A Learning Optimizer for a Federated Database Management System

S. Ewen[#] M. Ortega-Binderberger* V. Markl[+]

[#]IBM Germany
Am Fichtenberg 1
71083 Herrenberg
Germany

*IBM Silicon Valley Lab
555 Bailey Road
San Jose, CA
USA

[+]IBM Almaden Research Center
650 Harry Road
San Jose, CA
USA

ewens@de.ibm.com
{mortega, marklv}@us.ibm.com

Abstract: Optimizers in modern DBMSs utilize a cost model to choose an efficient query execution plan (QEP) among all possible ones for a given query. The accuracy of the cost estimates depends heavily on accurate statistics about the underlying data. Outdated statistics or wrong assumptions in the underlying statistical model frequently lead to suboptimal selection of QEPs and thus to bad query performance. Federated systems require additional statistics on remote data to be kept on the federated DBMS in order to choose the most efficient execution plan when joining data from different datasources. Wrong statistics within a federated DBMS can cause not only suboptimal data access strategies but also unbalanced workload distribution as well as unnecessarily high network traffic and communication overhead.

The maintenance of statistics in a federated DBMS is troublesome due to the independence of the remote DBMSs that might not expose their statistics or use different models and not collect all statistics needed by the federated DBMS.

We present an approach that extends DB2s learning optimizer to automatically find flaws in statistics on remote data by extending its query feedback loop towards the federated architecture. We will discuss several approaches to get feedback from remote queries and present our solution that utilizes local query feedback and remote query feedback, and can also trigger and drive iterative sampling of remote data sources to retrieve information needed to compute statistics profiles. We provide a detailed performance study and analysis of our approach, and demonstrate in a case study a potential query execution speedup in orders of magnitude while only incurring a moderate overhead during query execution.

Acknowledgements

We want to acknowledge the following people for providing the inspiration, insights and guidance that helped make this work possible: Guy Lohman, Hamid Pirahesh, Ashraf Aboulnaga, Holger Kache, Aakash Bordia, Tina Mukai and Anjali Grover.

1. Introduction

Modern database management systems (DBMSs) perform query optimization, i.e., the selection of the best possible query execution plan (QEP), by enumerating and costing all or a subset of possible QEPs, and then selecting the cheapest one. A query execution plan is a directed data flow graph, where nodes denote operations, and edges are input streams from other operators or tables in the database. Estimating the cost of a QEP requires computing the cardinality, i.e., the number of rows to be processed, for each node (intermediate step) in the QEP. The cost model uses statistics and various assumptions to compute the selectivity of any selection and join node in the QEP, as well as distinct values for grouping, projection, and aggregation nodes. Statistics that are kept in the system catalog include the number of rows in a table, the distribution of values in columns, and joint statistics on the correlation of groups of columns for more advanced optimizers. The most important and troublesome assumption is the independence assumption, which states that the data in two or more columns is independent, unless otherwise stated by column group statistics. This assumption simplifies the model and the need to store complex statistics, as it allows for multiplying the individual selectivities of individual predicates in order to compute the selectivity of a conjunctive predicate restricting multiple columns. Outdated statistics or the violation of assumptions can cause the optimizer to misestimate the intermediate cardinalities and may lead to the selection of a suboptimal plan, which in turn results in bad query performance. Most prevalent errors are the wrong allocation of runtime resources, wrong join orders, or selection of the wrong physical implementation of an operator (e.g., nested-loop join instead of hash-join).

On a federated DBMS, the optimizer has the additional task to determine how to distribute the workload over the datasources, considering the overhead of communicating with the remote source. For instance, joining tables from different sources may be realized by transferring a complete table and performing the join locally, or transferring only the rows matching the join predicate. Because of this local vs. remote decision, the performance loss through poor QEPs in federated systems is potentially a lot higher than for purely local database systems or database systems in a distributed, non federated architecture. Federated plans are costed with the size of the remote query results, estimated through the statistics the optimizer has on the remote data.

For statistics in non-federated environments several approaches have been suggested to help keeping them up to date, by monitoring Update/Delete/Insert (UDI) activity and changes to proactively determine when and where statistics need to be recomputed. For statistics on remote data in a federated DBMS, this clearly is not applicable, as the majority of the workload on the remote datasource will most likely not go through the federated system. Our approach utilizes an autonomic query feedback (QF) loop following the architecture of DB2s learning optimizer LEO, where plans and runtime monitor output are consecutively analyzed to find flaws in statistics and create recommendations for gathering or refreshing statistics.

The major difference to the non federated learning optimizer is the query monitoring component. Since an integrated runtime monitor does not exist for remote datasources in general, monitoring needs to utilize monitoring tools of the remote databases or cleverly re-write SQL-statements in order to piggy-back on query execution. Alternatively, a set of additional query related count statements can obtain the cardinalities from remote base tables, possibly accelerated by sampling techniques. The learning optimizer can use the statistics obtained through any of these methods to compute profiles that declare what statistics are needed to overcome the estimation errors. In that sense, this approach behaves reactively and helps the optimizer to more accurately estimate cardinalities for later queries that use same or similar selections of predicates.

The remainder of this paper is organized as follows: Section 2 provides background on federated database systems and the special considerations taken into account when optimizing queries for a federated database system. The section also describes the query-feedback architecture of DB2s learning optimizer LEO. Section 3 describes the mechanism of automated statistics profiling used by the learning optimizer, especially the analysis of predicates and column correlations. In Section 4 we discuss and evaluate several approaches to implement a runtime monitor for federated queries. Section 5 shows how to exploit the query feedback. Section 6 presents a case study for a realistic workload scenario. Sections 7 surveys related work. We give our conclusions as well as an outlook on future work in Section 8.

2. Background

Our approach is to extend the learning optimizer towards federated database systems. This section gives an overview of both the Learning Optimizer (LEO) used in DB2s non-federated mode, as well as of DB2 federated technology.

2. 1 LEO – DB2s Learning Optimizer

LEO [MLR03] exploits empirical results from actual executions of queries to validate the optimizer's model incrementally, deduce what part of the optimizer's model is in error, and compute adjustments to the optimizer's model. LEO is comprised of four components: a component to save the optimizer's plan, a monitoring component, an analysis component, and a feedback exploitation component. The analysis component is a standalone process that may be run separately from the DB2 server, and even on another system. The remaining three components are modifications to the DB2 server: plans are captured at compile time by an addition to the code generator, monitoring is part of the runtime system, and feedback exploitation is integrated into the optimizer. The four components can operate independently, but form a consecutive sequence that constitutes a continuous learning mechanism by incrementally capturing plans, monitoring their execution, analyzing the monitor output, and computing adjustments to be used for future query compilations.

Figure 1 shows how LEO is integrated into the architecture of DB2. The left part of the figure shows the usual query processing flow with query compilation, QEP generation and optimization, code generation, and code execution. The gray shaded boxes show the changes made to regular query processing to enable LEO's feedback loop: for any query, the code generator dumps essential information about the chosen QEP (a plan "skeleton") into a special file that is later used by the LEO analysis daemon. In the same way, the runtime system provides monitored information about cardinalities for each operator in the QEP. Analyzing the plan skeletons and the runtime monitoring information, the LEO analysis daemon computes adjustments that are stored in the system catalog. The exploitation component closes the feedback loop by using the adjustments in the system catalog to provide adjustments to the query optimizer's cardinality estimates.

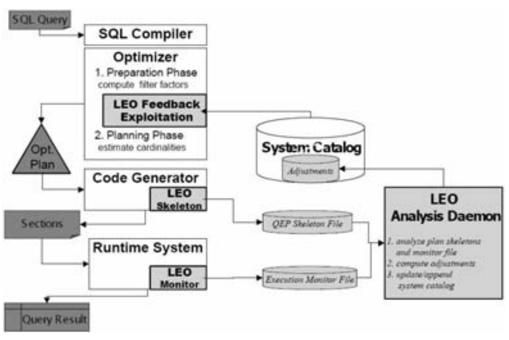


Figure 1: LEO Architecture

2. 2 Federated DBMS – DB2 II

Federated Database Management Systems are DBMSs that are able to interface with independent, external datasources and provide a relational view over remote data. Among those external datasources can be independent instances of the same database, 3rd party relational databases and also non-relational datasources like spreadsheets and flat files.

DB2 Information Integrator (DB2 II) extends DB2 UDB with federated capabilities. DB2 II contains an extended query compiler that includes a remote statement generator and extended pushdown analysis as well as a set of wrappers that encapsulate what is unique to each remote datasource and mediate all requests between the DB2 II server and the datasources. Figure 2 depicts the DB2 II architecture. The light gray shaded boxes are Information Integrator specific extensions.

In a typical query that involves a nickname (a view of remote data, such as a table on another relational DBMS), the optimizer develops an overall execution plan, which also includes those parts of the QEP that will later on be executed by the remote sources, based on the statistics it has on the remote data, to estimate the cardinalities of the results that will come back from the remote source. It considers the additional costs of the federated overhead and places a so-called ship-operator to determine at which point of the query, the results should be communicated between the remote datasource and the DB2 II server; this point is very dependent in the capabilities of the queried datasource.

For all parts of the query that appear below a ship-operator and are thus marked to be executed on a remote datasource, the statement generator creates a SQL statement, in the dialect of the targeted datasource, which represents this part of the QEP, to be sent to the wrapper during query execution. The wrapper finally acts as a client to the remote datasource, accesses it to execute the received query statement and parses the result data into DB2s proprietary format. Though the capabilities of DB2 II go far beyond connecting only relational datasources, we will focus on this subset of possible remote datasources, as those are the ones used in the larger scenarios and warehouses that the approach we present here targets.

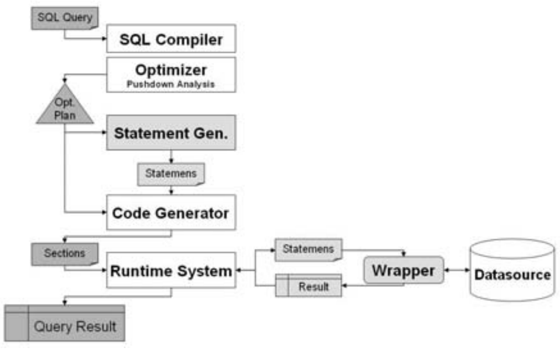


Figure 2: DB2 II Architecture

3. Automated Statistics Profiling

Direct adjustments to catalog statistics from Query Feedback is not applicable due to consistency reasons, as the QF only reflects isolated aspects of the data. What is generated instead is a ranked set of statistic profiles, which declare what sorts of statistics are needed and with what priority.

The continuous feedback process stores estimation errors determined by the plan and runtime monitor in the Query Feedback Warehouse (QFW) where the Query Feedback Analyzer (QFA) analyzes them to determine which tables have outdated statistics or lack special sort of statistics. This QFA is in our case comprised of the components “Table Cardinality Analyzer” (TCA), which finds deviations in estimated and actual table sizes, and the “Correlation Analyzer” (COA), which can detect intra-table correlations and recommend column group statistics. The architecture is that of figure 1; this section describes the analysis daemon.

3.1 The Query Feedback Warehouse

The QFW (see Figure 3) is populated periodically using the information generated by the Plan Monitor (PM) and the Runtime Monitor (RM). The data in the QFW is organized into relational tables. A detailed description can be found in [AHLL04].

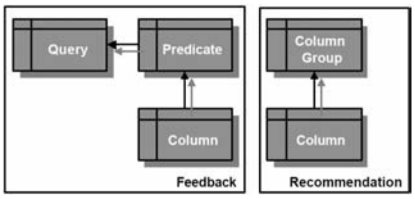


Figure 3: Tables in the QFW

3.2 Table Cardinality Analyzer

The TCA simply compares the estimated table cardinalities, with the actually observed cardinalities to determine if the statistics for this table are outdated.

3.3 Correlation Analyzer

The COA focuses on pair-wise correlations between columns in a table, because experiments indicate that the marginal benefit of correcting for higher-order correlations is relatively small; see [IMHB04]. For each pair of columns that appear jointly in a QFW record, the COA compares the actual selectivity of each conjunctive predicate to the product of the actual selectivity of the Boolean factors of the conjunct, assuming that this information is available. Denote by α_1, α_2, and α_{12} cardinalities of simple equality predicates that are observed during execution of a query, and denote by *m* the cardinality of the entire table. Then the COA deems the independence assumption to be valid if and only if $1-\Theta \leq \frac{\alpha_{12} m}{\alpha_1 \alpha_2} \leq 1+\Theta$, where $\Theta \in (0, 1)$ is a small pre-specified parameter. Otherwise, the COA declares that a correlation error of absolute magnitude $|\alpha_{12} - (\alpha_1 \alpha_2 / m)|$ has occurred. The analysis becomes more complicated when one or more of the actual cardinalities are not available, as is often the case in practice. The COA deals with the problem by estimating the missing information and adjusting the error-detection threshold and estimate of the error magnitude accordingly.

4. Plan and Runtime Monitor for Federated Queries

Plan- and Runtime Monitoring is the mechanism used to gather the different cardinalities used by the QFA to detect flaws in the statistics that were used to develop the QEP. Statistics can be analyzed only through feedback from operators that are directly influenced by them; in the case of statistics on remote data, those are the parts of the QEP that occur below a ship operator. They are used to construct the remote query statements and represent the optimizer's assumption of how those statements will get executed.

4.1 Plan Monitor

The Plan Monitor (PM) is the component that stores a skeleton of the optimizer selected QEP. In DB2, the QEP is translated into an internal format suitable for later execution, so called sections. Only these sections are retained, the original QEP is dropped after compile-time. To assemble Query Feedback, a slim version of the QEP is stored as a skeleton containing only information relevant to the QFA.

For the federated PM, the skeleton is extended to also store the remote parts of the QEP developed by the federated server's optimizer, which is translated into a SQL statement and executed on remote datasources.

The skeleton hence contains the local optimizer's assumption of how the statement should get executed on the remote source, based on local statistics available about the remote data. The actual QEP chosen by the remote server's optimizer will in many cases deviate due to different statistics and capabilities.

4.2 Runtime Monitor

The Runtime Monitors (RM) task is to collect the actual cardinalities that correspond to the estimates recorded by the Plan Monitor. In order to profile detailed column statistics, cardinalities must be monitored predicate-wise rather than operator wise. For an operator with three applied Boolean factors p_1, p_2, p_3, it is inapplicable to collect the associated cardinalities α_1, α_2, α_3 individually, as this requires applying each predicate isolated to the operators' input stream. Rather than that, joint cardinalities are collected by applying the next predicate to the output of the previous one, collecting in the above case the actual cardinalities α_1, α_{12}, α_{123}.

DB2s local RM, which is part of the LEO learning optimizer, piggy-bags on query execution and counts the number of rows that pass through the runtime operators. For federated queries, this RM monitors all local query parts. To supplement this runtime information with the cardinalities for operators in the remote query parts, several approaches are possible that can be categorized into three groups:

Immediate feedback can be obtained through the use of the remote datasources proprietary monitoring mechanisms and finding the matching parts between the federated optimizer's QEP and the remote server's QEP. This method has least overhead of all but is also least applicable as it requires those mechanisms to be available on the remote datasource. Utilizing query debugging tools is a method of this category.

A second way of obtaining immediate feedback is piggy-backing on the queries by cleverly rewriting the SQL statements so that besides executing the query they also return the intermediate cardinalities. The rewriting process ensures a query execution plan that guarantees that the collected cardinalities match the estimates recorded by the plan monitor. This piggy-backing is realized by either inserting table functions into the query plan or splitting up the query into a set of common table expressions and aggregating intermediate results as a side effect. This piggy-backing approach is highly applicable, but has moderate performance overhead when using query rewriting on databases that support common sub expressions, and high performance overhead when using table functions.

The third possibility is using deferred feedback. At query compilation time, a set of additional statements is generated that collect the intermediate cardinalities. Those statements are executed only for remote queries that show problems; this method hence has a selective overhead, which is potentially high, but applies only to a subset of queries and appears in reserved timeframes (e.g., together with scheduled maintenance). Furthermore, the performance of this method can be greatly improved by the use of sampling techniques.

Detailed evaluation of each method is presented in the next sections. Though DB2s RM collects cardinalities for all operators and can thus detect correlations between columns in different tables, the majority of the corrections are computed from feedback on single tables, coming from table access operators (QEP leafs). Restricting the federated RMs to collect only feedback from table access operators offers a lot of space for improvements and the application of advantageous techniques. Figure 4 shows the average overhead of monitoring a set of queries through the methods described above, where *scan* means restricting the method to be applied to base table scans only.

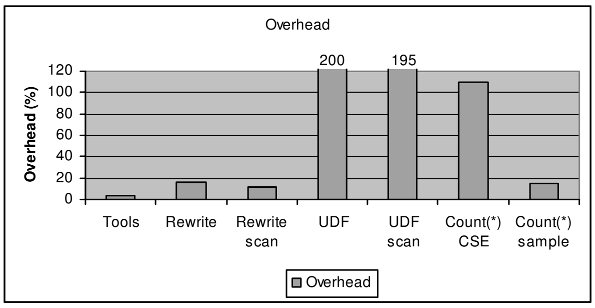


Figure 4: Performance of federated RMs

The overhead is given in percent of the original query execution time. It was measured from remote statements executed on a commercial DBMS accessed through DB2 UDB 8.2 with Information Integrator.

Concluding from the specific overhead, the monitoring through proprietary tools provides the best performance and least impact on the remote server. Still, this approach is not applicable, as those mechanisms are not generally present. For immediate feedback, the query rewriting method is the best approach, but has moderate overhead and is only applicable on selected DBMSs that work efficiently with common sub expressions.

In general, the sampling of table access predicates through additional count(*) statements is an efficient approach. It works selectively for problem queries only, has moderate overhead, and works asynchronously, hence not affecting regular query execution and being able to use reserved maintenance timeframes.

4.2.1 Proprietary Monitoring Tools

The utilization of product specific monitoring tools, usually intended for manual debugging of underperforming queries, is a way of using mechanisms built into the runtime system of a remote datasource to obtain cardinalities of intermediate results. Tools that can be used for that approach need to record the query execution plan with the actually observed cardinalities for each operator.

A RM built on that method reads the cardinalities from the output of that tool and transfers them back to the QFA for to be matched against the recorded cardinality estimates. Naturally, as the remote datasources are independent from the federated server, the actually chosen QEP will in many cases differ from the federated server's assumptions and observed cardinalities cannot be matched back to estimates.

The biggest problem turns out to be a different choice in join order. Operators below the joins, specifically base tables scan operators, can be matched regardless of that problem and provide the majority of the feedback on predicates as they are supported in the current QFW implementations.

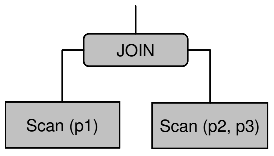
This method offers minimal overhead during query execution; specifically the costs of the remote server's proprietary monitor or trace generator, which usually ranks within few percent. This approach is after all not applicable, as there is no tool known to the author that collects all predicate cardinalities needed by the QFA. For all operators that apply multiple predicates, the cardinalities for applying only a subset of the predicates could only be observed from different queries.

4.2.2 Query Rewriting through CTE

The method of query rewriting with common table expressions (CTE) provides immediate feedback on intermediate cardinalities in one set with the regular query results. The rewritten query represents each operator or predicate with a common table expression, where expressions for non-leaf-level operators select from their children's expressions to build a continuous data stream not re-executing any query parts. The select query then builds a union of the output from the CTE that represents the root operator and a count(*) statement with operator id and predicate id for each CTE. To be able to bring those two parts into one result set, three numeric columns for the query feedback data are appended to the columns of the query result. Columns not used are filled by selecting null values. Figure 5 shows a simple and illustrative example how a query is rewritten. Figure 6 gives the prototype algorithm to build the list of CTEs for the rewritten query. When receiving the remote query results, the wrapper separates the actual query results from the feedback information by checking the additional columns of the input stream for null-values.

Original Query & Execution Plan

```
SELECT col1, col2
  FROM tab1 t1, tab2 t2
 WHERE t1.X = 'AAA'
   AND t2.Y = 'BBB'
   AND t2.Z = 'CCC'
   AND t1.A = t2.B
```



Rewritten Query

```
WITH
(SELECT col1, A    FROM tab1   WHERE X = 'AAA') AS Q1,
(SELECT col2, Y, B FROM tab2   WHERE Z = 'CCC') AS Q2,
(SELECT col2, B    FROM Q2     WHERE Y = 'BBB') AS Q3,
(SELECT col1, col2 FROM Q1, Q3 WHERE A = B    ) AS Q4

(SELECT col1, col2, NULL, NULL, NULL FROM Q4) UNION ALL
(SELECT NULL, NULL, 2, 1, COUNT(*) FROM Q1  ) UNION ALL
(SELECT NULL, NULL, 3, 1, COUNT(*) FROM Q2  ) UNION ALL
(SELECT NULL, NULL, 3, 2, COUNT(*) FROM Q3  ) UNION ALL
(SELECT NULL, NULL, 1, 1, COUNT(*) FROM Q4  )
```

Figure 5: Query rewriting with CTE

```
int buildCTEs(PlanOp op, String cteList, int pred, int tab) {

    String thisCTE = "(";
    addSelectList(thisCTE, op.propagatedCols);

    if (pred <= 1) {    // input from children or base table
        if (input is base_table) {
            addTableReference(thisCTE, op.base_table);
        } else {
            // build and reference all children
            for (int i = 0; i < children(op); i++) {
                tab = bldLists(op->child[i], cteList,
                               op->child[i].numPredicates, tab);
                addTableReference(thisCTE, tab);
            }
        }
    } else {    // input from another predicate, same operator
        tab = bldLists(op, cteList, pred - 1, tab);
        addTableReference(thisCTE, tab);
    }

    // add current predicate
    appendPredicate(thisCTE, pred);

    // append other clauses (group by/order by/...)
    ...

    // label this CTE
    tab++;
    thisCTE += ") AS Q" + tab;

    // append this CTE to CTE list
    cteList.append(thisCTE);

    // return this CTE's number
    return tab;
}
```

Figure 6: Query rewriting algorithm (CTE Lists)

The given algorithm is base for further tuning. As an example, this algorithm blocks the use of some index/fetch combinations, e.g. in nested loop joins. A possibility to overcome this problem is to treat those operators as one unit, gaining speedup at the costs of loosing some intermediate cardinalities.

The performance of this method is dependent on the remote server's ability to efficiently work with common sub-expressions (CSE). Figure 7 shows the execution time overhead of four queries compared with their rewritten statements on a commercial DBMS as the remote datasource. Query one and two are simple table scans with one respectively two predicates applied. Query three applies one predicate to a table and joins with another table after an indexed key. Query four is comparable to the query used in figure 5. For more complex queries, the overhead ranks roughly around 15%.

The drawback of this method is not the performance overhead of the side aggregations. The rewriting enforces the rough structure of the execution plan and prohibits the remote datasource's optimizer from selecting its very own plan, which might look totally different due to proprietary features, unmapped indexes or additional statistics. Restricting this method to rewrite the queries partially and collect only predicate cardinalities for table scans gets around those problems as soon as the federated optimizer is aware of all available indexes. The restricted query rewriting method is fast and applicable, providing all feedback necessary to detect flaws in single table statistics.

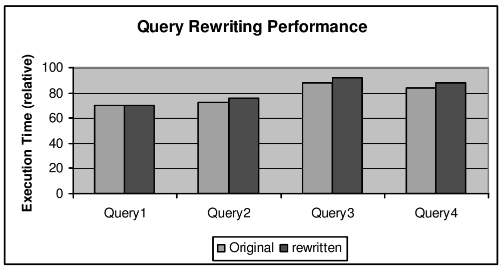


Figure 7: Performance of query rewriting

4.2.3 Table functions

A runtime monitor that collects immediate feedback by piggy-backing on query execution can be implemented by utilizing user defined functions (UDFs). A piped table function that does not modify the data and simply increments a counter for each row it pipes, is inserted between every operator respectively predicate in the QEP to count the intermediate cardinalities as shown in figure 8. The pseudo code for such a function is given in figure 9; operator and predicate id for the UDF's cardinality are passed as parameters.

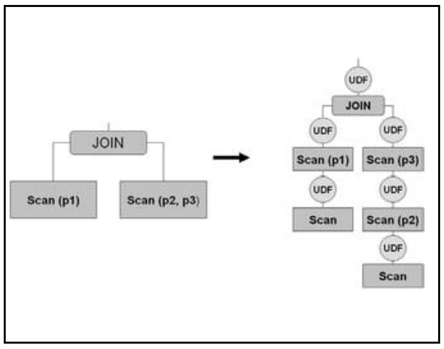


Figure 8: Runtime monitoring through UDFs

```
FUNCTION udf (table tab, int opId, int
predId)
RETURNS TABLE PIPED

INT counter = 0;

BEGIN
   WHILE (next row from tab available)
   {
      PIPE next row;
      counter = counter + 1;
   }

   INSERT opId,predId,counter INTO table;
END
```

Figure 9: Runtime Monitor UDF pseudo code

The overhead of the UDFs compared to the native data stream is extreme. Further more, the federated optimizer enforces his best remote QEP through the insertion of the UDFs, as through rewriting (4.2.2). Figure 10 presents a performance comparison for the same four queries as used in the performance evaluation of the query rewriting method. Shown is the performance of using UDFs for all cardinalities (UDF Comp.) and UDFs for all predicates but no base table cardinality (UDF w/o base card).

A first improvement to this method is not to use the UDFs to obtain the base table cardinality, because this requires the whole table to be read and piped. Instead use UDFs only after the first predicate has been applied and collect the base table cardinality through a count(*) statement at the time of the query feedback analysis. The performance gain of this improvement is illustrated also in figure 10. Some additional applicability can be obtained through restricting this method to monitor base table cardinalities only, as suggested in section 4.2.2, but this method still stays behind the rewriting method.

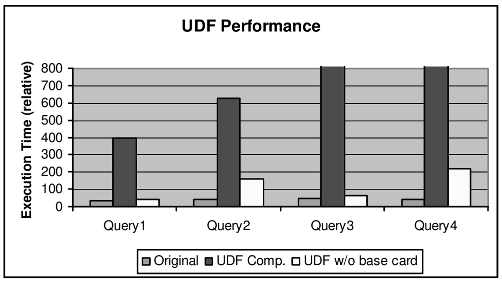


Figure 10: Performance of UDF monitored queries

4.2.4 Count(*) Statements

Deferred feedback on cardinalities for operators and predicates in the QEP can be obtained by issuing count(*) statements that are build upon the parts of the QEP below the targeted operator or predicate. A RM build on this method creates during compile time an additional count(*) statement for each operator in the QEP, containing in the select clause additional constants to identify the associated operator and predicate. To reduce the overhead, these statements can be concatenated using *"union all"*. Further more, certain operators, which do not change the cardinality (e.g. basic sorts), are excluded from the statement. A basic algorithm for creating those statements is given in Figure 11, utilizing the federated server's statement generator.

```
generate_RM (PlanOperator op, String sql) {
   if (op.input_card != op.output_card) { // check if relevant
      // for all predicates
      for (int i = 1; i < op.num_preds; i++) {
         // invoke statementgen for op, including first i predicates
         SQLStatement stmt = translate(op, i);
         stmt.select_clause = "count(*)"
         // nest in case of group by
         if (stmt.groupby_clause)
            sql += "SELECT count(*) from (" +stmt.getFullString()+ ")";
         else
            sql += stmt.getFullString();
         sql += ") UNION ALL (";
      }
   }
   for (int i = 0; i < op.numInputs; i++) // go over children
      generate_RM(op.input[i], sql);
}
```

Figure 11: Algorithm for count(*) RM statements

The overhead of this method is extreme again, Figure 12 shows for the same four queries, as used in the performance analysis of the query rewriting and UDFs, the execution times of the count(*) RM statements (overhead only) and the original queries. The extremely bad performance of those statements originates from the fact that for each operator and predicate, their whole input plan has to be re-executed, yielding an overall complexity class for a statement with n operators of $O(n^2)$. This can be greatly improved by reusing intermediate results through Common Subexpression Elimination (CSE). Instead of translating a separate statement for each plan operator and predicate, a list of common table expressions (CTE) is built at the beginning of the statement, where each CTE consists of the translation of this plan operator alone and references the CTEs of the plan inputs. The actual statement is then simply a union of count(*) statements over each CTE. This corresponds to the query rewriting method without returning the query result and thus without extending the result set. The algorithm is exactly the same (see figure 6), the performance overhead is slightly lower than the execution time of the rewritten statements.

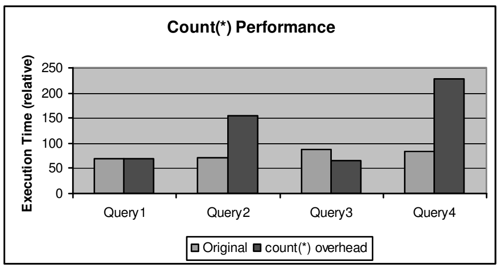


Figure 12: Performance overhead of count(*) RM

The fact that this approach works with deferred feedback brings a lot of advantages. It has no overhead during query execution at all. The overhead in this approach are the count(*) queries themselves, the time of their execution is somewhat independent from the original query for which they were created. For several reasons, it makes sense to issue those queries together with the analysis of the local QF. During this analysis, it can be determined if the remote query part suffers from bad statistics at all, by comparing the cardinalities at the ship operator. Only if the actual cardinalities fall outside a confidence interval, the federated RM would be invoked; that way queries that perform well would not suffer from any overhead. For all queries that are marked to be analyzed, the conjunct predicate sets are duplicate eliminated so predicate subsets that appear in multiple queries are analyzed only once, reducing the overall workload largely.

Updating the statistics though this method can be summed up as a three step procedure. First, find the remote queries that suffer from deficiencies in the statistics by analysis of local QF. Second, profile the statistics that are needed by collecting remote QF for those queries. Third, compute the profiled statistics.

4.2.5 Sampling Statements

This approach is an improvement to the count(*) approach. Restricting it to collect feedback only on table access operators, sampling can be used to drastically reduce the monitoring overhead of the count(*) statements.

Sampling is possible on different levels. A sampling process on row level would provide representative samples usable for analysis, but would not reduce the overhead too much, as the number of I/Os is unrelated high. A lot of pages would need to be read for one single record contained. Hence, big savings are only observed when the records grow very large, as then more pages can be skipped. The level used for this approach is system level, where the pages themselves are sampled, thus reducing the number of I/Os dramatically and speeding up the process. A problem with that method is that the samples obtained through that method might be heavily biased and statistically not robust and representative. This can be due to the fact that data is often clustered on pages with respect to certain columns. To overcome that problem, multiple series of sampling are run with changing sampling rates, observing where the results converge.

This approach comprises all the advantages of an asynchronous runtime monitor with applicable performance and support on the targeted DBMSs.

5. A Query Feedback Analyzer for Federated Queries

The analysis of the QF gathered by the RM works as described in section 3. As the federated system is more complex in architecture, it offers several aspects that can be targeted beyond simple profiling of local statistics on remote data. Several actions are recommended either for the federated server or for the remote datasource.

5.1 Actions on the federated server

A quick response to heavily misestimated cardinalities for remote queries is the creation of statistics on non materialized data for that query, which behaves like a non materialized view that has catalog statistics assigned. During query compilation, the optimizer can match this view to a part of the query and take the output cardinality of this part of the QEP to be the cardinality found in the catalog statistics of the view. When for a remote query the estimated and actual cardinalities deviate greatly, such a view is created from the SQL statement of the remote query and gets the locally observed cardinality at the ship operator assigned. This response has no performance overhead on the remote datasource at all, as it works only with local QF and would not need a RM for the remote query. It has for that particular remote query the effect that the result cardinality can be precisely estimated. However, this has no benefit for the optimization of similar remote queries that differ in their selection of predicates and is thus only applicable for small sets of repeatedly bad performing federated queries.

5.2 Actions on the remote datasource

The federated QFA can indicate that the remote datasource should refresh its statistics. In order to do that, it needs access to cardinality estimates computed by the remote server's optimizer. Most DBMSs come with an explain plan feature where the optimizer selected QEP is stored in a set of tables, which can be used for manual query debugging. For our purpose, those tables are queried by the QFA to obtain the cardinality estimates. Though the plans might deviate, the estimated cardinality at the topmost operator in the QEP, which represents the estimated number of rows in the final result set, is in any case comparable and gives conclusion if the remote server's optimizer has made grave estimation errors for predicates throughout the QEP as a whole. This method is again cheap, as no additional remote runtime monitoring is necessary, since the result set cardinality is observed locally at the ship operator. Still, this method does not provide enough feedback to make recommendations about the remote server's statistics, but it will indicate that the remote server might not be using the optimal execution plan and that actions need to be taken.

Another possible response is that the federated QFA can recommend is the usage of plan hints, or similar features, to push the remote server's optimizer towards certain plan considerations. General usage of those plan hints makes not too much sense, as it could prohibit the remote server from taking advantage of proprietary features such as certain indexes or clustered file structures.

In connection with the previously mentioned way to validate the remote servers estimated result size, it can be used to compensate for join orders or implementations, in case the remote datasource is unable to correctly model the data through its statistics.

6. Case Study

To validate the usefulness of a learning optimizer for federated DBMSs, a small case study will point out what special problems bad statistics on remote data can cause for a federated optimizer and how statistics as recommended by the QFA can lead to QEPs that perform better in orders of magnitude.

This case study illustrates the performance gain through distribution and multi-column statistics on remote data. The database used for this purpose is STEST, a synthetic four table database holding information on cars and accidents. Its schema and setup is illustrated in figure 13. A realistic scenario for such a setup would be the following: The department of motor vehicles holds data about owners and cars, the police maintains an accidents history and the social security offices tracks demographical data. The data used in this database has several correlations and soft functional dependencies. Relations between columns within the same tables are expressed in figure 14, where dotted lines indicate soft functional dependencies and full lines correlations; the thicker the line, the stronger the correlation. Table sizes are 1,000,000 rows for owner and demographics 1,500,000 rows for cars and 2,500,000 rows for accidents.

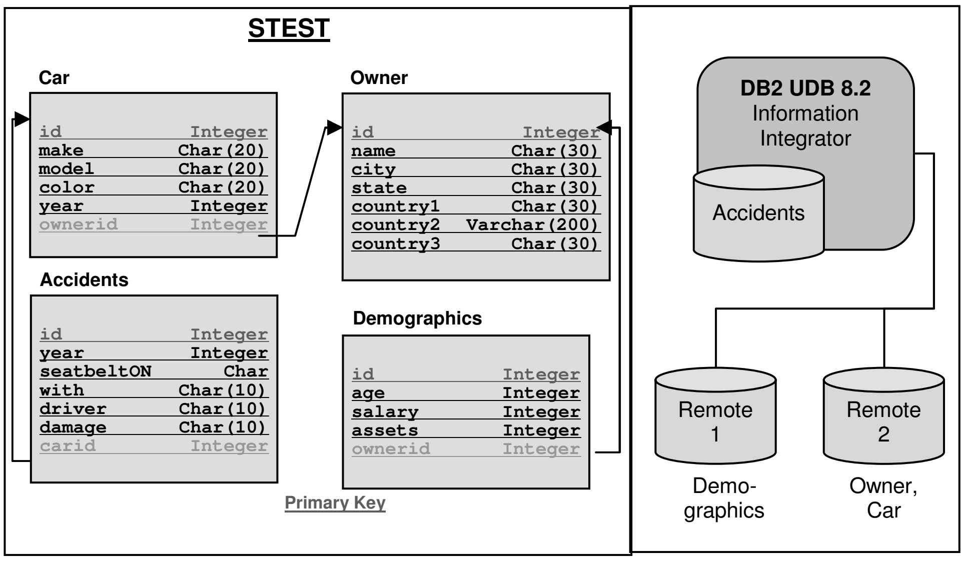


Figure 13: Schema and database setup

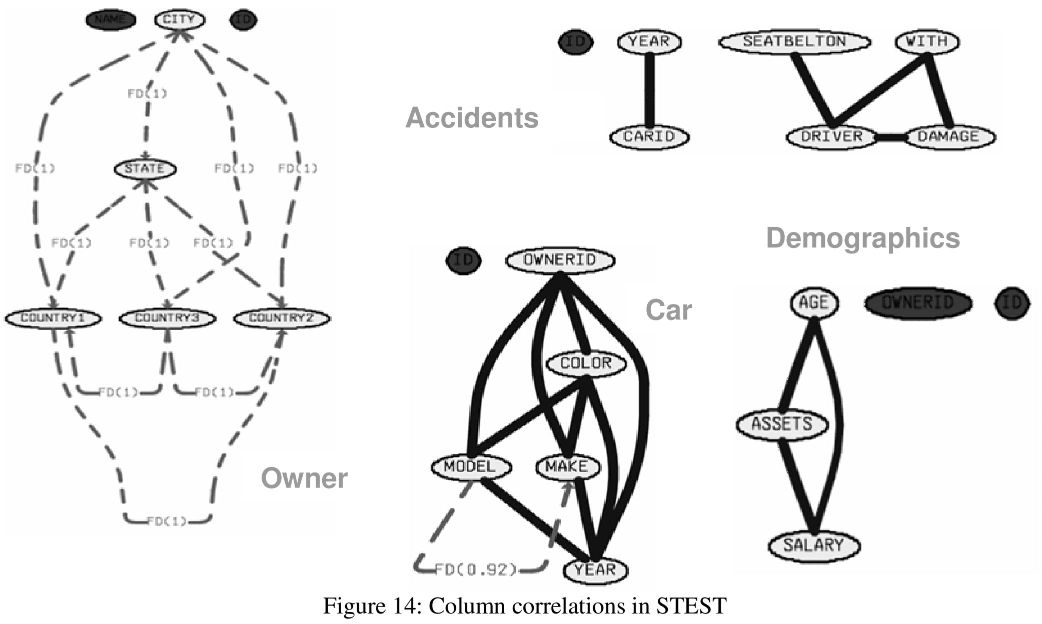


Figure 14: Column correlations in STEST

The runtime monitor used with this scenario provided deferred feedback through the count(*) with CSE method. For the performance comparison, we ran 50 queries on the database where every query joins two to four tables and applies multiple, mostly correlated, predicates.

The scatter plot in figure 15 shows the performance of those queries running with and without the statistics as profiled by the QFA. Note that almost all of the points lie below the line of equivalence, i.e. almost all queries benefited from the adjustments, some in orders of magnitude. Increases in query execution time were small and resulted from small inaccuracies in the cost model. Most queries with average execution times benefited modestly, while 'worst case' queries benefited dramatically.

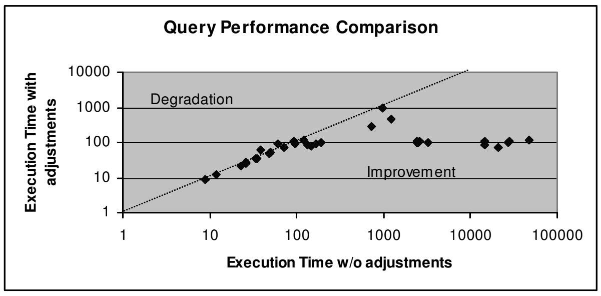


Figure 15: Performance scatter plot

The graph solely illustrates the advantage of column distribution and –group statistics for federated queries, based on query feedback; the overhead of runtime monitoring is not included. This is justifiable as the query execution and runtime monitoring work asynchronously. Furthermore, an autonomic component like this is mostly used in development environments rather than production environments. A common scenario is to enable the learning optimizer while executing sample workloads during development time and have it analyze the database and profile the needed statistics. During production time, it would be switched off and the statistic profiles are used during maintenance time to refresh the catalogue statistics.

Two special issues that arise only in federated queries are the local join strategies and the placement of the ship operator. Both have big impact on the query execution speed and resource consumption. How those issues profit from statistical adjustments is illustrated with two selected queries

6.1 Local join strategies

The joining of data from different sources is performed locally on the federated server. Besides the join order, the type of join operator has a grave impact on the overall performance. How big the impact can be shows the following query, where the tables are distributed (owner and car on remote server 1, accidents on remote server 2).

```
SELECT  o.name, a.driver
  FROM  owner o, car c, accidents a
 WHERE  o.id=c.ownerid
   AND  c.id=a.carid
   AND  o.country3='US'
   AND  o.state='California'
   AND  o.city='San Francisco
```

Through the high correlation between the columns country3, state and city, the estimated cardinalities for all operators above the scan on table 'owner' are very low. The optimizer thus chooses an execution plan as in the left of figure 16, where the table 'accidents' is not completely transferred to be joined locally, but instead queried multiple times and only the rows that match the join predicate are transferred. This is a good strategy if the number of rows to be joined is in fact very small, as the overhead of several scans is potentially smaller than that of transferring millions of records. Since the number of rows is actually a lot higher, this access strategy results in critically bad performance. Correcting for correlations by column group statistics as suggested by the QFA, results in the plan displayed in the right of figure 16. The speedup factor of this query through the recommended statistics is more than 100.

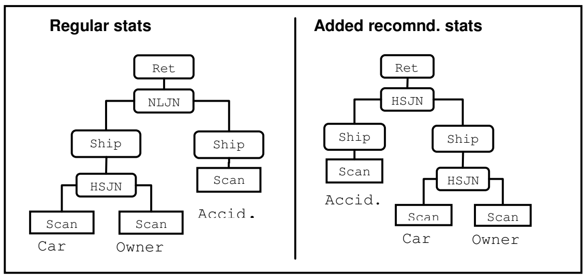


Figure 16: QEPs for federated query

6.2 Workload distribution

Different intermediate cardinality estimates cause the optimizer in some situations to place the ship operator and distribute the workload differently. For example it might choose to have an operator executed locally instead of remotely when this turns out to be cheaper in costs. Suppose a scenario, where the tables referenced in the following query reside all on the same remote server and the query is thus completely pushdownable.

```
SELECT  city, COUNT(*), avg(assets)
  FROM  owner, car, demographics
 WHERE  car.ownerid = owner.id
   AND  demographics.ownerid = owner.id
   AND  make = 'Ford'
   AND  model = 'Taurus'
   AND  salary > 516
GROUP BY city
```

Using tables with schema and data as describes in the general case study, the optimizer originally estimates 31 rows to result from the joins and to transfer those rows and aggregate locally. Due to the heavy correlations in the underlying data, this is vastly underestimated; the actual cardinality after the last join is 125144 rows, which get transferred. After the generation of column group statistics, as suggested by the QFA, the estimation was close enough for the optimizer to push the aggregation down and transfer the aggregated results, 246 rows. The overall reduction in network traffic through this adjustment was by a factor of more than 500.

7. Related Work

This paper discusses Federated database extensions to IBM's DB2 Learning Optimizer. A good overview of the DB2 optimizer can be found in [SACLP79] and [LMH97]. [MLR03] and [ML02] discuss LEO, the optimizer extensions to support the learning framework, which forms the basis of our work.

There are many papers in the literature on the topic of federated optimizer design [LOG93, LD99], but the majority assumes the pre-existence of statistics and focus on the communications protocol and the method of searching the query plan space. Likewise, numerous papers exist discussing extensible optimizers [PGH98, GD87, GM93, PHH92, SJ01]. Our work is orthogonal to these efforts since we focus on obtaining better selectivity statistics for the complex subclass of correlated columns.

Our work essentially centers on learning the cost models of the underlying data sources. [DH02] is closest to our work when it writes how the "cost of costing" in a federated database is a major factor in the overall cost. Unlike our approach, which allows data sources to remain autonomous, [DH02] uses a distributed set of optimizer/ bidder components. More importantly, it assumes that accurate statistics are already available and focuses on a distributed negotiation of these statistics across its optimizer/ bidder components, while we are focused on the practical problem of obtaining such statistics in the first place. Incidentally, [DH02] also uses as a base a System-R type optimizer [SACLP79] as its core.

In [ZL94], the authors obtain accurate estimates for the cost parameters (e.g., table, index access costs, etc.) by executing remote queries from several carefully chosen categories. Categories are based on the existence of indices, predicates with constants, etc. [DKS92] is a similar paper where the authors focus on obtaining estimates for the cost parameters themselves. Our approach differs in that we already know of the existence of indices and other access paths, but also know the base table access costs. What our approach focuses on is to determine the selectivity parameters for correlated columns in a federated database.

8. Conclusions

Our approach extends a learning optimizer for non-federated databases with federated technologies. We have shown a set of methods to implement a runtime monitor for remote queries, providing immediate feedback during query execution or deferred feedback. By helping federated systems to learn from underperforming queries, this method pushes the idea of autonomic computing further into federated environments.

A prototype of an asynchronous runtime monitor has been implemented into a development build of DB2 UDB 8.2. The case study based on this prototype shows, how reliable statistics that correctly model the remote data are very important for federated systems to estimate the query result sizes. Performance gain for queries, especially when joining over several remote sources, can be in orders of magnitude.

Future work on this topic is the improvement of the runtime monitor through sampled count(*) queries with dynamically adjusted statistical confidence intervals. Further more, analysis of the remote server's QEP and the included estimates from an explain interface, their validation and methods to use those as a replacement query feedback are to be explored as follow up.

References

[AHLL04] A. Aboulnaga, P. Haas, S. Lightstone, G. Lohman, V. Markl, I. Popivanov, V. Raman. Automated Statistics Collection in DB2 Stinger. Proc. VLDB 2004.

[DH02] Amol Deshpande, Joseph M. Hellerstein. Decoupled Query Optimization for Federated Database Systems. Proc. IEEE ICDE, 2002.

[DKS92] W. Du, R. Krishnamurthy, and M.-C. Shan. Query optimization in a heterogeneous DBMS. Proc VLDB, 1992.

[GD87] G. Graefe, D. J. Dewitt. The EXODUS Optimizer Generator. In Proc ACM SIGMOD, 1987.

[GM93] G. Graefe, W. McKenna. The Volcano Optimizer Generator: Extensibility and Efficient Search. In Proc 12th IEEE ICDE, 1993.

[IBM02] DB2 Universal Database for iSeries - Database Performance and Query Optimization. IBM Corp., 2002.

[IBM04] DB2 v8.2 Performance Guide. IBM Corp., 2004.

[IMHB04] I. F. Ilyas, V. Markl, P. J. Haas, P. G. Brown, A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. Proc. 2004 ACM SIGMOD, June 2004.

[LD99] Anhai Doan and Alon Levy. Efficiently ordering query plans for Data Integration. Proc. IEEE ICDE 1999.

[LLZ02] S. Lightstone, G. Lohman, D. Zilio. Toward autonomic computing with DB2 Universal Database. SIGMOD Record, 31(3), 2002.

[LMH97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, Jun Yang.Optimizing Queries across Diverse Data Sources. Proc. of the 23rd VLDB conference.

[LOG93] Hongjun Lu, Beng-Chin Ooi, Cheng-Hian Goh. Multidatabase Query Optimization: Issues and Solutions. Proc RIDE, 1993.

[ML02] Volker Markl, Guy Lohman. System performance and benchmarking: Learning table access cardinalities with LEO. Proc. ACM SIGMOD June 2002.

[MLR03] V. Markl, G. M. Lohman, V. Raman. LEO: An autonomic query optimizer for DB2. January 2003 IBM Systems Journal, Volume 42 Issue 1

[PGH98] Yannis Papakonstantinou, Ashish Gupta, Laura Haas. Capabilities-Based Query Rewriting in Mediator Systems. Proc. 4th International Conference on Parallel and Distributed Information Systems, 1998.

[PHH92] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In Proc. ACM SIGMOD, June 1992.

[SACLP79]P. G. Selinger, M. M. Astrahan, D. D. Chamberlain, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database. Proc. ACM SIGMOD, pp23-34, 1979.

[SJ01] Giedrius Slivinskas and Christian S. Jensen. Enhancing an Extensible Query Optimizer with Support for Multiple Equivalence Types. Lecture Notes in Computer Science, vol. 2151, 2001.

[SLMK01] M. Stillger, G. M. Lohman, V. Markl, M. Kandil. LEO - DB2s Learning Optimizer. Proc. 27th VLDB, 19-28, 2001.

[ZL94] Q. Zhu and P.A. Larson. A query sampling method of estimating local cost parameters in a multidatabase system. Proc IEEE ICDE, 1994.