Exploiting Common Subqueries for Complex Query Optimization*

Yingying Tao† Qiang Zhu† Calisto Zuzarte‡

Abstract

As database technology is applied to more and more application areas, user queries on a database become more and more complex. Existing query optimization techniques were not developed for dealing with complex queries and may suffer from some serious problems such as intolerably long optimization time and poor optimizing results. To tackle this challenge, we introduce a new technique to improve the quality of complex query optimization in this paper. The key idea is to exploit the common subqueries that often appear in a complex query and reuse the optimization work done for each common subquery. An algorithm to identify common subqueries efficiently and generate a high-quality execution plan for a given complex query is proposed. The complexity of the algorithm is analyzed. Experiments demonstrate that this proposed polynomial-time technique is quite promising in optimizing complex queries for a database management system.

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. People have conducted extensive study on query optimization in the last three decades. Many query optimization techniques have been proposed. Good surveys of this area is given in [2, 3, 6].

However, the database field is a rapidly growing one. As database technology is applied to more and more application areas, user queries become more and more complex in terms of (1) the number of operations (e.g., more than 50 joins), (2) the query structure (e.g., star queries, snowflake queries), and (3) the heterogeneity of operations (e.g., regular joins and spatial joins). These are the three main dimensions of query complexity among others. The query optimization techniques adopted in the existing DBMSs cannot cope with these new challenges.

In this paper, we focus on studying a technique to tackle the challenges of the first dimension mentioned, namely, the rapidly increasing number of operations in a complex query. However, we also make use of some special structure information (i.e., common subqueries) in complex queries (i.e., the second dimension) to develop our technique.

As we know, the most important operation for a query is the join operation[1]. A query typically involves a sequence of joins. To determine a good join order[2] for the query, two types of algorithms are adopted in the current DBMSs. The first type of algorithms are 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 algorithms are based on heuristics. Although such an algorithm has a polynomial time complex-

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

†Department of Computer and Information Science, The University of Michigan, Dearborn, MI 48128, USA.

‡IBM Toronto Laboratory, Markham, Ontario, Canada L6G 1C7

[1] In this paper, we consider a relational DBMS only.

[2] Among many decisions contained in a QEP, we focus on studying the join order decision in this paper.

ity, it often uses a suboptimal solution. In traditional database applications, queries involving more than 15 joins are considered to be unrealistic. For such queries, both types of algorithms mentioned above are acceptable to a certain degree. Although the time complexity of the former is high, the actual time taken for optimizing a query is still acceptable because of the small input size. Although the solution used by the latter may not be optimal, it usually does not cause a disaster due to the small size of a query and the high-performance of a modern machine.

However, as database applications become more and more complex and database sizes become larger and larger, queries with a large number of joins (e.g., ≥ 50) occur more and more often in the real world. Existing techniques can no longer be used to optimize such queries. For an algorithm with exponential complexity, it may take months or years to optimize a query. For instance, if such an algorithm takes 1 minute to find an optimal join order for a query with 15 joins in the worst case, it could take about 65,372 years to find an optimal join order for a query with 50 joins in the worst case. 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 algorithms do not take the characteristics of a complex query into consideration, which often leads to a bad solution when optimizing the query.

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 impossible to find an optimal plan within a polynomial time. A few studies have been done to find a good plan for a complex query within a polynomial time recently. The proposed techniques include the iterative improvement (II), simulated annealing (SA) [5], Tabu Search (TS) [7], AB algorithm (AB) [9], genetic algorithm (GA) [1]. These algorithms represent a compromise between the time the optimizer takes for optimizing a query and the quality of the result plan. Clearly, the quality of a query execution plan is no longer the most important thing, and an efficient optimization algorithm is also crucial. It should have a polynomial time complexity in the worst case, and in practice it can "perform well on the average, and very rarely performs poorly" [10].

These techniques are mostly based on the randomization method. That is, they pick up a sample of plans randomly, choose the best one using some criteria, and then repeat this procedure until a stop condition is met. One advantage of this approach is it is applicable for all kinds of queries, no matter whether simple or complex. On the other hand, since the randomization method has little "intelligence", it does not make use of some special features that may lie within the underlying queries.

We notice that many complex queries contain some "common factors". That is, some substructures in a query are exactly the same. This phenomenon appears more often when predefined views are used in the query, the underlying database system is a distributed one, or some parts of the query are automatically generated. The more complex the query is, the more "common subqueries" there will possibly be.

Based on this observation, we introduce a new technique to reduce optimization time for a complex query by exploiting its common subqueries. This is a two-phase optimization procedure. In the first phase, it identifies common subqueries in a given query. For each group of subqueries that are of the same query structure, it performs optimization for only one such subquery and uses the result optimization plan for all the subqueries in the group. After each common subquery is replaced by its (estimated) result table obtained by using the corresponding optimization plan in the original query, the revised query is then optimized in the second phase. Since the complexities of common subqueries and the revised final query are reduced, they can be optimized using a conventional approach (e.g., dynamic programming) or a randomization approach.

The rest of this paper is organized as follows. Section 2 introduces the definitions of a query graph and its common subqueries, 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 shows some experimental results. Section 5 summarizes our conclusions.

2 Query Graph and Common Subqueries

In this section, we introduce the definitions of a query graph and common 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 operations (σ), join operations ($\bowtie$), and projection operations (π). These are the most common operations studied for query optimization in the literature, which are also the operations considered in this paper. Note that π is usually processed together with other operations (σ or $\bowtie$) it follows in a pipelined fashion rather than processed independently. For simplicity, we assume that there always is a projection operation following each $\sigma/\bowtie$ operation to filter out the columns that are not needed for processing the rest of a given query. We also assume that the query condition is a conjunction of simple predicates of the following form: $R.a\ \theta\ C$ 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 $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

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. Note that one table in T may be referenced several times 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. There exists a many-to-one function $\delta : x \mapsto R$, where $x \in V$ and $R \in T$. In other words, several table instances may reference the same table in the database. In G, each $x \in V$ is labeled by $\delta(x) = R$, i.e., R is the "weight" of vertex x.

- 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. Note that if x and y are different, the conjunction of all predicates on their edge is the join condition for a join between x and y. 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. The conjunction of all predicates on the loop is the (selection) condition for the selection operation on the table instance. 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 $\varphi(e) = c$, i.e., c is the weight of edge e.

  For example, if vertices x and y have a join relationship and the join condition is the conjunction of predicates p_1 and p_2, then there is an edge e between x and y and $\varphi(e) = \{p_1, p_2\}$. If vertex z has a selection operation on it and the condition is simple predicate p_3, then there is a self-loop edge e_1 on z and $\varphi(e_1) = \{p_3\}$.

  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, and $vertices(e)$ denotes the set of (one or two) vertices connected by edge e in a query graph.

Therefore, a query graph for query Q is a 6-tuple $G(V, E, T, P, \delta, \varphi)$, with each component defined as above.

Note that, based on our assumption, projection operations are implied in a query and thus not shown explicitly in the query graph. For any pair of distinct vertices (table instances), if there is no join condition between them, a Cartesian product can be performed on them. However, a Cartesian product usually receives the lowest priority when determining a join order for a given query. 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.

Let us consider an example. Given the following SQL query,

SELECT $X.a_1$, $R_2.a_2$
FROM R_1 X, R_1 Y, R_2, R_3, R_4, R_5, R_6,
V_1 as (**SELECT** $R_2.a_3$
FROM R_1, R_2, R_3, R_6
WHERE $R_1.a_1 = R_2.a_1$ and $R_2.a_2 = R_3.a_2$ and $R_2.a_6 = R_6.a_6$)
WHERE $X.a_1 = R_2.a_1$ and $R_2.a_2 = R_3.a_2$
and $R_2.a_5 = R_6.a_5$ and $X.a_3 = R_5.a_3$
and $R_3.a_4 = R_5.a4$ and $R_5.a_1 = R_4.a_1$
and $R_5.a_2 = R_4.a_4$ and $R_4.a2 = Y.a_2$
and $R_4.a_3 = V_1.a_3$ and $R_4.a_8 = 10$;

its query graph is shown in Figure 1.

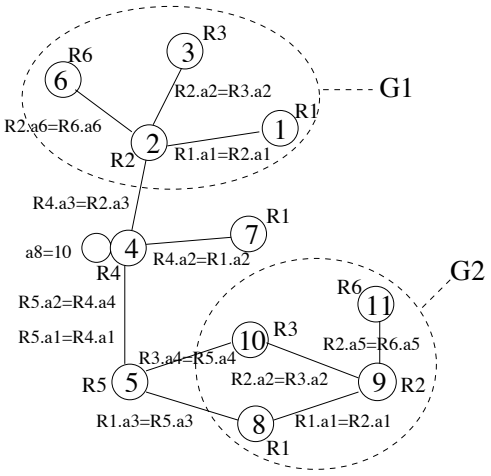

Figure 1: Example of a query graph

2.3 Common 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

$E|_{V'} = \{\ e \mid e \in E$ *and* $vertices(e) \subseteq V'\ \}$,
$T|_{V'} = \{\ R \mid R \in T$ *and there exists* $x \in V'$ *such that* x *is an instance of* $R\ \}$,
$P|_{V'} = \{\ p \mid p \in P$ *and* $p \in w$ *and* w *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* $\delta(x) = R$,
$\varphi|_{V'} : e \mapsto c$, *where* $e \in E|_{V'}$, $c \in 2^{P|_{V'}}$ *and* $\varphi(e) = c$.

We call $E|_{V'}$, $T|_{V'}$, $P|_{V'}$, $\delta|_{V'}$ and $\varphi|_{V'}$ be 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 $edge(x, y)$ between vertices x and y in G is called an interconnected edge of subquery graph G' if one of x and y is in V' and the other is not in V'.

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. If they satisfy the following conditions, we regard them as a pair of common subquery graphs (namely, the corresponding subqueries are a pair of common subqueries):

- $T|_{V'} = T|_{V''}$ and $P|_{V'} = P|_{V''}$. That is, the corresponding label sets for the vertices and edges are the same.

- 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 $\delta|_{V'}(x) = \delta|_{V''}(f(x))$. That is, the labels (tables) for the corresponding vertices (instances) are the same.

- 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 $\varphi|_{V'}(e) = \varphi|_{V''}(g(e))$. That is, the join

conditions for the corresponding pairs of join vertices are the same.

For example, there exists a pair of common subquery graphs G_1 and G_2 in the query graph shown in Figure 1. Intuitively, a pair of common subquery graphs have the same inner graph structure, the same corresponding vertex labels and the same corresponding edge labels, although the IDs of the corresponding vertices are different.

2.4 An Efficient Data Structure for Query Graph

A query graph gives a logical representation of a user query. To implement an algorithm based on a query graph, we need to adopt an efficient data structure to represent the query graph in a computer system. For the query optimization technique to be discussed in this paper, we introduce a data structure called a network ring representation (RNR) for a query graph. As we will see, this data structure is an efficient way to represent the input query and facilitate the search for common subqueries.

Given a query graph $G(V, E, T, P, \delta, \varphi)$, its ring network representation (RNR) is a comprehensive data structure with several components that are described below.

Ring Network

To represent the vertices, edges and their labels in the query graph, we use a network of nodes linked by rings (simply called a ring network) as follows:

- Every vertex x in the query graph has a node[3] x in the ring network. 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 a query graph is the owner of one ring. It also participates in other rings as a member. More specifically, node x participates in $n+1$ rings, in one of which it plays the role as the owner, and in others it plays a role as a member. Hence, node x has 1 owner pointer to point to the first member[4] that it owns and n member pointers to point to the next member[5] in the other rings in which it participates(See Figure 2).

- All vertex labels (i.e., the tables) and edge labels (i.e., the join conditions) are kept in two link lists, respectively. To indicate which table (vertex label) a node in the ring network represents, the node (e.g., x) has a table pointer to point to the corresponding table (e.g., $\delta(x)$) entry in the above first link list. To indicate which join condition (edge label) a pair of nodes has, each member pointer is associated with a condition pointer to point to the corresponding join condition entry in the above second link list. To indicate which selection condition a node may have, each node has a pointer to point to the corresponding condition (if any), and if there is no self loop on a node, the pointer point to NULL. Figure 2 shows the structure of a node in the ring network.

For example, for a simple query graph in Figure 3, its ring network is shown in Figure 6.

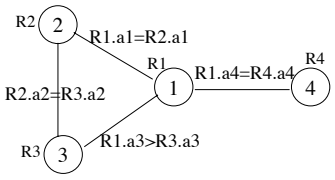

Figure 3: A simple query graph

Note that, to simply the view of Figure 6, we instantiate each table pointer with the table name it points to and instantiate each condition pointer with the join condition it points to. Furthermore, the bitmaps, which will be discussed later on, are not shown in the figure.

[3] For simplicity, we use the same symbol to denote both a vertex in the query graph and its corresponding node in the ring network representation.

[4] Assume that the IDs of all nodes are ordered. Members in a ring are linked in the ascending order of their IDs.

[5] If x is the last member in a ring, it points back to the owner of the ring.

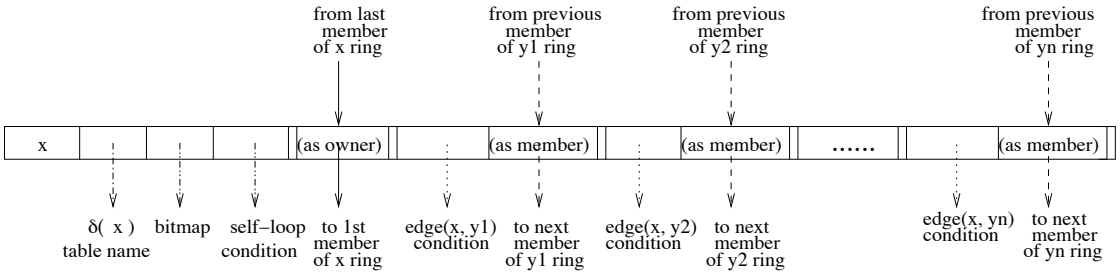

Figure 2: Structure of node x

Candidate Node Lists

Although a ring network well represents the information in a query graph, it is not efficient to directly use it to search for common subqueries. Some auxiliary structures are needed to facilitate the search.

To efficiently locate the candidate start nodes in a ring network for a search for common subqueries, we build a structure for candidate node lists. Figure 4 shows such an example.

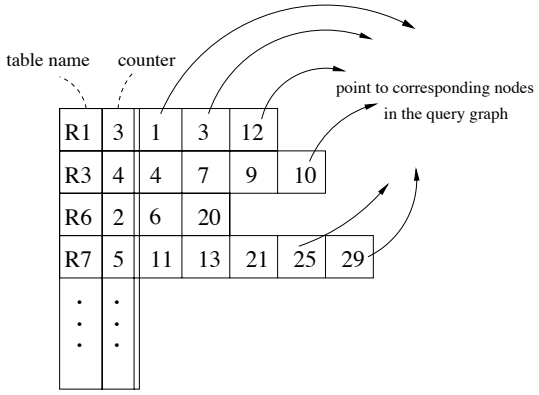

Figure 4: Example of candidate node lists

When we first scan the input query, we construct a candidate node list for each table in the query. The list has a header that contains the table name and a counter to keep the number of instances for the table used in the query. The list also contains pointers to the nodes in the ring network that represent the instances of the table. It is obvious that only those tables (lists) that have more than one instance (node) in the query (ring network) can be in common subqueries. The nodes in such lists are called candidate (start) nodes. Those lists that have only one node are useless for searching for common subqueries. Therefore, such a list is removed from the candidate node lists.

Bitmaps

To efficiently search for large common subqueries, we attach a bitmap to each node in a ring network. The length of the bitmap is the same as the size of T (i.e., the number of different tables) for the input query. Each table is represented by one bit in the bitmap. For each node x in a ring network, assume its adjacent nodes are $y_1, y_2, ..., y_n$, representing tables $\delta(y_1), \delta(y_2), ..., \delta(y_n)$, respectively. We set the bits corresponding to these tables to 1 in the bitmap for node x, and set other bits to 0. Note that $\delta(y_1), \delta(y_2), ..., \delta(y_n)$ may not be distinct. An example of bitmaps is shown in Figure 5.

We will use $\beta(x)$ to represent the bitmap of node x in our further discussion. The functions $count0(\beta(x))$ and $count1(\beta(x))$ give the numbers of 0's and 1's in $\beta(x)$, respectively.

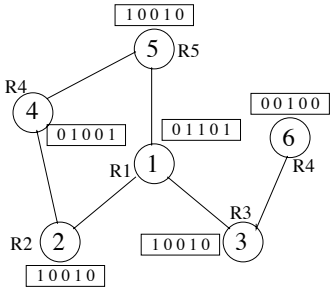

Figure 5: Example of bitmaps

The ring network, candidate node lists and bitmaps together comprise a ring network representation for the given query

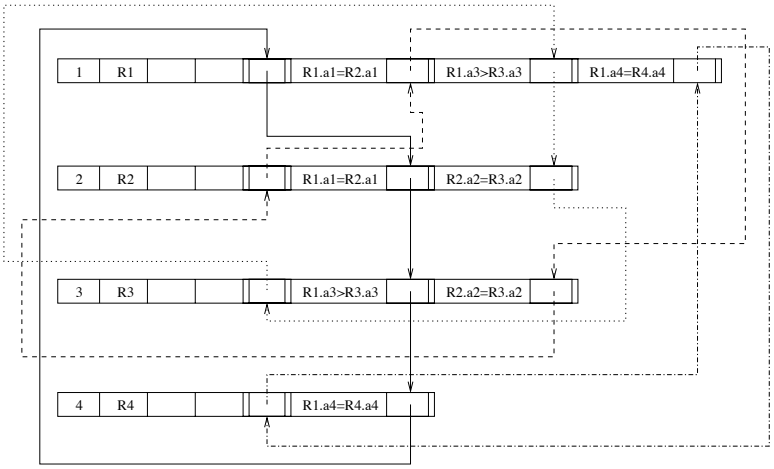

Figure 6: An example of the ring network

graph. As mentioned before, the query graph $G(V, E, T, P, \delta, \varphi)$ is a logical representation of a given query Q, while the ring network representation is an implementation representation of the query graph. Hence information in all components V, E, T, P, δ, and φ in query graph G are represented. In some cases we use the two representations for a query interchangeably in discussions of the rest of this paper.

To apply our optimization technique, which is to be discussed in the next section, we need to first convert a user query into a ring network representation. The nodes in the RNR, their labels (i.e., table names), and the candidate node lists can be created during a scan of the table instances involved in the query. The time complexity is $O(n)$, where n is the number of table instances referenced in the query. The edges among the nodes in the RNR and their labels can be established during a scan of the join predicates in the query. The time complexity is $O(m)$, where m is the number of join predicates in the query. The time complexity to compute the bitmaps is $O(n^2)$ (in the worst case). Therefore, the total time complexity to construct an RNR for a given query in the worst case is $(m+n^2)$, which is polynomial.

3 Query Optimization Exploiting Common Subqueries

In this section, we will introduce our algorithm to perform query optimization based on common subqueries. The basic idea is to (1) identify common subqueries; (2) for each group of subqueries that are common, optimize one subquery and share the resulting plan with others; and (3) replace common queries with their (estimated) result tables in the query graph (in fact, the ring network representation) and optimize the resulting query.

3.1 Algorithm Description

A high-level description of the algorithm is given as follows. Note that this algorithm makes use of pairs of common subqueries only. However, it can be generalized to utilize more than two subqueries that are common.

ALGORITHM **1** : Query Optimization Based on Common Subqueries (QOBCS).
Input: Ring network representation of query graph $G(V, E, T, P, \delta, \varphi)$ for user query Q.
Output: Execution plan for query Q.
Method:
1. **begin**
2. Initialize flags selected(x)=0 for each node x;
3. Initialize the sets of identified pairs of com-

```
    mon subquery graphs S_accept = S_hold = ∅;
4.  while there are candidate nodes in the
    candidate node lists do
5.    Use bitmaps to find the pair of candidate
      nodes x and y with δ(x) = δ(y) in the
      candidate node lists such that x and y
      have the maximum number of common
      adjacent node pairs;
6.    if (β(x) ∧ β(y)=0) break;
7.    if φ(self − loop(x)) = φ(self − loop(y))
8.    then Let V_1 = {x};  // V_1 is the node
           set for current subquery graph
           G_1 //
9.         Let V_2 = {y};  // V_2 is the node
           set for current subquery graph
           G_2 //
10.        Let selected(x)=selected(y)=1;
11.      while V_1 and V_2 have unexpanded
         nodes do
12.        Pick the next pair x ∈ V_1 and
           y ∈ V_2 based on the FIFO order
           for expanding;
13.        for each member x_1 with sele-
           cted(x_1)=0 in the ring owned
           by x do
14.          for each member y_1 with
             selected(y_1)=0 in the ring
             owned by y do
15.            if φ (self-loop(x_1))=φ (self-
               loop(y_1))=0
16.            and δ(x_1) = δ(y_1)  and
             φ(edge(x, x_1)) = φ(edge(y, y_1))
17.          and x_1 and y_1 do not violate
             the commonality between
             current subquery graphs G_1
             and G_2
18.            then V_1 = V_1 ∪ {x_1};
19.                 V_2 = V_2 ∪ {y_1};
20.                 selected(x_1)=sele-
                    cted(y_1)=1;
21.                 break;
22.          end if;
23.        end for;
24.      end for;
25.    end while;
26.    Evaluate the resulting pair of com-
       mon subquery graph < G_1, G_2 >;
27.    if it is worth to accept < G_1, G_2 >
       then
28.      Remove any < G'_1, G'_2 > from S_hold
         that shares some nodes with
         < G_1, G_2 >;
29.      S_accept = S_accept ∪ {< G_1, G_2 >};
30.      Adjust bitmaps for the unselected
         nodes that are connected to a node
         in G_1 or G_2;
31.    else if it is worth to hold < G_1, G_2 >
       then
32.      Reset selected(x) = 0 for all x in
         G_1 and G2;
33.      S_hold = S_hold ∪ {< G_1, G_2 >};
34.    else Reset selected(x)=0 for all x
       in G_1 or G2;
35.         Discard < G_1, G_2 >;
36.    end if;
37.    Remove x, y and any node z with
       selected(z)=1 from the candidate
       node lists;
38.    Remove any candidate node list that
       has less than two nodes;
39.  end if;
40. end while;
41. while S_hold ≠ ∅ do
42.   Remove the largest pair < G_1, G_2 >
      from S_hold;
43.   if < G_1, G_2 > does not share any node
      with any pair in S_accept
44.   then S_accept = S_accept ∪ {< G_1, G_2 >};
45. end while;
46. for each common subquery pair < G_1, G_2 >
    ∈ S_accept  do
47.   Optimize G_1 and share the execution
      plan with G_2;
48.   Replace G_1 and G_2 in the original query
      graph G with their (estimated) result
      tables;
49. end for;
50. Optimize the revised G to generate an
    execution plan;
51. return the execution plan for Q, which
    includes the plan for the revised G and
    the plans for all the common subquery
    graphs;
52. end.
```

Lines 4 - 40 search for all pairs of common subquery graphs starting with a pair of nodes chosen from the candidate node lists. Lines 11 - 25 expand the current pair of common subquery graphs using the ring network. Lines 26 - 36 determine if we should accept, hold, or reject

the pair of identified subquery graphs. Lines 41 - 45 accept the remaining pairs of common subquery graphs that are not overlapped with any accepted common subquery graphs. Lines 46 - 50 optimize all common subquery graphs and the revised final query graph. Lines 51 return the execution plan for the given query.

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

3.2 Choosing starting nodes

How to choose a pair of nodes x and y in the candidate node lists as the starting nodes to search for a pair of common subquery graphs G_1 and G_2 is important in reducing the query optimization time. Our goal is to make the subquery graphs as large as possible. The larger the common subquery graphs, the more optimization work can be shared between them, and, therefore, the more time is saved for query optimization.

Unfortunately, we do not know how the subquery graphs will grow from a pair of starting nodes until the subquery graphs are found. Hence, a greedy approach is adopted.

First of all, to be qualified as a pair of starting nodes, x and y must satisfy $\delta(x) = \delta(y)$, i.e., they are in the same candidate node list for a particular table.

To choose a pair of nodes that have a potential to lead to a pair of larger common subquery graphs, we use a heuristic, namely, choose x and y that maximize the size of the set of their common tables. We say that x and y share a common table R if each of them has at least one adjacent node representing R. Note that one node may have a different number of adjacent nodes representing R than the other. The more pairs of adjacent nodes representing the same table, the greater the chance the pair of common subqueries will grow larger. The bitmaps $\beta()$ associated with the nodes help to implement this heuristic efficiently. More specifically, we only need to compute a bitwise AND operation: $z = \beta(x) \wedge \beta(y)$ (e.g., $0110 \wedge 1100 = 0100$). Then $count1(z)$ tells us how many common tables x and y share. Clearly, if $count1(z) = 0$, the algorithm should exit the loop (i.e., Line 6) since no more common subquery graphs (with more than one table) can be found in the graph.

It is possible that there is more than one pair of nodes in the candidate node lists that share the same maximum number of common tables. In such a case, we use a secondary heuristic to choose which pair of nodes to start, namely, choose the pair of nodes x and y with $count1(\beta(x) \oplus \beta(y))$ is minimum, where $\oplus$ is a bit-wise exclusive OR operation. $count1(\beta(x) \oplus \beta(y))$ (e.g., $0110 \oplus 1100 = 1010$) indicates how many different tables x and y have. This heuristic tries to choose a pair of nodes that has a good chance to lead to a pair of common subqueries with the minimum number of interconnected edges (with the rest of the query). Hence, the common subqueries can be optimized independently without worrying too much about the rest of the query. If there is still a tie, a pair can be chosen randomly.

Note that any node that has been chosen as a starting node or is in an accepted common subquery graph should not be considered as a candidate starting node. It is, therefore, removed from the candidate node lists.

3.3 Searching for common subquery graphs

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

To find the pairs of common adjacent nodes for the current pair of x and y, a nested-loop method is used to check the two rings owned by the current pair of common nodes. Note that a pair of adjacent nodes x_1 and y_1 are considered to be common if (1) they are not in any previously selected common subquery graphs (i.e., Line 14); (2) they represent the same table and have the same join condition with the current pair of common nodes (i.e., Line 15, 16); and (3) including them does not make the current

pair of subquery graphs become not common (i.e., Line 17).

The first two conditions are easy to check. Caution needs to be taken when checking condition (3). Notice that condition (2) cannot guarantee that x and y can be added into G_1 and G_2 respectively to still keep G_1 and G_2 common. See the example in Figure 7. Assume that x_2 and y_2 have been included in the two subgraphs. When we check the pair of x_1 and

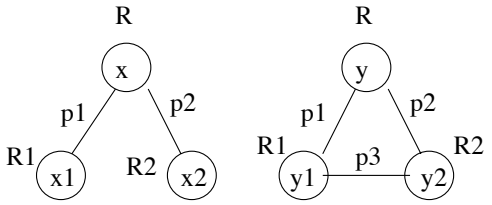

Figure 7: Example of pair of subquery graphs that are not common

y_1, they satisfy conditions (1) and (2). Adding them into the subgraphs would make the subgraphs become not common. Hence, condition (3) has to be checked.

Let V_1 and V_2 be the current set of nodes selected for G_1 and G_2, respectively. Condition (3) is violated by a pair of adjacent nodes x_1 and y_1 when at least one of the following two cases occurs:

(i) There exist a pair of $x_2 \in V_1$ and $y_2 \in V_2$ such that $edge(x_1, x_2)$ exists, but $edge(y_1, y_2)$ does not, and vice versa.

(ii) There exist a pair of $x_2 \in V_1$ and $y_2 \in V_2$ such that there are $edge(x_1, x_2)$ and $edge(y_1, y_2)$, but $\varphi(edge(x_1, x_2)) \neq \varphi(edge(y_1, y_2))$.

Hence, cases (i) and (ii) must be checked to ensure condition (3) is met when a new pair of x_1 and y_1 is considered to be added into common subquery graphs G_1 and G_2.

3.4 Selecting common subquery graphs

After we find a pair of common subquery subgraphs and decide to use them in our query optimization, those nodes in the graphs are no longer able to participate in other common subquery graphs. Hence, we should remove them from the query graph (by setting their 'selected' flag to 1) and the candidate node lists. On the other hand, if we decide not to use them, their nodes can be used by other common subquery graphs.

If a pair of common subquery graphs are very small (in terms of the number of nodes in each graph), e.g., containing only 2 - 3 nodes, it is most likely that not much optimization work can be shared by them. If we use them for our optimization, their nodes may not be able to participate in other possibly larger common subquery graphs. In such a case, it is better not to use them (Lines 34-35).

On the other hand, if a pair of common subquery graphs are large, it is most likely that much optimization work only needs to be done once and be shared by the subquery graphs. In this case, we should accept the common subquery graphs for our optimization (Lines 27-30).

If a pair of common subquery graphs are marginal, it is hard to decide whether to use them in our optimization or not. In this case, we hold the common subquery graphs, but allow other common subquery graphs to use their nodes. If we find a pair of larger common subquery graphs using some of the nodes, we use the new common subqueries and discard the held ones (Line 28). Otherwise, we will still use the held common subquery graphs (Lines 41 - 45).

To determine whether to accept, reject, or hold a pair of common 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 common 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 common subquery graphs:

- If $n \geq c_2$, then the new pair of common 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 via experiments. Note that our QOBCS algorithm can also incorporate other evaluation methods for selecting common subquery graphs.

3.5 Optimizing the query

As described in QOBCS algorithm, after all common subquery subgraphs are selected, we can apply any efficient optimization algorithm, such as II, or AB, to optimize a common subquery graph of each pair and the final query graph with common subquery graphs replaced by their result tables. Since the complexities of the common subqueries and the revised final query are less than that of the original query, these query optimization techniques usually perform very well. If a common 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.

3.6 Time Complexity Analysis

Let n be the number of tables instances (i.e., nodes in RNR) in given query Q. Based on the description of QOBCS algorithm, we can see that:

- Line 2 requires at most $O(n)$ time.
- Loop 4-40 loops at most $O(n^2)$ time.
- Line 5 requires at most $O(n^2)$ time.
- Lines 3, 6-10 require $O(1)$ time.
- Loop 11-25 loops at most $O(n)$ time.
- Lines 12-24 require at most $O(n^2)$ time.
- Lines 26-38 require at most $O(n^2)$ time.
- Lines 4-40 together require at most $O(n^4)$ time.
- Lines 41-45 require at most $O(n^2)$ time.
- Lines 46-49 require at most $O(T(n))$ time, where $T(n)$ is the complexity of the optimization technique used to optimize the common subqueries and the revised final query.

In summary, the time complexity for QOBCS algorithm is $O(max\{n^4, T(n)\})$. As mentioned before, unless n is less than a small constant, we employ an efficient polynomial-time technique such as II with our algorithm. Therefore, our technique is of polynomial time complexity.

4 Experiments

In Section 3.6, we have shown that QOBCS is an efficient polynomial time technique, which is suitable for being used to optimize 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 since it takes some complex query characteristics into account. To check the quality of the execution plans generated by our technique, we have conducted some experiments.

In the experiments, we chose to compare our QOBCS algorithm with two other popular techniques that can also handle complex queries, i.e., the Iterative Improvement algorithm (II) and a heuristic-based technique using the rule "always join two smallest tables first" (JSF). The former is a randomization technique that often generates better query plans than other randomization techniques. The latter employs a popular simple heuristic. Both are of polynomial time complexity. To make a fair comparison, we employ II algorithm with our QOBCS technique for optimizing common subqueries and the revised final query. The join access method used to perform each join in a query for all three techniques is the nested-loop join algorithm.

The experiments were done on an IBM PC with Pentium III 1 G and 512 MB memory, running Linux OS kernel 2.4.7-10. All techniques were implemented in C.

Experimental data was randomly generated. We use a program to randomly generate the following parameters for each test query: the number of different tables (between 10-30), the number of table instances for each table (between 1-8, with 1's occurring more often), and the join relationships between every two table

instances.

Let n be the number of nodes in a query graph. In the experiments, the following threshold values were used to accept a pair of identified common subquery graphs (see Section 3.4):

- If $n < 25$, we set $c_1 = c_2 = 3$. That is, common subquery graphs with $>= 3$ nodes are accepted.
- If $n >= 25$, we set $c_1 = 3$ and $c_2 = 4$. That is, common subquery graphs with $>= 4$ are accepted; common subquery graphs with < 3 are rejected, and common 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.

From the experimental results, we can see that QOBCS has the best performance of the three techniques. The execution plans generated by QOBCS reduce 1.3 $\sim$ 184.2 times of the costs of the ones generated by II. On average, our QOBCS technique is about 33.1 times better than the popular II method. Although II can improve its results by taking a longer time to try more random plans, our experiments showed that with the same number of tries our QOBCS technique is always superior to the II method. This observation verifies that making use of the characteristics (e.g., the structure information) of a complex query can significantly improve the quality of the execution plan.

The JSF method always gave much worse performance for all test queries. This shows that a simple heuristic rule that works fine for simple queries but does not take the characteristics of a complex query into consideration can perform very poorly for such a query.

We also noticed from our experiments that usually, the more common subquery graphs a query graph has and the higher the quality (e.g., larger size and less interconnected edges) of common subquery graph is, the better our QOBCS technique performs. For example, the improvement of QOBCS over II for query no.2 in the experiment is much better than others. The main reason for that is, there is relatively more nodes in the common subquery pairs in query no.2 than in other query samples.

5 Conclusions

As database technology being 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 generate a poor execution plan (e.g., simple heuristics). Although some randomization-based techniques (e.g., II, SA, AB) 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 (QOBCS) exploiting common subqueries for optimizing complex queries. The key idea is to use an efficient ring network representation to represent a complex query, search for common subquery graphs, optimize each representative subquery, share the optimization result with other common subqueries, reduce the original query by replacing each common subquery with its result, and then optimize the revised final query. Any efficient technique such as II can be used together with our QOBCS technique for optimizing identified common subqueries. It has been shown that QOBCS is an efficient polynomial time technique, and furthermore, that it usually generates better execution plans for complex queries, since it takes the structure information of a query into account, compared with other polynomial time algorithms. 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 QOBCS technique is quite promising in optimizing complex queries. It outperforms the popular II technique by 33.1 times on aver-

query no.	# of table instances	# of pairs of CSGs	CSG avg size (nodes)	I/O cost of QOBCS	I/O cost of II (times)	I/O cost of JSF (times)	improvement of QOBCS over II	improvement of QOBCS over JSF
1	27	1	4	0.3622e+7	0.4172e+8	0.4030e+10	0.1150e+2	0.1113e+4
2	30	2	5.5	0.3681e+8	0.6781e+10	0.5168e+15	0.1842e+3	0.1404e+8
3	37	1	5	0.1224e+13	0.4229e+13	0.1436e+19	0.3500e+1	0.1173e+7
4	39	2	4	0.1998e+15	0.2554e+15	0.6192e+21	0.1300e+1	0.3099e+7
5	47	2	4	0.3347e+16	0.3584e+17	0.1902e+24	0.1070e+2	0.5683e+8
6	57	2	5	0.4156e+19	0.4838e+20	0.5058e+28	0.1160e+2	0.1217e+10
7	61	2	4	0.2154e+17	0.1990e+18	0.2100e+29	0.9200e+1	0.9750e+12

Table 1: Comparison of I/O costs for execution plans generated from three techniques

age and improves even much more for a simple heuristic 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.

Acknowledgments

The authors are grateful to Berni Schiefer for his valuable suggestions and insights for the research issues and work discussed in this paper. We would also like to thank Kelly Lyons, Wing Lau, Joe Wigglesworth for their useful discussions and support for the project.

Trademark Statement

About the Author

Yingying Tao is an M.Sc. student in the Dept of Comp. and Inf. Sci. at The Univ. of Michigan - Dearborn. She is also a research assistant in the department. She received her B.Sc. in Comp. Sci. from Tsinghua Univ. in 2000. Her research interests include query processing and optimization in database systems.

Qiang Zhu is an Associate Professor in the Department of Computer and Information Science at The University of Michigan, Dearborn, MI, USA. He received his Ph.D. in Computer Science from the University of Waterloo in 1995. Dr. Zhu is a principal investigator for a number of database research projects funded by sources including the US National Science Foundation and IBM at The University of Michigan. He has over 40 research publications in various journals and conference proceedings. Some of his research results have been included in several well-known database research/text books. Dr. Zhu has served as a program committee member and session/workshop chair for a number of international conferences. His current research interests include query optimization for advanced database systems, multidatabases, self-managing databases, Web-based database technology, and data mining.

Calisto Zuzarte is a senior technical manager of the SQL Query Rewrite development team at the IBM Toronto laboratory. He has been involved in several projects leading and implementing many features related to the DB2 SQL compiler. His main expertise is in the area of query optimization including cost based optimizer technology and automatic query rewriting for performance. Calisto is also a research assistant associated with the Centre for Advanced Studies (CAS) with the responsibility of database related projects within CAS.

References

[1] K. Bennett, M. C. Ferris, and Y. Ioannidis A genetic algorithm for database query optimization. In *Proceeding of 4th International Conference on Genetic Algorithms*, CA, July 1991, pages 400-407.

[2] Surajit Chaudhuri: An overview of query optimization in relational systems. In *Pro-*

ceeding of ACM PODS 1998, pages 34-43, Seattle, 1998.

[3] G. Graefe: Query Evaluation Techniques for Large Databases. In *ACM Computing Surveys*, 25(2) 111-152, June 1993.

[4] Toshihide Ibarake and Tiko Kameda: On the optimal nesting order for computing N-relational joins. In *ACM Transactions on Database Systems*, 9(3) 482-502, September 1984.

[5] Y. E. Ioannidis and E. Wong: Query Optimization by Simulated Annealing. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 9-22, 1987.

[6] M. Jarke and J. Koch: Query Optimization in Database Systems. In *ACM Computing Surveys*, 16(2) 111-152, June 1984.

[7] Maciej Matysiak: Efficient Optimization of Large Join Queries Using Tabu Search. In *Information Science*, v83, n1-2, Mar 1995.

[8] P. G. Selinger, M. M. Astrahan, D. D. Chanmberling, R. A. Lorie, and T. G. Price: Access Path Selection in a Relational Database Management System. In *Proceedings of ACM-SIGMOD International Conference on Management of Data*, pages 23-34, 1979.

[9] A. Swami and B. R. Iyer: A polynomial time algorithm for optimizing join queries. In *Proceeding of the 9th IEEE Conference on Data Engineering*, Vienna, Austria, 1993. pages 345-354.

[10] A. Swami and A. Gupta: Optimization of Large Join Queries. In *Proceeding of ACM-SIGMOD International Conference on Management of Data*, pages 8-17, 1988.