Cost Based Optimization

Alireza Samadian (asamadian@google.com)

Overview

Currently, Beam SQL does not have a cost based optimizer; therefore, it does not reorder joins and it does not uses the cost estimation. We are going to make all the sources to implement row count estimation and then add rules to reorder joins. Since Beam SQL can have unbounded sources, we are also going to change the cost model of Apache Calcite and implement a new cost model that has rate and window as well.
In this document we are explaining the challenges we are facing and changes that are needed.

Goals

- Implement cost estimation for all sources and all Beam relation nodes.
- Implement a new cost model that supports streams.
- Add new rules to Volcano Optimizer so that it reorders the joins.
- Implementing cost estimation for different input tables

Non-goals

- Implementing a new cost based optimizer.
- Estimating cost for custom PTransforms in Beam.
- Optimization of entire beam execution graph.
- Dynamic optimization

Background

Beam SQL uses Apache Calcite to parse and optimize its queries. Apache Calcite [1] optimizer is an implementation of Cascade Optimizer. [2] Cascade is an object-oriented implementation of Volcano Optimizer [3], and it is referred as Volcano optimizer in Calcite.

Cascade is a cost-based optimizer and in order to perform its optimization, it needs a set of rules and a cost estimator. The cost for bounded sources are mostly based on row count estimation of the input and output of the node, and there are evidences that suggest row count by itself is enough for the cost estimation [7]. However, for unbounded sources there are other suggestions for the cost. One is implementing the cost based on rate and window size [6]. In this approach, every node should estimate two values for its output, a rate and a window size. Then the cost of each node is based on its input rates and/or output rate. For instance, if we have a join one way of estimating these two values would be:

$$r_o = (r_1 w_2 + r_2 w_1) f$$

$$w_o = w_1 w_2 f$$

Where r_o is the rate of output, w_o is the estimate of the number of tuples in the window of the output, and f is the estimated selectivity of the join. The advantage of this cost model is it can be easily combined with bounded sources because we can assume bounded sources have windows equivalent with their row count and rate of zero.

The current implementation of calcite optimizer can reorder joins and it uses only a single double value row count. That means we need a new cost model and cost estimation for the relational nodes in Beam SQL.

Cost Model

The cost model that we are going to use is a function of two parameters:

1. CPU
2. CPU_Rate

The first one shows the number of processed tuples in batch mode and the second one will be the rate of tuples processed for streaming mode. Then in order to compare two costs, we compare their CPU and CPU_Rate. However, we give much more weight to CPU_Rate. The default setting would be CPU_Rate is 3600 times more important than CPU. This means, every unit of CPU cost per hour (for streams) will have the same weight of one CPU cost of bounded sources.

The Cost model needs to extend the class RelOptCost, and there must be a Cost Factory implementing the following interface:

```
public interface RelOptCostFactory {
 RelOptCost makeCost(double rowCount, double cpu, double io);
 RelOptCost makeHugeCost();
 RelOptCost makeInfiniteCost();
 RelOptCost makeTinyCost();
 RelOptCost makeZeroCost();
}
```

Then all the nodes in Apache Calcite use these methods to return their cost. Since it is assumed by default that the cost is a function of three double values, Beam's cost model should return infinity when RelOptCostFactory.makeCost is called with three variables. (Because this means it is called in one of the logical nodes) Then when the node is becoming physical, Beam's node can return a cost that has been created by all of its values set. (And it will not be infinite)

NodeStats

Since for estimating CPU_Rate of each node we need rate and window size estimate (in terms of tuples) of its input, we cannot use calcite getRowCount method anymore. We need to define our own metadata and auxiliary class to represent all rowcount, rate and window.

For bounded sources, Window and Row Count are the same values and rate will be 0. For unbounded sources, Row Count (and as a result the CPU in the cost model) will be 0, Rate and Window are the output estimation of rate and window size (in terms of number of tuples).

The correct way to get the node stats for some rel node is calling BeamSqlRelUtil#getNodeStat for that node. This method will preprocess the input if the input is RelSubset and then invoke the metadata handler and returns the result.

Estimating Cost and NodeStats of Relational Nodes

Each relational node in calcite, implements a the method computeSelfCost(). We can implement our cost estimation either by overriding this method, or implementing some new method such as beamComputeSelfCost() and pluging a new handler for NonCumulativeCost in MetadataQuery. The benefits of the second approach is that we can include this method in BeamRelNode abstract class and force the future rels to implement it. Also, since in JDBC driver the cost factory is not configurable yet, we can implement this cost model incrementally by first implementing it for the other path and then after changes in JDBC driver and overriding JDBCFactory, we can implement for both of them together.

We also need to add a new method for estimating row count, rate and window for each node. Similarly, we can add a metadata for these values, and implement default methods for some of the relational nodes that are in calcite. (This way we can get an estimate of the cost for upper nodes faster and we do not need to wait until all the elements are physical).

Note that since we are returning infinite cost when the default method of the cost factory is used, we have to implement this method in all of the relnodes, otherwise, it will use Calcite's default implementation and returns infinite cost. Having an infiniti cost causes Calcite to throw an exception at the end of optimization because it means the plan is not executable.

During planning, the input of a node can be an instance of RelSubset that has different sub plans that all have the same output. For instance, if we are joining three tables the upper join will be Join(AB,C) where AB will be a RelSubset that can have the following plans in it:

- Logical_join(A,B)
- Logical_join(B,A)
- Physical_join(A,B)
- Physical_join(B,A)

Note that these different plans will be created gradually during the planning; therefore, at the beginning it may only have one of them. In order to get the cost of the input, if the input is a RelSubset, we can take its BestCost.

The cost of the input can be infinite, because the input is not physical yet. Similarly the row count and rate estimation of the input might be infinite because the estimation might not be implemented for that logical node in Calcite. However, this will not make a problem, because later if the input subset gets a physical alternative, its bestCost will be changed and that change will trigger a propagation propagated to its parent and as a result the cost of the parent will be recalculated. This behavior can be seen in RelSubset.propagateCostImprovements0 method.

In the following we are discussing the cost estimation used for some of the physical nodes. These estimations are similar to the Calcite's estimations, but consider rate and window of the input in them as well. In all of the following we denote the window, rate, and row count by w, r, c. We subscribe them based on whether they are estimates of the input or the output. For instance the rate of the output is denoted by r_o.

BeamAggregationRel

For aggregation, the CPU cost will be the same as the input row count, and CPU_Rate will be similar to the input rate. The output row count can be estimated based on the number of columns that we are grouping by. If the GroupCount is zero then the window size and number of rows will be 1 otherwise it will be a factor of input window and output window based on the number of columns that we are grouping by. The output rate will be

$$r_i \frac{w_o}{w_i}$$

Since the windows are also represented as a aggregation and they can possibly increase the rate or row count, (because some of the tuples appear in multiple windows) we need to also recognize if the aggregation is a windowing operation and in that case, estimate the cost differently based on the windowing strategy.

BeamCalcRel

We use the same method as calcite for getting the selectivity (which depends on the number of conditions in the Calc). Then use that selectivity for estimating rate, row count and window size of the output.
For CPU and CPU_Rate we use the number of expression multiplied by the input row count and rate.

BeamEnumerableConverter

The cost is Huge because we want to discourage the planner to pick this plan.

BeamIntersectRel

We estimate the row count, window, and rate to be the minimum of row count, window, and the rate of the inputs.
The cost will be the summation of row counts and CPU_rate will be summation of input rates.

BeamIOSinkRel

The CPU and CPU_Rate will be the size of the input. There is no window or output rate or output row count.

BeamIOSourceRel

This is done by cost estimation of the sources and it is estimated inside the table. The cost estimation of the input tables are discussed in Section Cost Estimation for Sources.

BeamJoinRel

For joins we estimate the selectivity of the join using calcite method. (It uses the number of conditions and their types) Let f be the selectivity, we estimate the output values by:

$$r_o = (r_1 w_2 + r_2 w_1) f$$

$$w_o = w_1 w_2 f$$

$$c_o = c_1 c_2 f$$

And the CPU will be input row counts plus output row count estimation multiplied by the estimation of the tuple sizes. The CPU_rate will be input rates plus output rate multiplied by the estimation of the tuple sizes.
In Beam Join, the most expensive step is Shuffle step. However, since we do not have n-way join, we cannot reduce the number of joins and as a result we cannot reduce the number of shuffles. Currently, we can only optimize the number of tuples processed in each join/shuffle.
Later, we need to separate different physical Joins (such as side input vs standard) into different RelNodes and also add rels for nway joins. This way, we can add rules to merge joins into n-way joins and have different cost estimation for them that also consider shuffling. (This can be done by adding one extra constructor in CostFactory and also extend the cost to consider shuffling in its cost).

BeamMinusRel

The CPU and CPU_Rate is the summation of input row counts and the summation of input rates. For A-B similar to calcite, we estimate the row count, rate, and window size as the following.
$c_0 = c_A - \lambda c_B$, $r_0 = r_A - \lambda r_B$, $w_0 = w_A - \lambda w_B$
Where λ is some constant (In calcite 0.5 is used).

BeamSortRel

The rate, window, and row count will be similar to the input. The CPU_Cost will be n log (n) and the CPU_rate will be r log(w). Similar to Calcite we should also multiply those costs by the number of fields to encourage projection push down.

BeamUncollectRel

This will make multiple rows per input row. Since there is no estimation of how many rows it will create, we are assuming it will produce $\lambda = 2$ rows per input row.
Therefore, CPU and the output row count will be λc_i and the window size will be λw_i , and CPU_Rate and the output rate will be λr_i.

BeamUnionRel

In the union we can assume the output rate, row count and the window are the summation of these estimates from the input. However, there will be intersections between the values. Since we don't have any statistics about the values we can assume a constant fraction of them are similar to each other. In Calcite they use 0.5 for the fraction. CPU and CPU_rate will be the summation of input row counts and the summation of input rates respectively.

BeamUnnestRel

This will be similar to BeamUncollectRel.

BeamValuesRel

This rel just represents a finite set of tuples. Therefore, there is no rate and CPU_rate. The window and row count and CPU will be the same as the number of values.

Metadata Handler

Calcite uses RelMetadataQuery for getting cost, rowcount,... and RelMetadataQuery has multiple handlers to support different operations.

BeamCostModel Metadata Handler

The handler that is used for getting the cost of a node is NonCumulativeCost handler. It is implemented along with two other handlers in RelMdPercentageOriginalRows. The only thing that this handler does is calling computeSelfCost of the relNode for getNonCumulativeCost queries. We are going to implement our own handler to make sure that we are always calling our cost computation method and not used any other estimation later. The other reason that we are going to implement our own handler is Calcite's RelMetadataQuery has a map that caches previous results. This can potentially make a problem, because our cost estimation cannot estimate a cost if its inputs are not physcall yet. Therefore, we need to clean the cache every time that we are trying to get the cost.

Here is the proposed implementation for the handler:

```
class NonCumulativeCostImpl
   implements MetadataHandler<BuiltInMetadata.NonCumulativeCost> {

  public static final RelMetadataProvider SOURCE =
      ReflectiveRelMetadataProvider.reflectiveSource(
          BuiltInMethod.NON_CUMULATIVE_COST.method, new NonCumulativeCostImpl());

  @Override
  public MetadataDef<BuiltInMetadata.NonCumulativeCost> getDef() {
    return BuiltInMetadata.NonCumulativeCost.DEF;
  }

  @SuppressWarnings("UnusedDeclaration")
  public RelOptCost getNonCumulativeCost(RelNode rel, RelMetadataQuery mq) {
    // This is called by the generated code in calcite MetadataQuery.
    // If the rel is Calcite rel or we are in JDBC path and cost factory is not set yet we should use calcite cost estimation
    if (!(rel instanceof BeamRelNode)){
      return rel.computeSelfCost(rel.getCluster().getPlanner(), mq);
    }

    // We need to first remove the cached values.
    List<List> costKeys =
        mq.map.entrySet().stream()
            .filter(entry -> entry.getValue() instanceof BeamCostModel)
            .map(entry -> entry.getKey())
            .collect(Collectors.toList());

    for (List key : costKeys) {
      mq.map.remove(key);
    }

    return ((BeamRelNode) rel).beamComputeSelfCost(rel.getCluster().getPlanner(), mq);
  }
}
```

Note that this handler will be called by the generated code in calcite. It is not directly used. That generated code populate the cache as well. (Which we are removing our costs from it because it might be infinite at the beginning).

Then we can plug this handler into the MetadataQuery by the following code in CalciteQueryPlanner:

```
root.rel
  .getCluster()
  .setMetadataProvider(
      ChainedRelMetadataProvider.of(
          ImmutableList.of(
              NonCumulativeCostImpl.SOURCE, root.rel.getCluster().getMetadataProvider())));
RelMetadataQuery.THREAD_PROVIDERS.set(
  JaninoRelMetadataProvider.of(root.rel.getCluster().getMetadataProvider()));
root.rel.getCluster().invalidateMetadataQuery();
```

Node Stats Metadata Handler

We have defined a Metadata Handler for getting Nodestats. We could do it without metadata handler but there are two benefits of having a handler:

1. It will cache the results of previously estimated nodes
2. We can provide default implementation for logical nodes without overriding the nodes.

The implementation of the metadata handler can be found in RelMdNodeStats. Currently, the only thing that the metadata handler does is calling estimateNodeStats for BeamRelNodes and return unknown for all other nodes (Calcite Nodes).

We can define default implementation for all other nodes. This won't affect the final result of the optimization but it will make it faster. (See RelMdRowCount in calcite). We should be careful after implementing the default implementations, our nodes are not passed to those implementation. For instance, if we define a default implementation for Join we should make sure that the calcite instances of joins are passed to that default implementation and BeamJoinRel is not passed there. There are two ways to handle this:

- Creating Another Metadata handler for default implementation and then register it in CalciteQueryPlanner and invoke that handler in RelMdNodeStats for all the nodes are not instance of Beam.
- Handle the BeamRelNodes in BeamSqlRelUtil#getNodeStats by calling their estimateNodeStats and pass the other nodes to MetadataHandler

The second approach is simpler but calcite won't be able to cache the results for us.

JDBC Path

The JDBC driver does not plug the BeamCostModel and it does not also plug the handler. As a future work both of them should be implemented.

The reason that we are not overriding computeSelfCost at this stage is also this. If we do that, since the cost model of the JDBC path is still VolcanoCost, it will not be compatible with our cost model.

Reordering Joins

In order to have an optimal plan, we have to reorder joins. Note that reordering outer joins will change the semantic of the plans, so we are only going to reorder inner joins. The rules used to reorder the joins are the followings:

- JoinCommuteRule.INSTANCE
- JoinAssociateRule.INSTANCE
- JoinPushThroughJoinRule.RIGHT
- JoinPushThroughJoinRule.LEFT

Having only JoinCommuteRule.INSTANCE and JoinAssociateRule.INSTANCE will be enough; however, by adding the other two rules, we make transformation faster. Therefore, we can expand and reach to new states faster.

There are some considerations: Currently, Beam does not support some types of joins. For instance, it does not support Cross join or any join that has OR or inequality in its condition. Permuting joins, can cause some of these conditions to appear. Therefore, we need to check if the resulting join will be supported or not. This can be done in two ways:

1- Reimplementing the Join Reordering Rules so that they don't fire if they produce such a join.
2- Return infinite cost if the join is not supported.

The problem with the second approach is if the input query cannot be converted into a legal join (it is a cross join in the first place) then it is hard to return appropriate error. That is because the planner tries to find a physical plan with a cost smaller than infinity and it cannot. Therefore, the user gets an error regarding planning instead of an appropriate exception about the join.

We are going to use the first approach. There are two new rules implemented:

1. BeamJoinAssociateRule.INSTANCE
2. BeamJoinPushThroughJoinRule

Both of them are a wrapper for the calcite versions. They just check if the resulting join is legal before applying the rule. Since calcite applies the rule by creating the new RelNode and then sending the RelNodes to a callback method, we had to override the call back class in order to check if the join is legal. Then we pass our own fake callback instance to the original rule. For instance the onMatch method for BeamJoinAssociationRule is implemented similar to the following code:

```
public void onMatch(final RelOptRuleCall call) {
 super.onMatch(
    new JoinRelOptRuleCall(
       call,
       rel -> {
         Join topJoin = (Join) rel;
```

```
        Join bottomJoin = (Join) ((Join) rel).getRight();
        return BeamJoinRel.isJoinLegal(topJoin) && BeamJoinRel.isJoinLegal(bottomJoin);
      }));
}
```

We also need to implement a method that can check if the join is a legal type of join. This can be hard because some of the information about the input such as windowing strategy are hard to be detected during planning. Note that it is OK if a condition is violated (such as compatibility of windows) if it is also violated in the original ordering of the joins. What is not ok is the case that a condition is not violated in the original ordering, but is violated in the new order. For instance, if we get an inequality condition in the new ordering of the joins, we can always conclude that there was an inequality condition in the original order as well. Therefore, there is no need to check that condition. Currently, the only condition that might be violated in the reordered join and it will not be violated in the original is having cross join.

Effect of Reordering Joins in Beam

In order to demonstrate the effect of reordering joins in the optimization step we performed a small experiment. The data is based on Stackoverflow dataset available in BigQuery public datasets. We have separated all posts with more than 20 comments in a table called post and separated all of the comments related to them and all the users made those comments in a separate table.
Then we performed the following two queries both before and after activating join reordering, and the runner was set to direct runners. For each of the queries we ran the query 6 times before activating join reordering and 6 times after it and a two-tailed independent samples t-test is performed on the values.

The first query looks for all the posts that a particular user has commented on:

```
SELECT postt.* FROM postt
JOIN commentt ON postt.id = commentt.post_id
JOIN usert ON usert.id=commentt.user_id
where usert.display_name='Vincent'
```

The execution plan without reordering will be:

```
BeamCalcRel(expr#0..39=[{inputs}], proj#0..19=[{exprs}]):
  BeamJoinRel(condition=[=($27, $24)], joinType=[inner]):
    BeamJoinRel(condition=[=($0, $23)], joinType=[inner]):
      BeamIOSourceRel(table=[[bigquery, postt]]):
      BeamIOSourceRel(table=[[bigquery, commentt]]):
    BeamCalcRel(expr#0..12=[{inputs}], expr#13=['Vincent':VARCHAR], expr#14=[=($t1, $t13)],
    proj#0..12=[{exprs}], $condition=[$t14]):
      BeamIOSourceRel(table=[[bigquery, usert]])
```

The execution plan after reordering will be:

```
BeamCalcRel(expr#0..39=[{inputs}], proj#0..19=[{exprs}]):
  BeamJoinRel(condition=[=($0, $23)], joinType=[inner]):
      BeamIOSourceRel(table=[[bigquery, postt]]):
      BeamJoinRel(condition=[=($7, $4)], joinType=[inner]):
          BeamIOSourceRel(table=[[bigquery, commentt]]):
          BeamCalcRel(expr#0..12=[{inputs}], expr#13=['Vincent':VARCHAR], expr#14=[=($t1, $t13)],
          proj#0..12=[{exprs}], $condition=[$t14]):
              BeamIOSourceRel(table=[[bigquery, usert]])
```

The second query looks for the people commented on all the posts created by users in seattle.

```
SELECT commentt.user_display_name FROM commentt
JOIN postt ON postt.id = commentt.post_id
JOIN usert ON usert.id=postt.owner_user_id
where usert.location = 'Seattle, WA, USA'
```

The execution plan without reordering is:

```
BeamCalcRel(expr#0..39=[{inputs}], user_display_name=[$t5]):
  BeamJoinRel(condition=[=($27, $21)], joinType=[inner]):
      BeamJoinRel(condition=[=($7, $3)], joinType=[inner]):
          BeamIOSourceRel(table=[[bigquery, commentt]]):
          BeamIOSourceRel(table=[[bigquery, postt]]):
      BeamCalcRel(expr#0..12=[{inputs}], expr#13=['Seattle, WA, USA':VARCHAR], expr#14=[=($t6, $t13)],
      proj#0..12=[{exprs}], $condition=[$t14]):
          BeamIOSourceRel(table=[[bigquery, usert]]):
```

After reordering the plan becomes:

```
BeamCalcRel(expr#0..39=[{inputs}], user_display_name=[$t5]):
  BeamJoinRel(condition=[=($7, $3)], joinType=[inner]):
      BeamIOSourceRel(table=[[bigquery, commentt]]):
      BeamJoinRel(condition=[=($20, $14)], joinType=[inner]):
          BeamIOSourceRel(table=[[bigquery, postt]]):
          BeamCalcRel(expr#0..12=[{inputs}], expr#13=['Seattle, WA, USA':VARCHAR], expr#14=[=($t6, $t13)],
          proj#0..12=[{exprs}], $condition=[$t14]):
              BeamIOSourceRel(table=[[bigquery, usert]]):
```

The results are in Table 1 and shows significant improvement in the execution time for both of the queries.

Table 1: Effect of reordering joins on Query 1 and Query 2.

	Query 1	Query 2
Average time (ms) without reordering	236812	150980
Average time (ms) after reordering	83311	74620
T-test p value	< 0.0001	< 0.0001

Cost Estimation for Sources

All tables in calcite can implement getStatistic() that returns an object implementing org.apache.calcite.schema.Statistic. While Calcite's statistics can return many informations, such as row count or distribution of the column, it does not have rate and window size for the streams. Therefore, we are creating a new class for Statistics that also implements org.apache.calcite.schema.Statistic. This way we can also include rate and window.
The window size for unbounded sources are going to be represented as a constant. (The rate is still going to be gathered by the previous tuples. The reason is it is hard to estimate window size without gathering statistics of the actual data. This will be hard specially for windowing strategies such as session windows.

Implementation of getting row count for different sources is discussed in this doc separately.

References

[1] Begoli, Edmon, et al. "Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources." *Proceedings of the 2018 International Conference on Management of Data*. ACM, 2018.
[2] Graefe, Goetz. "The cascades framework for query optimization." *IEEE Data Eng. Bull.* 18.3 (1995): 19-29.
[3] Graefe, Goetz, and William J. McKenna. "The volcano optimizer generator: Extensibility and efficient search." *ICDE*. Vol. 93. 1993.
[4] Viglas, Stratis D., and Jeffrey F. Naughton. "Rate-based query optimization for streaming information sources." *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, 2002.
[5] Ngo, Hung Q., et al. "Worst-case optimal join algorithms." *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*. ACM, 2012.
[6] Ayad, Ahmed M., and Jeffrey F. Naughton. "Static optimization of conjunctive queries with sliding windows over infinite streams." *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. ACM, 2004.
[7] Leis, Viktor, et al. "How good are query optimizers, really?." *Proceedings of the VLDB Endowment* 9.3 (2015): 204-215.
[8] https://console.cloud.google.com/marketplace/details/stack-exchange/stack-overflow

Appendix

Future Work

- Implementing an API for PTransform to estimate its rate, window size, and row count based on its input and its logic, so that we have estimation for PCollection tables.
- Make nodes aware of window strategy so that they can predict impossibility of the plan during construction.
- Support cross join and theta joins.
- Dynamic rescheduling of stream jobs based on newer information if our estimations are not correct.
- Implement selectivity estimation by sketching and gathering statistical data of the input sources.
- Recognize equality of fields in different tables to avoid unnecessary cross joins. For instance: t1 is joined with t2 and t3 using the same column, then we should be able to join t2 and t3 using those columns.
- Having multiple physical join rels and instead of deciding what physical join we are picking during execution (Standard vs Side join), we can convert the logical join to physical during planning.