsto y_1$;
16. Record one-to-one mapping g : $edge(x_1, x') \mapsto edge(y_1, y')$
if such edges exist for any $x' \in V_1$ and $y' = f(x') \in V_2$
17. Break;
18. **end if**
19. **end for**
20. **end for**
21. **end while**;
22. Evaluate the resulting pair of similar subquery graphs $< G_1, G_2 >$;
23. **if** it is worth to accept $< G_1, G_2 >$
24. **then** Remove any $< G_1', G_2' >$ from S_{hold} that shares node(s) with $< G_1, G_2 >$;
25. $S_{accept} = S_{accept} \cup \{< G_1, G_2 >\}$;
26. **else if** it is worth to hold $< G_1, G_2 >$
27. **then** Reset selected(x) = 0 for all x in G_1 or G_2;
28. $S_{hold} = S_{hold} \cup \{< G_1, G_2 >\}$;
29. **else** Reset selected(x)=0 for all x in G_1 or G_2;
30. Discard $< G_1, G_2 >$;
31. **end if**;
32. **end while**;
33. $S_{accept} = S_{accept} \cup S_{hold}$;
34. **for** each similar subquery pair $< G_1, G_2 > \in S_{accept}$ **do**
35. Optimize G_1 and share the execution plan with G_2;
/* after an appropriate mapping using f and g */
36. Replace G_1 and G_2 in original G with their (estimated) result tables;
37. **end for**;

38. Optimize the modified G to generate an execution plan;
39. **return** the execution plan for Q, which includes the plan for the modified G and the plans for all accepted similar subqueries.

Lines 3 - 32 search for all pairs of similar subquery graphs in query Q. Lines 8 - 21 expand the current pair of similar subquery graphs using the ring network. Lines 22 - 31 determine if we should accept, hold, or reject the pair of identified subquery graphs. Line 33 accepts the remaining pairs of held similar subquery graphs, which are not overlapped with any accepted similar subquery graphs as guaranteed by Line 24. Lines 34 - 38 optimize all similar subquery graphs and the modified final query graph. Line 43 returns the execution plan for the given query.

More details of some steps and the reasons why some decisions in the algorithm were made are discussed in the following subsection.

3.3 Detailed Explanation of the Algorithm

Choosing starting nodes

For a given query Q, when we construct its ring network, we also construct a set of similarity starting lists. Each list in the set is for one base table in the query, which has a header containing the base table name R_i and two sublists – one OL_i contains all its instances (nodes) in the query and the other SL_i contains other table instances whose sizes are within error bound r_t with respect to the size of R_i. For example, for the query graph in Fig. 2 (selectivities on the edges are omitted), using error bound parameter $r_t = 0.3$, its similarity starting lists set is shown in Fig. 3.

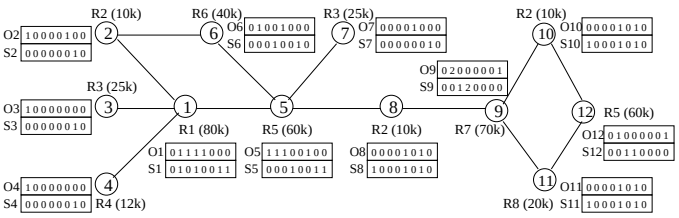

Fig. 2. Example of indicator arrays

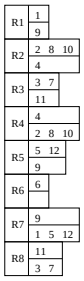

Fig. 3. Example of similarity lists set

In principle, any node in OL_i together with another node in OL_i or SL_i can be used as a pair of potential starting (similar) nodes. However, our goal is to find a pair of as large as possible similar subquery graphs. This is because the larger the similar subquery graphs in a pair, the more the optimization work can be shared.

Unfortunately, we cannot predict how the subquery graphs will grow from a pair of starting nodes. Hence, a greedy approach is adopted. We attempt to choose a pair of nodes with the maximum number of adjacent node pairs which

satisfy: (1) the adjacent nodes in each pair are unselected and different, and (2) the sizes of tables represented by the adjacent node pair are within the given error bound r_t.

To efficiently search for such a pair of starting nodes from the set of similarity starting lists, we attach two indicator arrays O (occurrence) and S (similarity) to each table instance in the query. The lengths of arrays O and S are the size of T for the given query. Let x be a table instance vertex in the given query graph, and R_i be a base table. Array element $O_x[i]$ indicates how many current adjacent nodes representing base table R_i that x has. Array element $S_x[i]$ indicates how many current adjacent nodes whose table sizes are within the given error tolerance with respect to R_i that x has, excluding $O_x[i]$. Fig. 2 shows examples of indicator arrays O and S.

Let m be the size of set T for query Q. For any pair of potential starting nodes x and y selected from a similarity starting list, we can calculate the following value:

$$matching\_pairs(x, y) = \sum_{i=1}^{m} \omega(x, y, i), \tag{1}$$

where $\omega(x, y, i)$ is defined as follows:

$$\omega(x, y, i) = \begin{cases} O_x[i] + min\{O_y[i] - O_x[i], S_x[i]\}, & O_x[i] \leq O_y[i], \\ O_y[i] + min\{O_x[i] - O_y[i], S_y[i]\}, & otherwise. \end{cases} \tag{2}$$

If we first match the adjacent table instances representing the current i-th base table for the starting pair and then match the remaining such table instances for one node (if any) in the pair to the other table instances whose sizes are within the given error bound for the other node, formula (2) gives the total number of such matchings for the current base table. Note that arrays O and S need to be updated every time when matched nodes are removed from consideration based on (2) before the next new base table is considered in (1).

Formula (1) essentially gives the total number of matching pairs of adjacent nodes for the pair of starting nodes x and y in a matching way described above. Our heuristic for choosing a pair of starting nodes is to choose the pair that maximizes the value of formula (1). If there is a tie, we pick up a pair with the smallest difference of table sizes. If there is still a tie, a random pair is chosen.

Searching for similar subquery graphs

In Algorithm 1, we use the breadth-first search strategy to expand a pair of similar subquery graphs G_1 and G_2. That is, all adjacent nodes for the current pair of expanding nodes x and y are considered before a new pair of nodes is selected in the first-in first-out (FIFO) fashion for further expansion. This procedure is repeated until no pair of nodes can be expanded.

We use a nested-loop method to check all the unselected nodes (x_1 and y_1) in the two rings owned by the current pair of expanding nodes (x and y). If the table sizes of this new pair of nodes x_1 and y_1 are within error bound r_t, then we check all nodes that are already in similar subquery pair G_1 and G_2 to see if adding x_1 and y_1 into the current subquery pair will violate the similarity of subqueries or not.

There are two cases in which the similarity of G_1 and G_2 may be violated if we add x_1 and y_1 into them: (i) there exists a pair of $x' \in V_1$ and $y' = f(x') \in V_2$ such that $edge(x_1, x')$ exists, but $edge(y_1, y')$ does not, or vice versa; and (ii) there exists a pair of $x' \in V_1$ and $y' = f(x') \in V_2$ such that there exist both $edge(x_1, x')$ and $edge(y_1, y')$, but $sel(edge(x_1, x'))$ and $sel(edge(y_1, y'))$ are not within error bound r_s.

Selecting similar subquery graphs

If the similar subquery graphs in an identified pair are very small (in terms of the number of nodes in each graph), e.g., containing only 2 nodes, not much optimization work can be shared between them. Furthermore, if we use them in optimization, their nodes cannot participate in other possibly larger similar subquery graphs. In such a case, it is better not to use them (Lines 29-30). If the pair is worth to keep, we remove all nodes in the subquery graph pair from the original query graph by setting their "selected" flag to 1.

If the sizes of similar subquery graphs in a pair are marginal, it is uncertain if it is worth to use it in optimization. In this case, we hold this pair, but allow other similar subquery graphs to use their nodes. If we can find a pair of larger similar subquery graphs using some of the nodes, we adopt the new similar subqueries and discard the held ones. Otherwise, we will still use the held similar subquery graphs (Line 33).

To determine whether to accept, reject, or hold a pair of similar subquery graphs, we use two threshold values c_1 and c_2, where $c_1 < c_2$. Let n be the number of nodes in a similar subquery graph (note that G_1 and G_2 are of the same size). The following rules are used to decide the fate of a pair of similar subquery graphs: (1) if $n \geq c_2$, then the new pair of similar subquery graphs is accepted; (2) if $c_1 \leq n < c_2$, then we put this pair on hold; and (3) if $n < c_1$, then the new pair is rejected.

Optimizing query

After all similar subquery pairs are selected, we can apply an optimization algorithm, such as AB or II, to optimize one of the subquery graphs in each pair. As all corresponding relationships of the nodes in each pair of similar subqueries are recorded during searching them, we can map the execution plan for one subquery to the one for its partner. For example, consider a pair of similar subqueries G_1 and G_2 that have the following corresponding relationship for the nodes: $x_1 \leftrightarrow y_1, x_2 \leftrightarrow y_2, x_3 \leftrightarrow y_3$, where x_1, x_2, x_3 are in G_1, and y_1, y_2, y_3 are in G_2. By optimizing G_1, assume that we get such a plan: $((x_1 \bowtie x_2) \bowtie x_3)$. The mapped plan for G_2 is $((y_1 \bowtie y_2) \bowtie y_3)$.

By replacing all accepted similar subqueries with their result tables in the original query graph, we reduce the number of nodes in it. Since the complexities of the similar subqueries and the modified final query are less than that of the original query, many existing query optimization techniques (such as AB and II) usually perform well for them. If a similar subquery or the modified final query is sufficiently small (e.g., less than a chosen small constant), a dynamic-

programming-based optimization technique could also be used to find a truly optimal plan for it.

It is not difficult to see that the time complexity of the algorithm is $O(max\{n^4, T(n)\})$, where n is the number of tables instances (vertices) in the query graph for input query Q, and $T(n)$ is the complexity of the optimization technique used to optimize the similar subqueries and the modified final query. As mentioned before, unless n is less than a small constant, we employ a polynomial-time technique such as AB or II with our algorithm. Therefore, our technique is of polynomial time complexity.

4 Experiments

As mentioned above, our technique is an efficient polynomial-time technique suitable for optimizing complex queries. Although our technique may not be more efficient than a randomization method or a heuristic-based technique in the worst case, it is expected that our technique usually generates a better execution plan for a complex query since it takes some complex query characteristics into account. To examine the quality of the execution plans generated by our technique, we have conducted some experiments.

In the experiments, we chose to compare our technique with the most promising randomization technique, i.e., the AB algorithm, for optimizing complex queries. To make a fair comparison, we employ the same AB algorithm with our technique for optimizing similar subqueries and the modified final query (i.e., Lines 35 and 38 in Algorithm 1). It is assumed that the nested-loop join method is used to perform joins in a query for all the techniques.

The experiments were conducted on a SUN UltraSparc 2 workstation running Sun OS 5.1. All techniques were implemented in C. Test queries and their operand tables in the experiments were randomly generated.

Let n be the number of nodes in a query graph. In the experiments, the following threshold values were used to accept, hold, or reject a pair of identified similar subquery graphs: (1) If $n < 25$, we set $c_1 = c_2 = 3$. That is, similar subquery graphs with $>= 3$ nodes are accepted; otherwise, they are rejected. (2) If $n >= 25$, we set $c_1 = 3$ and $c_2 = 4$. That is, similar subquery graphs with $>= 4$ nodes are accepted; similar subquery graphs with < 3 nodes are rejected; and similar subquery graphs with 3 nodes are put on the hold list.

We focused on comparing the I/O costs (i.e., the number of I/O's) when queries were performed by using the execution plans generated from different techniques. Table 1 shows such a comparison for a set of randomly-generated test queries. The error bounds used for the similarity-based technique in the experiments are $r_s = r_t = 0.3$.

From the experimental results, we can see that the performance of our similarity-based technique is better than that of the AB algorithm. In fact, it reduces the query costs (on average) by 72.1%. Although AB can improve its results by taking longer time to try more random plans, our experiments showed that with the same number of tries our technique is always superior to the AB

Table 1. Comparison of I/O costs for execution plans generated by two techniques

Q#	# of table instances in Q	$I/O\#$ by AB algo	$I/O\#$ by similarity tech	improve %
1	29	0.1161e+37	0.4730e+36	59.3%
2	21	0.7514e+25	0.3615e+25	51.9%
3	28	0.9714e+38	0.2313e+38	76.2%
4	34	0.8102e+42	0.7046e+41	91.3%
5	24	0.1649e+30	0.5320e+29	67.7%
6	26	0.1107e+31	0.6304e+30	43.1%
7	15	0.1308e+18	0.2218e+17	83.0%
8	23	0.5953e+27	0.1726e+27	71.0%
9	37	0.8535e+47	0.6774e+46	92.1%
10	14	0.7197e+17	0.1089e+17	84.9%

algorithm. This observation verifies that making use of the characteristics of a complex query can improve the quality of its execution plan.

To examine the effect of error bounds on the performance of our technique, we conducted an experiment, in which a typical test query (i.e., query # 9 in Table 1) was used, and the error bounds r_t and r_s were changed from 0.1 to 1. The result is shown in Fig. 4. From the figure, we can get the following

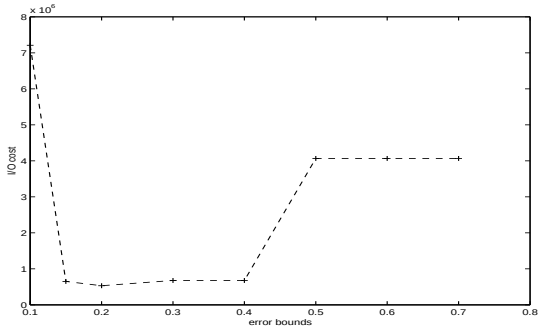

Fig. 4. Effect of changing error bounds on I/O costs

observations: (1) very small error bounds cannot yield good performance, since the smaller the error bounds, the smaller the similar subqueries; (2) moderate error bounds ($0.15 \sim 0.40$) yield the best performance, since similar subqueries with reasonable sizes can be found; and (3) large error bounds lead to poor performance, since subqueries are less similar in such cases, which makes that sharing execution plans between them may not be appropriate; and (4) when the error bounds are very large (close to 1), the performance of the similarity-based technique stays at the same level. The reason for (4) is that similar subqueries must have the same query structure, and the sizes of similar subqueries may reach their maximums when the error bounds are beyond a certain limit.

In summary, the similarity-based technique with moderate error bounds can significantly improve the performance of complex queries.

5 Conclusions

As database technology is applied to more and more application domains, user queries become increasingly complex. Query optimization for such queries be-

comes very challenging. Existing query optimization techniques either take too much time (e.g., dynamic programming) or yield a poor execution plan (e.g., simple heuristics). Although some randomization techniques (e.g., AB, II, and SA) can deal with this problem to a certain degree, the quality of the execution plan generated for a given query is still unsatisfactory since these techniques do not take the special properties of a complex query into consideration.

In this paper, we propose a new technique for optimizing complex queries by exploiting their similar subqueries. The key idea is to identify pairs (groups) of similar subqueries for a given query, optimize one subquery in each pair, share the optimization result with the other similar subquery, reduce the original query by replacing each similar subquery with its result, and then optimize the modified final query. An efficient technique such as the AB algorithm can be used together with our technique for optimizing identified similar subqueries as well as the modified query. It has been shown that our technique is efficient and can usually generate good execution plans for complex queries since it takes the structural characteristics of a query into account and divides a large query into small parts. Our experimental results demonstrate that the proposed technique is quite promising in optimizing complex queries.

References

1. Bennett, K., Ferris, M.C., Ioannidis, Y. A genetic algorithm for database query optimization. In *Proc. of Int'l Conf. on Genetic Algorithms*, pp. 400-07, 1991.
2. Chaudhuri, S. An overview of query optimization in relational systems. In *Proc. of ACM PODS*, pp. 34-43, 1998.
3. Graefe, G. Query Evaluation Techniques for Large Databases. In *ACM Comp. Surveys*, 25(2) 111-152, 1993.
4. Ibaraki, T., Kameda, T. On the optimal nesting order for computing N-relational joins. In *ACM Trans. on DB Syst.*, 9(3) 482-502, 1984.
5. Ioannidis, Y. E., Wong, E. Query Optimization by Simulated Annealing. In *Proc. of ACM SIGMOD*, pp. 9-22, 1987.
6. Jarke, M., Koch, J. Query Optimization in Database Systems. In *ACM Comp. Surveys*, 16(2) 111-52, 1984.
7. Matysiak, M. Efficient Optimization of Large Join Queries Using Tabu Search. In *Infor. Sci.*, v83, n1-2, 1995.
8. Selinger P. G., *et al.* Access Path Selection in a Relational Database Management System. In *Proc. of ACM SIGMOD*, pp. 23-34, 1979.
9. Swami, A., Iyer, B. R. A polynomial time algorithm for optimizing join queries. In *Proc. of IEEE ICDE*, pp. 345-54, 1993.
10. Swami, A., Gupta, A. Optimization of Large Join Queries. In *Proc. of ACM SIGMOD*, pp. 8-17, 1988.
11. Tao, Y., Zhu, Q., Zuzarte, C. Exploiting Common Subqueries for Complex Query Optimization. In *Proc. of CASCON*, pp. 21-34, 2002.
12. Tao, Y., Zhu, Q., Zuzarte, C. Optimizing Complex Queries by Exploiting Similarities of Subqueries. *Technical Report CIS-TR-0301-03*, CIS Dept, U. of Michigan, Dearborn, MI 48128, USA, March 2003.