Cloud Databases Midterm Project: Selinger Optimizer

Group 2
102061105 Yun-Sheng, Chang
102061212 Jian-Hao, Huang

May 17, 2017

1 Paper Review

1.1 Summary the key idea and design choice of the paper.

This paper proposes a cost-based query planner, using statistic results to estimate the cost for each possible plan, and the cheapest plan among all the plans is selected. It first describes how to estimate the cost for a single plan and the cost of joining two plans, which can be further generalized to joining multiple plans. Then, algorithms and heuristics are given to efficiently compute and cost. We briefly describe the key ideas in this paper.

1.1.1 join order matters

The join order has a great impact on the performance of a plan, especially for many-table join, which can have orders of magnitude difference between a good plan and a bad plan.

1.1.2 dynamic programming

Although the cost of joining in different orders can be substantially different, the cardinality is the same regardless of the join order, i.e., the problem exists optimal substructure. A dynamic programming approach can be adopted to efficienty solve the problem.

1.1.3 heuristics to reduce search space

Even though the dynamic-programming based algorithm can avoid much recomputation, the search space may still be quite large. Thus, some heuristics are required to reduce the search space.

1.2 How does Selinger estimate single-table operation plan's cost?

The single-table operation plan's cost is determined by the properties of it access path.

- Is the access path using segment scan or index scan?
- For index scan, is the index clustered?
- For index scan, does the index match any boolean factor?

1.3 How does Selinger estimate multi-table operation plan's cost?

To estimate multi-table operation, we first consider two-table operation. Two types of join methods are mentioned in this paper, nested-loop join and merge scan, and formulas are given respectively to estimate their cost. These formulas take the cost for plans being joined as input parameters. Once we know how to join two tables, we can join the first two tables and then join the composite table with the third table if we want to join three tables, and so on so forth. Moreover, a dynamic programming approach can be adopted due to the fact that the cardinality of a plan is the same regardless of its join order. For example, assume that we have computed the cardinality and cost of AB and BA and assume AB has a lower cost; then, we don't need to consider BAC anymore since ABC must be superior to BAC.

1.4 How does Selinger solve the problem of a huge number of plans?

The number of plans is huge because there are $n!$ permutations of plans and $(n-1)!$ join tree for each permutation if n tables are being joined. For example, if we want to join 3 tables, the 12 possibilities are:

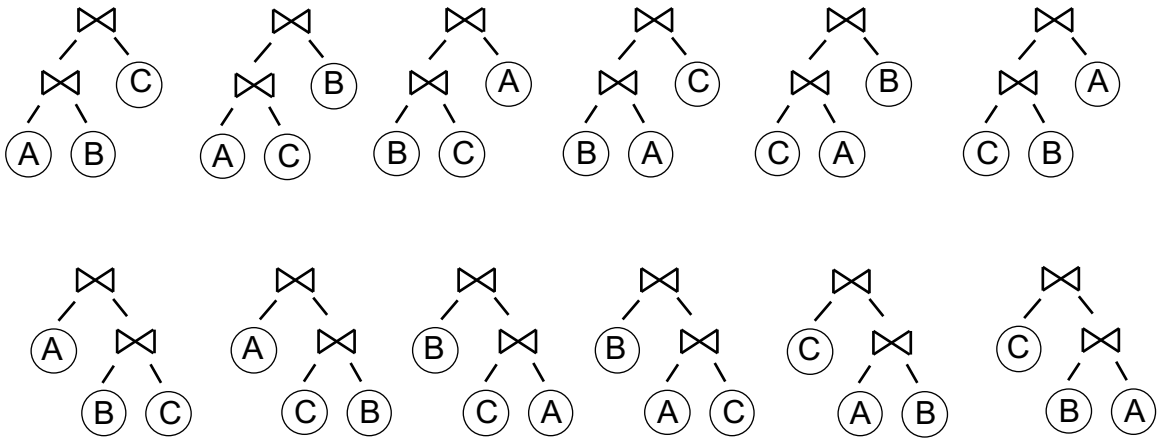

Figure 1: Directly joining three tables

To reduce the search space, two heuristics are adopted. The first one is **left-deep tree only**, removing the $(n-1)!$ term.

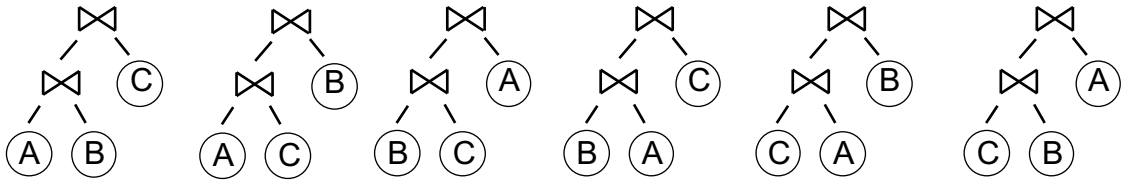

Figure 2: Left-deep tree only

Further, all joins requiring **Cartesian products are performed as late in the join sequence as possible**.

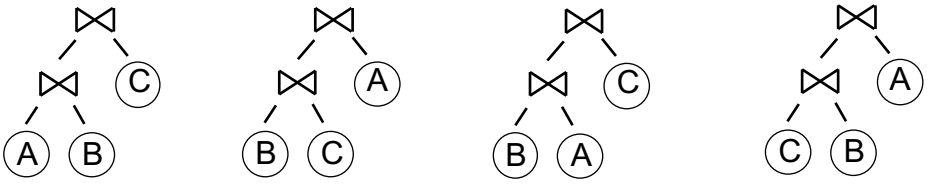

Figure 3: No Cartesian product (assuming no join predicate between A and C)

2 Join Benchmark Analysis

2.1 Summary the workload of the JoinbenchmarkProc.

The ER model of the Join Benchmark is shown below.

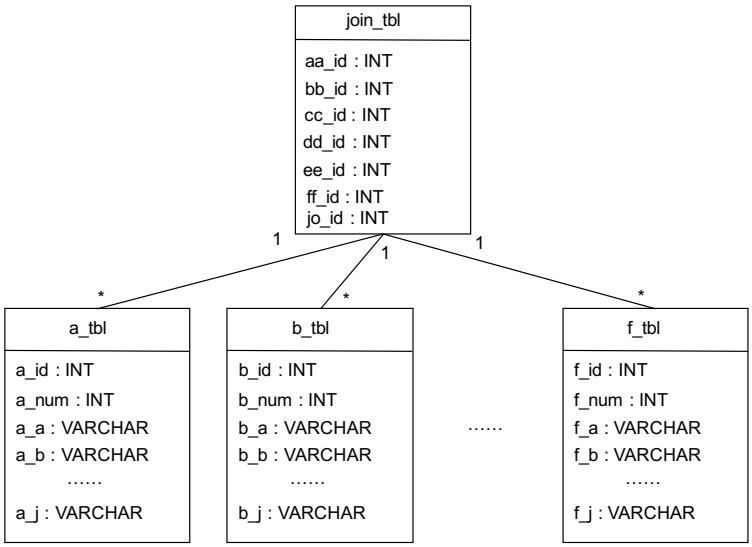

Figure 4: ER model of Join Benchmark. The *join_tbl* has a foreign key refering to every other table.

Every table involed in a query, except *join_tbl*, has a join predicate with the *join_tbl* on column *id* and a local predicate on column *num*. For example, if *a_tbl* and *b_tbl* are involved, the predicates might be like:

```
WHERE a_id = aa_id AND b_id = bb_id AND a_num > 36 AND a_num < 75 AND b_num > 27 AND b_num < 78
```

2.2 Run Join Benchmark and print out the plan which HeuristicQueryPlanner came out.

We first compare plan generated by the *BasicQueryPlanner* and plan generated by the *HeuristicQueryPlanner*.

```
->ProjectPlan  (#blks=18180218, #recs=3)
  ->SelectPlan pred:(aa_id=a_id and bb_id=b_id and cc_id=c_id and a_num>2526.0 and a_num<5875.0
    and b_num>2873.0 and b_num<7863.0 and c_num>2224.0 and c_num<8712.0) (#blks=18180218, #recs
    =3)
    ->ProductPlan  (#blks=18180218, #recs=1000000000)
      ->TablePlan on (a_tbl) (#blks=18, #recs=100)
      ->ProductPlan  (#blks=180218, #recs=1000000)
        ->TablePlan on (c_tbl) (#blks=18, #recs=100)
        ->ProductPlan  (#blks=218, #recs=10000)
          ->TablePlan on (join_tbl) (#blks=2, #recs=100)
          ->TablePlan on (b_tbl) (#blks=18, #recs=100)
```

Plan generated by BasicQueryPlanner

```
->ProjectPlan  (#blks=170, #recs=0)
  ->SelectPlan pred:(cc_id=c_id) (#blks=170, #recs=0)
    ->SelectPlan pred:(c_num>2224.0 and c_num<8712.0) (#blks=170, #recs=0)
      ->IndexJoinPlan (#blks=170, #recs=0)
        ->TablePlan on (c_tbl) (#blks=18, #recs=100)
        ->SelectPlan pred:(bb_id=b_id) (#blks=170, #recs=0)
          ->SelectPlan pred:(b_num>2873.0 and b_num<7863.0) (#blks=170, #recs=41)
            ->IndexJoinPlan (#blks=170, #recs=82)
              ->TablePlan on (b_tbl) (#blks=18, #recs=100)
              ->SelectPlan pred:(aa_id=a_id) (#blks=2, #recs=86)
                ->MultiBufferProductPlan (#blks=2, #recs=3176)
                  ->TablePlan on (join_tbl) (#blks=2, #recs=100)
                  ->SelectPlan pred:(a_num>2526.0 and a_num<5875.0) (#blks=18, #recs=31)
                    ->TablePlan on (a_tbl) (#blks=18, #recs=100)
```

Plan generated by HeuristicQueryPlanner

We find that the most significant difference between these two query planner is that the *HeuristicQueryPlanner* always try to push selections down while the *BasicQueryPlanner* does not. However, plan generated by *HeuristicQueryPlanner* is still slow because it sometimes uses *MultibufferProductPlan*, rather than *IndexJoinPlan*, which is much faster, to join two tables. This plan can be improved by not using *MultiBufferProductPlan*; to achieve this goal, we can fix the *join_tbl* to be the first outer relation and since *join_tbl* has foreign key to every other tables, *IndexJoinPlan* can always be selected to join two tables.

2.3 Summary the planning strategy of HeuristicQueryPlanner.

The planning strategy of *HeuristicQueryPlanner* is basically a **greedy** strategy. We illustrate the strategy with the flow chart below.

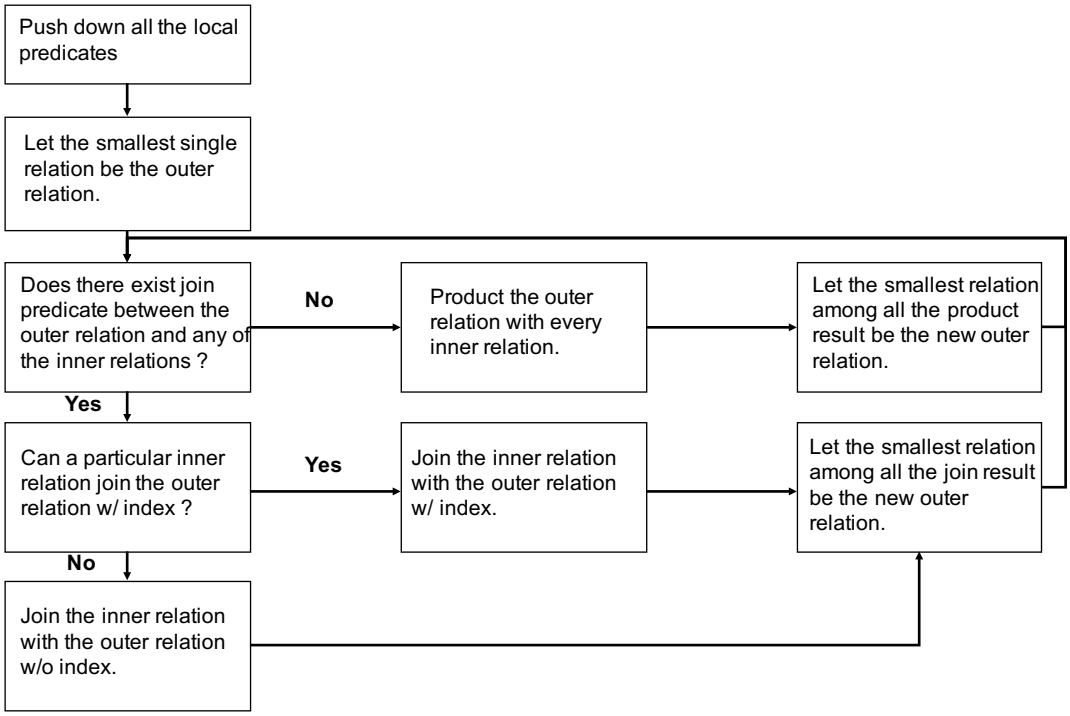

Figure 5: Planning strategy of HeuristicQueryPlanner

The main downside of the *HeuristicQueryPlanner* is that it sees only the **local** best choice rather than the **global** best choice.

3 Implmentation Details

3.1 How do you implement your own Selinger-like cost-based query planner.

3.1.1 Cost Estimation

At first, we used the cost formulas given in the paper, however, the result is less effective and the code is much complicated than using the histogram of a plan to estimate its cost. Thus, we estimate the cost of a plan simply by invoking its method *recordsOutput()*.

3.1.2 Algorithm

We illustrate our algorithm with an example. Assume that we have table A, B and C. A and B has a join predicate; B and C has join predicate;

1. At first, we encapsulate every tables. $\{A\}$, $\{B\}$ and $\{C\}$.
2. Add these sets to an empty C. $C = \{\{A\}, \{B\}, \{C\}\}$
3. Construct an empty set D. $D = \emptyset$
4. For each set T in C, if there exists a join predicate between any table in T and table $u, u \notin T$, add u to T.

5. Check if D already contains T. If not, add T to D. Otherwise, compare the cost of the same element in D with T, choose the one with lower cost to stay in D. $D = \{\{A, B\}, \{B, C\}\}$

6. Let C be D, and go back to step 3. Iterate n - 1 times, in this case, 2 times.

7. Finally, choose the lowest cost plan in D.

3.1.3 Time complexity

We analyze the time complexities of finding a plan for both *HeuristicQueryPlanner* and *SelingerQueryPlanner*. Let n be the number of tables involved in a query. Initially, the *HeuristicQueryPlanner* will find the smallest table as the outer relation; then it will join other table and choose the smallest result as the new outer relation, repeating $n-1$ times. Thus, the worst-case time to find a plan for *HeuristicQueryPlanner* is $O(\sum_{i=1}^{n} i)$, or $O(n^2)$. On the other hand, the *SelingerQueryPlanner* will estimate the cost for every subset of tables when every two tables has at least one join predicate; that is, the worst-case time to find a plan for *SelingerQueryPlanner* is $O(2^n)$. That means,

- the *SelingerQueryPlanner* is more sensitive to the number of tables and the number of join predicates;
- however, the *SelingerQueryPlanner* is much more likely to find a good plan.

3.1.4 Results

We run the Join Benchmark with *HeuristicQueryPlanner* and *SelingerQueryPlanner* and compare the results to prove that *SelingerQueryPlanner* is superior. The experiment is conducted with **4 RTEs**, the bufferpool size is **100000**, and the whole experiment consists of a 1-minute warm up phase followed by a 1-minute benchmark phase. We first profile the transaction execution time of both plans:

	Find the plan (ms)	Execute the plan (ms)	Ratio
SelingerQueryPlanner	10	4	2.5
HeuristicQueryPlanner	3	686	0.004

We can see that the *SelingerQueryPlanner* spends more time on finding a plan than the *HeuristicQueryPlanner*, but the time is worthwhile since the plan selected by *SelingerQueryPlanner* takes much less time to execute than the one selected by *HeuristicQueryPlanner*. This result matches our previous analysis. We compare the throughput and average latency between the two query planners:

	Throughput (txs)	Average Latency (ms)
SelingerQueryPlanner	17250	14
HeuristicQueryPlanner	347	689

The speed up is roughly **50** for both throughput and average latency.