Progressive Optimization in Action

Vijayshankar Raman Volker Markl David Simmen Guy Lohman Hamid Pirahesh

IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
U.S.A.
{ravijay,marklv,simmen,lohman,pirahesh}@us.ibm.com

Abstract

Progressive Optimization (POP) is a technique to make query plans robust, and minimize need for DBA intervention, by repeatedly re-optimizing a query during runtime if the cardinalities estimated during optimization prove to be significantly incorrect. POP works by carefully calculating validity ranges for each plan operator under which the overall plan can be optimal. POP then instruments the query plan with checkpoints that validate at runtime that cardinalities do lie within validity ranges, and re-optimizes the query otherwise. In this demonstration we showcase POP implemented for a research prototype version of IBM's DB2 DBMS, using a mix of real-world and synthetic benchmark databases and workloads. For selected queries of the workload we display the query plans with validity ranges as well as the placement of the various kinds of CHECK operators using the DB2 graphical plan explain tool. We also execute the queries, showing how and where re-optimization is triggered through the CHECK operators, the new plan generated upon re-optimization, and the extent to which previously computed intermediate results are reused.

Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004

1. Introduction

Virtually every commercial query optimizer chooses the best plan for a query using a cost model that relies heavily on accurate cardinality estimation. Cardinality estimation errors can occur due to the use of inaccurate statistics, invalid assumptions about attribute in-dependence, parameter markers, and so on. These errors can lead to substantially sub-optimal plans.

"Progressive query optimization" (POP) is an approach to make query processing more robust, and substantially reduce the need for DBA intervention to debug problem queries. POP makes query plans robust by automatically detecting and recovering from cardinality estimation errors. POP validates cardinality estimates against actual values as measured during query execution. If there is significant disagreement between estimated and actual values, execution might be stopped and re-optimization might occur. Oscillation between optimization and execution steps can occur any number of times. A re-optimization step can exploit both the actual cardinality and partial results, computed during a previous execution step. Checkpoint operators (CHECK) validate the optimizer's cardinality estimates against actual cardinalities. Each CHECK has a condition that indicates the cardinality bounds within which a plan is optimal. We compute this validity range through a sensitivity analysis of query plan operators. If the CHECK condition is violated, CHECK triggers re-optimization. POP places CHECK operators judiciously in query execution plans.

POP is implemented in a research prototype version of IBM's DB2 DBMS, and an experimental evaluation of POP using TPC-H queries as well as a real-world data base and workload showed that POP can provide speedups up to two orders of magnitude for complex OLAP queries [1].

Figure 1 depicts the control flow of POP. Upon compilation of a SQL query, the optimizer generates a variety of alternative query execution plans. Whenever

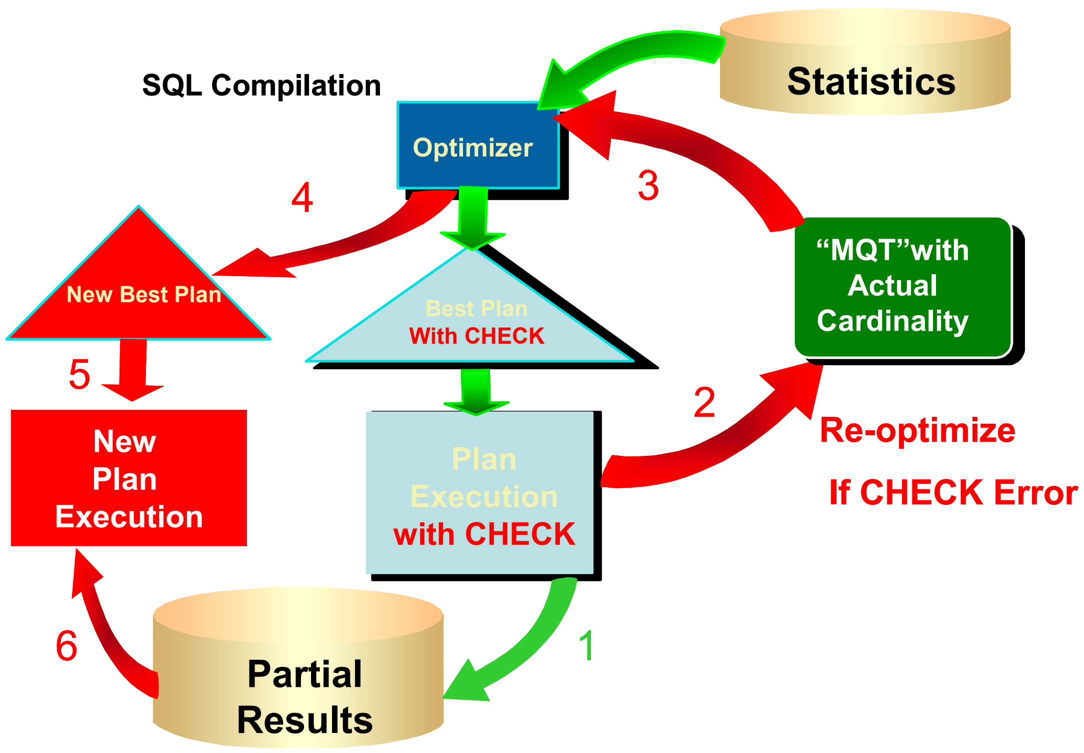

Figure 1: Overview of Progressive Optimization

one plan is pruned by another plan during optimization, POP uses a variant of the Newton-Raphson method to determine the cardinality at which the cost functions of the dominating and the dominated plan intersect. This cross-over cardinality is used to continuously refine the validity range of the optimal plan. Once the optimal plan has been determined, POP adds CHECK operators at selected points in the plan. During query runtime, these CHECKs act as safeguards to ensure that the cardinality of tuples flowing through them is indeed within the validity ranges determined during optimization.

In the Figure, step 1 shows the initial execution of the plan with the check, and the computation of intermediate results. If the validity range for any CHECK operator is violated, re-optimization is triggered (step 2), creating a materialized query table "MQT" for each intermediate result computed by the previous partial execution. The actual cardinalities of these MQTs are fed back into the optimizer (step 3), and the query is optimized again (step 4). The optimizer also generates access plans over the MQTs into its mix of possible plans, and thus has the option to reuse these pre-computed results. This second iteration (steps 5 and 6) can again employ CHECK operators and start the re-optimization cycle again, thus continuously improving the query plan over several iterations.

Further details on POP can be found in [MRS+04], available upon request for the reviewers. Further references and a discussion of related work can also be found in [MRS+04] and are omitted in this proposal in order to describe the demonstration system in more detail.

2. Demonstration

We demonstrate our implementation of POP using a research prototype version of IBM's DB2 DBMS. We show a case study with a workload of queries that are hard to optimize accurately using typical optimizer statistics such as single-column histograms. The databases we are going to use to demonstrate POP are (1) a DMV database storing cars and accidents, as well as (2) a TPC-H database. The DMV database stores tables of car models, makes, owners, accident records, and owner demographics. The main feature of this database is extensive correlation between various columns, both within and between tables. Next to these correlations, other source of error in our workloads for TPC-H and DMV are LIKE predicates and parameter markers. These errors in many cases cause the optimizer to choose a suboptimal plan. POP detects this during runtime, as the validity range for a specific part of a query plan is violated, and triggers re-optimization. In the following we describe the two major components of our demonstration: (1) the validity range computation and CHECK placement, and (2) the re-optimization of an example query.

3.1 Validity ranges and CHECK placement

The first part of our case study is to use our graphical explain tool to demonstrate validity ranges and checkpoint placement. We will use our graphical explain tool to visually navigate the query plan for each of our example queries. The query plan is shown as a tree, and clicking on tree nodes (plan operators) pops up a window with various properties of these operators. One of the properties is the validity range – the range of cardinalities flowing out of this operator under which this plan could be optimal. The span of these validity ranges denotes the robustness of these plans to cardinality errors, independent of whether we re-optimize or not. We will pick a single query, we will force alternative query plans by altering optimizer configurations, and show the validity ranges, to highlight the tradeoff between plan robustness and plan cost.

We will also use these plans to illustrate the different flavors of CHECK operators (see [MRS+04]) and the tradeoff between their risk and opportunity. Looking at the plan below a CHECK helps us gauge how much of the work below the CHECK will have to be redone upon re-optimization, and thus how risky it is. Likewise the distribution of CHECKs in the query plan shows the opportunity provided by each kind of CHECK.

3.2 Re-Optimization of an example query

The second part of our case study is the complete execution of example queries, with POP enabled. The query is initially submitted and optimized. After showing this initial query plan, we let the plan run until a validity range is violated. The execution engine outputs a warning message that cardinality estimates are highly inaccurate. Then the system automatically goes through another round of optimization. In the demo, again using the DB2 graphical explain tool, we show the new plan that has been computed during re-optimization, using the knowledge about actual cardinalities, correlations, parameter marker values, etc. learned from the previous partial execution. Figure 2 shows both the initial plan and the re-optimized plan for a query listing all lineitems for all orders placed in the 1990s against the TPC-H database:

```
SELECT * FROM ORDERS, LINEITEM
WHERE L_ORDERKEY = O_ORDERKEY
AND  O_ORDERDATE LIKE '%199%'
```

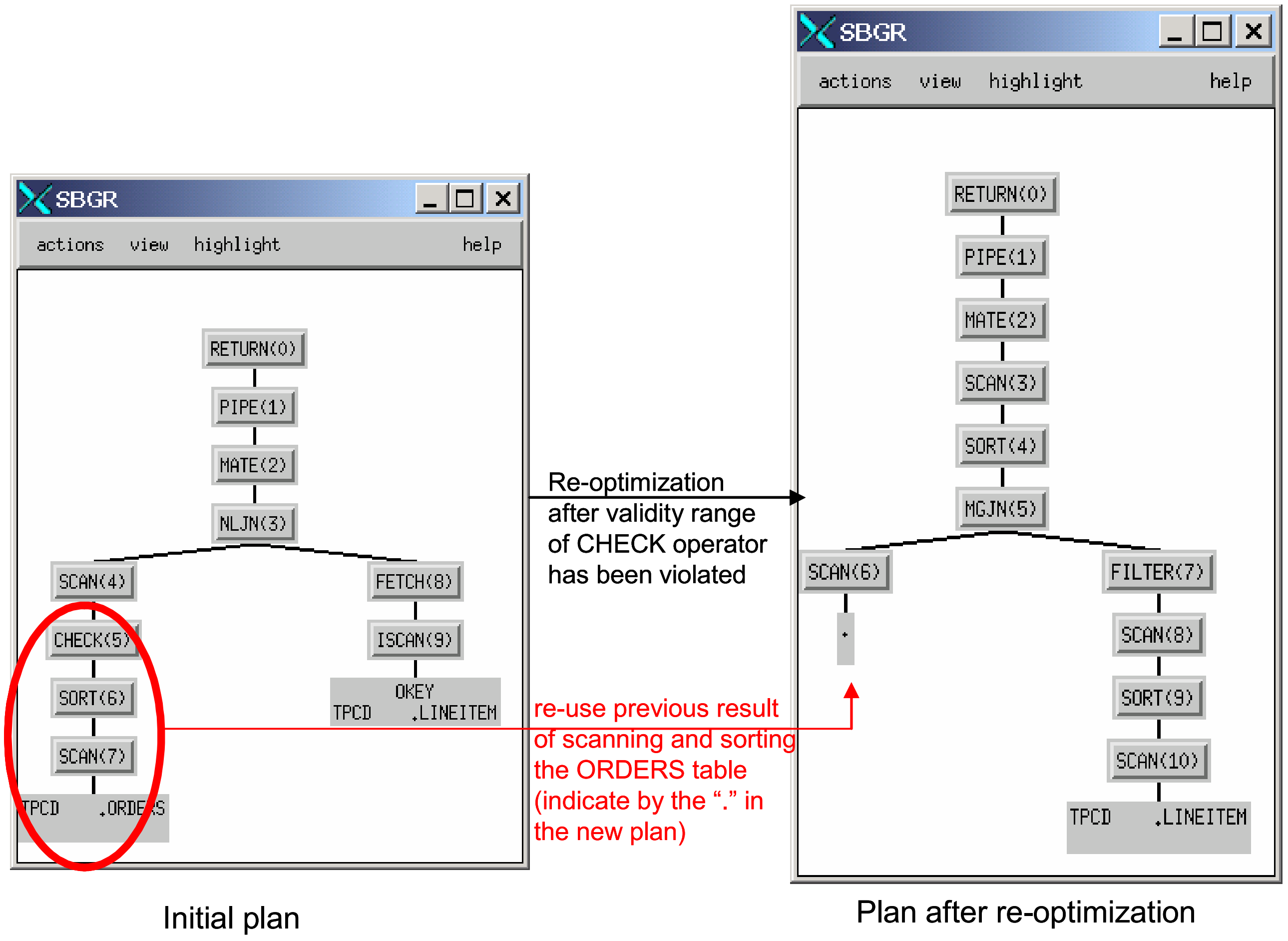

Figure 2: Comparing the Initial Plan with the Plan after Re-optimization

The initial plan for that query located at the left of Figure 2 shows that POP has placed a CHECK operator as operator number 5 into the plan to safeguard the nested-loop join (NLJN, operator 3) by ensuring that the cardinality of the outer leg of the nested loop join is within bounds. Because of the LIKE predicate, the output cardinality for the SCAN and SORT operators is highly underestimated (most of the records in the database are from the 1990s). This causes the CHECK operator to trigger re-optimization, which changes the initial plan to the plan displayed in the right part of Figure 2. Note that this plan has changed the join method from a nested-loop join (NLJN in the Figure) to a merge-join (MGJN in the Figure). Also note that the new plan re-uses the intermediate result generated by the partial execution of the initial plan. This intermediate result (indicated by a "dot" in the new plan in the right part of Figure 2) contains a subset of the ORDERS table after the application of the LIKE predicate and is already sorted by ORDERKEY.

This query runs orders of magnitude faster with POP than it would when continuing along the initial plan, as we also show in our demonstration by running the query both with and without POP enabled.

For the demonstration of POP on the DMV database, most estimation errors are not due to like predicates, but due to correlations within and between tables, parameter markers, and user-defined functions. All of these sources of errors can trigger re-optimization because of a violation of the validity ranges. Our demonstration also includes showing the robustness POP adds to query optimization for these sources of errors.

3. Conclusions

Progressive Optimization (POP) is a major step towards making query optimization a dynamic process, where the DBMS continually adjusts the plan based on a closed feedback loop between the runtime and the optimizer. Our demonstration shows the effectiveness and efficiency of POP.

We illustrate where CHECK operators are placed in query plans in a research prototype version of IBM's DB2 DBMS. We also show the conditions under which re-optimization takes place, using both a real-world DMV database and the TPC-H database. POP makes query plans more robust to estimation errors; and speeds-up query execution by orders of magnitude for some realistic queries.

4. References

MRS+04 V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh. Progressive Optimization for Robustness in Query Processing. SIGMOD 2004.

This full paper has detailed references to other related work.