Query Rewrite using Materialized Views

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

Date: 04/13/2017

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

Query Rewrite using Materialized Views

Scope:

The library aims to improve Spark query processing performance by rewriting queries to run against Materialized View (MV), rather than base tables. We assume metadata for MVs are available in the Hive meta-store with a form same as views (size of a MV is saved in the field comment of the entry) and can be fetched via extension of the Spark class InMemoryCatalog for SparkSQL Catalyst. The extended catalog essentially supports a key-value map, where each MV has an entry in the map with the set of table identifiers as the key and the corresponding sets of logical plans and sizes of the materialized tables as the value. The logical plan of the MV can be obtained via SparkSQL API. The library will use the main memory catalog to find good MVs to rewrite a SQL query to run against the MVs instead. The rewritten query will be converted into SQL and issued to the Spark cluster for result. If there are multiple sets of MVs can be used to rewrite a query, we will choose the one with the highest query reduction factor to rewrite the query. For the first phase, costs are estimated based on sizes of MVs. In the future, we will consider other factors associated with Carbon Data, such as index. We assume all the UDFs needed for the query are available with the Spark cluster. In addition, for simplicity, for the first phase, queries corresponding to the MVs are only of the SPJG (aggregate-select-project-join) form in that no outer join will be allowed in the MV definitions.

Goals:

For a Spark cluster, there are four ways to run a query: (1) submit the query remotely and use the RPC call of the JDBC client to send to the Spark Thrift server, (2) submit the query locally to the Spark cluster via the spark-sql shell, (3) submit the query via the REPL (Read-Evaluate-PrintLoop) shell (4) submit the query via a web service call to the cluster, where the web service is started as an application of the Spark cluster, using such framework as Dropwizard. The four methods all use the tool spark-submit to start the corresponding drivers, via initializing SparkContext, of Spark applications. Queries processed with the four servers are all via HiveContext/SQLContext/SparkContext created with the tool. So we will provide one library that can be used by all the four server to rewrite queries before submitting to the Spark cluster.

Approach:

What follows, we discuss developing a single library that can be installed alongside the SparkSQL engine, captures the user query, rewrite the query and “re-submit” the query to SparkSQL. We address in more detail how this can be added to Spark query tools in Appendix.

Query Rewrite Algorithm:

In this section, we present a simplified algorithm to implement ToolContext.queryrewrite(query, hivecontext) for the first phase. Given a user query and hivecontext, we use Spark to parse and compute the corresponding optimized logical query plan, which is a tree of relational algebra nodes such as FILTERs, JOINs, and PROJECTs, in that all view definitions have been expanded and resolved, and thus, the logical query plan contains only the base tables. In other words, we will be matching base tables against base tables, and there won’t be any views during this optimization phase. The relational algebra tree, however, is too low level for us to do query rewrite in matching the tree with (relational algebra) logical plans of the MVs in the catalog. We, therefore, convert the logical plan into a tree whose nodes corresponding to those of Query Graph Model of DB2. I.e. Nodes of the tree are of the type SELECT, GROUPBY, TABLE, etc. We

call such plans as modular plans. Both logical plans of the MVs and user queries are converted into modular plans for matching.

Like DB2, we build a matching framework to match modular plans between a user query and candidate MVs ordered by their sizes. The framework is simplified to consider queries and MVs whose plans are of the form of SELECT-GROUPBY-SELECT-TABLE only. Effectively, the queries and the MV definitions are in the form of:

SELECT <grouping columns>, <aggregate functions>

FROM T1, T2, … /* no subqueries */

WHERE <join predicates and filtering predicates>

GROUP BY <grouping columns>

HAVING <filtering on grouping columns or aggregate functions>

In our first implementation, we focus only on

- INNER JOINs (no LEFT/FULL/RIGHT JOIN yet)
- Grouping columns do not involve grouping set and cube, e.g., "group by cube(a,b), rollup(c,d)" is not considered
- Aggregate functions are simple functions like: max, min, sum, count (without distinct)

With this simplification, given a user query, the candidate MVs for matching must be involved with base tables that are exactly the same base tables involved in the query. The corresponding delta of the first candidate MV (subsumer) that matches with the user query (subsumee) will be converted into SQL as the rewritten query of the user query. If none of the candidate MVs match with the user query, we just go ahead to submit the user query to the Spark cluster for the answer. The reason that we choose the first MV is that it is of the smallest size.

The pair of SELECT-GROUPBY-SELECT-TABLE blocks of a user query and MV is matched in a bottom up fashion. With a set of matching functions implemented using the rule-based framework, i.e. Rule Executor, of Spark catalyst, where each function corresponds to a rule. At the first phase, we consider the following three functions only, which will explain later:

- SelectsWithNoChildDelta
- GroupbysWithSelectOnlyChildDelta
- SelectsWithGroupbyChildDelta

Consider the following two tables:

Trans(tid, fpgid, flid, date, faid, price, qty, disc)

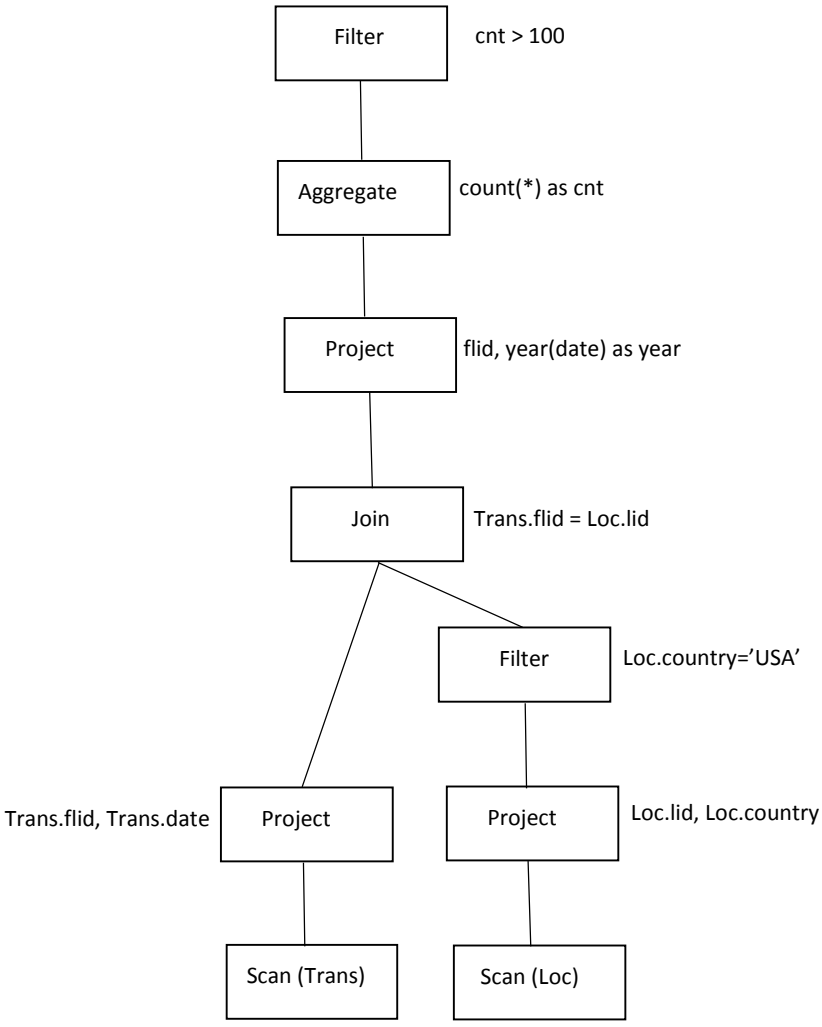
Loc(lid, city, state, country)

What follows is an example of query rewrite we consider:

```
Q1:  select flid, year(date) as year, count(*)
as cnt
from Trans, Loc
```

where flid = lid and country = ‘USA’ group
by flid, year(date)
having count(*) > 100

The logical plan of the query is as follows. Note that, when we generate this optimized logical plan, we configure the Spark optimizer to exclude the rules that create Local relation (temporary table), which is difficult to handle in query rewrite.



The Logical Plan is transformed to a Modular Plan by applying a set of extracting patterns. Every pattern uses pattern matching to convert a subtree of nodes in a Tree to a node of another kind of Tree. E.g.

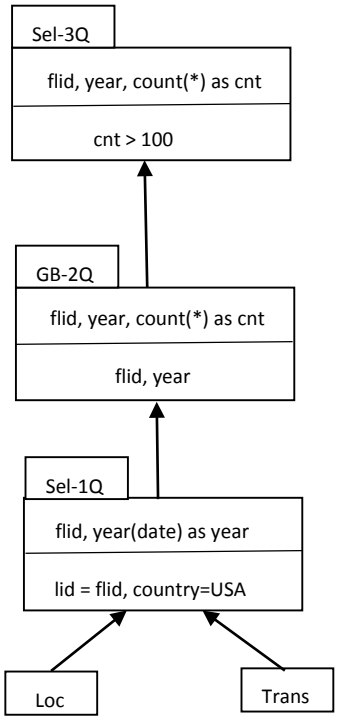
```
object GroupByBlock extends Pattern with PredicateHelper {
  …
```

```
   def apply(plan: LogicalPlan): Seq[ModularPlan] = plan match
{        case ExtractGroupBy(output, input, predicate, child) =>
makeGroupByBlock(output, input, predicate, matchLater(child))
case _ => Nil
   }
}
```

Object ExtractGroupBy is defined below and matchLater triggers other extracting patterns.

```
object ExtractGroupBy extends PredicateHelper {
   type ReturnType = (Seq[NamedExpression],Seq[Expression],Seq[Expression],LogicalPlan)
def unapply(plan: LogicalPlan):  Option[ReturnType] = plan match {
      case logical.Aggregate(groupingExpressions, aggregateExpressions, child) =>
Some(aggregateExpressions, child.output, groupingExpressions, child)          case
other => None
   }}
```

The modular plan transformed from the above logical plan is shown below:



We considered only 3 types of node blocks; they are Table, Select, GroupBy:

```
case class Table(databaseName: String,
                 tableName: String,
                 outputList: Seq[NamedExpression]) extends LeafNode
{   …
```

```
}
```

The Table node is transformed from the scan node in the logical plan. Essentially it is the base table. For relations associated with the scan node, the corresponding Table node has three fields denoting the database name, table name, and list of column expressions respectively. Finally, Table nodes do not have children (i.e., they are leaf nodes).

```
case class Select(outputList: Seq[NamedExpression],
                  inputList: Seq[Expression],
                  predicateList: Seq[Expression],
                  aliasMap: Map[Int, String],
                  joinEdges: Seq[JoinEdge],
                  children: Seq[ModularPlan]) extends ModularPlan
{  …
}
```

Effectively a SELECT node represents SELECT-FROM-WHERE in the SQL. The SELECT node has one or more input table(s). For a single input table, the SELECT node means that there is only 1 table in the FROM clause, and the operation is merely filtering rows from the input table and projecting out the selected columns. For multiple input tables, the SELECT node means that the query is doing a join involving multiple tables.

We note that a SELECT node may represent a subtree of filter/project/join nodes in a SparkSQL logical plan. In a way, multiple nodes in a logical plan will be "collapsed" into a single SELECT node, primarily for the ease of query matching and rewrite. The SELECT node collects lists of input, output, and predicate expressions extracted from the corresponding connected sub-tree of all the filter, project, and join nodes in the logical plan. The field input list comprises of all the base tables involved in the logical plan. The field output list comprises of the output columns. The field predicateList includes all the predicates, including joins, in these logical plan nodes. Children of the SELECT node is collected into the field children. Aliases of the tables are collected into the field aliasMap, which is a map where keys are indexes of tables relative to the children list and values are the corresponding aliases. To facilitate processing, we also represent joins with a list of edges, joinEdges, of the corresponding join predicates, where each edge is of the form (i, j, type), representing ith and jth tables relative to the children list and types of joins, i.e. equal, left/right outer joins (for the 1st release, we consider equijoin only). If the SELECT node involves an equijoin, to canonicalize the output list, if one of the join columns is in the list, we will replace with the other one if it has a larger hash value.

```
case class GroupBy(outputList: Seq[NamedExpression],
                   inputList: Seq[Expression],
                   predicateList: Seq[Expression],
                   child: ModularPlan) extends UnaryNode
{  …
```

}

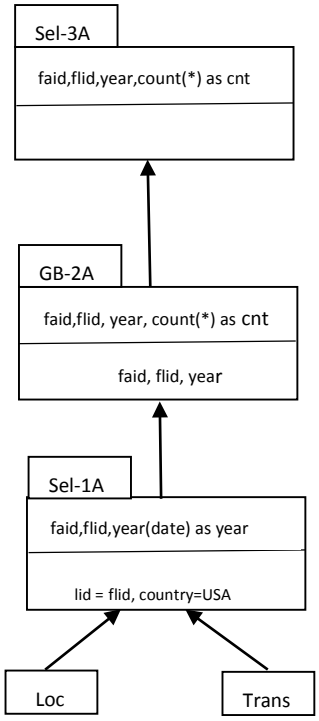
The GroupBy node collects lists of input, output, and predicate expressions from the Aggregate node of the logical plan, where predicateList is a list of groupby expressions. The node has a single child.

The materialized view we consider is MV1:

select faid, flid, year(date) as year, count(*) as cnt from
Trans, Loc
where flid = lid and country = 'USA'
group by faid, flid, year(date)

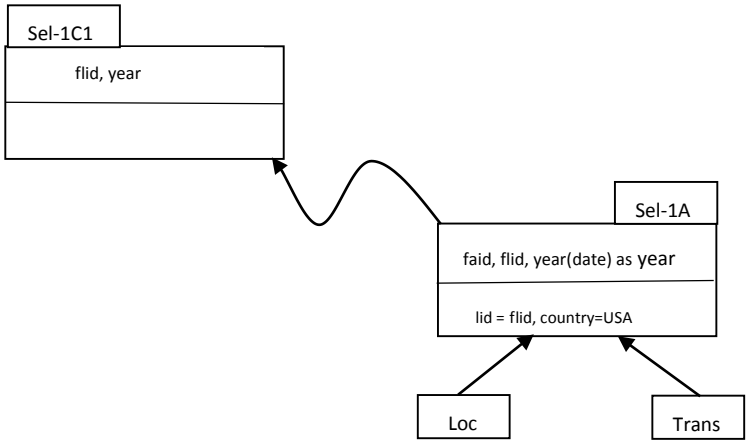
The modular plan for MV1 is shown as follows:

MV1:



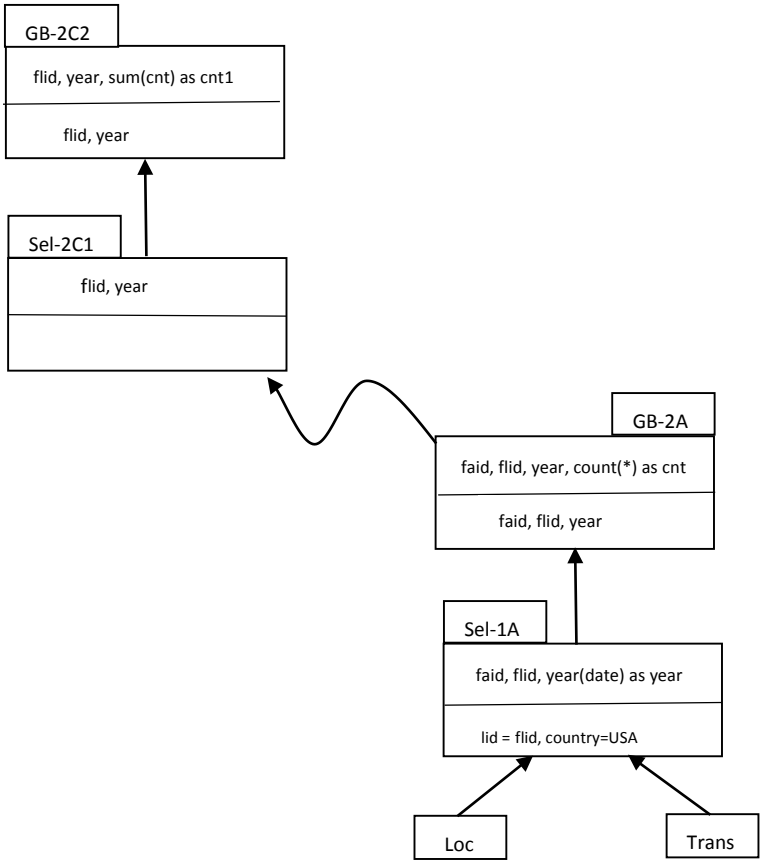
Note that Sel-3A is actually not needed for MV1. But for a unified SELECT-GROUPBY-SELECT-TABLE representation, we add a dummy node for MV1.
Matching between modular plans of Q1 and MV1 is a bottom-up process, starting from matching between Sel-1Q and Sel-1A. We try the first matching function, i.e. SelectsWithNoChildDelta. The matching is successful because (1) both nodes are of the Select box, (2) the children of Sel-1A matches with those of Sel-1Q respectively and the predicate list (where-clause) of Sel-1Q "semanticEquals" to

that of Sel-1A, where “semanticEquals” is a method in the Spark expression library, which we will elaborate later. The delta sub-plan of the matching is shown below:



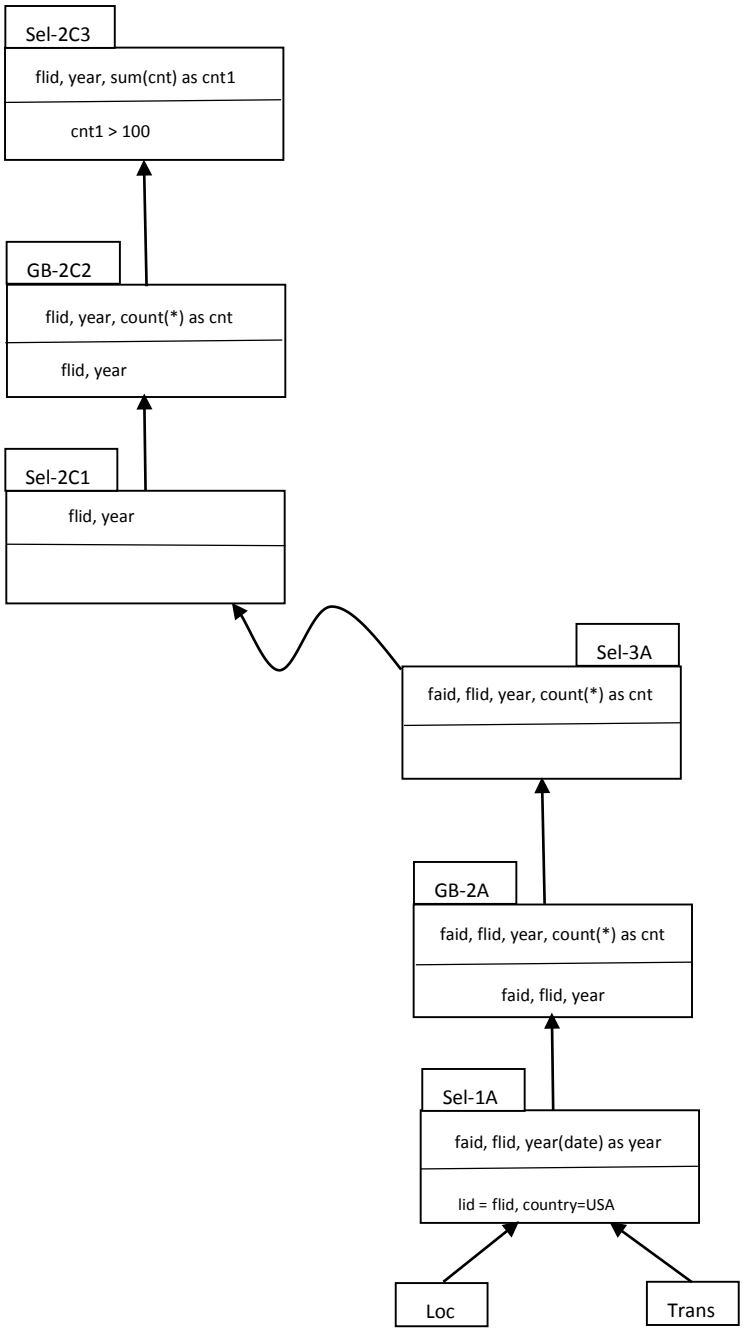
Note that, for this matching to be general enough, the optimized logical plan from which the query modular plan is transformed cannot be computed with an optimizer that has the rule of predicate push down through Groupby. In general, Sel-1Q can match Sel-1A even they have different numbers of children. Under our environment, however, it is very difficult to identify if the relationship of keys is of the 1 : n mapping. It is also difficult to make sure such column as country in the table Loc is at the higher level of a dimension hierarchy than the column lid. So, for the 1st phase of the project, this matching function applies only to the Select pair where the sets of base tables of these two Selects are of 1-1 mapping and have the same predicates. The order of input tables does not play a role in matching. Expressions of the output and predicate lists of Sel-1Q have to be derivable from those of Sel-1A respectively. Since we use the SparkSQL expression libraries to implement this, we obtain such conditions as the column orders do not matter and complex expressions is derivable from component output columns of Sel-1A for free.

Given the above delta, we match between GB-2Q against GB-2A. Only the matching function GroupbysWithSelectOnlyChildDelta can successfully matches these two blocks in that (1) the GB-2Q and GB-2A are all of Groupby nodes, (2) the child delta mentioned above has a Select node only, i.e. Sel-1C1, without any Groupby node, (3) grouping columns of GB-2Q have to be derivable from the output list of GB-2A, (4) expressions of the predicate list of Sel-1C1, i.e. those in the corresponding where-clause of the Select node, have to be derivable from the output list of GB-2A. If the set of grouping columns (in predicateList) of GB-2Q is exact the same as those in GB-2A, then we don’t need a regrouping delta. In this case, aggregates in GB-2Q have to be ‘semanticEquals’ aggregates in GB-2A. Otherwise, we need a regrouping delta as shown below where Sel-1C1 is pulled up to become Sel-1C2.



The node Sel-2C1 is pulled up when adding GB-2A for grouping on top of Sel-1A. Note that as a result of matching the two GroupBy nodes GB-2Q and GB-2A, the aggregate function count(*) of the query is derived as sum(cnt) and GB-2C2 is added on top of Sel-2C1 for regrouping. For the 1st phase, we do not consider aggregates such as count(distinct x) or sum(distinct x).

Finally we match Sel-3A with Sel-3Q, with the child delta GB-2C2. The matching function SelectsWithGroupbyChildDelta can be successfully applied in this case because (1) both Sel-3A and Sel-3Q are select blocks, (2) each has one child, (3) the child delta includes grouping, (4) the aggregates in Sel-3Q have to be translated into the corresponding aggregates in GB-2C2 via GB-2Q to determine if they are semantically equivalent to the corresponding aggregates in Sel-3A. For simplicity we only consider Sel-3A that has no where-clause so the semantic equivalence always remains true, (5) expressions in the output list and predicate list (where-clause) of Sel-3Q are derivable from Sel-3A, (6) for pull-up, grouping expressions of GB-2C2 and expressions in the predicate list of Sel-2C1 are derivable from the output list of Sel-3A. This results in the following delta plan. Note that Sel-2C1 is pulled up again when we add Sel-3A on top of GB-2A.



For each pair of node, the three matching functions mentioned above is tried sequentially until a successful matching is found. The same matching process is applied to the next pair of nodes, i.e. GB-2Q and GB-2A, then Sel-3A and Sel-3Q. Any failure of matching will result in backtracking to the previous pair of nodes to find another possible matching function for this pair. If the exhaustive search process can successfully match this two plans, the resulting delta plan is converted into a plan against the MV and the plan is equivalent to the query plan. The final plan is converted into a SQL query against MV1 as the result of query rewrite. Note that Spark SQL has an API, SQLBuilder, to convert the resolved logical plan back to

SQL. Most of the work done in this API is via methods called sql in the Spark Expression library. We can use these methods as well to convert the modular plan back to SQL

Cost-Based Rewrite:

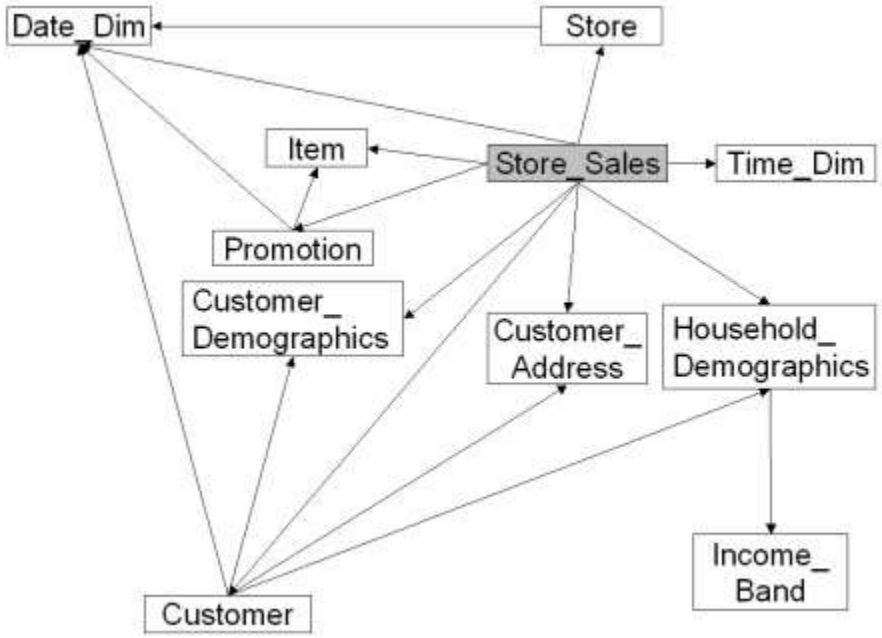
Query rewrite will be available with cost-based optimization in the following fashion. Costs are estimated via Spark catalyst for the input query with and without rewrite and the least costly alternative is selected for execution. The query is rewritten one or more query blocks, one at a time.

If the rewrite logic has a choice between multiple materialized views to rewrite a query block, it will select one to optimize query reduction factor, i.e. the ratio of the sum of the cardinality of the tables in the rewritten query block to that in the original query block.

After a materialized view has been picked for a rewrite, we perform the rewrite, and then test if the rewritten query can be rewritten further with another materialized view. This process continues until no further rewrites are possible. Then the rewritten query is optimized and the original query is optimized. We compare these two optimizations and selects the least costly alternative to submit to Spark for execution.

TPC-DS Examples:

Examples are based on the TPC-DS schema of Store Sales (SS). The Store Sales ER-Diagram is as follows, where Store_Sales is a fact table and the others are dimension tables:



Given the following MV, we show that all the sub-queries in TPC-DS, involving exactly the tables store_sales, item, and date_dim, can be rewritten to run against the MV. The MV is a covering query of the sub-queries. When we find a covering query, however, if the estimated size is too big, we need to split it into several queries for MVs to increase the query reduction factors.

MV:

```
SELECT dt.d_date, item.i_brand, item.i_brand_id, item.i_item_id, item.i_item_desc,
substr(item.i_item_desc, 1, 30) itemdesc, item.i_category, item.i_class,
item.current_price, item.i_item_sk, store_sales.ss_store_sk,
SUM(store_sales.ss_ext_sales_price) sum_agg,
SUM(store_sales.ss_quantity*store_sales.ss_list_price) sales, count(*) number_sales
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
      AND store_sales.ss_item_sk = item.i_item_sk
GROUP BY dt.d_date, item.i_brand, item.i_brand_id, item.i_item_id, item.i_item_desc,
substr(item.i_item_desc, 1, 30), item.i_category, item.i_category_id,
item.i_class, item.i_class_id,item.current_price, item.i_manager_id,
item.i_item_sk,store_sales.ss_store_sk
```

Re-writable queries for 1st phase (not in TPC-DS):

```
SELECT dt.d_year, item.i_brand brand, SUM(ss_ext_sales_price) sum_agg
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
      AND store_sales.ss_item_sk = item.i_item_sk
GROUP BY dt.d_year, item.i_brand, item.i_brand_id
```

Re-writable queries for future phases (sub-queries extracted from TPC-DS):

The following queries are sub-queries extracted from the TPC-DS test queries. Given the test queries, the sub-queries can be found and re-written with a navigator that navigates through test queries of TPC-DS. The navigator should walk through nodes corresponding to the operators UNION, UNION ALL, EXCEPT, INTERSECT, WINDOW expressions, IN sub-queries expressions, etc. to find rewritable sub-queries.

```
// d_year derivable from MV

SELECT substr(item.i_itemdesc,1,30) itemdesc, item.i_item_sk item_sk, date_dim.d_date
solddate, count(*) cnt
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
      AND store_sales.ss_item_sk = item.i_item_sk
      AND date_dim.d_year in (2000, 2000+1, 2000+2, 2000+3)
GROUP BY substr(item.i_item_desc,1,30), item.i_item_sk, date_dim.d_date
```

```
// d_year, d_moy derivable from MV

SELECT ‘store’ channel, item.i_brand_id, item.i_class_id, item.i_category_id,
SUM(store_sales.ss_quantity*store_sales.ss_list_price) sales, count(*) number_sales
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
      AND store_sales.ss_item_sk = item.i_item_sk
      AND date_dim.d_year = 1999 + 2
      AND date_dim.d_moy = 11
GROUP BY item.i_brand_id, item.i_class_id, item.i_category_id

// i_manager_id, d_moy, d_year derivable from MV

SELECT item.i_brand_id brand_id, item.i_brand brand, SUM(ss_ext_sales_price) ext_price
FROM store_sales, item, date_dim
Where store_sales.ss_sold_data_sk = d_date_sk
      AND store_sales.ss_item = item.item_sk
      AND item.i_manager_id = 28
      AND date_dim.d_moy = 11
      AND date_dim.d_year = 1999
GROUP BY item.i_brand, item.i_brand_id



// i_category and d_date derivable from MV, Spark SQL expression of ‘interval  //
30 days’ will be convertible back into SQL expression, window function will be
// supported


SELECT item.i_item_desc, item.i_category, item.i_class, item.current_price,
SUM(ss_ext_sales_price) as itemrevenue,
SUM(ss_ext_sales_price)*100/sum(sum(ss_ext_sales_price)) over (partition by
item.i_class) as revenueratio
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
      AND store_sales.ss_item_sk = item.i_item_sk
      AND item.i_category in (‘Sport’, ‘Books’, ‘Home’)
      AND dt.d_date between cast(‘1999-02-22’ as date)
                                AND (cast(‘1999-02-22’ as date) + interval 30 days)
GROUP   BY   item.i_item_id,   item.i_item_desc,   item.i_category,   item.i_class,
item.i_current_price

// ss_store_sk derivable from MV
// note that this query is extracted from q76 (in appendix) of the TPC_DS test queries.
// To get this query, the optimizer used to compute the optimized logical plan of q76
// for rewrite should be configured with a rule that pushes the Aggregrate operator //
down through the UNION ALL operator and removes the first grouping column, i.e.
// channel


SELECT ‘store’ as channel, store_sales.ss_store_sk col_name, date_dim.d_year,
date_dim.d_qoy, item.i_category, store_sales.ss_ext_sales_price ext_sales_price
FROM store_sales, item, date_dim
Where store_sales.ss_store_sk IS NULL
      AND store_sales.ss_sold_data_sk = d_date_sk
      AND store_sales.ss_item_sk = item.item_sk
GROUP BY col_name, date_dim.d_year, date_dim.d_qoy, item.i_category
```

```
// expression IN will be supported, d_date derivable from MV

SELECT item.i_item_id item_id, sum(ss_ext_sales_price) ss_item_rev
FROM store_sales, item, date_dim
Where store_sales.ss_sold_data_sk = d_date_sk
      AND store_sales.ss_item_sk = item.item_sk
      AND date_dim.d_date in (SELECT d_date
                              FROM date_dim
                              WHERE date_dim.d_week_seq= (SELECT d_week_seq
                                                          FROM date_dim
                                                          WHERE d_date = '2000-01-03'))
GROUP BY item.i_item_id
```

Non-rewritable queries (not in TPC-DS):

```
//  i_manufact_id not derivable from MV

SELECT dt.d_year, item.i_brand_id brand_id, item.i_brand brand, SUM(ss_ext_sales_price)
sum_agg
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
      AND store_sales.ss_item_sk = item.i_item_sk
      AND item.i_manufact_id = 128
      AND dt.d_moy = 11
GROUP BY dt.d_year, item.i_brand, item.i_brand_id



// not the same sets of tables in the FROM clause

SELECT store_sales.ss_customer_sk customer_sk, store_sales.ss_item_sk item_sk
FROM store_sales, date_dim
WHERE store_sales.ss_sold_date_sk = date_dim.d_date_sk GROUP
BY dt.d_year
```

SQL Supportability:

Only a subset of (Spark) SQL will be supported for query rewrite for the first phase. We will extend the supportability for the future in a phase by phase fashion.

In general, operators and expressions that we are NOT considered currently are as follows, some due to limitations on translating back to SQL from logical (modular) plans.

- Operators ○ UNION ○ UNION ALL ○ INTERSECT ○ EXCEPT
    - SCRIPT TRANSFORMATION
    - AGGREGATE after EXPAND (GROUPING SET)
    - ORDER BY

- SORT BY & SORT BY after HAVING ○ DISTRIBUTE BY ○ CLUSTER BY ○ WINDOW

- Expressions ○
  DISTINCT ○
  EXPLODE ○
  GROUPING ○
  CALENDAR
  INTERVAL LITERAL
  ○ Unpublished
  UDFS ○ multi-alias
  UDF

Examples of queries/MVs NOT supported are as follows:

Example 1. INTERSECT

```
SELECT CustomerKey
FROM FactInternetSales
INTERSECT
SELECT CustomerKey
FROM DimCustomer
WHERE DimCustomer.Gender = 'F'
ORDER BY CustomerKey;
```

Example 2. EXCEPT

```
SELECT CustomerKey
FROM FactInternetSales
EXCEPT
SELECT CustomerKey
FROM DimCustomer
WHERE DimCustomer.Gender = 'F'
ORDER BY CustomerKey;
```

Example 3. GROUPING

```
SELECT SalesQuota, SUM(SalesYTD) 'TotalSalesYTD',
GROUPING(SalesQuota) AS 'Grouping'
FROM Sales.SalesPerson
GROUP BY SalesQuota WITH ROLLUP
```

Example 4. CALENDAR INTERVAL LITERAL

```
SELECT product_id, p.name, (sum(s.units) * (p.price -
p.cost)) AS profit
FROM products p JOIN sales s USING (product_id)
WHERE s.date > CURRENT_DATE - INTERVAL '4 weeks'
GROUP BY product_id, p.name, p.price, p.cost
```

Example 5. unpublished test UDFs (test_max)

```
SELECT 1+1, 2+2 as zz, src.key, test_max(length(src.value)),
count(src.value), sin(count(src.value)),
count(sin(src.value)), unix_timestamp(),
       CAST(SUM(IF(value > 10, value, 1)) AS INT), if(src.key >
            1,1,0)
 FROM src group by src.key
```

Example 6. SCRIPT TRANSFORMATION

```
SELECT TRANSFORM(stuff)
USING 'script'
AS (thing1 INT, thing2 INT)
```

Example 7. SORT after HAVING

```
SELECT COUNT(value) FROM parquet_t1 GROUP BY key HAVING MAX(key)
> 0 SORT BY key
```

Example 8. multi-alias i.e. UDF as (e1,e2,e3)

```
SELECT stack(2, *) as (e1,e2,e3) from (
  SELECT concat(*), concat(a.*), concat(b.*), concat(a.*, b.key),
concat(a.key, b.*)
  FROM src a join src b on a.key+1=b.key where a.key < 100) x
LIMIT 10
```

MVs we consider for the 1st phase are as follows:

- MV
  - Simple SELECT-FROM-WHERE-GROUPBY, with aggregate functions without distinct: COUNT(), MAX(), MIN(), SUM(), in other words, we consider SPJG (SELECT-PROJECT-JOIN-GROUP BY) queries ○ When there are 2 or more tables involved, only equi-join (with simple column reference) is supported, i.e., the join predicates must be in the form of "Col1=Col2" ○ NO subquery (in SELECT list or WHERE clause)
  - NO table expression or views in the FROM clause; all table references are "base" tables
  - NO windowing/OLAP functions
  - NO additional clause like ORDER BY, SORT BY, DISTRIBUTE BY, CLUSTER BY, LIMIT ○ NO DISTINCT in SELECT list
  - GROUP BY items can be expressions or simple column reference ○ NO HAVING clause

Example 1. Simple select/filtering.

```
SELECT c1, c2, c3+c4
FROM T
WHERE c5>0
```

```
SELECT sales_trans_id, sales_vol, sales_items, sales_date
FROM Sales
WHERE sales_date > '2017-01-01'
```

```
SELECT c1, c2
FROM T
WHERE c5=1
```

Example 2. Simple select/join.

```
SELECT t_key, tc0, tc1, s_key, sc0, sc1
FROM  T, S
WHERE T.tc=S.sc
```

Example 3. Simple groupby with a single table

```
SELECT c1, c2, sum(c3), count(*)
FROM t
WHERE c5=1
GROUP BY c1, c2
```

Example 4. Simple groupby with a single table and complex grouping expressions

```
SELECT c1+c2, sum(c2), count(*)
FROM t
GROUP BY c1+c2
```

```
SELECT year(sales_date), month(sales_date), sum(sales_rev), count(*)
FROM  sales
GROUP BY year(sales_date), month(sales_date)
```

Example 5. Groupby with multiple tables (natural joins)

```
SELECT t.c1, s.c2, sum(t.c3), count(*)
FROM  t, s
WHERE t.tc=s.sc
```

GROUP BY t.c1, s.c2

User queries we consider, on the other hand, are of the SPJGH (SELECT-PROJECT-JOINGROUPBY-HAVING) form. All the MV queries mentioned above are also considered for user queries. In addition, we also consider queries that have the HAVING clause also.

Appendix A: TPC-DS q76

```
SELECT
    channel, col_name, d_year, d_qoy, i_category, COUNT(*) sales_cnt,
SUM(ext_sales_price) sales_amt
FROM (
   SELECT 'store' as channel, ss_store_sk col_name, d_year, d_qoy, i_category,
ss_ext_sales price ext_sale_price
   FROM store_sales, item, date_dim
   Where ss_store IS NULL
         AND ss_item_sk = item.item_sk
         AND ss_sold_date_sk = d_date_sk
   UNION ALL
   SELECT 'web' as channel, ws_ship_customer_sk col_name, d_year, d_qoy, i_category,
ws_ext_sales_price ext_sales_price
   FROM web_sales, item, date_dim
   WHERE ws_ship_customer_sk IS NULL
         AND ws_sold_date_sk = d_date_sk
         AND ws_item_sk= i_item_sk
   UNION ALL
   SELECT 'catalog' as channel, cs_ship_addr_sk col_name, d_year, d_qoy, i_category,
cs_ext_sales_price ext_sales_price    FROM catalog_sales, item, date_dim
   WHERE cs_ship_addr_sk IS NULL
         AND cs_sold_date_sk = d_date_sk
         AND cs_item_sk = i_item_sk) foo
GROUP BY channel, col_name, d_year, d_qoy, i_category
ORDER BY channel, col_name, d_year, d_qoy, i_category  Limit
100
```

Appendix B:

SparkSQL Internal Libraries:

We will extensively reuse the SparkSQL internal libraries, such as TreeNode, QueryPlan, SparkPlan, and Expressions to facilitate the implementation of query rewrite. The two methods mentioned in the matching functions are semanticEquals and canEvaluate defined in the class Expression and PredicateHelper respectively. semanticEquals returns true when two expressions will always compute the same result, even if they differ cosmetically, e.g. distinct orders of child expressions. canEvaluate(expr: Expression, plan: LogicalPlan) returns true if expr can be evaluated using only the output of plan.

Spark Thrift Server:

With the Spark Thrift Server, queries are sent to the server via the RPC client side call executeStatement(…). This call is handled by the corresponding executeStatement(…) in CLIService of the Thrift Server. The method will eventually call to the method newExecuteStatementOperation(…) of SparkSQLOperationManager. We insert a call ToolContext.queryrewrite(…) of our library before line 8 of the implementation of the method and passing the argument 'statement' of this method and the variable 'hiveContext' retrieved inside this method to the call, i.e. adding ToolContext.queryrewrite(statement, hiveContext) before line 8 of newExecuteStatementOperation(…), we will get a rewritten SQL statement against MVs instead for processing. Note that the approach handles query rewrite synchronously. We will extend it to asynchronous rewrite in the next phase.

Spark-SQL Shell:

With the interactive interface of spark-shell, when the method processCmd(…) of SparkSQLCLIDriver eventually calls to run(…) of SparkSQLDriver, we modify the implementation of run(…) by inserting ToolContext.queryrewrite(…) of our tool before line 5 of the method run, with the arguments 'command' and 'context', i.e. adding ToolContext.queryrewrite(command,context) of our tool before line 5 of the method run(…). The query 'command' will be first rewritten to against MVs before submitting to the Spark cluster.

Spark-Shell:

With the interactive interface of spark, in SparkILoop, we add an overriding method command(line: String) to call ToolContext.queryrewrite(…) so that we call rewrite the query prior to fetching the result.

Web Service:

The micro-service framework, Dropwizard, has been integrated into the Spark cluster, with an approach similar to that of the Spark Thrift Server and spark-sql shell. The integration launches a long-running application with the SparkContext/HiveContext created during the launch. So we can call ToolContext.queryrewrite(query, hivecontext) to rewrite the query in micro-service applications before submitting to the Spark cluster via HiveContext.