A Cost-Estimation Component for Statement Sequences

Tobias Kraft
Institute of Parallel and Distributed Systems
University of Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany
Tobias.Kraft@ipvs.uni-stuttgart.de

ABSTRACT

Query generators producing sequences of SQL statements are embedded in many applications. As the execution time of such sequences is often far from optimal, their optimization is an important issue. Therefore, in [5] we proposed a rule-based optimization approach, which we called CGO (Coarse-Grained Optimization). Our first prototype used a heuristic, priority-based control strategy to choose the rewrite rules that should be applied to a given statement sequence. This worked well but there is still potential for improvements. Thus, in [4] we have introduced an approach to provide cost estimates for statement sequences which is the basis for a cost-based CGO optimizer. It exploits histogram propagation and the optimizer of the underlying database system for this purpose. In this demonstration, we want to showcase the functionality and the effectiveness of our approach. Thereto, we present a prototype of a cost-estimation component for statement sequences which implements this approach. It includes a graphical user interface to explain the histogram-propagation process and to report the results of the cost-estimation process. In the setup for this demonstration, we use a TPC-H benchmark database with an appropriate set of sequences as sample scenario.

1. INTRODUCTION

Tools that generate SQL statements can be found in many of today's database application areas. Especially in the data warehouse area, tools even generate complex sequences of SQL statements. These sequences break down a complex information request into a set of subsequent steps. We analyzed such sequences and found out that their execution time is often far from optimal. Therefore, in [7] and [5] we proposed an optimization approach that supports the optimization of statement sequences outside the database system. This approach is called CGO (Coarse-Grained Optimization). It is based on a set of rewrite rules, i.e., the rules transform a SQL statement sequence into a semantically equivalent SQL sequence.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

CGO addresses statement sequences that are composed of CREATE TABLE, INSERT and DROP TABLE statements with the following properties. The CREATE TABLE and DROP TABLE statements implement temporary tables that only exist during the execution of the sequence in order to store intermediate results of the sequence. Moreover, one of the CREATE TABLE statements creates the table that stores the final result of the sequence which is not dropped within the sequence. The intermediate results as well as the final result are produced by the INSERT statements which represent the computational steps of the sequence. Thus, each INSERT statement has a query within its body that may access base tables as well as intermediate results of previous steps. So, multiple steps may share results of previous steps.

In the first CGO prototype [5][3], we employed a heuristic control strategy to determine the order in which rewrite rules are applied to a statement sequence. Thereto, we assigned a priority to each rule. Among the rules that can be applied to a sequence, the one with the highest priority is being chosen. This is repeated until no more rule applications are possible. This strategy works well but there is still potential for improvements. Therefore, in [4] we proposed a practical approach to compute cost estimates for statement sequences which is the basis for an enhanced cost-based control strategy. Cost estimates would allow to compare several alternative sequences without executing them and to select the presumably most efficient one from the search space that is made up by the set of CGO rewrite rules. Our approach exploits the cost-estimation component of the underlying database system which should later on execute the statement sequence. Furthermore, it makes use of histogram propagation to provide statistics for the intermediate-result tables increasing the accuracy and usability of the cost estimates returned by the optimizer of the underlying database system. This means, it retrieves histogram data and cost estimates from the underlying database systems and again stores the results of histogram propagation in the database catalog. Therefor, it makes use of Statistics API [2], a DBMS-independent application programming interface for management of DBMS statistics.

The remainder of this paper is organized as follows. Section 2 gives an overview of the demonstration setup and introduces our cost-estimation approach for statement sequences. Section 3 describes the purpose of the demonstration in more detail and shows some screenshots of the prototype's GUI (graphical user interface).

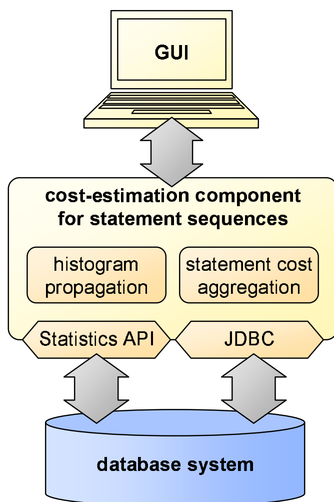

Figure 1: Architectural overview.

2. SYSTEM OVERVIEW

Figure 1 gives an architectural overview of the demonstration setup. It comprises a GUI, the cost-estimation component for statement sequences including Statistics API, and the underlying database system. Each of these components is highlighted in one of the following sections.

2.1 Database System

In the demonstration setup, we employ IBM DB2 as underlying database system. However, the underlying database system is interchangeable. It can be replaced by any other database system that provides access to its optimizer statistics and cost estimates and that is supported by a corresponding Statistics API implementation.

We use a TPC-H benchmark database [8] in the demonstration, applying the original benchmark tools to create the data. For the creation of table and column statistics as well as histograms, we execute RUNSTATS on all tables of the TPC-H benchmark database using the WITH DISTRIBUTION option.

2.2 Statistics API

Statistics API is a JDBC-based programming interface for DBMS-independent access and management of DBMS statistics. It defines a set of methods to retrieve and manipulate histograms as well as a set of methods to retrieve database meta data and cost estimates for arbitrary SQL statements from different DBMSs. Furthermore, it provides DBMS-independent data structures to hold the meta data and the statistics data which comprises a flexible histogram format that abstracts from the proprietary data structures of different DBMSs. For each supported database management system, a corresponding implementation has to exist that extracts the requested data from the target database and transforms the proprietary data structures into the DBMS-independent formats. Additionally, for histograms this has to work vice versa. At the moment there exists an implementation for IBM DB2, for Oracle and for Microsoft SQL Server. As Statistics API uses ordinary JDBC connections to communicate with the target database system, these connections can be reused for other data management purposes if needed. Due to the use of Statistics API, the cost-estimation component is largely independent from a certain underlying DBMS.

More information about these implementations and about Statistics API can be found in [2].

2.3 Cost-Estimation Component

The cost-estimation component for statement sequences expects a statement sequence as input and returns a cost estimate as output. The measuring unit of the cost value depends on the underlying database system. At first, the cost-estimation component traverses the statement sequence and executes the CREATE TABLE statements that create the intermediate-result tables. The existence of these tables is necessary for the subsequent steps. Afterwards, it parses the INSERT statements of the sequence and translates them into algebraic operator trees. This trees are used for histogram propagation, i.e., the operators of this trees do not process relations but sets of histograms. The histograms resulting from histogram propagation are stored proprietaryly in the catalog of the underlying database system as statistics for the target table of the corresponding INSERT statement. For this purpose as well as for the retrieval of histograms for the base tables used in the INSERT statements, we make use of Statistics API. After histogram propagation, the cost-estimation component retrieves a cost estimate for each INSERT statement from the optimizer of the underlying database system by the use of Statistics API. It aggregates the retrieved cost estimates to obtain a cost estimate for the whole sequence. Finally, it drops the intermediate-result tables that have been created for the purpose of cost estimation. The CREATE TABLE and DROP TABLE statements are executed via JDBC reusing the JDBC connection that has been opened by the Statistics API implementation.

The propagation of histograms and the storage of the propagation results in the database catalog solves the problem of missing statistics for intermediate-result tables. This means, the propagated histograms assist the optimizer of the underlying database system in producing more accurate query plans and cost estimates for the INSERT statements that depend on intermediate-result tables. Otherwise, the optimizer of the underlying database system would use default values for the cardinality of these tables and default selectivities for predicates that include attributes of these tables. For this purpose, we extended previous work on approximate query answering [1][6] and added support for arithmetic terms, grouping and aggregation.

More detailed information on the cost estimation approach and on the extensions to previous work on histogram propagation can be found in [4].

2.4 GUI

The GUI has been developed for demonstration purposes. It explains the cost-estimation process and in particular the histogram-propagation process. This means, it displays the algebraic operator tree obtained by translating the queries of the INSERT statements of a given sequence into our algebra. Each algebraic operator provides a set of attributes and a corresponding histogram for each attribute as output. The GUI is able to display these histograms. Moreover, the user can choose between different representations, i.e., for readability, the histogram resulting from propaga-

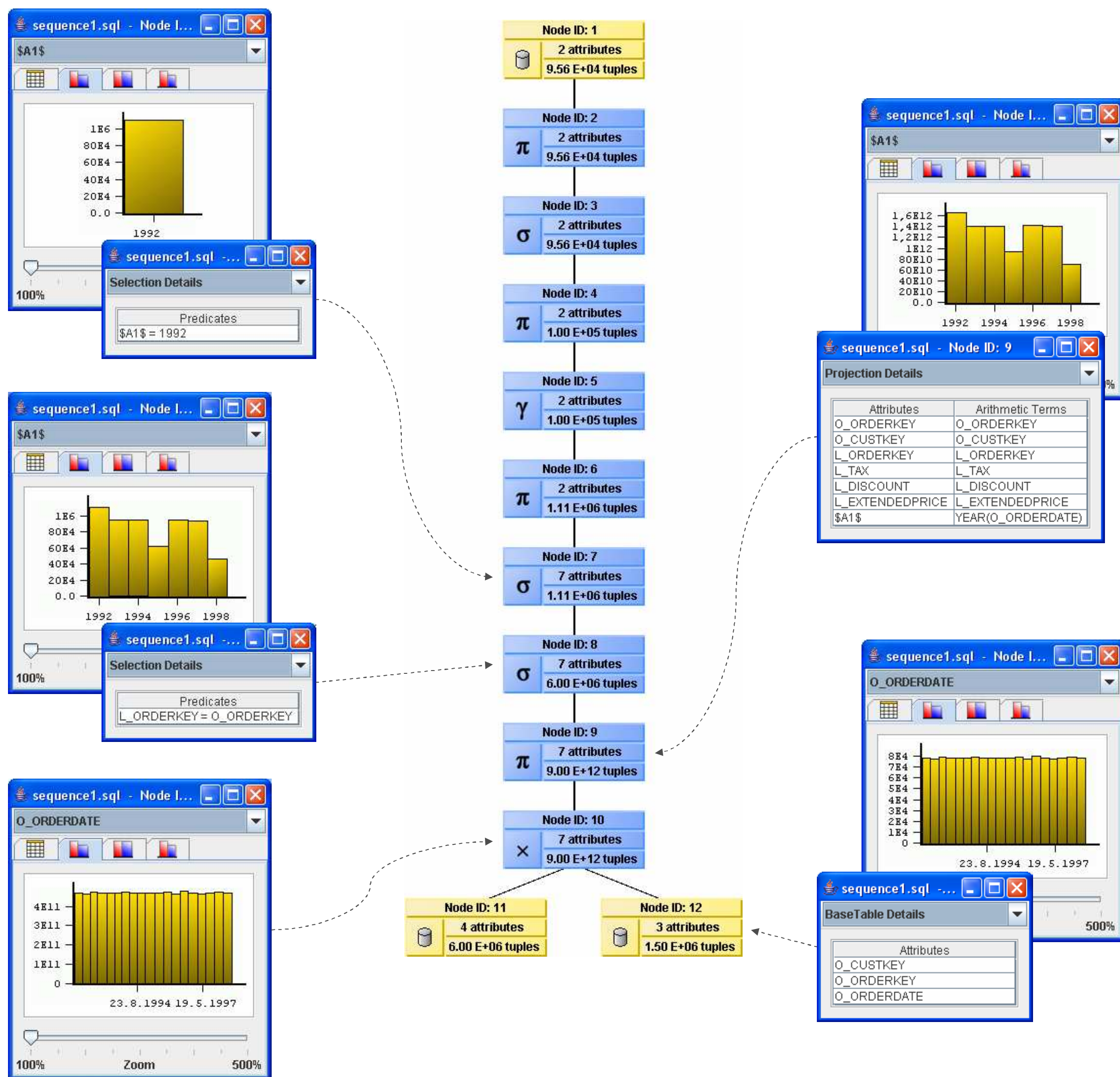

Figure 2: GUI screenshots showing an algebraic operator tree and some windows that display histograms and operator-specific information.

tion may be transformed into an equi-width or an equi-depth histogram. Besides, the GUI presents operator-specific information. Furthermore, for each INSERT statement in the sequence, the GUI reports the cost estimate and cardinality estimate retrieved from the optimizer of the underlying database system as well as the cardinality estimate obtained by histogram propagation. The GUI also allows to modify configuration parameters concerning the algorithms used for histogram propagation. For example, you can turn on and off the normalization step that reduces the number of buckets after creation of a new histogram or after modification of an existing histogram; or you can vary the maximum number of buckets concerning normalization.

Figure 2 shows a collection of some screenshots of the GUI. It contains an algebraic operator tree which is surrounded by some windows that display histograms or additional operator-specific information.

3. DEMONSTRATION

The demonstration has two purposes:

- First, we want to provide an insight into the cost-estimation process, especially we want to explain the histogram-propagation process.
- Secondly, we want to show the successful application of our cost-estimation approach.

For the first purpose, we employ the GUI that explains the histogram-propagation process in detail. Therefor, we use a set of sequences that results from the application of the CGO rewrite rules to a statement sequence which has been generated by the MicroStrategy DSS tool suite. So, these sequences represent a search space of semantically equivalent but syntactically different sequences. Some constants used for filtering in the sequences can also be varied to produce

```
INSERT INTO temptable1
  SELECT
    o.o_custkey,
    sum(l.l_extendedprice*(1-l.l_discount)*
        (1+l.l_tax))
  FROM
    lineitem l,
    orders o
  WHERE
    l.l_orderkey = o.o_orderkey AND
    year(o.o_orderdate) = 1992
  GROUP BY
    o.o_custkey
  HAVING
    sum(l.l_extendedprice*(1-l.l_discount)*
        (1+l.l_tax)) >= 100000
```

Figure 3: An INSERT statement of the generated statement sequence.

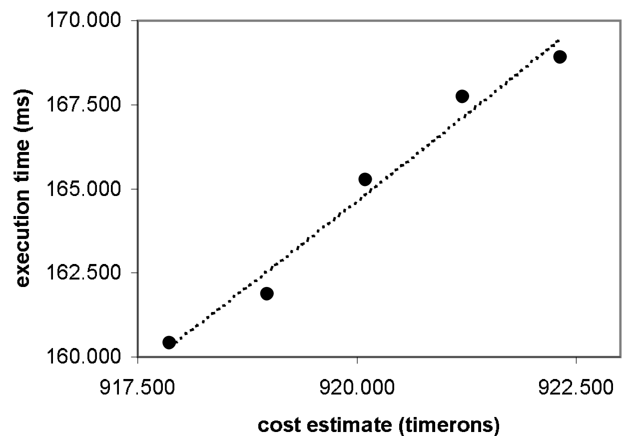

Figure 4: Measurement results for different variants of the generated sequence (in a DB2 environment).

additional variants of the sequences with a different selectivity. To give an example of what is being demonstrated, we again refer to the collection of screenshots in Figure 2 that shows the algebraic operator tree obtained for the INSERT statement in Figure 3. This INSERT statement is part of the generated sequence. It computes the turnover for each customer in the year 1992 and returns all customers that have a turnover equal to or greater than 100000. For this purpose, it accesses the base table *LINEITEM* and the base table *ORDERS* which are both part of the TPC-H schema. The result of the computation is stored in the intermediate-result table *temptable1* which is also created and dropped within the sequence. The windows that surround the algebraic operator tree depict how the attribute *O_ORDERDATE* provided by the base table *ORDERS* is adapted by histogram propagation. The window in the bottom right corner shows the corresponding histogram extracted from the catalog tables of the underlying database system by the use of Statistics API. The cartesian product changes the total cardinality of the histogram but it does not change the relative data distribution. The subsequent projection applies the arithmetic function *YEAR* to the attribute *O_ORDERDATE*. The generated attribute *$A1$* stores the result which is represented by the histogram in the upper right corner of Figure 2. The subsequent selection operator applies the join condition to the result of the projection. This again just modifies the total cardinality of the histogram of *$A1$*. The second selection operator applies the filter condition $A1$ = 1992. Thus, the resulting histogram, shown in the upper left corner of Figure 2, contains only a single bucket that represents the year 1992.

For the second purpose of the demonstration, we execute the sequences and compare the execution times with the corresponding cost estimates. This should depict the successful application of our cost-estimation component. Figure 4 shows the measurement results when using different filter values in the generated sequence, i.e., when varying selectivities within the sequence. As the trend line shows, the cost estimates provided by our prototype are a good indicator for the corresponding execution times. Further experiments have also shown that the optimizer of the underlying database system profits from histograms on intermediate-result tables provided by histogram propagation.

4. ACKNOWLEDGMENTS

We would like to thank Benjamin Höferlin for his help in implementing the graphical user interface.

5. REFERENCES

[1] Y. Ioannidis and V. Poosala. Histogram-Based Approximation of Set-Valued Query-Answers. In *Proc. VLDB*, 1999.

[2] T. Kraft and B. Mitschang. Statistics API: DBMS-independent Access and Management of DBMS Statistics in Heterogeneous Environments. In *Proc. ICEIS*, 2007.

[3] T. Kraft and H. Schwarz. CHICAGO: A Test and Evaluation Environment for Coarse-Grained Optimization. In *Proc. VLDB*, 2004.

[4] T. Kraft, H. Schwarz, and B. Mitschang. A Statistics Propagation Approach to Enable Cost-Based Optimization of Statement Sequences. In *Proc. ADBIS*, 2007.

[5] T. Kraft, H. Schwarz, R. Rantzau, and B. Mitschang. Coarse-Grained Optimization: Techniques for Rewriting SQL Statement Sequences. In *Proc. VLDB*, 2003.

[6] V. Poosala, V. Ganti, and Y. Ioannidis. Approximate Query Answering using Histograms. *IEEE Data Engineering Bulletin*, 22(4), 1999.

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

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