Constructing Materialized View from Similar Subexpressions for Spark SQL

Prepared By: Ey-Chih Chow, Ting Yu Leung (Cliff), Yingpong Chen

Date: 11/14/2017

Huawei Technologies Co., Ltd.
华为技术有限公司

Constructing Materialized Views from Similar Subexpressions

Scope:

The majority of application scenarios with Spark are related to analytics via ad-hoc or batch-oriented queries. With such applications, query workload is often relatively static, even with interactive queries. Much of the system's activity can be predicted in advance. Under such environment, it is highly likely to improve Spark performance via materialized views recommended from historic query workload. This document presents ways to construct materialized views for this purpose.

Goals:

Assume that queries, interactive and/or batch, submitted by tenants to a Spark platform are all logged. The logged queries can be extracted and processed into a query batch. With such a batch, we build an advisor to recommend Materialized Views that can improve query execution performance. In contrast to traditional databases that use an accurate model to predict the runtime behavior of a candidate query plan, the advisor utilizes as much as possible high-level knowledge of the query batch to identify candidate MVs, because constructing an accurate model for massive clusters has been viewed as difficult in the absence of accurate statistical summaries of the input data, and in the presence of machine failures.

Approach:

We present a method, based on the SparkSQL rule engine, to construct candidate MVs from a query batch. We use two rule engines, preprocessor and MV generator, to accomplish our goal. Queries in a workload are first transformed into the corresponding Modular Plans. Each rule in the preprocessor and MV generator is to transform a batch of the Modular Plans into another batch. The preprocessor includes rules such as CanonicalizePlan, FindSPJGs, ThrowAwayNonStarJoin, RemoveFiltersIfNeeded, etc. The batch of Modular Plans after preprocessing is grouped into several sub-batches of Modular Plans based on the set of tables involved and if there is a GROUPBY in a plan. Plans in each of the sub-batches involve the same set of tables with GROUPBY. We use the MV generator to transform each of the sub-batches into candidate MVs, in the form of Modular Plans. Finally these candidate MVs are converted back to SQLs. The MV generator has rules such as FindPromisingTrivialCandidates, CanonicalizeSPJGs, CreateCandidateMVs, DiscardCheapMVs, ExcludeCandidateWithHugeResults, KeepWhenBeneficial. For cost estimation, each component Modular Plan will be associated with the number of potential consumers from original queries.

Detail Procedure:

The core of the approach is generation of candidate MVs. Given a batch of modular plans, a simple solution would be to create a single MV that covers all the plans. However, this is not necessarily the best solution. The MV may produce a very large result that does not fit any of its consumers (queries) well. This may happen, for example, if the queries require different sets of columns or different sets of tuples. In that case, processing each query may have to scan through a lot of data that it does not need. In that case, we should consider multiple candidate MVs, each covering some subset of the queries. We use a greedy algorithm to do this. We start

with creating missing opportunities. Code for such generation is as follows: trivial MVs, each corresponding to a Modular Plan in the batch, and greedily merge in one other trivial MV at a time to maximize the merging benefit until no more beneficial merging is available. If there are still trivial MVs that have not been merged, we apply the algorithm again to the remaining trivial MVs.

During generation, several heuristic rules are applied to prune out candidate MVs that are not promising or are less promising that other candidates. The goal is to reduce the number of candidate MVs generated.

```
object CreateCandidateCSEs extends CoveringRule[ModularPlan] with PredicateHelper {
  def apply(batch: Seq[ModularPlan]): Seq[ModularPlan] = {
    val candidateSet = mutable.ArrayBuffer[ModularPlan]()
    val trivialCandidateSet = new Array[ModularPlan](batch.length)
    batch.copyToArray(trivialCandidateSet)
    val trivialCandidateBufR = mutable.ArrayBuffer[ModularPlan]()
    trivialCandidateSet.foreach{trivialCandidateBufR += _ }
    val trivialCandidateBufM = mutable.ArrayBuffer[ModularPlan]()
    while (!trivialCandidateBufR.isEmpty) {
      var candidateR = trivialCandidateBufR.remove(0)
      trivialCandidateBufM.clear
      trivialCandidateSet.foreach{trivialCandidateBufM += _}
      trivialCandidateBufM -= candidateR
      var isCandidate = false
      var done = false
      while (!trivialCandidateBufM.isEmpty && !done) {
        val bestCandidateWithBenefit = computeCandidateWithMaximalBenefit(candidateR,trivialCandidateBufM)
        if (bestCandidateWithBenefit._1 > 0) {
          val result = combinePlans(candidateR, bestCandidateWithBenefit._2)
          (result._1,result._2) match {
            case (true, Some(combinedPlan)) =>
              candidateR = combinedPlan
              isCandidate = true
              trivialCandidateBufM -= bestCandidateWithBenefit._2
              trivialCandidateBufR -= bestCandidateWithBenefit._2
            case _ =>
          }
        }
        else done = true
      }
      if (isCandidate) candidateSet += candidateR
    }
    collection.immutable.Seq(candidateSet:_*)
  }
```

As mentioned above, we consider SPJG candidates with the Star Join only. The materialized table with a plan is essentially the pre-aggregation of the fact table enriched with some of the columns in the dimension tables. The method computeCandidateWithMaximalBenefit(_, _), is to find a candidate plan so that, when it combines with the chosen plan, the result of pre-aggregation should be as small as possible and the combined plan should have as many consumers as possible. For two modular plans that are all in the form of Select/Groupby/Select, the following steps are needed for either this method or the method combinePlans(_, _) to find a covering plan:

(1) For the bottom pair of Select objects, check if the predicateLists are join compatible, i.e. the equijoin constructed from the intersection of equijoins in the two object is non-empty. Create a covering Select object with equijoin predicates from the intersections. Simplify the selection predicate in the pair of Select objects by deleting any conjunct already included in the join predicate of the new Select object. Add a covering selection predicate, if any, by

OR'ing the simplified predicates. AND the covering predicate to the join predicate just constructed for the new covering Select object.

(2) Add a covering Groupby object on top of the covering Select object created in (1). The grouping columns consists of the union of (1) grouping columns of the two Groupby objects of the two plans to be combined, (2) all columns referenced in the covering selection predicate constructed in step 1.

(3) Add a top covering Select object on top of the covering Groupby object created in step 2. Include all columns and aggregate expressions that are required to compute the result of potential consumers as outputList.

Integration with Spark SQL Cost Framework for MV recommendation:

Operators in the Logical Plan employs the method computeStats(conf: SQLConf): Statistics (or stats(conf: SQLConf): Statistics) to return cost estimation in the form of Statistics. Given a query, the corresponding Statistics object in the logical plan has information on physical size in bytes and estimated number of rows for the query result. The object also has statistics on columns involved in the query and query hints. Statistics collected for a column include the (1) number of distinct values, (2) minimum value, (3) maximum value, (4) number of nulls, (5) average length of the values, and (6) maximum length of the values. The configuration options for Spark SQL COB are in SQLConf.scala and StaticSQLConf.scala. What follows, we present a scheme to incorporate this Spark capabilities for cost-based construction of MVs.

We only consider MV candidates that involve the star join on a set of tables in which one of them is the fact table and the others dimensions. Also, candidates must involve GroupBy operators. In other words, we consider MVs that are pre-aggregations of the fact table joined with some dimension ones. We assume MVs are saved in the columnar format such as Parquet or CarbonData. We focus on lowering the overall cost of query execution under no disk space constraint.

For a set of initial candidates, involving the same set of tables, we first convert each candidate back to SQL and compute the corresponding statistics using Spark. They will saved in the corresponding modular plans. What follows are three cost-based heuristics used to efficiently construct final MVs. We apply heuristic 1 to prune initial candidates. Then we apply heuristic 2 to merge candidates for final MVs.

Heuristic 1. ***Discard the candidate if there is not enough potential savings***

Suppose that, for a candidate, denote the estimated number of rows (cardinality) of the result by C_a and the cardinality of the corresponding fact table by C_b, then discard the candidate if $C_a > \beta \times C_b$ where β is a configuration option, e.g. $\beta = 90\%$. This will discard candidates that have many group-by columns.

Heuristic 2. ***Merge Only When Beneficial***

Merging two candidates can save redundant computation but it is not always beneficial because the new candidate may produce a larger result and thus significantly increase the materialization cost and reading cost for its consumers.

For two candidates *Q1* and *Q2*, f_{Q1} and f_{Q2} denote frequencies of the candidates in the batch respectively, $speedup_{Q1}$ and $speedup_{Q2}$ denote estimated speedups of the two candidates with the merged MV. The net gain of the combined candidate is:

$$netgain = \frac{f_{Q1} * speedup_{Q1} + f_{Q2} * speedup_{Q2}}{f_{Q1} + f_{Q2}}$$

Merge the two candidates if $netgain > threshold_{speedup}$. The estimation of speedup is a simple approximation of the I/O savings, i.e. the ratio of the size of the MV on disk against the size of tables involved in the MV.

Another heuristic that might worth to consider is about the size of the result of the candidate as follows. We will dig it more in the future.

Heuristic 3. ***Exclude candidates with Huge Results***

Consider a candidate Q that has the frequency f_Q. Denote the cost of materializing the candidate by C_W, and the cost of using the result by C_R. Exclude the candidate if benefit-to-cost ratio smaller than 1, i.e. (f_Q * (C_E - C_R)) / (C_E + C_W) < 1. C_R and C_W can be estimated based on the cardinality of the candidate and the set of output columns using the statistics from Spark. C_E is the execution cost. We have yet to figure how to estimate the cost.