DOI 10.1007/s10115-004-0189-y
Springer-Verlag London Ltd. © 2005
Knowledge and Information Systems (2005) 8: 350–373

Knowledge and Information Systems

Optimizing complex queries based on similarities of subqueries*

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

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

Abstract. As database technology is applied to more and more application domains, user queries are becoming increasingly complex (e.g. involving a large number of joins and a complex query structure). Query optimizers in existing database management systems (DBMS) were not developed for efficiently processing such queries and often suffer from problems such as intolerably long optimization time and poor optimization results. To tackle this challenge, we present a new similarity-based approach to optimizing complex queries in this paper. The key idea is to identify similar subqueries that often appear in a complex query and share the optimization result among similar subqueries in the query. Different levels of similarity for subqueries are introduced. Efficient algorithms to identify similar queries in a given query and optimize the query based on similarity are presented. Related issues, such as choosing good starting nodes in a query graph, evaluating identified similar subqueries and analyzing algorithm complexities, are discussed. Our experimental results demonstrate that the proposed similarity-based approach is quite promising in optimizing complex queries with similar subqueries in a DBMS.

Keywords: Algorithm; Complex query; Computational complexity; Database system; Query optimization; Query similarity

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. Many query optimization techniques have been proposed. Good surveys of this area can be found in the literature (Chaudhuri 1998; Graefe 1993; Jarke and Koch 1984).

However, the database field is a rapidly growing one. As database technology is applied to more and more application domains, user queries become more and more

* Research supported by the IBM Toronto Laboratory and The University of Michigan.

Received 8 January 2004
Revised 7 May 2004
Accepted 9 September 2004
Published online 2 February 2005

complex in terms of the number of operations (e.g. more than 50 joins) and the query structure (e.g. snowflake queries). 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 specified in a QEP, two types of algorithms are adopted in the current DBMSs. The first type of algorithm 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, i.e. $O(2^n)$. The second type of algorithm is based on heuristics. Although such an algorithm has a polynomial time complexity, it often yields a suboptimal solution. In traditional database applications, queries involving more than 15 joins were considered to be unrealistic. For such queries, both types of algorithms mentioned above are acceptable to a certain degree.

However, as database applications become more and more complex and database sizes become larger and larger, queries with a large number of joins occur more and more often in the real world. Existing techniques can no longer be used to optimize such queries. For an algorithm with an exponential complexity, it may take months or years to optimize a complex (large) 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 such a 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 (Ibarake and Kameda 1984). 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) (Ioannidis and Wong 1987), Tabu Search (TS) (Matysiak 1995), AB algorithm (AB) (Swami and Iyer 1993), and genetic algorithm (GA) (Bennett et al. 1991). These algorithms represent a compromise between the time the optimizer takes to optimize a query and the quality of the optimization plan. However, these techniques are mostly based on the randomization method. One advantage of such an approach is that it is applicable to general queries, no matter how simple or complex the query is. 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 (Tao et al. 2002), we introduced a technique to exploit common subqueries that appear in a complex query to optimize the query. This technique is shown to be more effective than a pure randomization-based method because it takes the structural characteristics of a complex query into consideration when optimizing the query. However, because this technique requires existence of common subqueries in a query, its application is limited.

We noticed 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. 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 year. 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). Another more concrete real-world query example is given in the Appendix.

Based on the above observation, we introduce a new similarity-based approach to reducing optimization time for a complex query by exploiting its similar subqueries (the similarity is defined by some error bounds on query parameters) in this paper. It is a two-phase optimization procedure. In the first phase, it identifies similar subqueries in a given query. For each group of subqueries that are similar to a representative subquery, it performs optimization for the representative subquery and shares the resulting execution plan for all the subqueries in 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 revised query is then optimized in the second phase. Because the complexities of similar subqueries and the revised final query are reduced, they can be optimized using a conventional approach (e.g. dynamic programming) or a randomization approach (e.g. AB).

Although our work has some similarity with multiple-query optimization (Chesnais et al. 1983; Cosar et al. 1993; Dalvi et al. 2001; Dalvi et al. 2003; Kalnis and Papadias 2003; Krolikowski et al. 2000; Mistry et al. 2001; O'Gorman et al. 2002), there are significant differences. The purpose of multiple-query optimization is to optimize multiple user queries collectively instead of individually. Such typical techniques include caching (intermediate) results of one query for sharing with others, choosing an access method (e.g. one involving column sorting or index creation) for one query that can also benefit the others, sharing page accesses among multiple queries and parallelizing execution to maximize the system throughput. The similarity-based optimization technique introduced in this paper optimizes one individual complex (large) query, which became an issue only recently when a user query was getting very complex. Because similar subqueries used in our technique are not necessarily identical, we only share their execution plans (to reduce optimization time) rather than share their results. Although different parts of a complex query that contain similar subqueries sharing an execution plan can be logically considered as separate queries, such a (logical) split of the query is not clear until the beneficial similar subqueries are identified. In other words, we cannot split the given query into multiple queries in advance and treat them as independent multiple user queries for optimization because different splits may significantly affect optimization results. Hence, our technique searches for beneficial similar subqueries within a given query without any predefined boundaries. To our knowledge, no similar work has been reported in the literature.

The rest of this paper is organized as follows. Section 2 introduces the definitions of a query graph and its similar subquery graphs, which are the key concepts used in our technique. We then present an efficient ring network representation to implement a query graph. Section 3 gives the details of an algorithm that is the kernel for our technique. Section 4 defines similarities at higher levels and discusses the revisions needed to extend the previous algorithm. Section 5 shows some experimental results. Section 6 summarizes our 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. We also

present an efficient data structure, called the ring network representation, to represent a query graph for our 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 operations 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 $\theta \in \{=, \neq, <, \leq, >, \geq\}$; R, R_1 and R_2 are tables (relations); a, a_1 and a_2 are columns (attributes) in the relevant tables; and C is a constant in the domain of $R.a$. For example, *EMP.salary* > 20000 and *EMP.dno* $=$ *DEPT.dno* are two valid predicates. The predicates of the second form are also called join predicates. Another assumption made in our paper 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 tables referenced in Q and $P = \{p_1, p_2, ..., p_n\}$ be the set of all 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 (node) in the query graph G for Q. Let V be the set of vertices in G. There exists a many-to-one function[1] $\delta : x \mapsto R$, 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 labelled on the edge. If x and y are the same table instance (vertex), the edge, in fact, is a self-loop (edge) for the vertex in query graph G. Let E be the set of all edges in G. There exists a function $\varphi : e \mapsto c$, where $e \in E$ and $c \in 2^P$. $\varphi(e)$ gives the set of all simple predicates on 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. If $x = y$, $edge(x, x)$ is denoted by $self\text{-}loop(x)$. $sizeof(x)$ 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 $\varphi(e)$.

[1] Note that several table instances may reference the same table in the database. For example, table *EMP* is referenced twice in the following SQL query: *SELECT X.name, Y.name FROM EMP X, EMP Y WHERE X.supervisor_SSN = Y.SSN.*

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.

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

$$
\begin{aligned}
&E|_{V'} = \{\ e \mid e \in E \textit{ and } \textit{vertices}(e) \subseteq V'\},\\
&T|_{V'} = \{\ R \mid R \in T \textit{ and there exists } x \in V' \textit{ such that } x \textit{ is an instance of } R\},\\
&P|_{V'} = \{\ p \mid p \in P \textit{ and } p \in w \textit{ and } w \textit{ is the set of predicates labeled}\\
&\qquad\qquad \textit{on } e \in E|_{V'}\},\\
&\delta|_{V'} : x \mapsto R,\ \textit{where } x \in V',\ R \in T|_{V'} \textit{ and } \delta(x) = R,\\
&\varphi|_{V'} : e \mapsto c,\ \textit{where } e \in E|_{V'},\ c \in 2^{P|_{V'}} \textit{ and } \varphi(e) = c.
\end{aligned}
$$

We call $E|_{V'}$, $T|_{V'}$, $P|_{V'}$, $\delta|_{V'}$ and $\varphi|_{V'}$ the restrictions of E, T, P, δ and φ under subset V' of vertices in G, respectively. 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. Note that a subquery graph is uniquely determined by the subset V' of vertices in G. An edge $\textit{edge}(x, y)$ between vertices x and y in G is called an interconnecting edge of subquery graph G' if one of x and y is in V' and the other is not.

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[2]. If G' and G'' satisfy the following conditions, we regard them as a pair of similar subquery graphs with respect to the 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 $\varepsilon_t(x, f(x)) = \textit{max}\left\{\frac{|\textit{sizeof}(x) - \textit{sizeof}(f(x))|}{\textit{sizeof}(x)}, \frac{|\textit{sizeof}(x) - \textit{sizeof}(f(x))|}{\textit{sizeof}(f(x))}\right\}$.
- 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 $\textit{vertices}(e) = \{x, y\}$, then $\textit{vertices}(g(e)) = \{f(x), f(y)\}$ and $\varepsilon_s(e, g(e)) < r_s$, where $\varepsilon_s(e, g(e)) = \textit{max}\left\{\frac{|\textit{sel}(e) - \textit{sel}(g(e))|}{\textit{sel}(e)}, \frac{|\textit{sel}(e) - \textit{sel}(g(e))|}{\textit{sel}(g(e))}\right\}$.

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 dashed-line circles marked as G' and G''. (The light dotted-line circles show a pair of subqueries with a higher level similarity, which will be discussed in Sect. 4.) 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. Note that the size of a table and the selectivity of a query condition are the two most important factors that influence the decision of a query optimizer. In fact, a selectivity can be determined/estimated based on a set of other basic parameters/factors such as the number of distinct values in a column, the maximum and minimum values

[2] Assume that no table size or selectivity is zero. Otherwise, the result of the entire query will be empty—no optimization is needed in such a case.

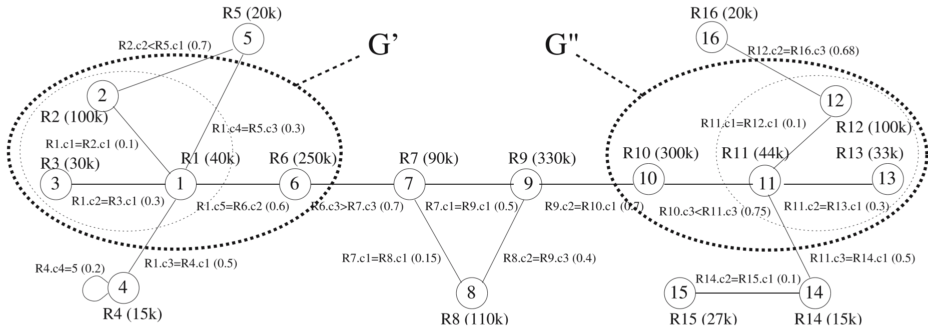

Fig. 1. Example of a query graph with similar subqueries

of a column and the data distribution of a column. We could directly use the basic parameters to define the similarity. However, using the integrated factor (i.e. the selectivity) from the basic ones greatly simplifies the definition of similarity. On the other hand, it is also possible that some additional factors need to be considered in some environments (e.g. index-related parameters if indexes exist). In such a case, the above similarity definition can be easily extended by examining the additional parameters with respect to their given error bounds.

2.4. Ring network representation

A query graph gives a logical representation of a user query. To implement an algorithm based on it, we need to adopt an efficient data structure to represent the graph in a computer system. For the query optimization technique to be discussed in this paper, we introduce a data structure called the ring network representation to represent a query graph.

In the ring network representation, every vertex x in the query graph G has a node x. Assume that node x has a set of adjacent nodes, $y_1, y_2, \ldots, y_n$. We use a closed link list (i.e. a ring): $x \rightarrow y_1 \rightarrow y_2 \rightarrow \ldots \rightarrow y_n \rightarrow x$ to represent such an adjacency relationship. Node x is called the owner (node) of the ring, while nodes $y_1, y_2, \ldots, y_n$ 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. The structure of a node[3] is shown in Fig. 2. For a simple query graph in Fig. 3, which is subquery graph G' in Fig. 1, its ring network is shown in Fig. 4.

3. Query optimization via exploiting similarity of subqueries

In this section, we will introduce a technique to perform query optimization via exploiting similarity of substructures in a given complex query. We will discuss the basic idea of this technique, give a high-level description of the algorithm for the technique and analyze the complexity of the algorithm.

[3] For simplicity, we only include the selectivity of each edge e and omit the relevant predicates, which are not directly used in our algorithms.

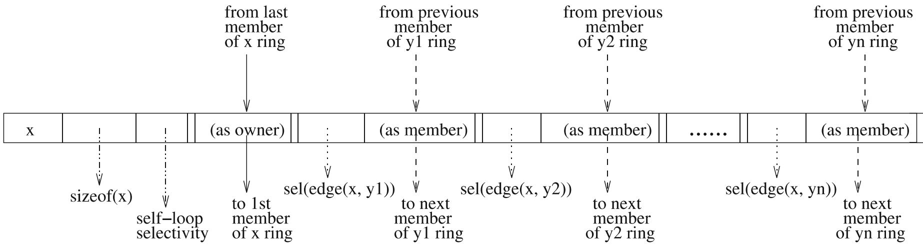

Fig. 2. Structure of node x in the ring network

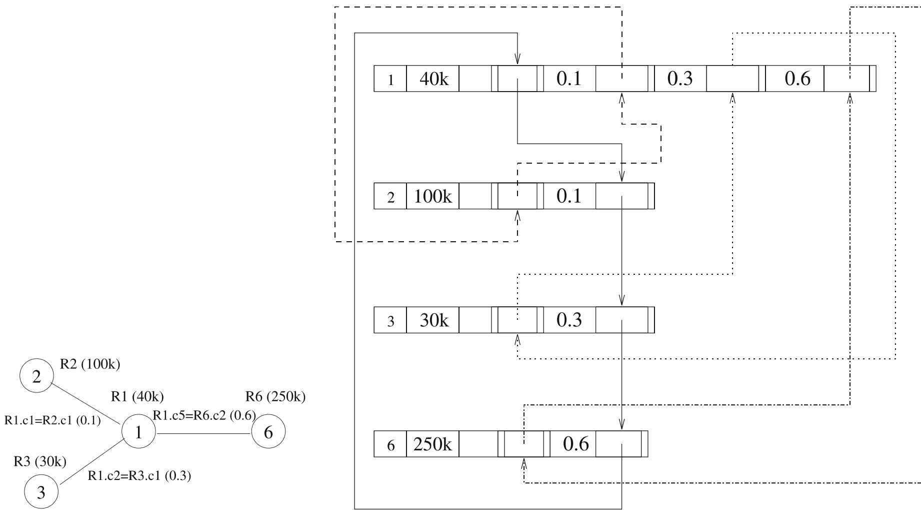

Fig. 3. A simple query graph

Fig. 4. An example of the ring network

3.1. Basic idea

The basic idea of this technique is as follows:

- Identify pairs of similar subqueries in a given query with respect to given error bounds r_t and r_s.
- Optimize one subquery in each similar pair and map and apply the resulting execution plan to the other subquery in the pair. In this way, we reduce the optimization time by sharing the same (mapped) plan between two similar subqueries.
- Replace similar subqueries with their (estimated) result tables in the query graph and optimize the resulting revised query. As the number of vertices in the query graph is reduced, less overhead is needed to optimize the revised query compared with the original query.

Because the error bounds r_t and r_s are adjustable, by setting r_t and r_s larger, we may get a pair of larger similar subqueries (that is, with more vertices in each subquery),

and thus more optimization work may be shared between the similar pair. On the other hand, if the error tolerances are higher, the similarity between corresponding vertices and edges in two similar subqueries is worse, and therefore, the optimization quality (the performance) after sharing the plan may be worse. We will discuss this issue in more detail in Sect. 5.

Note that we will describe how to generalize our technique to utilize a group of more than two similar subqueries for query optimization in Sect. 3.5.

3.2. Algorithm description

A high-level description of the algorithm for our technique is given below. Note that, if there is a self-loop on an 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 3.1. Query optimization based on similar subqueries

Input: Ring network representation of 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. **begin**
2. Initialize flag selected(x) $= 0$ for each node x; /* no node selected yet */
3. Initialize the sets of identified pairs of similar subquery graphs
   $S_{accept} = S_{hold} = \emptyset$;
4. **while** there are unselected node pairs with similar table sizes within
   error bound r_t **do**
5. Pick up the pair of nodes x and y by following the rules for choosing start
   nodes in Sect. 3.3;
6. Let $V_1 = \{x\}$; $V_2 = \{y\}$; /* V_1 and V_2 are the node sets for current similar
   subquery graphs G_1 and G_2 */
7. record one-to-one mapping $f: \ x \mapsto y$; /* i.e., $y = f(x)$ */
8. Let selected(x) = selected(y) = 1;
9. **while** V_1 and V_2 have unexpanded nodes **do**
10. Pick the next pair $x \in V_1$ and $y \in V_2$ based on FIFO order for expanding;
11. **for** each member x_1 with selected(x_1) $= 0$ in the ring owned by x **do**
12. **for** each member y_1 with selected(y_1) $= 0$ and $y_1 \neq x_1$ in the ring owned
    by y **do**
13. **if** $\varepsilon_t(x_1, y_1) < r_t$ and
14. [$edge(x_1, x')$ exists for any $x' \in V_1$ if and only if
    $edge(y_1, y')$ exists for $y' = f(x') \in V_2$
15. and $\varepsilon_s(edge(x_1, x'), edge(y_1, y')) < r_s$]
16. **then** $V_1 = V_1 \cup \{x_1\}$; $V_2 = V_2 \cup \{y_1\}$;
    /* a new pair of similar nodes has been found */
17. selected(x_1) = selected(y_1) = 1;
18. record one-to-one mapping $f: \ x_1 \mapsto y_1$;
19. 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 $y' = f(x') \in V_2$;
20. break;
    /* no need to check other similar nodes for x_1 after y_1 is found */
21. **end if**

22. **end for**
23. **end for**
24. **end while**;
25. Evaluate the resulting pair of similar subquery graphs $< G_1, G_2 >$;
26. **if** it is worth to accept $< G_1, G_2 >$
27. **then** Remove any $< G'_1, G'_2 >$ from S_{hold} that shares some nodes with $< G_1, G_2 >$;
28. $S_{accept} = S_{accept} \cup \{< G_1, G_2 >\}$;
29. **else if** it is worth to hold $< G_1, G_2 >$
30. **then** Reset selected$(x) = 0$ for all x in G_1 and G_2; /* allow nodes to be re-considered */
31. $S_{hold} = S_{hold} \cup \{< G_1, G_2 >\}$;
32. **else** Reset selected$(x) = 0$ for all x in G_1 and G_2; /* unselect nodes */
33. Discard $< G_1, G_2 >$;
34. **end if**;
35. **end while**;
36. **while** $S_{hold} \neq \emptyset$ **do** /* accept some held similar subquery pairs */
37. Remove the largest pair $< G_1, G_2 >$ from S_{hold};
38. **if** $< G_1, G_2 >$ does not share any node with any pair in S_{accept}
39. **then** $S_{accept} = S_{accept} \cup \{< G_1, G_2 >\}$;
40. **end while**;
41. **for** each similar subquery pair $< G_1, G_2 > \in S_{accept}$ **do**
42. Optimize G_1 and share the execution plan with G_2; /* after an appropriate mapping using f and g */
43. Replace G_1 and G_2 in the original query graph G with their (estimated) result tables;
44. **end for**;
45. Optimize the revised G to generate an execution plan;
46. **return** the execution plan for Q, which includes the plan for the revised G and the plans for all the similar subquery graphs;
47. **end.**

Lines 4–35 search for all pairs of similar subquery graphs in query Q. Lines 9–24 expand the current pair of similar subquery graphs using the ring network. Lines 25–34 determine if we should accept, hold or reject the pair of identified subquery graphs. Lines 36–40 accept the remaining pairs of held similar subquery graphs that are not overlapped with any accepted similar subquery graphs. Lines 41–45 optimize all similar subquery graphs and the revised final query graph. Line 46 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 subsections.

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 initial lists, called the similarity starting lists set. 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,

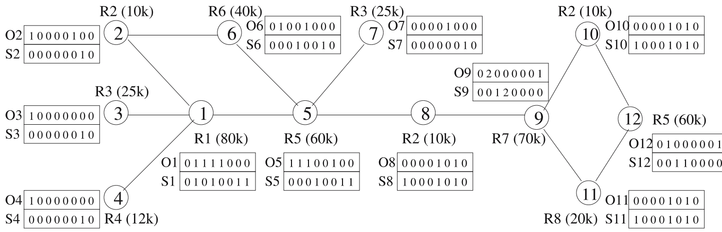

Fig. 5. Example of indicator arrays

R1	1
	9
R2	2 8 10
	4
R3	3 7
	11
R4	4
	2 8 10
R5	5 12
	9
R6	6
R7	9
	1 5 12
R8	11
	3 7

Fig. 6. Example of similarity lists set

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. 5 (selectivities on the edges are omitted), using error bound parameter $r_t = 0.3$, its similarity starting lists set is shown in Fig. 6.

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 the pair of similar subquery graphs as large as possible. This is because the larger the similar subquery graphs in a pair, the more the optimization work can be shared between them, and, therefore, the more time is saved for query optimization.

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, based on 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 (i.e. the number of distinct base tables) 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[4]. 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[5]. Figure 5 gives an example 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:

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

[4] As we have mentioned before, one base table can be represented by several instances; thus, the value of $O_x[i]$ can be greater than 1.

[5] This number does not include the number of adjacent table nodes representing R_i.

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

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

If we first match the adjacent table instances for the current ith base table of the starting pair and then match the remaining table instances of one node (if any) to other table instances whose table sizes are within the given error bound, 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 our algorithm, 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 similar 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 V_1 and V_2 to see if adding x_1 and y_1 into the current node 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.
(ii) There exists a pair of $x' \in V_1$ and $y' = f(x') \in V_2$ such that there are $edge(x_1, x')$ 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 pair of similar subquery graphs that we have just found 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 by 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 32–33). If the pair is worth keeping, we remove all nodes in the subquery graph pair from the original query graph by setting their "selected" flag to 1.

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

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:

- If $n \geq c_2$, then the new pair of similar subquery graphs is accepted.
- If $c_1 \leq n < c_2$, then we put this pair on hold.
- If $n < c_1$, then the new pair is rejected.

The threshold values can be calibrated through experiments. Note that our algorithm can also incorporate other evaluation methods for selecting similar subquery graphs.

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 for 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 that is generated from optimization to the one for its partner. For example, consider a pair of similar subqueries, G_1 and G_2, that has the following corresponding relationship for the nodes: $x_1 \leftrightarrow y_1, x_2 \leftrightarrow y_2, x_3 \leftrightarrow y_3$, where $x_1, x_2, x_3 \in G_1$ and $y_1, y_2, y_3 \in G_2$. By optimizing G_1, 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 similar subquery pairs with their result tables in the original query graph, we reduced the number of nodes in it. Because the complexities of the similar subqueries and the revised final query are less than that of the original query, many existing query optimization techniques (such as AB and II) usually perform well. If a similar subquery or the revised final query is sufficiently small (e.g. less than a chosen small constant), a dynamic-programming-based optimization technique can also be used to find a truly optimal plan for it.

3.4. Time complexity analysis

By counting the number of operations required by each line in Algorithm 3.1, it is not difficult to see that the worst-case 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 revised 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.

3.5. Sharing optimization work among more than two similar subqueries

Algorithm 3.1 assumes that optimization work is shared between two similar subqueries in each pair. Sharing work among more than two similar subqueries in each group can be done as follows. Algorithm 3.1 is applied to find a pair of similar subqueries, $< G_1, G_2 >$, first. Other subqueries (other than G_2) similar to G_1 can be then found in a similar way, described in lines 9–24 in Algorithm 3.1. All the subqueries in this similar group share the same execution plan generated for G_1—the

representative subquery for the group. A trade-off should be made between the size of each similar subquery and the size of the similar group (i.e. the number of similar subqueries in the group). In general, the more similar subqueries in each group and the larger each similar subquery in the group, the more optimization work can be saved.

For simplicity, we will still assume that each group contains a pair of similar subqueries in the rest of the discussion in this paper.

4. Discussion of higher similarity level

The similarity we have defined in Sect. 2.3 can only guarantee that the table sizes for the corresponding vertices and the selectivities for the corresponding edges from two subquery graphs are within their given error bounds. As we know, when the selectivities of two join conditions and the sizes of corresponding operand tables are within their respective error bounds, the sizes of the two join result tables may not be within its error bound. Based on this observation, we can extend the definition of the similarity between subqueries into multiple levels.

4.1. Definition of similarity level

Let symbol $G' \approx |^{k}_{(r_t,r_s)} G''$ denote that two subquery graphs, G' and G'', are similar at level k with respect to error bounds r_t and r_s. The similarity definition given in Sect. 2.3 is for the one at level 1, i.e. $G' \approx |^{1}_{(r_t,r_s)} G''$. The higher level similarities are recursively defined as follows.

Let $G_1(V_1, E|_{V_1}, T|_{V_1}, P|_{V_1}, \delta|_{V_1}, \varphi|_{V_1})$ be a subquery graph in G. Let e_1 be an edge in G_1, that is, $e_1 \in E|_{V_1}$. If we replace the two nodes connected by e_1 with their result table and rearrange the edges of the related tables in G_1 to the new result table, we get a new graph G'_1. We call this operation a β-operation, denoted by $\beta(G_1, e_1) = \hat{G}_1$.

Let G' and G'' be two subquery graphs in query graph G. r_t and r_s are the given error bounds for the table sizes and selectivities, respectively. If G' and G'' satisfy the following condition, we regard them as a pair of similar subquery graphs at similarity level k, i.e. $G' \approx |^{k}_{(r_t,r_s)} G''$:

- For $\forall e \in E'$ and $g(e) \in E''$, we have $G' \approx |^{k-1}_{(r_t,r_s)} G''$ and $\beta(G', e) \approx |^{k-1}_{(r_t,r_s)} \beta(G'', g(e))$.

The meaning of the similarity at level k (also called the k-level similarity) is: if two subqueries, G' and G'', are k-level similar, then their inner structures are the same, and if we join any corresponding $2, 3, \ldots$ or k connected tables/vertices in each subgraph, the corresponding (unchanged and merged) table sizes and the corresponding selectivities are within their given error bounds, r_t and r_s. For example, in Fig. 1, a pair of 2-level similar subquery graphs using error bounds $r_t = r_s = 0.3$ are shown in the light dotted-line circles.

4.2. Algorithm modification

For a given similarity level k, we can revise our algorithm described in Sect. 3.2 to optimize a complex query by exploiting its k-level similar subqueries.

Let x be a node in query graph $G(V, E, T, P, \delta, \varphi)$, that is, $x \in V$. Let $G_1(V_1, E|_{V_1}, T|_{V_1}, P|_{V_1}, \delta|_{V_1}, \varphi|_{V_1})$ be a subquery graph in G. We define a Boolean function $\eta(G_1, x)$. If $x \in V_1$, that is, x is a node in subquery graph G_1, then $\eta(G_1, x) = \text{TRUE}$. Otherwise, $\eta(G_1, x) = \text{FALSE}$. We will use this function in describing the algorithm based on the k-level similarity. Let $|G_1|$ denote the number of nodes in the subquery graph G_1.

In order to find the k-level similar subqueries for a given query G, we replace lines 11–23 in Algorithm 3.1 by the following code:

Algorithm 4.1. Revised code segment for Algorithm 3.1

1′. **for** each member x_1 with selected(x_1) = 0 in the ring owned by x **do**
2′.   **for** each member y_1 with selected(y_1) = 0 and $y_1 \neq x_1$ in the ring owned by y **do**
3′.   flag = 1;
4′. $V_1 = V_1 \cup \{x_1\}$; $V_2 = V_2 \cup \{y_1\}$;
5′.   record one-to-one mapping $f: \; x_1 \mapsto y_1$;
6′.   **for** i from 1 to k **do**
7′.   **for** each $G'_1 \in \{G' \mid G'$ *is a subquery graph of* G_1, $|G'| = i$, *and* $\eta(G', x_1) = TRUE\}$ **do**
8′.   follow the one-to-one mapping relationships (for vertices and edges (if any)) between G_1 and G_2 to find the corresponding subquery graph G'_2 of G_2;
9′.   Let res_1 and res_2 be the result tables for G'_1 and G'_2, respectively;
10′.   **if** $\varepsilon_t(res_1, res_2) \geq r_t$ **or** { $\exists x' \in V_1 - V'_1$ and $y' = f(x') \in V_2 - V'_2$ such that [only one, not both, of $edge(x', res_1)$ and $edge(y', res_2)$ exists] **or** $\varepsilon_s(edge(x', res_1), edge(y', res_2)) \geq r_s$ }
11′.   **then** flag = 0; break; **end if**;
        /* find a violation of k-level similarity—no need to further check */
12′.   **end for**;
13′.   **if** flag = 0 **then** break; **end if**;
      /* not k-level similar—no need to further check */
14′.   **if** i = 1 **then**
15′.   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 $y' = f(x') \in V_2$;
        /* we know G_1 and G_2 have the same structure after 1st loop */
16′.   **end if**;
17′.   **end for**;
18′.   **if** flag = 1 /* k-level similar */
19′.   **then** selected(x_1) = selected(y_1) = 1; break;
20′.   **else** $V_1 = V_1 - \{x_1\}$; $V_2 = V_2 - \{y_1\}$;
21′.   **if** i > 1 **then**
22′.   unrelate the mapping relationships $g: \; edge(x_1, x') \mapsto edge(y_1, y')$ if such edges exist for any $x' \in V_1$ and $y' = f(x') \in V_2$;
23′.   **end if**;
24′.   unrelate the mapping relationship $f: \; x_1 \mapsto y_1$;
        /* unselect nodes x_1 and y_1 */
25′.   **end if**
26′.   **end for**
27′. **end for**;

Lines $6'$–$17'$, which are the main part of the code, check if the current pair of subquery graphs with new nodes x_1 and y_1 included are k-level similar. The time complexity of the above algorithm segment is $O(k * n^{k+2})$, where n is the total number of table instances in the given query, Q, and k is the similarity level.

Note that, for the same given error bounds, the higher the similarity level, the more similar the two corresponding subquery graphs. Hence, a good execution plan generated for one subquery is more likely to be also good for the other. On the other hand, if the similarity level is very high, the chance to find similar subqueries is greatly reduced and the similar subqueries (if any) to be found may be small. Hence, less optimization work may be shared in this case. Besides, to find similar subqueries at a very high similarity level requires much time. Fortunately, as we will see in the next section, a low similarity level is usually satisfactory in practice. Therefore, we assume $k \ll n$. Note that, to search for identical (common) subqueries, our previous efficient technique based on common subqueries can be applied (Tao et al. 2002).

5. Experiments

In the last section, we have shown that our technique is an efficient polynomial time technique, which is 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 than others because it takes some complex query characteristics into account. Experiments were conducted to examine the quality of the execution plans generated by our technique.

In the experiments, we chose to compare our technique with the most promising randomization technique, i.e. the AB algorithm, for optimizing complex queries. We also compared the results of our technique based on similarities at levels $1 \sim 3$. For simplicity, we call our technique based on the i-level similarity the i-level ($i = 1, 2, 3$) similarity method in the following discussion. To make a fair comparison, we employ the AB algorithm with our technique for optimizing identified similar subqueries and the revised final query (i.e. lines 42 and 45 in Algorithm 3.1).

The experiments were run on a PC with PIII 500 MHz CPU and 512 MB RAM. All techniques were implemented in C. A synthetic database was used for our experiments, which is described as follows.

The experimental database consists of 80 tables with their sizes shown in Table 1. This database has the following characteristics:

- The table sizes cover a wide range from 1 to 19,521,020 rows.
- The database contains two groups of tables to be chosen as dissimilar ones in a test query: $R_1 \sim R_{10}$ and $R_{76} \sim R_{80}$. As shown in Table 1, the relative error $(|R_{i+1}|-|R_i|)/|R_{i+1}|$ between the sizes of tables R_i and R_{i+1} is 60% (i.e. significantly different/dissimilar) in these two groups. This feature allows us to include some dissimilar tables in test queries. Note that the database contains both small dissimilar tables (i.e. the first group) and large dissimilar tables (i.e. the second group) to increase the diversity.
- The database contains a group of tables to be chosen as similar ones in a test query: $R_{11} \sim R_{75}$. The relative error between the sizes of two consecutive tables in the group is from 1.5% (near R_{75}) to 27.5% (near R_{11}). Furthermore, the closer the two tables, the more similar they are. For example, tables R_{11} and

Table 1. Sizes of test tables

Group 1	Table R_i	R_1	R_2	R_3	......	R_{10}
	Size $\|R_i\|$	1	3	8	......	4895
	Relationship	$\|R_{i+1}\| = \lceil \|R_i\|/0.4 \rceil$				
Group 2	Table R_i	R_{11}	R_{12}	R_{13}	......	R_{75}
	Size $\|R_i\|$	7895	10895	13895	......	199895
	Relationship	$\|R_{i+1}\| = \|R_i\| + 3000$				
Group 3	Table R_i	R_{76}	R_{77}	R_{78}	......	R_{80}
	Size $\|R_i\|$	499738	1249345	3123363	......	19521020
	Relationship	$\|R_{i+1}\| = \lceil \|R_i\|/0.4 \rceil$				

R_{12} are more similar to each other than tables R_{11} and R_{15}. This feature allows us to choose tables with different degrees of similarity in a test query.

- The tables provide different selectivities for queries. Each table in the database has 15 columns. Values for the ith column ($1 \leq i \leq 15$) are randomly generated from domain [1, $100 * i^2$]. In other words, a lower column has a fewer number of distinct values than a higher column. Hence, different selectivities are provided for queries on different columns.

Clearly, the above synthetic database allows us to have some controllable parameters, such as the number of similar/dissimilar table instances in a query, the error bounds for similarity and selectivities for query conditions in our experiments. Hence, we can evaluate our technique for different situations. A real-world database, on the other hand, would usually represent only one particular application scenario. Therefore, we chose to use the above more flexible synthetic database for our experiments.

Because our technique was developed for optimizing large queries with similar subqueries, we need to have a set of such test queries to evaluate our technique. A test query Q with similar subqueries in our experiments was generated as follows.

- *Determining the total number N of tables (instances) in test query Q and the number of tables (instances) in each similar subquery in Q*. For a test query, its N is randomly chosen from the range [20, 80]. We reserve a certain percent $p\%$ (e.g. 20%) of tables for dissimilar ones. For the rest of the tables, we randomly pick up a number from the range [2, *upperbound*] as the number of tables in each subquery of a similar pair, where $upperbound = N * (1 - p\%)/2$. This process is repeated to choose more pairs of similar subqueries until either the $(1 - p\%)$ quota for similar tables is used up or adding the next pair of similar subqueries (because too large) would exceed the quota for similar tables. In the latter case, the next pair of similar subqueries is not added, and the remaining quota for similar tables is redistributed to dissimilar ones. To avoid increasing the quota (beyond the initial p) for dissimilar tables too much in such a case, we properly reduce *upperbound* during the process to allow more (smaller) similar subqueries to be generated within the quota. In general, the number of tables outside the similar subqueries generated from the above process for a test query is usually larger than the initial quota p.
- *Determining the tables in each pair of similar subqueries $< Q_1, Q_2 >$*. To do so, we randomly pick a table R from $R_{11} \sim R_{75}$ for Q_1. We then randomly pick a table from $R_{11} \sim R_{75}$ with a size within the range $[(1 - r_t) * |R|, (1 + r_t) * |R|]$,

where r_t is the error bound used to define similar tables for the given test query Q. This process is repeated until all tables in subqueries Q_1 and Q_2 are determined.

- *Determining the join edges (connections) for each pair of similar subqueries* $< Q_1, Q_2 >$. Let m be the number of tables in Q_1 (also for Q_2). Consider its adjacency matrix $A[1..m, 1..m]$. That is, if $A[i, j] = 1$, there is a join edge between the ith table and the jth table ($1 \leq i, j \leq m$) in Q_1; if $A[i, j] = 0$, there is no join edge between these two tables. In fact, we only need to consider the upper triangular part of matrix A due to its symmetry. To ensure subquery Q_1 is connected, we set all elements in the first row of A to be 1. For the rest of the upper triangular part of A, its elements are determined as follows. We randomly pick a number k from the range $[0, m*(m-1)/2]$ as the additional number of edges (besides m edges for the first row of A) in subquery Q_1. We then randomly pick a number from the range $[1, m*(m-1)/2]$, map this number into a position in the upper triangular part of A and set the element at this position to be 1. This process is repeated until k additional join edges are chosen. Once the join edges for Q_1 are determined, the corresponding join edges for Q_2 are set accordingly.
- *Determining the join predicates on the join edges for each pair of similar subqueries* $< Q_1, Q_2 >$. For each join edge in Q_1, we randomly choose a join predicate with selectivity S and associate (label) it to the edge. To avoid having an extremely large result for a large query, we restrict the selectivity of a join predicate within the range (0, 0.3]. For the corresponding join edge in Q_2, we randomly choose a join predicate with a selectivity within the range $(\max\{(1-r_s)*S, 0\}, \min\{(1+r_s)*S, 0.3\}]$, where r_s is the error bound used to define similar selectivities for the given test query Q.
- *Determining the tables that are not in any similar subquery*. To do so, we repeatedly choose a table (randomly) from $R_1 \sim R_{10}$ and $R_{76} \sim R_{80}$ until all such tables in the given query are determined.
- *Determining the join edges (connections) and predicates for the tables that are not in any similar subquery*. To do so, we consider each similar subquery determined previously in the given query Q as one single (merged) table. We then apply the above adjacency matrix method to determine the join edges among the (merged and unmerged) tables. If an end of join edge e is connected to a merged table representing a similar subquery Q', a table R' within Q' is randomly chosen to connect to that end of edge e in the original (unmerged) query. For each join edge, a join predicate is randomly chosen for it.

Clearly, a test query generated from the above process has a random number of tables, a random number of similar pairs of subqueries, a random size for each similar subquery, a random selectivity for each join predicate, and a certain number of tables outside the similar subqueries, within their respective allowed ranges. Such test queries can be used to effectively evaluate the performance of our technique.

In the experiments, the following threshold values were used to accept, hold or reject a pair of identified similar subqueries (subgraphs) in our technique: $c_1 = 3$ and $c_2 = 5$. That is, similar subqueries with $>= 5$ tables (nodes) are accepted; similar subqueries with < 3 tables (nodes) are rejected and similar subqueries with 3 or 4 tables (nodes) are put on a hold list.

As mentioned before, both the AB algorithm and our technique have a polynomial time complexity. Therefore, both of them are efficient in optimizing large queries. Note that the AB algorithm can achieve better results by taking a longer

Table 2. Comparison of I/O costs for execution plans generated by different techniques

Q#	$\|V\|$	AB-$I/O\#$	1-sim$I/O\#$ (imp%)	2-sim$I/O\#$ (imp%)	3-sim$I/O\#$ (imp%)
1	66	0.212e+59	0.273e+58 (87.1%)	0.340e+58 (84.0%)	0.713e+58 (66.4%)
2	33	0.622e+27	0.111e+26 (82.2%)	0.175e+26 (71.9%)	0.175e+26 (71.9%)
3	55	0.444e+44	0.655e+43 (85.2%)	0.355e+43 (92.0%)	0.528e+43 (88.1%)
4	38	0.486e+29	0.208e+29 (57.2%)	0.208e+29 (57.2%)	0.208e+29 (57.2%)
5	51	0.524e+44	0.720e+43 (86.3%)	0.989e+43 (81.1%)	0.137e+44 (73.9%)
6	44	0.648e+36	0.518e+35 (92.0%)	0.706e+35 (89.1%)	0.706e+35 (89.1%)
7	53	0.415e+42	0.350e+41 (91.6%)	0.234e+41 (94.4%)	0.363e+41 (91.3%)
8	72	0.309e+67	0.199e+66 (93.6%)	0.368e+66 (88.1%)	0.584e+66 (81.1%)
9	46	0.484e+36	0.138e+36 (71.5%)	0.151e+36 (68.8%)	0.151e+36 (68.8%)
10	63	0.716e+55	0.739e+54 (89.7%)	0.210e+55 (70.6%)	0.276e+55 (61.5%)
11	36	0.623e+23	0.957e+22 (84.6%)	0.295e+23 (52.6%)	0.295e+23 (52.6%)
12	57	0.370e+47	0.469e+46 (87.3%)	0.525e+46 (85.8%)	0.927e+46 (74.9%)
13	41	0.529e+33	0.154e+33 (71.0%)	0.313e+33 (40.8%)	0.313e+33 (40.8%)
14	49	0.687e+43	0.152e+43 (77.9%)	0.293e+43 (57.4%)	0.293e+43 (57.4%)
15	66	0.272e+56	0.303e+55 (88.8%)	0.218e+55 (92.0%)	0.300e+55 (89.0%)
16	31	0.591e+23	0.156e+23 (73.5%)	0.607e+23 (-2.7%)	0.607e+23 (-2.7%)
17	37	0.488e+31	0.665e+30 (86.4%)	0.725e+30 (85.2%)	0.909e+30 (81.4%)
18	60	0.612e+48	0.241e+48 (60.1%)	0.346e+48 (43.4%)	0.346e+48 (43.4%)
19	22	0.513e+14	0.719e+13 (86.0%)	0.306e+14 (40.2%)	0.306e+14 (40.2%)
20	50	0.437e+40	0.326e+39 (92.5%)	0.505e+39 (88.4%)	0.541e+39 (87.6%)

$|V|$, the number of table instances in the given query; $I/O\#$, the number of I/O's;
imp%, the percent of improvement of the underlying technique over the AB algorithm.

time to try more random plans. If we let the stand-alone AB and the AB used within our technique have the same number of tries, we would actually allow more optimization time for our technique because our technique needs some time to search for similar subqueries. From a set of experiments, we found that our technique took about 26.6% $\sim$ 68.5% (47.7% on average) more time to optimize a large query in such a case although both techniques can finish optimizing a query within 4 $\sim$ 30 minutes. To make a fair comparison, we let the stand-alone AB have twice as many tries (i.e. double its execution time) as the AB used within our technique in the following experiments. Hence, the execution of the stand-alone AB took no less time than that of our technique when optimizing a large query in the experiments.

Because no complex computation is required for the set of queries considered in the paper, the dominant cost for executing such a query is the I/O cost (i.e. the number[6] of I/O's). Like related work in the literature, we focused on comparing the I/O costs when queries were performed by using the execution plans generated from different techniques. Table 2 shows a typical comparison of I/O costs for the execution plans generated by different techniques for a set of test queries. The error bounds used to search for similar subqueries in the experiments were $r_s = r_t = 0.3$, and the initial quota p for tables outside the similar subqueries in a test query was set to 20%.

[6] A typical block size (8 KB) was used in the experiments.

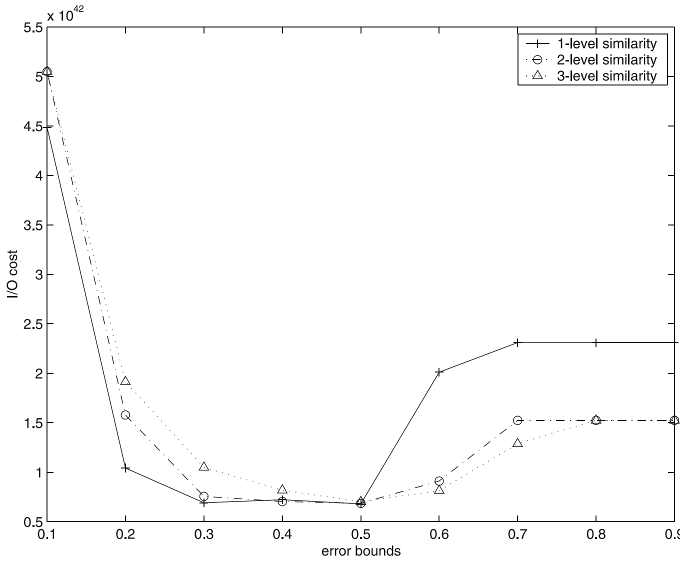

Fig. 7. Experimental result of changing error bounds

From the experimental result, we can see that the performance of execution plans generated by our 1-level, 2-level and 3-level similarity methods for the test queries is better than the performance of execution plans generated by the AB algorithm. In fact, our 1-level, 2-level and 3-level similarity methods improve the performance of the queries (on average) by 82.2%, 69.0% and 65.7%, respectively. This observation verifies that making use of the characteristics of a complex query can improve the performance of its execution plan.

The experimental result also demonstrates that increasing the similarity level usually does not improve the performance. The reason for this phenomenon is that, for the same error bounds, the higher the similarity level, the smaller the similar subqueries that can be found (due to the stronger requirement). Although a good execution plan for a subquery has a better chance to be also good for its similar counterpart when the similarity level is higher, the smaller size of the subquery reduces this benefit (because using a good execution plan for a small subquery may not have a significant impact on the performance of the overall execution plan, compared with a large subquery).

We also conducted experiments to examine the effect of error bounds on the performance of query execution plans generated by our technique for different similarity levels. Figure 7 shows the typical experimental result for a large query with similar subqueries at different levels for different error bounds (assuming $r_t = r_s$). From the figure, we can get the following observations:

- Very small error bounds cannot yield good performance, because, the smaller the error bounds, the smaller the similar subqueries found.
- Moderate error bounds (e.g. $0.3 \sim 0.5$) yield the best performance because similar subqueries with reasonable sizes can be found.
- Large error bounds degrade the performance because subqueries are less similar in such cases, which means that sharing execution plans among them may be inappropriate.

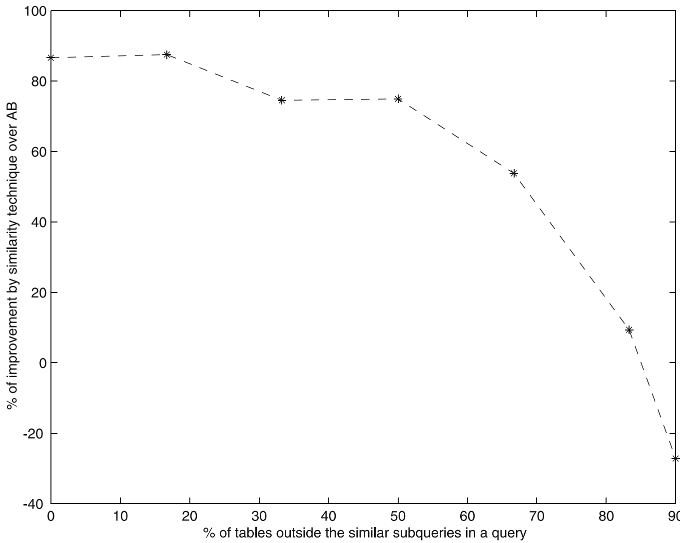

Fig. 8. Experimental result of changing percentage of tables outside the similar subqueries

- When the error bounds are sufficiently large (e.g. ≥ 0.8), the performance of an execution plan generated by our technique (for every similarity level) stays the same. The reason for this phenomenon is that similar subqueries must have the same query structure. When the error bounds are beyond a certain limit, the sizes of similar subqueries may reach the maximum (in terms of the common structure).
- When the error bounds are large, similarity at a higher level helps to improve the performance. This is because larger error bounds usually lead to larger similar subqueries and the higher level similarity ensures that subqueries are more similar to each other.

Another factor that affects the performance of a large query optimized by our technique is the percentage of tables that are outside its similar subqueries. To examine how this factor affects the performance, we conducted an experiment. In the experiment, we considered queries with different percentages of tables outside their similar subqueries. For each percentage, we applied our technique (1-level as a representative) to optimize 10 test queries and calculated the average performance improvement from our technique over the AB algorithm. Figure 8 shows the experimental result.

From the figure we can see that, the smaller the percentage of tables outside the similar subqueries (in other words, more tables in the similar subqueries), the more the performance improvement achieved by our technique. When the percentage of tables outside the similar subqueries in a query is very large (e.g. $\geq 85\%$), our technique does not outperform the AB algorithm. Therefore, our technique is not suitable for queries without or with little similarity among its subqueries.

In summary, our similarity-based technique can significantly improve the performance of complex queries that possess certain similarity among subqueries. To achieve good performance, the error bounds should be moderate and the similarity level does not need to be high. However, for large error bounds, a high similarity level is needed to ensure reasonable performance.

6. Conclusions

As database technology is applied to more and more application domains, user queries become more and more complex. Query optimization for such queries becomes 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-based 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 because these techniques do not take the special characteristics of a complex query into consideration.

In this paper, we propose a new technique for optimizing complex queries based on exploiting similar subqueries. The key idea is to use an efficient ring network representation to represent a complex query, search for similar subqueries, optimize each representative subquery, share the optimization result with other similar subqueries, reduce the original query by replacing each similar subquery with its result table and then optimize the revised final query. Any efficient technique such as AB algorithm can be used together with our technique for optimizing identified similar subqueries and the revised final query. It has been shown that our technique is an efficient polynomial time technique, and furthermore, it usually generates good execution plans for complex queries with similar subqueries because it takes the structure characteristics of a query into account and divides a large query into small parts. Using our technique, the query optimizer only needs to optimize a subquery with a small number of operand tables at each time rather than directly deal with the original query with a large number of operand tables.

Our experimental results demonstrate that the technique is quite promising in optimizing complex queries with similar substructures. It outperforms the popular AB algorithm. Experimental results also show that different situations need different similarity levels to obtain the best performance for our technique.

Our work is just the beginning of further research that needs to be done in the future in order to completely solve the query optimization issues for complex queries.

Acknowledgements. The authors would like to thank Berni Schiefer, Kelly Lyons and William Grosky for many useful discussions. We also wish to extend our sincere thanks to the anonymous reviewers for their careful reading of the manuscript and valuable suggestions for improving the paper. The preliminary work of this paper was presented at DEXA'03 (Tao et al. 2003).

Trademarks. IBM is a registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Appendix: A real-world complex query with similar subquery structures

In this appendix, we show an example of a complex (large) query with similar subquery structures from a real-world user application. To protect the user's information, we have masked the real names of the tables and attributes in the example. The complex query is expressed in SQL as follows.

```
SELECT
SUM(F.R_CMS) AS CMS, SUM(F.R_MPY) AS MPY, SUM(F.R_NCD) AS NCD,
SUM(F.R_SPI) AS SPI
```

```
FROM
FR_7 F, DR_7 D7, S_CLA S1, S_CLA S2, S_CLA S3, S_CLA S4, S_CLA S5,
DR_2 D2, SCARE S6, DR_5 D5, SCUSR S7, SCUSR S8, SCUSR S9,
SCUSR S10, SCUSR S11, SCUSR S12, SCUSR S13, DR_U DU, SUNIT S14,
DR_T DT, SDATE S15, SCMON S16, SCQUA S17, SCWEK S18, SCYER S19,
DR_B DB, SCSMR S20, DR_8 D8, SCTYP S21, SCRCY S22, DR_1 D1,
SCUPC S23, SMATR S24, SPLT S25, DR_A DA, SR_BAC S26, SR_BKY S27,
DR_C DC, SR_CDF S28, SR_CSR S29, SR_CSN S30, SR_CNB S31,
SR_CTP S32, SR_COD S33, SR_CON S34, SR_COT S35, DR_6 D6,
SR_CTR S36, SR_CDF S37, SR_PDR S38, SR_PME S39, SR_PON S40,
DR_9 D9, SR_PPR S41, DR_D DD, SR_PMO S42, DR_4 D4, SR_RCN S43,
SR_SGR S44, SR_SNO S45, DR_3 D3, SR_SHR S46, SR_STP S47,
SR_SER S48, SUNIT S49, STIME S50, SWDY1 S51, DR_P DP
WHERE
F.KC_1=D1.DID AND D1.ENP=S23.SID AND D1.MAT=S24.SID AND
F.KC_2=D2.DID AND D2.CUS1=S6.SID AND D2.PLT=S25.SID AND
F.KC_3=D3.DID AND D3.SHR=S46.SID AND D3.TIME=S50.SID AND
F.KC_4=D4.DID AND D4.RCN=S43.SID AND F.KC_5=D5.DID AND
D5.CUS2=S7.SID AND D5.CUS3=S8.SID AND D5.CUS4=S9.SID AND
D5.CUS5=S10.SID AND D5.CUS6=S11.SID AND D5.CUS7=S12.SID AND
D5.CUS8=S13.SID AND D5.SGR=S44.SID AND D5.SNO=S45.SID AND
F.KC_6=D6.DID AND D6.CTR=S36.SID AND D6.PON=S40.SID AND
F.KC_7=D7.DID AND D7.CLA1=S1.SID AND D7.CLA2=S2.SID AND
D7.CLA3=S3.SID AND D7.CLA4=S4.SID AND D7.CLA5=S5.SID AND
D7.CSR=S29.SID AND D7.CSN=S30.SID AND F.KC_8=D8.DID AND
D8.C_TYP=S21.SID AND D8.COD=S33.SID AND D8.CON=S34.SID AND
D8.COT=S35.SID AND F.KC_9=D9.DID AND D9.PPR=S41.SID AND
D9.STP=S47.SID AND F.KC_U=DU.DID AND DU.B_UOM=S14.SID AND
DU.D_CRY=S22.SID AND DU.S_UNI=S49.SID AND F.KC_T=DT.DID AND
DT.CDAY=S15.SID AND DT.CMON=S16.SID AND DT.CQUA=S17.SID AND
DT.CWEK=S18.SID AND DT.CYER=S19.SID AND DT.WDY1=S51.SID AND
F.KC_A=DA.DID AND DA.BAC=S26.SID AND DA.BKY=S27.SID AND
F.KC_B=DB.DID AND DB.CSM=S20.SID AND DB.CDF=S37.SID AND
F.KC_C=DC.DID AND DC.CDF=S28.SID AND DC.CNB=S31.SID AND
DC.CTP=S32.SID AND DC.PDR=S38.SID AND DC.PME=S39.SID AND
DC.SER=S48.SID AND F.KC_D=DD.DID AND DD.PMO=S42.SID AND
F.KC_P=DP.DID  AND ((((DP.RQD<=81)) AND
((S1.C_CLA BETWEEN 'K0' AND 'K6'))))
GROUP BY
S1.C_CLA, S2.C_CLA, S3.C_CLA, S4.C_CLA, S5.C_CLA, S6.CUSR,
S7.CUSR, S8.CUSR, S9.CUSR, S10.CUSR, S11.CUSR, S12.CUSR,
S13.CUSR, S14.UNIT, S15.DAT, S16.CMON, S17.CQUA, S18.CWEK,
S19.CYER, S20.CSMR, S21.CTYP, S22.CRCY, S23.CUPC, S24.MATR,
S25.PLT, S26.R_BAC, S27.R_BKY, S28.R_CDF, S29.R_CSR,
S30.R_CSN, S31.R_CNB, S32.R_CTP, S33.R_COD, S34.R_CON,
S35.R_COT, S36.R_CTR, S37.R_CDF, S38.R_PDR, S39.R_PME,
S40.R_PON, S41.R_PPR, S42.R_PMO, S43.R_RCN, S44.R_SGR,
S45.R_SNO, S46.R_SHR, S47.R_STP, S48.R_SER, S49.UNIT,
S50.TIME, S51.WDY1   .
```

This query involves 68 tables. To process this query, a database management system typically performs a join of all tables using the condition in the WHERE clause first. It then processes the GROUP BY clause. The join structure for this query is shown

in Fig. 9, which demonstrates a snowflake structure. Similar subqueries exist among the snowflakes in the query.

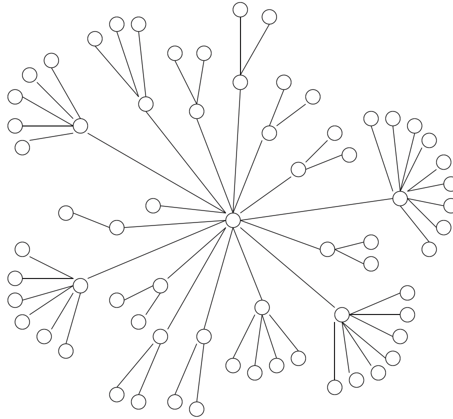

Fig. 9. The structure of a real-world large join query

References

Bennett K, Ferris MC, Ioannidis Y (1991) A genetic algorithm for database query optimization. In: Proceedings of 4th international conference on genetic algorithms, pp 400–407

Chaudhuri S (1998) An overview of query optimization in relational systems. In: Proceedings of ACM symposium on principles of database systems (PODS), pp 34–43

Chesnais A, Gelenbe E, Mitrani I (1983) On the modeling of parallel access to shared data. In: Commun ACM 26(3):196–202

Cosar A, Lim EP, Srivastava J (1993) Multiple query optimization with depth-first branch-and-bound and dynamic query ordering. In: Proceedings of international conference on information and knowledge management (CIKM), pp 433–438

Dalvi NN, Sanghai SK, Roy P, et al (2001) Pipelining in multi-query optimization. In: Proceedings of ACM symposium on principles of database systems (PODS), pp 59–70

Dalvi NN, Sanghai SK, Roy P, et al (2003) Pipelining in multi-query optimization. In: J Comput Syst Sci 66(4):728–762

Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25(2):111–152

Ibarake T, Kameda T (1984) On the optimal nesting order for computing N-relational joins. ACM Trans Database Syst 9(3):482–502

Ioannidis YE, Wong E (1987) Query optimization by simulated annealing. In: Proceedings of ACM international conference on management of data (SIGMOD), pp 9–22

Jarke M, Koch J (1984) Query optimization in database systems. ACM Computing Surveys 16(2):111–152

Kalnis P, Papadias D (2003) Multi-query optimization for on-line analytical processing. Inf Syst 28(5):457–473

Krolikowski Z, Morzy T, Bebel B (2000) Comparison of genetic and tabu search algorithms in multiquery optimization in advanced database systems. In: Proceedings of international conference on advances in information systems (ADVIS). Lecture notes in computer science, Vol 1909. Springer, Berlin Heidelberg, New York, pp 117–126

Matysiak M (1995) Efficient optimization of large join queries using tabu search. Inf Sci 83(1–2):77–88

Mistry H, Roy P, Sudarshan S, et al (2001) Materialized view selection and maintenance using multi-query optimization. In: Proceedings of ACM international conference on management of data (SIGMOD), pp 307–318

O'Gorman K, Agrawal D, Abbadi AE (2002) Multiple query optimization by cache-aware middleware using query teamwork. In: Proceedings of IEEE international conference on data engineering (ICDE), pp 274

Selinger PG, Astrahan MM, Chanmberling DD, et al (1979) Access path selection in a relational database management system. In: Proceedings of ACM international conference on management of data (SIGMOD), pp 23–34

Swami A, Gupta A (1988) Optimization of large join queries. In: Proceedings of ACM international conference on management of data (SIGMOD), pp 8–17

Swami A, Iyer BR (1993) A polynomial time algorithm for optimizing join queries. In: Proceedings of the 9th IEEE conference on data engineering (ICDE), pp 345–354

Tao Y, Zhu Q, Zuzarte C (2002) Exploiting common subqueries for complex query optimization. In: Proceedings of the 2002 CAS conference (CASCON), pp 21–34

Tao Y, Zhu Q, Zuzarte C (2003) Exploiting similarity of subqueries for complex query optimization. In: Proceedings of the 14th international conference on database and expert systems applications, pp 747–759

Author biographies

Qiang Zhu is an Associate Professor in the Department of Computer and Information Science at the University of Michigan–Dearborn, USA. He received his PhD from the University of Waterloo in Canada in 1995. He also holds an MSc from the McMaster University in Canada, an MEng and a BSc, both from the Southeast University in China. He was a faculty member in Computer Science at the Southeast University and a Visiting Scientist and a CAS Faculty Fellow at the IBM Toronto Lab. His research has been funded by competitive sources, including US NSF and IBM. He has published numerous papers in various journals and conference proceedings. Some of his research results have been included in several well-known database books. Dr. Zhu has served as a program committee member and workshop/session chair for a number of international conferences. His current research interests include query optimization, multidatabases, multidimensional indexing, self-managing databases and Web databases.

Yingying Tao is a graduate research assistant with an IBM CAS fellowship at the University of Michigan–Dearborn, USA. She received an MSc in Computer and Information Science from The University of Michigan-Dearborn, USA, in 2004 and a BSc in Computer Science from Tsinghua University (China) in 2000. Her research interests include query processing and optimization in database systems.

Calisto Zuzarte is a senior technical staff member at IBM in the data management software development organization. He manages the Query Rewrite component of DB2 on the Linux, Unix and Windows platforms and specializes in query optimization for performance. Calisto is also a research staff member in the Centre for Advanced Studies (CAS) at the IBM Toronto Lab.

Correspondence and offprint requests to: Qiang Zhu, Department of Computer and Information Science, The University of Michigan–Dearborn, Dearborn, MI 48187, USA. Email: qzhu@umich.edu