Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences

Tobias Kraft Holger Schwarz Ralf Rantzau Bernhard Mitschang

University of Stuttgart
Department of Computer Science, Electrical Engineering and Information Technology
Universitätsstraße 38, 70569 Stuttgart
Germany
{*firstname.lastname*}@informatik.uni-stuttgart.de

Abstract

Relational OLAP tools and other database applications generate sequences of SQL statements that are sent to the database server as result of a single information request provided by a user. Unfortunately, these sequences cannot be processed efficiently by current database systems because they typically optimize and process each statement in isolation. We propose a practical approach for this optimization problem, called "coarse-grained optimization," complementing the conventional query optimization phase. This new approach exploits the fact that statements of a sequence are correlated since they belong to the same information request. A lightweight heuristic optimizer modifies a given statement sequence using a small set of rewrite rules. Since the optimizer is part of a separate system layer, it is independent of but can be tuned to a specific underlying database system. We discuss implementation details and demonstrate that our approach leads to significant performance improvements.

1 Introduction

Query generators are embedded in many applications, such as information retrieval systems, search engines, and business intelligence tools. Some of these applications, in particular ROLAP tools, produce more than one query as a result of an information request that is defined by a user interacting with a graphical user interface, as illustrated in Figure 1. Typically, the query generators produce a *sequence* of statements for the sake of reduced query complexity although the statements could often be merged into a single large query. All but the last statement of the sequence produce intermediate results as temporary database objects, i.e., they either create or drop tables and views, or insert data into the newly created tables. The last INSERT statement of a sequence defines the final result, which is delivered to the ROLAP server and, after some optional reformatting, delivered to the client that visualizes the result. Since statements that appear later in the sequence refer to intermediate results produced earlier, the sequence can be considered a connected directed acyclic graph (DAG) with the final INSERT statement as the root.

Statements produced by query generators are typically tuned to a certain target database system. However, the response time of such an information request is often far from optimal. Improving the query generators is not a viable option because they depend on the application, the structure of the associated database as well as on the underlying database system. Alternatively, one can rewrite the statement sequence into an alternative sequence of one or more statements such that far less resources are consumed by the database system than for the original, equivalent sequence. In this paper, we refer to this approach of rewriting statement sequences as *coarse-grained optimization (CGO)*. The *distinct feature* of this multi-statement optimization problem is that the equivalence only refers to the result of the *last* statement in the sequence, i.e., the result table. The intermediate results (either views or tables) may differ completely from one sequence to another.

This work is motivated by experiments we conducted with the MicroStrategy DSS tool suite. We observed that the execution time of the SQL state-

Proceedings of the 29th VLDB Conference, Berlin, Germany, 2003

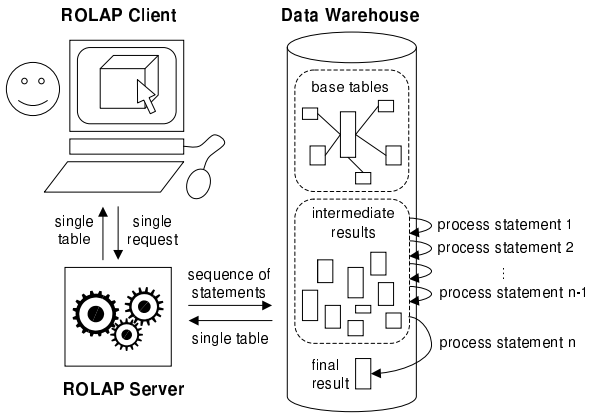


Figure 1: The typical ROLAP scenario

ment sequences generated by the tool were often far from optimal, so we analyzed the statement sequences more closely and achieved astonishing improvements through few manual rewrites [13]. The main conclusion we drew is that merely rewriting the statement sequence into a single query was not always the best solution. The rewritten sequence, which has prescribed materialization points (CREATE TABLE statements), performed better than both a single query (when the database system was free to decide on materialization points itself) and the original sequence. This paper extends that work significantly by identifying a more effective ruleset, automating the rewrite phase by a tool, and conducting in-depth performance experiments.

Outline

The rest of this paper is organized as follows: Related work is discussed in Section 2. In Section 3, we present the concept of statement sequences in greater detail before we introduce coarse-grained optimization in Section 4. The heuristic rules used in our approach as well as the internal representation of statement sequences are explained in Sections 5 and 6, respectively. Section 7 discusses the implementation of our prototype that was employed for performance experiments, presented in Section 8. Section 9 concludes this paper and comments on directions for future work.

2 Related Work

The special problem of optimizing SQL statement sequences is related to both conventional (single) query optimization and multi-query optimization, to be discussed briefly in the following.

Multi-query optimization (MQO) tries to recognize the opportunities of shared computation by detecting common inter- and intra-query subexpressions. Furthermore, MQO requires to modify the optimizer search strategy to explicitly account for shared computation and find a globally optimal plan. This includes deciding which subexpressions should be materialized, how they should be materialized (w.r.t. sort order), and what indexes should be created on these intermediate results. This is a very costly task and it is considered infeasible to employ exhaustive algorithms [12].

Multi-query optimization has a long history of research [4], see for example [14] for an overview. The cardinal problem of MQO, finding common subexpressions within a batch of queries, has been investigated, e.g., in [12]. The authors propose three cost-based heuristic algorithms that operate on an AND-OR DAG that has a pseudo root, which has edges to each query of the batch. One of the optimizer rules proposed is *unification*: Whenever the algorithm finds two subexpressions that are logically equivalent but syntactically different (see Section 2.1 of that paper), then it unifies the nodes, creating a single equivalence node in the DAG. Another strategy is *subsumption*: Given a number of selections on a common expression, create a single new node representing a disjunction of all the selection conditions. The performance experiments presented in that paper employ TPC-D query batches. For their experiments using Microsoft SQL Server, they transformed the plans generated by their multi-query optimizer back into SQL queries. They created, populated and deleted temporary tables, and created indexes on these tables according to the decisions of their algorithm. Unfortunately, there are no execution times given for running optimized query batches on SQL Server. They provide such numbers only for single queries as input that have been transformed into batches by materializing common intra-query subexpressions. This retranslation of queries into SQL was supposedly done because they were not able to use a DBMS interface for query plans. Our approach also transforms statements back to SQL.

In [3], the work of [12] is extended by a greedy heuristic for finding good plans that involve pipelining. The key idea is that multiple usages of a result can share a scan on the result of a subexpression. In particular, if *all* usages of the result can share a scan then the result does not need to be materialized.

Notice that CGO differs from MQO in that the optimization of statement sequences aims at optimizing the collection of SQL statements as if it was a *single* query, with the last statement in the sequence as the outermost query. It does not require that the intermediate queries (used for populating temporary tables) are actually computed as specified. It is sufficient that the *final* query delivers the required result. Hence, CGO allows additional ways of rewriting a given statement sequence compared to MQO.

Except for techniques to derive a set of materialized views for a given workload [5, 17, 18], we do not know of any viable multi-query optimization technique

available in state-of-the-art DBMS. The materialized view design does not cover the full range of multi-query optimization, it only deals with a true subproblem: finding common subexpressions that are worthwhile to materialize in order to support for a set of subsequent queries.

Instead of using MQO, we could employ *conventional single-query optimization (SQO)* for our problem. This is based on the fact that a statement sequence can be expressed by a single query, as we will show in Section 3. SQO searches for a plan for a single query that is cheaper than the total cost for the equivalent statement sequence. Hence, one can argue that statement sequence processing is actually an optimization problem involving a single, potentially very large query. In order to cope with complex queries involving many joins randomized and heuristic optimization techniques have been studied [8, 15]. However, our performance experiments, summarized in Section 8, show that commercial optimizers were not able to find an execution strategy for single-queries that was nearly as good as for several improved equivalent statement sequences, including the original sequence.

To the best of our knowledge there is no previous work combining rewrite rules in a way similar to our CGO approach. However, our ruleset consists of rules that are at least to some extent contained in optimization algorithms used for certain prototype multi-query optimizers or conventional single-query optimizers [1, 7, 10, 12]. We discuss the relationship of our ruleset with rules known from the literature in Section 5.

3 The Query Dependency Graph

Statement sequences are the result of information requests. We define an *information request* as an interaction of a user with the system typically by means of a graphical user interface. It consists of the specification of the data and processing needed to derive the desired information as well as its presentation style. Applications generate an entire statement sequence although it would be possible to represent the information request by a single SQL statement. There are several reasons for query generators to follow this approach:

- Since a collection of individual statements is typically less complex than a single large query it is possible to run these statements even on database systems that do not support the latest SQL standard. This reduces the number of special cases that have to be treated in the query generation process.
- It allows to keep the process of query generation and query verification simple.

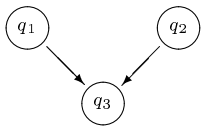
Figure 2(a) shows an example sequence S that consists of eight statements. In this paper, we focus on the following types of statements:

```
c1: CREATE TABLE q1 (custkey INTEGER, turnover1990 FLOAT);

i1: INSERT INTO q1
     SELECT o.custkey, SUM(o.totalprice)
     FROM   orders o
     WHERE  o.orderdate BETWEEN '1990-01-01'
                            AND '1990-12-31'
     GROUP BY o.custkey;

c2: CREATE TABLE q2 (custkey INTEGER, turnover1991 FLOAT);

i2: INSERT INTO q2
     SELECT o.custkey, SUM(o.totalprice)
     FROM   orders o
     WHERE  o.orderdate BETWEEN '1991-01-01'
                            AND '1991-12-31'
     GROUP BY o.custkey;

c3: CREATE TABLE q3 (custkey INTEGER, name VARCHAR(25));

i3: INSERT INTO q3
     SELECT c.custkey, c.name
     FROM   q1, q2, customer c
     WHERE  q1.custkey = c.custkey
       AND  q1.custkey = q2.custkey
       AND  q2.turnover1991 > q1.turnover1990;

d1: DROP TABLE q1;

d2: DROP TABLE q2;
```

(a) SQL statement sequence

$q_1 = (c_1, i_1, d_1)$
$q_2 = (c_2, i_2, d_2)$
$q_3 = (c_3, i_3, -)$

(b) Statement triples



(c) Query dependency graph

Figure 2: Representations of statement sequence S

- CREATE TABLE statements c_j that create tables to hold intermediate results or the final result of an information request.
- INSERT statements i_j that compute the intermediate results or the final result and insert them into tables. There is exactly one INSERT statement for each table created by a statement sequence. Its body is a *query* that may access base tables as well as any intermediate result of the sequence.
- DROP TABLE statements d_j that remove intermediate result tables. The final result table is not dropped as part of the sequence because it has to be retrieved by the application that generated the sequence.

We can identify statement triples $q_j = (c_j, i_j, d_j)$ within the sequence that consist of a CREATE TABLE, INSERT and DROP TABLE statement regarding the same table, as shown in Figure 2(b). A sequence that consists of k statement triples includes $n = 3 \cdot k - 1$ statements. There are less than $3 \cdot k$

statements because there is no DROP TABLE statement for the last statement triple which provides the result of the entire information request. The sequential dependencies between the INSERT statements i_j define a partial order on the statement triples. If we consider triples as nodes q_j and draw an edge from node q_m to node q_n if and only if the query expression of INSERT statement i_n refers to the table created by c_m and filled by i_m, we obtain a connected directed acyclic graph called *query dependency graph (QDG)*. It expresses the data flow and the direct sequential dependencies among the INSERT statements of a sequence. The corresponding QDG for statement sequence S is illustrated in Figure 2(c). A one-to-many relationship holds between QDGs and statement sequences, i.e., there are multiple correct and logically equivalent sequential orders of statements for a single QDG. We define two statement sequences to be equivalent if they answer the same information request, i.e., the content of the table that stores the result of the entire information request is the same for both sequences.

At the moment we restrict our approach to a subset of SQL-92 that is limited to *queries* without subqueries and set operations. This and the above specification of a query sequence do not impose a severe restriction because our experience has shown that most generated sequences adhere to these requirements. Hence, UPDATE, DELETE and additional INSERT statements are not in our focus yet. But we plan to extend our specifications and adapt our rewrite rules.

There are two alternative SQL representations providing the same information as the statement sequence within a single query. The first one replaces the references to temporary tables in i_k by subqueries containing the definition of the respective temporary tables. Figure 3 shows the resulting query for statement sequence S. One has to repeat this step recursively until i_k includes references to base tables only. This may result in deeply nested FROM clauses. The second option uses the WITH clause of SQL:1999 [9] to define all temporary tables before referring to them in the body of the query expression, which consists of i_k. This approach is illustrated in Figure 4. Note, that both options have several drawbacks. The first option adds much complexity to the process of query generation and query optimization because the entire information request has to be represented by a single, probably deeply nested query. One important drawback of the second option is that the WITH clause is not supported by all commercial database systems. Our experimental results, discussed in Section 8, show that the statement sequence was superior to the corresponding single query based on subqueries in most cases. Single queries based on the WITH clause performed even worse and did not finish in acceptable time for several queries. We conclude that current optimizer technology is not able to provide efficient query

```
CREATE TABLE q3 (custkey INTEGER, name VARCHAR(25));

INSERT INTO q3
 SELECT c.custkey, c.name
 FROM
       (SELECT o.custkey, SUM(o.totalprice)
        FROM   orders o
        WHERE  o.orderdate BETWEEN '1990-01-01'
                               AND '1990-12-31'
        GROUP BY o.custkey) AS q1 (custkey, turnover1990),
       (SELECT o.custkey, SUM(o.totalprice)
        FROM   orders o
        WHERE  o.orderdate BETWEEN '1991-01-01'
                               AND '1991-12-31'
        GROUP BY o.custkey) AS q2 (custkey, turnover1991),
       customer c
 WHERE  q1.custkey = c.custkey
   AND  q1.custkey = q2.custkey
   AND  q2.turnover1991 > q1.turnover1990;
```

Figure 3: Single query for statement sequence S using subqueries.

```
CREATE TABLE q3 (custkey INTEGER, name VARCHAR(25));

INSERT INTO q3
  WITH
    q1 (custkey, turnover1990) AS
      (SELECT o.custkey, SUM(o.totalprice)
       FROM   orders o
       WHERE  o.orderdate BETWEEN '1990-01-01'
                              AND '1990-12-31'
       GROUP BY o.custkey),
    q2 (custkey, turnover1991) AS
      (SELECT o.custkey, SUM(o.totalprice)
       FROM   orders o
       WHERE  o.orderdate BETWEEN '1991-01-01'
                              AND '1991-12-31'
       GROUP BY o.custkey)
  SELECT c.custkey, c.name
  FROM   q1, q2, customer c
  WHERE  q1.custkey = c.custkey
    AND  q1.custkey = q2.custkey
    AND  q2.turnover1991 > q1.turnover1990;
```

Figure 4: Single query for statement sequence S using a WITH clause.

execution plans for complex queries that represent an information request in a single SQL query.

4 Coarse-Grained Optimization

In today's commercial database systems each SQL statement of a statement sequence is optimized and executed separately. Relationships with other statements of the same sequence are not considered though dependencies and similarities among these statements offer great potential for optimization. It is possible, e.g., to combine similar statements into a single one, or to move predicates from one statement to a dependent one. These optimizations are similar to rewrite rules used in conventional single query optimizers as well as in multi-query optimization. In our approach of coarse-grained optimization (CGO), we adopt and combine few but effective heuristic rewrite rules in a ruleset providing an additional step of query rewriting, which complements the optimization steps of the underlying database system. The key idea of CGO is to produce several SQL statements that are less complex for a single-query optimizer compared to a single

merged query. It produces a new equivalent sequence of one or more statements that is most likely to be executed by the database system in less time than the original sequence.

Compared to conventional optimizers, CGO works on a more abstract level and allows to apply rewrite rules independent of the underlying database system. This level of abstraction has an impact on the internal representation of statement sequences as well as on the cost estimations that are available to guide the application of rewrite rules.

The rewrite rules that we will introduce in Section 5 refer to clauses of SQL statements. As we will show in Section 6, the rules operate on a simple "coarse-grained" data structure similar to the query graph model (QGM) of Starburst [10] that represents the statements (logical operations), not a "fine-grained" algebra tree as in SQO or MQO (physical operations). This representation of statement sequences allows to specify *what* the result should be like and *not how* it is computed.

Conventional optimizers need cost estimations in order to decide on the best plan for a given query. Detailed cost estimation is almost impossible for a CGO optimizer, i.e., cost estimations derived from operators, algorithms implementing these operators and physical properties of the data are not known on this level. This is a serious problem for the development of a rule engine for CGO. One way to cope with this problem is to use heuristics that control the application of rewrite rules. Heuristics of a CGO optimizer could be based on characteristics of the ruleset, on characteristics of the statement sequences as well as on characteristics of the underlying database. Our prototype is based on the first category of characteristics. The ruleset and its properties are described in the following section.

There is one possible extension to this purely heuristic approach that deals with cost estimations. The optimizer of the underlying database system could provide cost estimations for every statement sequence that the CGO optimizer produces during its rewrite process. In cases where several rules are applicable to a given statement sequence the decision on the next rule to be applied could be based on these cost estimations. This approach has several drawbacks: First, there is no standardized interface to force the database system to calculate the costs for a given query. Second, in a conventional database system there is no interface to simulate what-if scenarios as created by such statement sequences, i.e. there is no way to get the cost estimates for batches where tables are created and immediately used inside the same batch. This would be necessary to provide cost estimations for queries based on temporary tables. Moreover, statistics would not be available in advance for these temporary tables and default values would have to be used instead. Third, this approach is inconsistent with our objective of an optimizer that is independent of the underlying database system.

5 A Ruleset for Coarse Grained Optimization

This section is divided into two parts. First, we present the ruleset used for the CGO prototype. Second, we explain important properties of this ruleset.

5.1 Classes of Rewrite Rules

Each rule consists of two parts, condition and action. If the rule condition is satisfied the action can be applied to the affected nodes of the QDG. We identified three classes of rewrite rules:

1. Rules that are based on the similarity among a subset of the nodes of a QDG. The rule condition specifies in which components the queries of these nodes have to be equal and in which they may or have to differ. This class comprises the rules *MergeSelect*, *MergeWhere*, *MergeHaving*, and *WhereToGroup*.

2. Rules that are based on dependencies among nodes of a QDG. The rule condition specifies the requirements to be met by a subgraph of dependent nodes. The class includes the rules *ConcatQueries*, *PredicatePushdown*, and *EliminateUnusedAttributes*.

3. Rules that are restricted to the context of a single node in the QDG, including *EliminateRedundantReferences* and *EliminateRedundantAttributes*.

Some of these rules are based on rules that were proposed for conventional and multi-query optimizers. Our ruleset is adjusted to the specific needs of query sequence rewriting, i.e., the focus is on rules that cope with similar or dependent queries. In some cases we combine rules known from the literature. For example, several rules of class 1 merge clauses of two SQL statements by combining *unification* and *subsumption* introduced in [12]. *WhereToGroup*, another class 1 rule, is similar to the push-down rules for duplicate-insensitive generalized projections, described in [7]. The *ConcatQueries* rule is similar to view expansion and it is equivalent to the *Selmerge* rewrite rule used in Starburst [10] to merge a query and a sub-query in its FROM clause. However, *Selmerge* is limited to SPJ (select-project-join) queries, whereas we consider grouping and aggregation, too. Hence, *ConcatQueries* is a generalization of the *Selmerge* rule.

Figure 5 shows an example of the *WhereToGroup* rule. The left part of the figure shows a subgraph of a QDG that consists of four nodes. Due to lack of space only the INSERT statements are shown for each node.

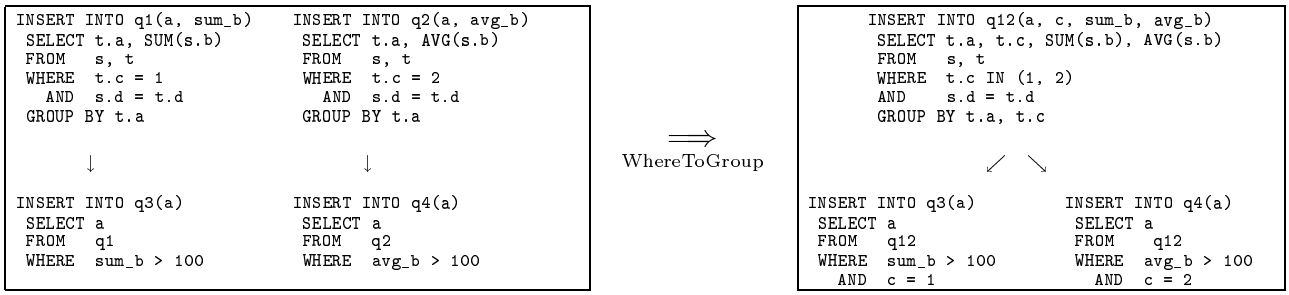


Figure 5: *WhereToGroup* rule example

Dependencies are represented by arrows in the QDG, i.e., node q_3 depends on node q_1 and q_4 depends on q_2. The new QDG after applying the *WhereToGroup* rule is shown on the right side of Figure 5.

The rule condition of *WhereToGroup* specifies a set of nodes whose queries differ in the WHERE clause and optionally differ in the SELECT clause but match in all other clauses. The queries have to include a GROUP BY clause but none of the queries may calculate the final result of the sequence. The rule condition also specifies that all queries have to match in the same predicates and differ in exactly one predicate of the form *attribute = constant*. The constant in this predicate may be different for each query but the attribute in this predicate is identical for all queries and it must not appear in any of the aggregate terms of the SELECT and HAVING clause. In our example, this condition holds for queries q_1 and q_2.

In the rule action a new node is created that replaces all nodes matching the rule condition. Accordingly, all references to these nodes must be adapted to the new node. The query of the new node, q_{12} in our example, contains the clauses and the predicates that are common to the queries of all nodes. Additionally, it contains a predicate of the form *attribute IN setOfConstants*, where the attribute is the one mentioned in the description of the rule condition and the set of constants is a collection of the appropriate constants. The attribute must also be added to the GROUP BY clause, the SELECT clause and the primary key. The SELECT clause of the query of the new node is built by appending the SELECT clauses of the queries of all affected nodes and eliminating duplicate expressions in the resulting SELECT clause. The predicates in which the selected queries differ have to be added to the appropriate referencing queries for each occurring reference. Hence, $c = 1$ is appended to q_3 and $c = 2$ is added to q_4.

As can be seen from Figure 5, this rule reduces the set of nodes in a QDG by unifying previously unrelated parts of a sequence. Joins that had to be executed for q_1 as well as for q_2 only have to be processed once for the new query q_{12}.

The *ConcatQueries* rule is an example of a class 2 rule. If a node q_1 is referred to by exactly one other node q_2 in the QDG, *ConcatQueries* allows to merge q_1 and q_2. In the following, s_i denotes the SELECT clause of the query of node q_i, f_i the FROM clause, w_i the WHERE clause, g_i the GROUP BY clause, and h_i the HAVING clause. An example scenario is illustrated in Figure 6.

In the rule condition we can distinguish between four cases:

1. $g_1 = \emptyset \ \wedge \ DISTINCT \notin s_1$.
2. $g_1 = \emptyset \ \wedge \ DISTINCT \in s_1 \ \wedge$
   $g_2 = \emptyset \ \wedge \ DISTINCT \in s_2$
3. $g_1 \neq \emptyset \ \wedge \ g_1 \subseteq s_2 \ \wedge$
   $g_2 = \emptyset \ \wedge \ f_2$ contains only a single reference
4. $g_1 \neq \emptyset \ \wedge \ g_1 \not\subseteq s_2 \ \wedge$
   $g_2 = \emptyset \ \wedge \ f_2$ contains only a single reference $\wedge$
   q_2 does not store the final result of the sequence

If one of these conditions is met, the rule can be fired. In the rule action the reference to q_1 is removed from f_2 and the elements of f_1 are appended to f_2. When case 1 or 2 occurs, q_1 has no GROUP BY clause and therefore we just have to add the predicates of w_1 to w_2. In case 3 and 4, q_2 has the function of a filter that simply selects rows of the result table of q_1. Hence, g_1 and h_1 have to be added to the query of node q_2; they become g_2 and h_2. Then the elements of w_2 are added to this new HAVING clause of q_2 and w_2 is replaced by w_1. In short, the old WHERE clause of the query of node q_2 becomes part of the new HAVING clause of the query of node q_2. In addition, the attributes of g_1 have to be added to s_2 in case 4. In any case, when merging the WHERE and HAVING clauses and adding attributes to the SELECT clause, duplicate elements can be detected and eliminated.

EliminateRedundantReferences is one of the rules of class 3. Its condition searches for queries that refer to the same source multiple times in their FROM clause and that directly or transitively equate the primary keys of these references. One of these references to the same source is still required, but all the others are removed by the rule action. The conditions of the WHERE clause that are affected by this elimination are also removed or adjusted. Hence, this rule eliminates joins when both inputs are the same source and

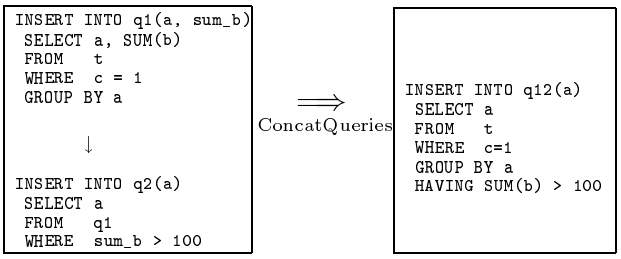


Figure 6: *ConcatQueries* rule example

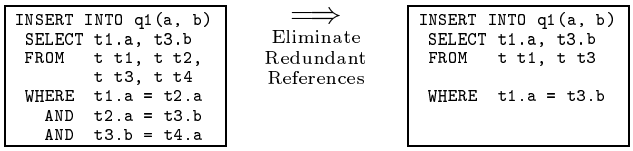


Figure 7: *EliminateRedundantReferences* rule example (attribute a is primary key of table t)

there exists a one-to-one relationship between the attributes of the join condition, because these joins add no new information but produce tuples with redundant fields. Figure 7 gives an example where attribute a is the primary key of table t. The first row of the WHERE clause shows a direct equation, the remaining rows show a transitive equation of the primary keys.

Rules of class 3 can also be found in the underlying database system. We added the *EliminateRedundantReferences* rule to our ruleset because it can initiate other rules to be applied based on the result of its transformation whereas these rules could not be applied to the original, unmodified query. *EliminateRedundantAttributes* is part of the ruleset because it eliminates redundant elements in the select clause of a query and the appropriate columns of the result table of the query. This reduces the storage used by an intermediate result.

5.2 Properties of the Ruleset

Rewrite rules for CGO have to ensure that the original and the rewritten sequence are equivalent. As we will see, this condition holds for our ruleset. Class 1 rules unify n previously unrelated nodes of a QDG into one node. None of these nodes may produce the final result of the whole sequence, i.e., no affected node is the root of the graph. Hence, the final statement of a sequence is not changed by class 1 rules. Other nodes that are dependent on the set of merged nodes produce the same result before and after rewrite processing. Therefore, in Figure 5 the WHERE clauses of queries q_3 and q_4 are transformed by the *WhereToGroup* rule as follows: Query q_3 contains those predicates of q_1 where q_1 differs from q_2, and query q_4 contains those predicates of q_2 where q_2 differs from q_1. The same holds for all other class 1 rules. Thus, this class of rules preserves the final result of a statement sequence. The same fact is also guaranteed for rules in class 2. These rules either do not change the structure of a QDG at all (*PredicatePushdown*) or they merge a subgraph of dependent nodes. In the second case, the *ConcatQueries* rule guarantees that the node that is the result of a merge step, q_{12} in Figure 6, produces the same result as the last query of the subgraph that was processed by *ConcatQueries*. Rules of class 3 also preserve the result of a statement sequence because they only remove redundant references and attributes and do not change the structure of the sequence.

The focus of our CGO approach is on lightweight optimization of statement sequences, i.e., no cost estimations are used to guide the rewrite process. A rewrite rule is applied to a given sequence whenever its condition is true. Hence, we have to guarantee the termination of rewrite processing. The ruleset given in the previous subsection mainly consists of rules that merge nodes and therefore continuously reduce the number of nodes in the QDG. Some of the rules, mainly those in class 3, do not change the structure of the sequence, i.e., they do not add nodes to the QDG. Hence, rewrite processing stops no later than when the QDG is reduced to a single node. Rules of class 3 do not produce loops because they only eliminate redundant information and the ruleset does not contain any rules that could add this redundancy again. Hence, we can conclude from the characteristics of rules in our ruleset that termination of rewrite processing is guaranteed.

6 Internal Representation of Statement Sequences

As discussed in Section 4, one of our key objectives is to develop an optimizer that is as independent of the target database system as possible. Hence, we were free to optimize the internal representation of query sequences w.r.t. our CGO processing. In the following, we present the major concepts of our CGO-XML representation, which is based on a descriptive model of SQL queries and reflects their clause structure similar to a parse tree. Instead of an extensive description we illustrate the CGO-XML representation of a QDG using an example: Figure 8 shows the CGO-XML representation of the QDG that belongs to the statement sequence in Figure 2. Due to lack of space we omit the CGO-XML representation of all but the last statement triple q_3. It consists of a CREATE TABLE and INSERT statement but no DROP TABLE statement since we need to keep the final result of the sequence.

Our XML representation is suitable for coarse-grained optimization for several reasons:

First, the queries are represented on the same descriptive level as the rewrite rules, i.e., we do not have to deal with physical properties of operators and data that are important whenever a fine-grained algebraic representation was chosen.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!DOCTYPE sequence SYSTEM "sequence.dtd">

<sequence>
  ...

  <query name="q3" distinct="no" type="table" result="yes">
    <referenced-by/>
    <select-clause>
      <attribute-definition name="custkey" type="INTEGER">
        <attribute name="custkey" source="c"/>
      </attribute-definition>
      <attribute-definition name="name" type="VARCHAR(25)">
        <attribute name="name" source="c"/>
      </attribute-definition>
    </select-clause>
    <from-clause>
      <source name="q1" alias="q1"/>
      <source name="q2" alias="q2"/>
      <source name="customer" alias="c"/>
    </from-clause>
    <where-clause>
      <equal>
        <attribute name="custkey" source="q1"/>
        <attribute name="custkey" source="c"/>
      </equal>
      <equal>
        <attribute name="custkey" source="q1"/>
        <attribute name="custkey" source="q2"/>
      </equal>
      <greater>
        <attribute name="turnover1991" source="q2"/>
        <attribute name="turnover1990" source="q1"/>
      </greater>
    </where-clause>
  </query>
</sequence>
```

Figure 8: CGO-XML representation of statement sequence S

Second, since we focus on a database independent optimization approach, the result of the transformation process has to be translated back into SQL, which is straightforward with our XML representation.

Third, there are statement sequences where no or only few statements are affected by the rewrite rules, i.e., there are few rule actions manipulating those statements. It is important that all unaffected parts of the original sequence are reproduced without structural modifications because even little modifications in the SQL statements might result in dramatically different execution plans on the target database system. Our internal representation supports this *SQL preserving retranslation*. If we would employ a fine-grained algebra representation, it would be more difficult to achieve this goal. Furthermore, for the sake of readability (for database administrators, for example), it is helpful if the original sequence, which served as input to CGO, and the output sequence look similar.

Several internal representations are proposed in the literature or are used in commercial database systems, but none of the representations we looked at fully met our needs [2, 10]. Therefore we decided to develop our own type of representation based on XML. It meets our requirements, because it is very close to SQL.

A variety of tools are available for transforming XML documents. This helped us to develop the CGO prototype quickly. Choosing an XML representation emphasizes our idea of a lightweight optimizer, because we do not have to pay attention to the physical representation of the XML document in memory and we do not have to implement a special interface to our representation. We were thus able to focus on the implementation of the rewrite rules and the rule engine.

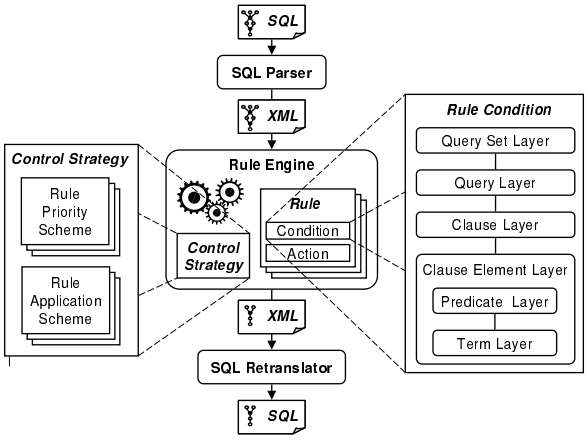


Figure 9: Architecture of CGO

7 Implementation

This section gives an overview on the CGO architecture. We considered the results and experiences of optimizer technology as described for example in [6, 11] and designed a modular CGO prototype. The rule engine and its control strategies are also described in the following sections.

7.1 CGO Architecture

Figure 9 illustrates the optimizer's architecture consisting of three components discussed below: *SQL parser*, *rule engine*, and *SQL retranslator*.

Given an SQL statement sequence, in a first step the SQL parser consumes the entire sequence and translates it into CGO-XML. At the moment, the parser, which is implemented using the parser generator JavaCC, accepts a subset of SQL-92 that is limited to queries without subqueries and set operations. The statement sequence has to comply with the definition in Section 3.

The optimization step that follows parsing is realized by a priority based *rule engine*. It employs the rewrite rules introduced in Section 5, which operate on the internal representation of statement sequences. For navigation and manipulation of CGO-XML we decided to use the document object model (DOM) and Sun's JAXP as the DOM interface implementation.

The final step in the optimization process is the retranslation from CGO-XML into SQL, i.e., both the input and the result of the rewrite process are a sequence of SQL statements. The retranslation component *SQL retranslator* is implemented by the XSLT processor that is part of JAXP.

```
INSERT INTO t1              INSERT INTO t2
  SELECT *                    SELECT *
  FROM   s s1, s s2           FROM   s s1, s s2
  WHERE  s1.a = s2.a          WHERE  s1.a = s2.a
    AND  s1.b > 0               AND  s2.b > 0
```

Figure 10: Matching of FROM clauses: Table correlation names s_1 and s_2 for the insertion into table t_1 match s_2 and s_1 for table t_2, respectively.

Each rule is implemented by a separate Java class, which is derived from a common super class providing two abstract methods that are invoked by the rule engine: a method checking the rule's condition and one realizing the rule's action, respectively. The condition method returns a boolean value that indicates whether parts of the queries in the query dependency graph satisfy the rule condition. During the evaluation of the rule condition information used in the rule action is stored temporarily to avoid that data is computed twice. The action method embodies the transformation of the query sequence.

The rule condition of class 1 rules can be divided into several layers according to the structure of an SQL query and its components. This layer model is shown on the right of Figure 9. Every layer uses the functionality of the layer below. The top layer realizes the search for sets of queries that are similar according to the rule condition. This problem is reduced to a pairwise comparison of the queries of a sequence. The comparison is done by the *query layer*, whose main task is to match the FROM clause elements of a pair of queries. This is nontrivial if the same table appears multiple times in the same query, of course each one having a unique correlation name. An example query pair is depicted in Figure 10. For every possible matching of the FROM clause elements, all the other clauses have to be compared whether they satisfy the rule condition. In the worst case all n correlation names in a FROM clause refer to the same table and $n!$ possible combinations have to be checked. The comparison of the clauses is implemented in the *clause layer*. For every clause type there is a method that compares two clauses of this type, taking into account the special characteristics of that clause type. These methods make use of the methods of the *clause element layer*. In case of a WHERE or HAVING clause the clause element layer can be further divided into the *predicate layer* and the *term layer*.

We were able to share and reuse source code by using class inheritance since some of the rules have several methods in common. The layer model reduces the complexity of the ruleset implementation and facilitates future extensions.

7.2 Control Strategies

Control strategies of the rule engine are based on rule priority schemes and rule application schemes. The *rule priority scheme* defines the order in which rule conditions are evaluated. In our prototye we use a static priority scheme. For this purpose, we have assigned a priority to each rule of the ruleset. The priority assignment is based on experience gained from experiments and corresponds to the effectiveness of the rules. If several rules could be applied to the same sequence, the rule of highest priority is picked. Rules of class 1 have the highest priority whereas class 3 rules have the lowest priority. We have chosen this order because class 1 rules mainly merge similar queries, which eliminates redundant processing and results in remarkable performance improvements according to our experiments.

The priority-based approach described so far works fine provided that each application of rewrite rules results in a reduced execution time for a given statement sequence. This is not true for all rules in our ruleset. In particular, the rules of class 1 may lead to a deterioration of execution times in some special cases. For example, merging nodes q_1 and q_2 in Figure 5 is not always advantageous. In some cases the processing of the modified queries q_3 and q_4 may add more execution time than is saved by the reduced redundancy that is achieved by q_{12}. The application of rules of class 2 and 3 never results in worse execution times except for *ConcatQueries*. The application of *ConcatQueries* merges two dependent queries into a single, but more complex query and therefore increases the complexity of calculating the best execution plan in the underlying database system. This can force the database to use pruning and heuristics that lead to suboptimal execution plans. Complex queries are also more vulnerable to wrong row estimations of intermediate results within the execution plan. Nevertheless, our experimental results in Section 8 show that considerable performance improvements can be achieved with our lightweight approach.

In the modular CGO architecture, the ruleset and the control strategies can easily be modified and extended. There are several ways to extend the control strategy used in our prototype. This includes a *rule application scheme* that comprises a set of rule application patterns. For each rule several patterns may be defined. A pattern describes conditions that have to be met, such that the application of a rewrite rule leads to performance improvements. These conditions could be based on statistics on base tables, on characteristics of the underlying database system as well as on characteristics of a statement sequence and the statements it contains. Some examples of relevant characteristics of statements are the number of joins or the number and type of predicates. Each database system has different features and optimization strategies. Hence, the performance gain of some rules is different depending on the database system that executes the resulting sequence. For example, database systems differ in the

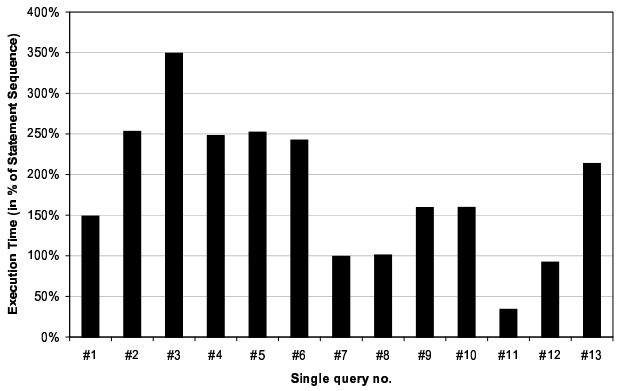


Figure 11: Runtime of single queries compared to their corresponding statement sequences. The statement sequence runtime is equal to 100%.

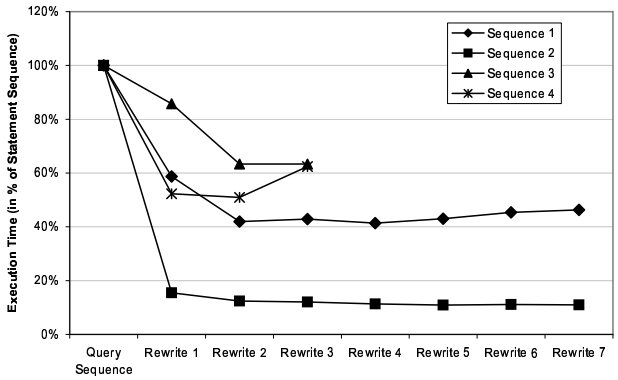


Figure 12: Runtime of different statement sequences after applying several rewrite rules step by step.

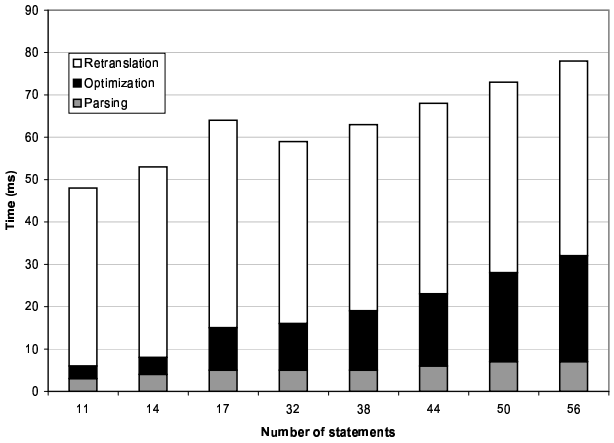


Figure 13: Runtime of CGO optimizer phases.

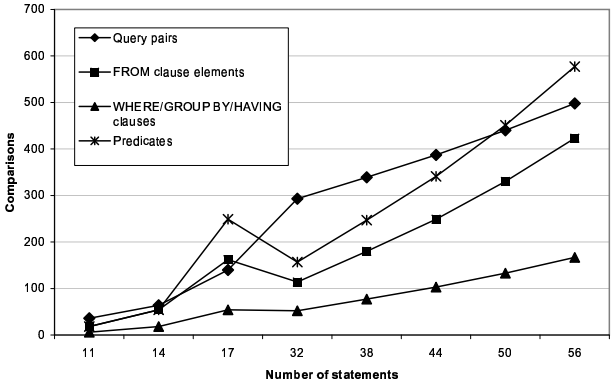


Figure 14: Number of comparisons in class 1 rule layers depending on the statement sequence size.

maximum number of joins that they are able to optimize exhaustively and process efficiently as part of a single statement. In CGO, this knowledge could be used to stop merging statements as soon as any further application of rewrite rules would lead to statements that include more than this maximum number of joins. Furthermore, one could define a dynamic rule priority scheme based on the same characteristics that are used by rule application patterns. Since extended control strategies are not in the focus of this paper, we do not further elaborate on this topic.

8 Experimental Results

In order to show the effectiveness of CGO and its independence of the underlying database system we measured the runtime of statement sequences in two prominent environments. One consists of a SUN Enterprise 4500 with 12 processors and DB2 V7.1 on Solaris. The other system is a 4-processor Dell PowerEdge 6400/900 with 4 CPUs and Microsoft SQL Server 2000 Standard Edition on Microsoft Windows 2000 Server. These two systems are not directly comparable. Therefore, we will not show absolute numbers in this paper. The experimental results presented in Figures 11 and 12 include relative numbers for both platforms. The results are similar in both environments.

We have chosen TPC-H [16] with scaling factor 10 (10 GB) as our benchmark database and the MicroStrategy DSS tool suite as a generator of statement sequences. This required to make some small additions to the TPC-H schema including some new tables representing different time dimensions and several views on the original tables of the TPC-H schema. Our experiments are based on a large set of statement sequences generated for typical information requests in customer relationship mangement, merchandise management and supply chain management. They range from 11 statements per sequence (i.e., 4 triples) to a maximum of 56 statements in a sequence (i.e., 19 triples) as shown in Figure 13. The DBMS produced very complex query execution plans, especially for the corresponding single queries. They consist of up to 200 database operators (measured via EXPLAIN for DB2 and Query Analyzer for SQL Server) including

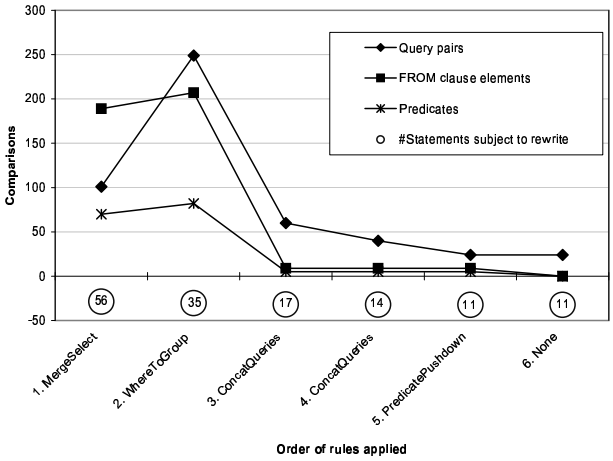


Figure 15: Number of comparisons per layer of the CGO optimizer depending on the rewrite steps for one specific statement sequence.

approximately 30 joins and 40 sorts. For presentation and discussion we have selected a small set of characteristic statement sequences.

The comparison of statement sequence and corresponding single query is covered by Figure 11. It shows the execution time for several single queries. The runtime of the corresponding initial statement sequence is assumed as 100%. The single queries are derived from the statement sequences by nesting the definitions of the respective temporary tables as described in Section 3. As one can see, single queries show a performance gain only in rare cases. For some statement sequences, e.g., sequence 3, the execution time for the corresponding single query grows tremendously. Hence, single query optimization is not a viable alternative to our CGO approach. Our experiments did not reveal a clear correlation between the complexity of a sequence and its single query performance.

We ran our sample sequences against the CGO prototype and picked the intermediate results of the rewrite process for measurements. We measured the total execution time, i.e., the total time taken by the system to evaluate the entire statement sequence as rewritten up to that point. Figure 12 shows the results for four statement sequences. The total execution time of each modified sequence is given as a percentage of the total execution time of the corresponding original sequence. Due to the differences in the four given statement sequences the appropriate rewrite processes include a varying number of steps. For statement sequences 1 and 2 the CGO optimizer was able to apply rewrite rules in seven successive steps. Statement sequences 3 and 4 allowed only three steps of query rewriting. Each step constitutes the application of a single rewrite rule. Remark, that different sequences have different rewrite rules applied in each step. For example, in step *rewrite 1* the rule applied to *sequence 2* is not the same as the rule applied to *sequence 3*. Figure 12 shows that for each of the statement sequences the execution time was reduced by at least 30% after rewriting. However, reducing the runtime by even an order of magnitude is possible, as can be seen for statement sequence 2.

The performance results depicted in Figures 13 to 15 focus on the efficiency and quality of our CGO prototype. We installed the prototype on a MS Windows 2000 system with two AMD Athlon 1800+ processors and JDK 1.3.1. Figure 13 shows the time consumed by the three components of the CGO prototype to optimize eight different sequences. This time is negligible compared to the execution time of a sequence, which is 20 minutes and more for some of the sequences. One can see that the time for optimization as well as the total execution time grows proportional to the number of statements in the sequence. This demonstrates the scalability of the CGO approach. Most of the time is spent for retranslation since we use XSLT for this purpose, which is not very efficient.

The number of comparisons for rewriting each sequence is depicted in Figure 14. The values are classified by the specific comparison operations in each layer of the rule condition model illustrated in Figure 9. Figure 15 shows the distribution of the comparison operations during the optimization process of a sample sequence initially consisting of 56 statements. Circled numbers show the decreasing number of statements as a result of rewriting the sequence. Due to the length of statement sequences a high number of comparisons typically appear at the beginning of the optimization. For the example in Figure 15 the number of comparisons even grows for the second rule application compared to the first rewrite step. This results from the fact that rule conditions of all class 1 rules have to be checked before the *WhereToGroup* rule is applied. In further steps, the number of comparisons decreases since the remaining set of comparable statements is considerably reduced.

We conclude that CGO is a scalable and effective approach for the optimization of statement sequences. It represents a feasible solution that offers significant performance improvements. The optimization process is very efficient with a manageable complexity compared to state-of-the-art optimization approaches. As the measurements of sequence 4 in Figure 12 show, a deterioration of execution times is possible. Improved control strategies as described in Section 7.2 shall avoid this.

9 Conclusions

The purpose of this paper is to consider a lightweight, heuristic, pre-optimization rewrite phase as a beneficial approach for processing large sequences of correlated SQL statements. Such a rewrite phase is meant to complement the conventional query optimization

phase for the individual statements of a sequence. SQL statement sequences are produced by query generators of applications like ROLAP tools and are likely to occur more frequently in the future, when more and more complex analysis and report tasks are automated. We argue that an exhaustive optimization of such problems with the help of commercial optimization techniques is intolerable, as our measurements revealed. The key idea of our approach, called coarse-grained optimization, is to rewrite the given SQL statement sequence into an equivalent statement sequence that can be processed more efficiently by the target database system than the original sequence. We are aware of the fact that heuristic rewriting does not necessarily lead to a solution that is optimal or at least better than the original problem. However, we believe that there is a considerable benefit in a rewrite phase that is independent of the target database systems.

Our performance experiments on two different commercial database systems revealed that our heuristic approach is able to contribute to significant response time reductions, in some cases by an order of magnitude.

In future work we will investigate the effectiveness of extended control strategies as mentioned before. Additional work will focus on fine-tuning CGO techniques to the specific characteristics of the underlying database systems.

Acknowledgement

We would like to thank the referees for their helpful comments as well as Peter Peinl for reading an early version of this paper.

References

[1] J. Chen, D. DeWitt, F. Tia, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. SIGMOD, Dallas, Texas, USA*, May 2000.

[2] M. Cherniack and S. Zdonik. Inferring Function Semantics to Optimize Queries. In *Proc. VLDB, New York, USA*, August 1998.

[3] N. Dalvi, S. Sanghai, P. Roy, and S. Sudarshan. Pipelining in Multi-Query Optimization. In *Proc. PODS, Santa Barbara, California, USA*, May 2001.

[4] S. Finkelstein. Common Subexpression Analysis in Database Applications. In *Proc. SIGMOD, Orlando, Florida, USA*, June 1982.

[5] J. Goldstein and P.-A. Larson. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. In *Proc. SIGMOD, Santa Barbara, California, USA*, May 2001.

[6] G. Graefe. The Cascades Framework for Query Optimization. *Data Engineering Bulletin*, 18(3), 1995.

[7] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-Query Processing in Data Warehousing Environments. In *Proc. VLDB, Zürich, Switzerland*, September 1995.

[8] Y. Ioannidis and Y. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proc. SIGMOD, Atlantic City, New Jersey, USA*, May 1990.

[9] J. Melton and A. Simon. *SQL:1999, Understanding Relational Language Components*. Morgan Kaufmann, 2002.

[10] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible Rule-Based Query Rewrite Optimization in Starburst. In *Proc. SIGMOD, San Diego, California, USA*, June 1992.

[11] A. Rosenthal and U. Chakravarthy. Anatomy of a Modular Multiple Query Optimizer. In *Proc. VLDB, Los Angeles, California, USA*, August/September 1988.

[12] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and Extensible Algorithms for Multi Query Optimization. In *Proc. SIGMOD, Dallas, Texas, USA*, May 2000.

[13] H. Schwarz, R. Wagner, and B. Mitschang. Improving the Processing of Decision Support Queries: The Case for a DSS Optimizer. In *Proc. IDEAS, Grenoble, France*, July 2001.

[14] T. Sellis. Multiple-Query Optimization. *TODS*, 13(1), 1988.

[15] M. Steinbrunn, G. Moerkotte, and A. Kemper. Heuristic and Randomized Optimization for the Join Ordering Problem. *VLDB Journal*, 6(3), 1997.

[16] TPC-H Standard Specification, Revision 2.0.0. *www.tpc.org/tpch*, 2002.

[17] J. Yang, K. Karlapalem, and Q. Li. Algorithms for Materialized View Design in Data Warehousing Environment. In *Proc. VLDB, Athens, Greece*, August 1997.

[18] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering Complex SQL Queries Using Automatic Summary Tables. In *Proc. SIGMOD, Dallas, Texas, USA*, May 2000.