Lecture #15: Optimizer Implementation III

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Introduction

Traditional query optimizers generate a query plan before the DBMS executes the query. The optimal plan, however, changes as the database's state evolves over time. This can be due to:

1. Changes in the physical design of the database
   - Adding or dropping indexes
   - Partition scheme changes
2. Changes in data
   - Adding or removing tuples
   - Change in the distribution of values for columns
3. Prepared statement parameters can change per invocation
4. If ANALYZE was run more recently, the optimizer's decision may be completely different

All this is to say that query plans that were **once** good may not **always** be good. Bad query plans may result from the changes listed above. If we are able to detect bad query plans (typically from poor join ordering), we may be able to *adapt* them instead of optimizing again from scratch.

1.1 Optimization Timing

There are three timings for query optimization.

1. Static Optimization
   - Select the best plan prior to execution
   - Plan quality dependent on cost model accuracy
2. Dynamic Optimization
   - Select plans on-the-fly, as queries execute
   - Each execution has a different optimization
   - Hard to implement and debug (non-deterministic)
3. Adaptive Optimization
   - Compile a query plan statically
   - If the estimation errors exceed some threshold, change the plan or re-optimize entirely.

In this lecture, we will be focusing on adaptive optimization.

2 Adaptive Query Optimization

The main idea of adaptive query optimization is detecting a bad query plan at execution time, and improving it. The query optimizer's estimates can be evaluated during execution, as the information becomes available [1].

2.1 Approaches

1. Modify Future Invocations
    - As we get information during execution, report it back to the optimizer.
    - Help the optimizer generate a better plan next time.
    - Current plan remains suboptimal.
2. Replan Current Invocations
    - If we have a bad plan, go back to the optimizer and start over.
    - Discard execution progress entirely.
3. Plan Pivot Points
    - Switch from one strategy to another mid-execution—do not go back to the optimizer.
    - Improve the current plan *while we are executing it.*

3 Modify Future Invocations

In this approach, we are focusing on improving future query plans. For example, if, while executing a scan, we found that the true selectivity is very different from the estimated selectivity, we can modify future invocations with the true selectivity.

1. Plan Correction
    - Fix the given query plan, for next time.
2. Feedback Loop
    - Merge the information into the statistical catalog.
    - Help future query cost estimations.

3.1 Reversion-based Plan Correction

In this strategy, the DBMS tracks the history of query invocations. This includes the cost estimations from the optimizer, the generated query plan, and the actual metrics gathered at runtime.

Then, when the DBMS generates a new plan for a query we have seen before, we can compare it to a previous plan. If it regresses (performs worse), we can switch back to an earlier plan.

For example, if we run the same query again after creating new indexes, it may be able to take advantage of index scans. However, the cardinality estimates may have resulted in a poor choice of physical join operator. If the new plan is worse, despite the indexes, we can throw it away and use the old query plan for future invocations.

4 Plan Stitching

Plan stitching builds on the reversion-based plan correction approach described in the previous section by allowing us to "stitch" logically equivalent sub-plans from different query plans together, to form an even better plan. [3]

Approach:

1. Identify which sub-plans are logically equivalent
2. Generate a graph containing all possible sub-plans, where OR operators indicate alternate paths
3. Perform bottom-up search in that graph, picking the best/cheapest node at each operator

4.1 Identifying Equivalent sub-plans

Determining if any arbitrary sub-plans are logically equivalent is undecidable. Instead, we can define equivalence as having the same logical expression and physical properties. We can also prune sub-plans that will never be equivalent using simple heuristics (e.g. accessing different sets of tables).

4.2 Encoding Search Space

The search space is encoded as a graph containing OR operators, where each OR operator represents an alternate ”path” through the plan, i.e. a different sub-plan.

Later, a bottom-up search is performed through this data structure. To find the best plan in this search, at each OR operator, the cheapest sub-plan is selected.

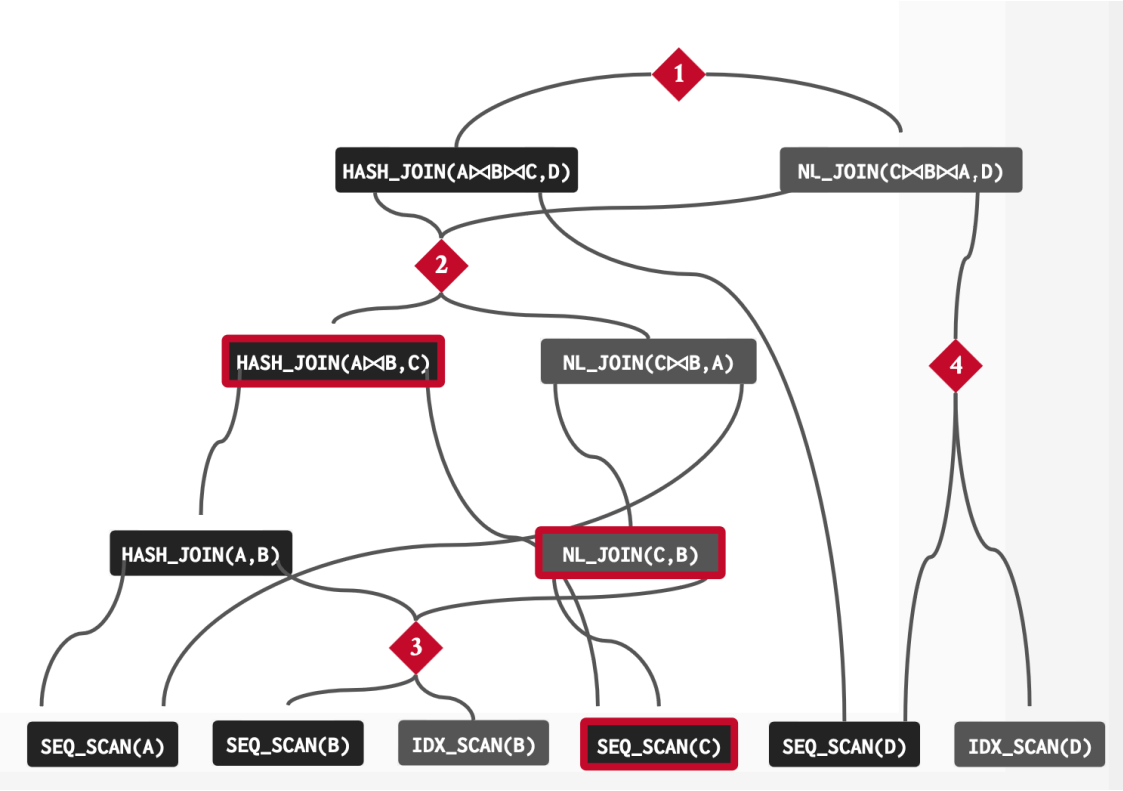

Figure 1: Bottom-up search through graph with red OR nodes

4.3 Redshift: Codegen Stitching

Redshift, a transpilation-based codegen engine, saves on compilation time by doing the stitching at the codegen level. When a sub-plan is stitched, instead of recompiling a sub-plan that we have seen before, the generated code is also available in the cache.

In addition, this cache is global to every customer in Redshift, allowing for a high cache hit rate and a large body of available sub-plans.

4.4 IBM DB2: Learning Optimizer

As the DBMS scans a table during normal execution, update the table statistics with new information. This is one of the earliest examples of a commercial system adopting adaptive query optimization techniques [5].

5 Replan Current Invocation

In the plan stitching approach described in the previous section, our goal is to improve the quality of future invocations of the same query. With this method, we intend to improve the quality of the current plan.

There are two approaches to replanning the current invocation:

1. Start over from scratch
   - Continuing with our current query plan will be worse than starting over.
2. Keep some intermediate results
   - If we have already done some expensive work, we can keep some of our results to save recomputation.

5.1 Quickstep: Lookahead Info Passing

At a high level, do some work at the beginning of the query, and pass information to later operators in the query to make a more informed decision for join ordering. This approach is limited to left-deep join trees and star schemas. [6]

Approach:

1. Compute bloom filters on dimension tables.
2. Probe these filters using fact table tuples to determine the ordering of the joins.

This gives us an approximation of the selectivity, which we can use to change the join ordering in the plan.

6 Plan Pivot Points

Introduce a special operator in the query plan that allows us to **pivot** the execution strategy during execution, selecting a sub-plan based on information about how execution is going thus far [4].

For example: There is a CHOOSE node above a SCAN node. When we reach the CHOOSE node, we decide to do a nested loop join if the data is very small, and do a hash join otherwise.

The advantages are that we do not have to go back to the optimizer in adapting our plan, and we do not have to lose any execution progress.

6.1 Proactive Reoptimization

When the optimizer generates our plan, generate "bounding boxes", putting a bound on the uncertainty we expect in our estimates. Just as before, the optimizer can select a different query plan at execution time. However, if the real data exceeds the bounding box thresholds, we can yield to the optimizer and re-optimize anyway [2].

References

[1] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. pages 238–249, 01 2005.

[2] S. Babu, P. Bizarro, and D. DeWitt. Proactive re-optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, page 107–118, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1595930604. doi: 10.1145/1066157.1066171. URL https://doi.org/10.1145/1066157.1066171.

[3] B. Ding, S. Das, W. Wu, S. Chaudhuri, and V. Narasayya. Plan stitch: harnessing the best of many plans. *Proc. VLDB Endow.*, 11(10):1123–1136, jun 2018. ISSN 2150-8097. doi: 10.14778/3231751.3231761. URL https://doi.org/10.14778/3231751.3231761.

[4] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, SIGMOD '89, page 358–366, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913175. doi: 10.1145/67544.66960. URL https://doi.org/10.1145/67544.66960.

[5] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. Leo - db2's learning optimizer. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, page 19–28, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558608044.

[6] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust: making the initial case with in-memory star schema data warehouse workloads. *Proc. VLDB Endow.*, 10(8):889–900, apr 2017. ISSN 2150-8097. doi: 10.14778/3090163.3090167. URL https://doi.org/10.14778/3090163.3090167.