POP/FED: Progressive Query Optimization for Federated Queries in DB2

Holger Kache
IBM Silicon Valley Lab
555 Bailey Avenue
San Jose, CA, USA
kache@us.ibm.com

Wook-Shin Han, Volker Markl, Vijayshankar Raman
IBM Almaden Research Center
650 Harry Road
San Jose, CA, USA
{wookshin,marklv,ravijay}@us.ibm.com

Stephan Ewen
IBM Boeblingen Lab
Schönaicher Str. 220
71032 Böblingen, Germany
ewens@de.ibm.com

ABSTRACT

Federated queries are regular relational queries accessing data on one or more remote relational or non-relational data sources, possibly combining them with tables stored in the federated DBMS server. Their execution is typically divided between the federated server and the remote data sources. Outdated and incomplete statistics have a bigger impact on federated DBMS than on regular DBMS, as maintenance of federated statistics is unequally more complicated and expensive than the maintenance of the local statistics; consequently bad performance commonly occurs for federated queries due to the selection of a suboptimal query plan. To solve this problem we propose a progressive optimization technique for federated queries called POP/FED by extending the state of the art for progressive reoptimization for local source queries, POP [4]. POP/FED uses (a) an opportunistic, but risk controlled reoptimization technique for federated DBMS, (b) a technique for multiple reoptimizations during federated query processing with a strategy to discover redundant and eliminate partial results, and (c) a mechanism to eagerly procure statistics in a federated environment. In this demonstration we showcase POP/FED implemented in a prototype version of WebSphere Information Integrator for DB2 using the TPC-H benchmark database and its workload. For selected queries of the workload we show unique features including multi-round reoptimizations using both a new graphical reoptimization progress monitor POPMonitor and the DB2 graphical plan explain tool.

1. INTRODUCTION

In a federated database, data is integrated from different remote data sources, without the requirement to replicate or otherwise copy the data to the federated database instance. Federated databases use references instead that point to the objects living in the remote data source. They are called *nicknames*, or *index specifications* and point to a physical object in the remote data source. While in a non-federated database, the query execution plan defines an access strategy for the local relational objects that reside in the database where the plan was compiled, a federated query execution also includes an access strategy for the objects in the remote data source. Based on the complete cost model, the query optimizer will choose the optimal global query access plan with regards to the total query execution time. It will consider different join strategies for the join of the remote and local data, different join orders, different points to transfer data between data sources for multiple joins, trade off between local and remote joins, and even push down of predicates to a remote data source as opposed to local processing.

The federated query optimizer, however, does not influence the remote query access plan. It sends a SQL string representing the remote sub-statement to the remote data source only and the plan decision for the sub-statement is entirely left to the remote data source. The remote data source, in turns, compiles the sub-statement and generates a local access strategy for the objects that it owns. It may or may not employ a query optimizer to do that, depending on the nature and capabilities of the data source. The federated server retrieves the data returned by the remote data source to join it with data from another remote data sources or local data stored at the federated server itself. At this point in time it executes the federated query plan.

2. DB2 FEDERATED QUERIES

In DB2, federated queries are prepared and processed like regular relational queries with a few extra steps specifically introduced for federated queries. In the query prepare phase, the query is first parsed and rewritten using semantic rules. During that phase, the query compiler determines which quantifiers, columns, and predicates can be sent to which remote data source and marks them with a push-down flag. In the next phase, the query is optimized with respect to that push-down flag using a cost-based optimizer. The plan costs are computed differently for the local parts of the plan and the parts that are to be processed by the

VLDB '06, September 12–15, 2006, Seoul, Korea.

remote data sources. After the query is optimized and the access strategy is clear, the query compiler generates the SQL string for every sub-statement to be sent to a remote data source. During query execution, DB2 uses the common APIs available for the remote data source to submit the SQL statements and fetches the results back into local data structures. The results are retrieved and the federated server starts processing the local plan operators using the results from the remote sources. In the final fetch phase, it delivers the query results back to the requesting application.

Cost based optimization of federated queries transparently extends optimization across data sources, by introducing communication cost, but otherwise treating remote tables similar to local tables and by introducing a *source-* or *server* property that describes where the processing of the current plan operator happens. A special operator (SHIP) describes the point in the QEP where intermediate results are communicated between a remote data source and the federated DBMS. The statistics that are used to estimate cardinalities for remote base tables are in most cases obtained from the remote data source, since the gathering of statistics on remote data is very expensive for the federated DBMS. The varieties of relational DBMSs, which can be a remote source, employ different optimizers and utilize different forms of statistics. Often, the federated server can only exploit very basic statistics about the number of rows in a table. The federated DBMS's optimizer is hence not able to model data distribution and correlation in detail, as this would require distribution and multivariate statistics. The worst cases are federated queries that access non-relational remote data sources or remote DBMSs that do not employ a cost based optimizer. In those cases, there are no statistics on the remote data available at all and the optimizer is forced to derive its cardinality estimates from default values.

3. POP/FED OVERVIEW

3.1 Concept

Progressive Optimization (POP) is a technique that breaks with the fixed sequence of query prepare, execute, and fetch. It is a compromise between static optimization and continuous dynamic optimization which allows us to optimize a query again during query runtime. Especially for federated queries, where the query execution phase is exceptionally complex and includes the remote query compilations, remote execution, and remote fetch phases, followed by the actual local execution, it makes sense to allow an additional query compile cycle. That compilation can now be based on actual cardinality values as discovered during query runtime, rather than cardinality estimates.

During the initial query compilation, POP determines criterions for estimated parameters that are required to hold if the plan is to be the optimal one. The current prototype uses only the estimated cardinality, which is the most important parameter and also the one subject to the gravest estimation error. It computes the *validity range* around it, an interval that describes for which cardinality range the current plan is truly the optimal one. It then places CHECK operators at strategic points, which in turn validate during plan execution that the actual cardinality, obtained from the runtime monitor, is within the validity range. If this is not the case, all intermediate results from fully materialized points are retained and the optimizer is called again. The actual cardinalities from the aborted query execution are made available to the optimizer so that it is able to develop a better plan, which is not subject to the estimation error that caused the reoptimization. Note that this makes POP suitable for any source of cardinality estimation error, be it bad statistics, wrong assumptions, or parameter markers. The retained intermediate results are treated as *materialized views*, also called *materialized query tables (MQT)* or *automatic summary tables* in DB2 [6]. The optimizer has the cost based choice to match them back into the plan, enabling the query to basically continue from the point it was aborted for reoptimization, avoiding the re-execution of previously executed parts. Figure 2 shows an example of this process.

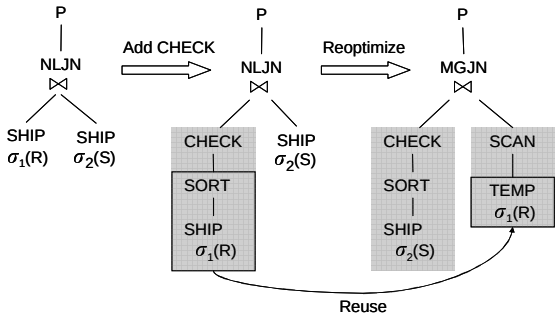


Figure 1. POP reoptimizing a sub-optimal nested loop join for 2 federated sources.

The left side shows the initial plan for the example used above. σ_1(R) represents the sub-statement to read from the owner table and σ_1(S) the statement to read from the car table. During optimization, POP computes the validity ranges around the edges of the plan and places CHECK operators at places that are suitable or performance critical. The CHECK operator, in this case with artificial materialization, takes the validity range of its child edge as parameter. During runtime, it identifies whether the actual cardinality is within validity ranges, and triggers reoptimization if not. The optimizer uses knowledge about the actual cardinality to develop the new plan; the intermediate result is matched into the plan as a temporary table (right side). [4] introduces different flavors of check operators for eager checking (tuple pipelines) and lazy checking (full materialization points in a QEP). The current prototype supports only the lazy variant, which is also the preferable one for federated queries, as it solely supports

the re-use of intermediate results; an effect that we very much want to utilize for federated queries to reduce communication cost. Furthermore, so far no research has proposed a good way to determine the validity range for eager checkpoints, which has to consider the cost inherent to partial re-execution.

3.2 Multiple Reoptimizations

For federated queries, the number of reoptimizations is commonly as high as the number of uncorrelated SHIP operators in the federated query plan, possibly higher if correlation on join predicates that span several SHIPs occurs. One potential problem associated with multiple rounds of reoptimization is the stockpiling of partial results, as each iteration introduces new temporary tables. POP is not forced to reuse partial results but rather performs the decision to reuse them on a cost base. Through this mechanism, it occurs that POP ignores partial results but reconsiders them after another round of reoptimization or decides to fall back to another partial result; this happens especially when new knowledge that was added in the course of another reoptimization compensated for correlation on join predicates. It is consequently dangerous and regressive to throw away partial results as soon as POP does not consider them during a reoptimization.

POP/FED provides a technique for multiple reoptimizations with a strategy to discover redundant and eliminate partial results. Dropping redundant partial results here ensures that the DBMS processes the query, and also other concurrently running queries, with the maximum possible temporary storage space. However, when we drop a redundant partial results, we keep statistics as a virtual statistical view.

Figure 2 shows a screenshot for POPMonitor, which is a new graphical progress monitor tool for the demo. In each round of reoptimization, POPMonitor 1) invokes the db2 graphical explain tool showing the reoptimized plan, and 2) displays partial results maintained as virtual materialized views and statistical views. With POPMonitor, one is able to understand how POP/FED reoptimizes plans and maintains partial results throughout multiple rounds of reoptimization. As shown in this figure, POPMonitor also shows QGM information for a given virtual materialized view using a pop-up window.

4. DEMONSTRATION

We demonstrate POP for federated queries with the TPC-H workload as defined by the Transaction Processing Performance Council (TPC). The TPC-H tables are distributed amongst an Oracle database and a DB2 database. The federated queries are executed using the IBM Information Integrator product, which implements the DB2 federated query processing functionality. POP/FED applied

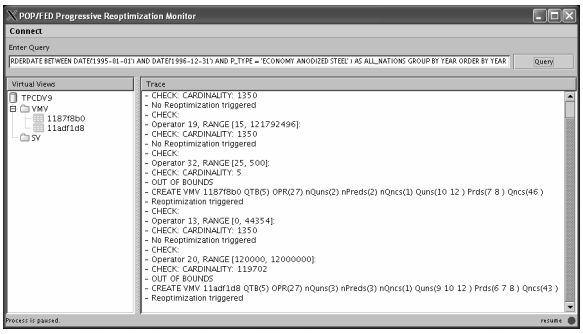


Figure 2. Screenshot for POPMonitor.

to the federated TPC-H queries demonstrates its ability to improve the quality of the federated query plans. POP does not interfere with the remote plan compilation and it is apparent that only the local portion of the plan can be optimized. Therefore, we will demonstrate queries that have complex local plans (e.g. complex joins) and put a high load on the federated server.

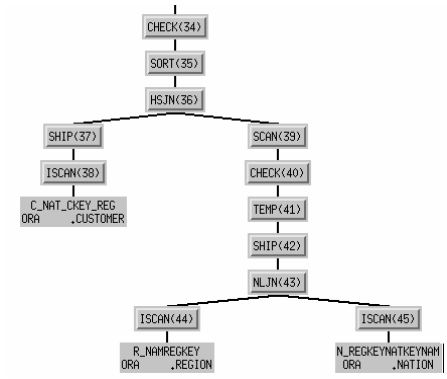
The story-line of the demonstration is the following.

1. We start with a partially loaded TPC-H Oracle database. Only the PART, SUPPLIER, CUSTOMER, NATION, and REGION tables are loaded and statistics are updated for the tables and indexes. The missing LINEITEM and ORDERS tables are loaded into a DB2 database, which is enabled as a federated database.

2. Federated connection to the remote Oracle database is set up as documented in [3], and nicknames to the Oracle tables are created in the local schema 'ORA'. Since we created the nicknames after we populated the data into the Oracle TPC-H database, we will automatically pick up the correct federated statistics for the underlying Oracle base tables.

3. To demonstrate a number of capabilities of POP for federated queries, we run TPC-H query 8 using the Oracle and DB2 tables.

4. The initial plan compiled by the federated query compiler uses a local hash join for the results of the remote NATION/REGION join and the remote CUSTOMER table. That is the cheapest option only if CHECK(40) returns at least 25 rows.

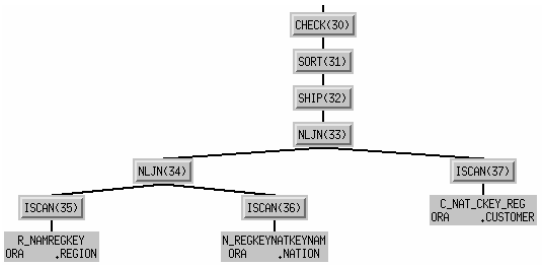


5. The remote statement represented at SHIP(42) is

```
SELECT A1."N_NATIONKEY" FROM
"TPCH"."REGION" A0, "TPCH"."NATION" A
WHERE (A0."R_NAME" = 'AMERICA') AND
(A1."N_REGIONKEY" = A0."R_REGIONKEY")
```

   and returns 5 rows. At CHECK(40) POP/FED interrupts query execution and creates a virtual materialized view consisting of 2 quantifiers (NATION, REGION) and 2 predicates (R_NAME, N_REGEIONKEY).

6. Reoptimization is triggered and a POP/FED proposes a new plan that pushes down the 3 table join between Oracle NATION, REGION, and CUSTOMER tables.



   The materialized view created in 6 is not used because the quantifiers and predicates don't match for the 3 table join.

7. During execution of the new plan, a second reoptimization is triggered by another CHECK point in the plan. POP/FED materializes the new results for the 3 table join into another virtual materialized view

```
CREATE VMV 11bdf1d8 QTB(5) OPR(27)
nQuns(3) nPreds(3) nQncs(1) Quns(9 10 12
) Prds(6 7 8 ) Qncs(43 )
```

8. The third plan proposed by POP/FED reuses the results materialized in step 7. The quantifiers and predicate lists match and the cost of computing the result suggests the use of the materialized virtual view.

5. CONCLUSION

POP/FED, an extension of POP for a single local source queries, is a powerful technique for progressively reoptimizing federated queries. The problem of incorrect or incomplete statistics is far greater for federated queries than for non-federated queries. The federated query compiler has to make assumptions about the complexity and costs of remote statements without actual knowledge about the remote query plans. Thus, the federated query plan faces the danger of being sub-optimal. POP/FED reoptimizes queries for an arbitrary number of times and avoids wasting storage space by analyzing partial results for redundancy and cleaning up after each reoptimization. For federated queries that were optimized with little knowledge, early materialization reorders the subplans in a way that data access, in the federated case access to the remote results, is done prior to the actual plan execution. It provides knowledge about actual cardinalities earlier and reduces number of reoptimizations. Through a more evenly provided knowledge, the optimizer runs less risk of getting into a plan bias.

6. REFERENCES

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

[2] Ewen, S., Kache, H., Markl, V., and Raman, V., Progressive Query Optimization for Federated Queries. *Proc. EDBT 2006 (accepted).*

[3] IBM DB2 Information Integrator *Federated Systems Guide* Version 8.2, IBM Corp 2004.

[4] Mark, V., Raman, V., Simmen, G., Lohman, G., Priahesh, H., and Cilimdzic, M. Robust Query Processing through Progressive Optimization. Proc. ACM SIGMOD 2004.

[5] Stillger, M., Lohman, G., Markl, V., and Kandil, M. LEO – DB2's Learning Optimizer, Proc. VLDB 2001.

[6] Zaharioudakis, M., Cochrane, R., Lapis, G., Pirahesh, H., and Urata, M. Answering Complex SQL Queries Using Automatic Summary Tables, Proc. ACM SIGMOD 2000.