Spark Materialized View

Background and Motivation

Cache is widely used in various fields and directions of information processing. At the cost of more resource consumption, cache pushes the data to a position closer to the calculation, thus speeding up the data processing, filling the gap between computing and IO access speed. Materialized views in RDBMS are very common applications for this type of cache. Materialized views are special types of aggregate views that improve query execution times by precalculating expensive joins and aggregation operations prior to execution and storing the results in a table in the database.

Spark natively supports RDD-level cache. Multiple Jobs can share cached RDD. The cached RDD is closer to the calculation result and requires less computation. In addition, the file system level cache such as HDFS cache or Alluxio can also load data into memory in advance, improving data processing efficiency.

A very common and important application scenario of Spark is interactive query. Currently, Spark has done a lot of optimization work in SQL engine, Spark Runtime, data storage format, etc. However, in the case of large data sets, it may not be possible to quickly complete the user's interactive query. The user's query is usually an ad-hoc query, but there may be many general query logic.

Materialized view is an important approach in DBMS to cache data to accelerate queries. By creating a materialized view through SQL, the data that can be cached is very flexible, and needs to be configured arbitrarily according to specific usage scenarios. The Materialization Manager automatically updates the cache data according to changes in detail source tables, simplifying user work. When user submit query, Spark optimizer rewrites the execution plan based on the available materialized view to determine the optimal execution plan.

Target Personas

Data scientists, data analysts, DBA, business Intelligences.

Goals

1. Mechanisms to *define* materialized views in SQL.
    a. Full SQL syntax includes CREATE/DROP/REFRESH/ALTER/DESC/SHOW CREATE/HINT etc.
    b. New logical plan nodes and new physical nodes in Spark plan for MV.
    c. Support using Hive 1.2 version or above.
    d. Support creating cross-session (cross-jvm) or user session level materialized views.
    e. Support using tables or views as materialized view's detail data sources.
    f. Support partitioning (range partitioning) and bucketing materialized view.
2. Multiple *refresh* mechanisms to ensure that all materialized views are updated when the underlying detail tables are modified.
    a. Manual refresh by SQL.
    b. Regular deactivate expired materialized view by default.
    c. Event triggered expired materialized view under some restrictions.
3. A query *rewrite* capability to transparently rewrite a query to use a materialized view[1][2].
    a. Query rewrite capability is transparent to SQL applications.
    b. Query rewrite can be disabled at the system level or on individual materialized view. Also it can be disabled for a specified query via hint.
    c. Query rewrite as a rule in optimizer should be made sure that it won't cause performance regression if it can use other index or cache.
4. A cost based comparator to pick the best plan to execute.

Non-Goals

- From Hive 3.x, Hive supports materialized view, upgrading Hive is not our goal.
- One challenging part of materialized view is how to recommend or generate materialized views automatically, it is actually based on the queries and types of business. It's not a goal here. However, in our internal Spark, we have had a Spark Auditor which is running in thrift server to collect and send query log to Kafka. With the Spark Auditor, the parts is almost done.
- Also, using lattices to create and populate is not current goal.

- ACL is not included in this proposal. Similarly, we had developed column level ACL in our internal Spark. It can cooperate with materialized view. But it’s too complex to backport into here. So it’s not current goal.

Proposed Changes

1. Add full SQL syntax includes CREATE/DROP/REFRESH/ALTER/DESC/SHOW CREATE/HINT etc.
2. Add new type in *CatalogTableType*, the new types won’t map to *HiveTableType*, instead, we add some preportes in Hive Metastore. So it doesn’t restrict to specific Hive version.
3. MV CREATE command is similar with CREATE VIEW AS command, it reuse *CreateTable* class. So we just modify the ResolveRelations like:

```
case p @ SubqueryAlias(table, view: View) =>
  val newChild =
    if (view.desc.tableType == CatalogTableType.MATERIALIZED) {
      resolveRelation(UnresolvedCatalogRelation(view.desc))
    } else {
      resolveRelation(view)
    }
  p.copy(child = newChild)
```

4. Add a new *CreateMaterializedViewCommand* node like *CreateDataSourceTableAsSelectCommand*. The entire execution plan looks like:

```
spark-sql> EXPLAIN EXTENDED CREATE TEMPORARY MATERIALIZED VIEW mv1 spark-sql>
CLUSTERED BY (prod_id) SORTED BY (prod_id, dt) INTO 2 BUCKETS
spark-sql> AS SELECT * FROM t1;

== Parsed Logical Plan ==
Time taken: 0.061 seconds, Fetched 1 row(s)
'CreateTable `mv1`, ErrorIfExists
+- 'Project [*]
  +- 'UnresolvedRelation `t1`

== Analyzed Logical Plan ==
CreateMaterializedViewCommand `mv1`, ErrorIfExists, [prod_id, dt], true
+- Project [prod_id#4, dt#5L]
  +- SubqueryAlias t1
    +- Relation[prod_id#4,dt#5L] parquet

== Optimized Logical Plan ==
CreateMaterializedViewCommand `mv1`, ErrorIfExists, [prod_id, dt], true,
Statistics(sizeInBytes=0.0 B, hints=none, atts=Map())
+- Relation[prod_id#4,dt#5L] parquet, Statistics(sizeInBytes=0.0 B, hints=none, atts=Map())

== Physical Plan ==
Execute CreateMaterializedViewCommand CreateMaterializedViewCommand `mv1`,
ErrorIfExists, [prod_id, dt], true
+- *(2) Sort [prod_id#4 ASC NULLS FIRST, dt#5L ASC NULLS FIRST], false, 0
  +- Exchange hashpartitioning(prod_id#4, 2, class
org.apache.spark.sql.catalyst.expressions.Murmur3Hash, false)
     +- *(1) FileScan parquet default.t1[prod_id#4,dt#5L] Batched: true, Format: Parquet, Location:
InMemoryFileIndex[file:/Users/lajin/git/ebay/carmel-spark_3/spark-warehouse/t1],
PartitionFilters: [], PushedFilters: [], ReadSchema: struct<prod_id:int,dt:bigint>
```

5. Add a new *MaterializationSourceScanExec* like *FileSourceScanExec*. The entire execution plan looks like:

```
spark-sql>EXPLAIN EXTENDED SELECT * FROM t1;
== Parsed Logical Plan ==
'Project [*]
+- 'UnresolvedRelation `t1`

== Analyzed Logical Plan ==
prod_id: int, dt: bigint
Project [prod_id#4, dt#5L]
+- SubqueryAlias t1
  +- Relation[prod_id#4,dt#5L] parquet

== Optimized Logical Plan ==
Relation[prod_id#4,dt#5L] parquet, Statistics(sizeInBytes=0.0 B, hints=none, atts=Map())

== Physical Plan ==
*(1) MaterializationScan parquet mv1[prod_id#4,dt#5L] Batched: true, Format: Parquet,
Location:
InMemoryFileIndex[file:/Users/lajin/git/ebay/carmel-spark_3/spark-warehouse/tmv-7a5681ea-3b
48-4a2..., PartitionFilters: [], PushedFilters: [], ReadSchema: struct<prod_id:int,dt:bigint>,
SelectedBucketsCount: 2 out of 2, TemporaryMV: true
```

6. Add a new Optimizer Rule *ChooseMaterializationToRewrite* to rewrite.

Design Sketch

DDL

CREATE

```
CREATE [TEMPORARY] [MATERIALIZED [VIEW] | MV] [IF NOT EXISTS]
[db_name.]table_name
  USING datasource
  [OPTIONS (key1=val1, key2=val2, ...)]
  [PARTITIONED BY (col_name1, col_name2, ...)]
  [CLUSTERED BY (col_name3, col_name4, ...) INTO num_buckets BUCKETS]
  [LOCATION path]
  [COMMENT table_comment]
  [TBLPROPERTIES (key1=val1, key2=val2, ...)]
  AS select_statement
```

DROP

```
DROP [TEMPORARY] [MATERIALIZED [VIEW] | MV] [IF EXISTS] [db_name.]view_name
```

SHOW CREATE

```
SHOW CREATE [MATERIALIZED [VIEW] | MV] [db_name.]table_name
```

DESCRIBE

```
[DESCRIBE|DESC] [MATERIALIZED [VIEW] | MV] [EXTENDED|FORMATED]
[db_name.]table_name
```

EXPLAIN

```
EXPLAIN [EXTENDED] CREATE [TEMPORARY] [MATERIALIZED [VIEW] | MV] ...
```

SET

```
SET QUERY_REWRITE_ENABLED = [true|false]
```

ALTER

```
ALTER [MATERIALIZED [VIEW] | MV] [DISABLE | ENABLE] QUERY REWRITE
```

DML

REFRESH

```
REFRESH [MATERIALIZED [VIEW] | MV] [db_name.]table_name
```

LOAD

```
LOAD [MATERIALIZED [VIEW] | MV] ON DATABASE db_name
```

HINT

```
SELECT /*+ NOREWRITE */ ...
```

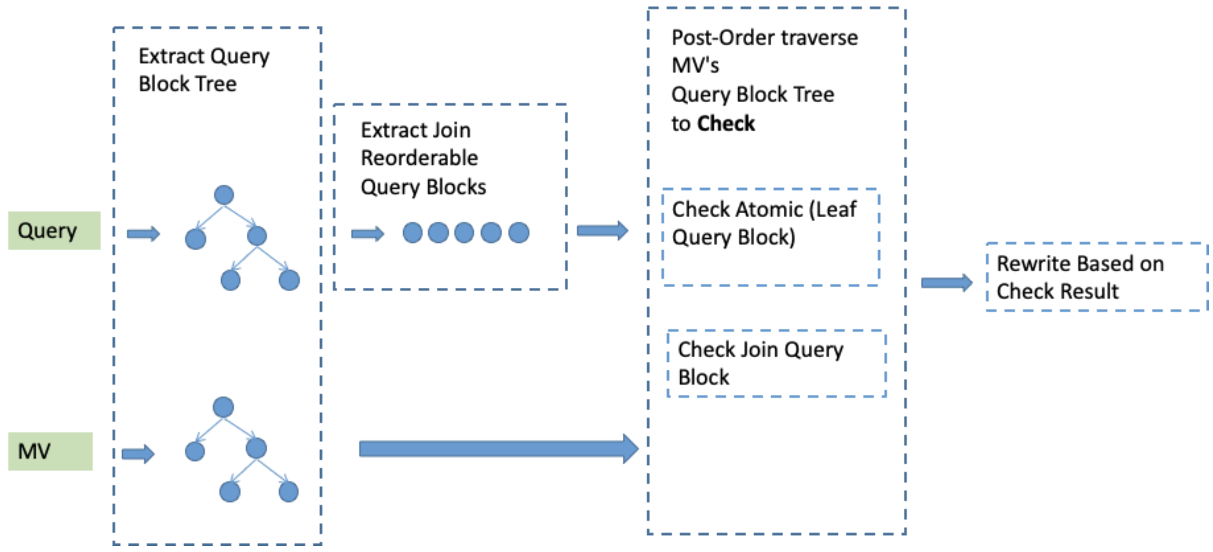
Query Block Based Rewrite Algorithm

Check Atomic Query Block

- Data sufficient check => Filter, Project
- Grouping Compatibility Check => Aggregate
- Aggregate Computability Check => Aggregate Expression

Check Join Query Block

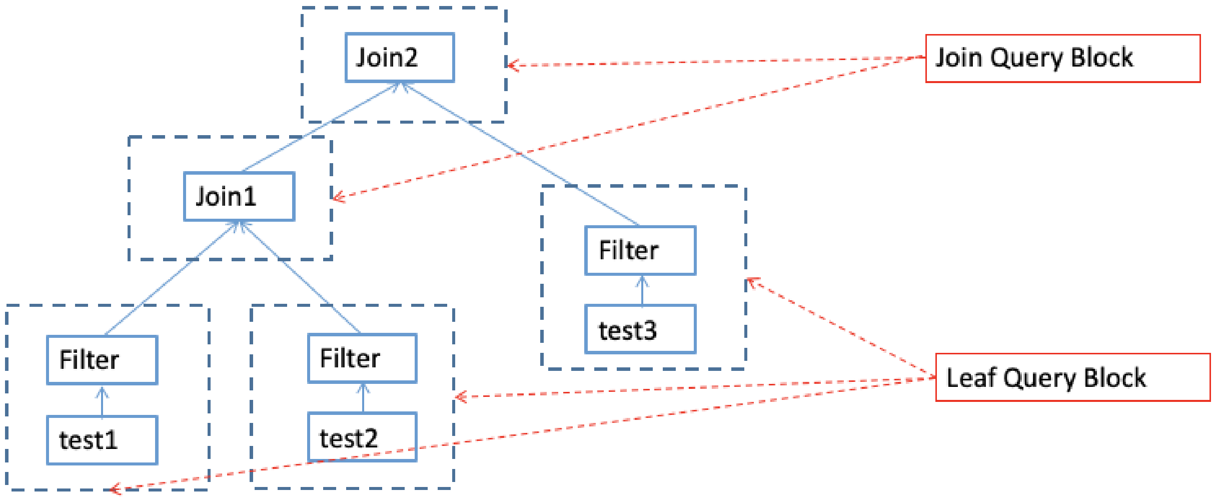
- Join Compatibility Check

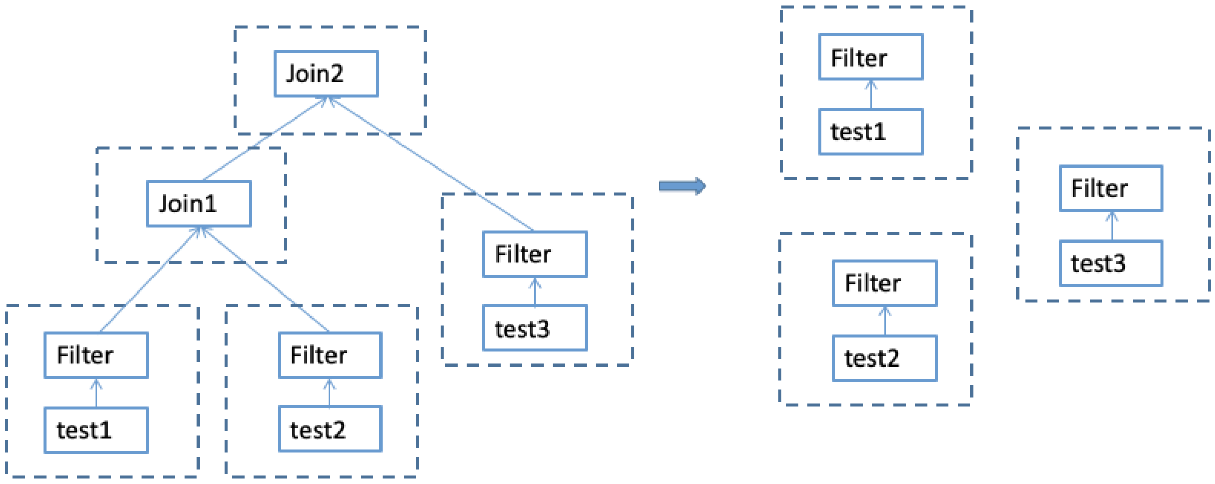

Query
MV
Extract Query Block Tree
Extract Join Reorderable Query Blocks
Post-Order traverse MV's Query Block Tree to **Check**
Check Atomic (Leaf Query Block)
Check Join Query Block
Rewrite Based on Check Result

Example

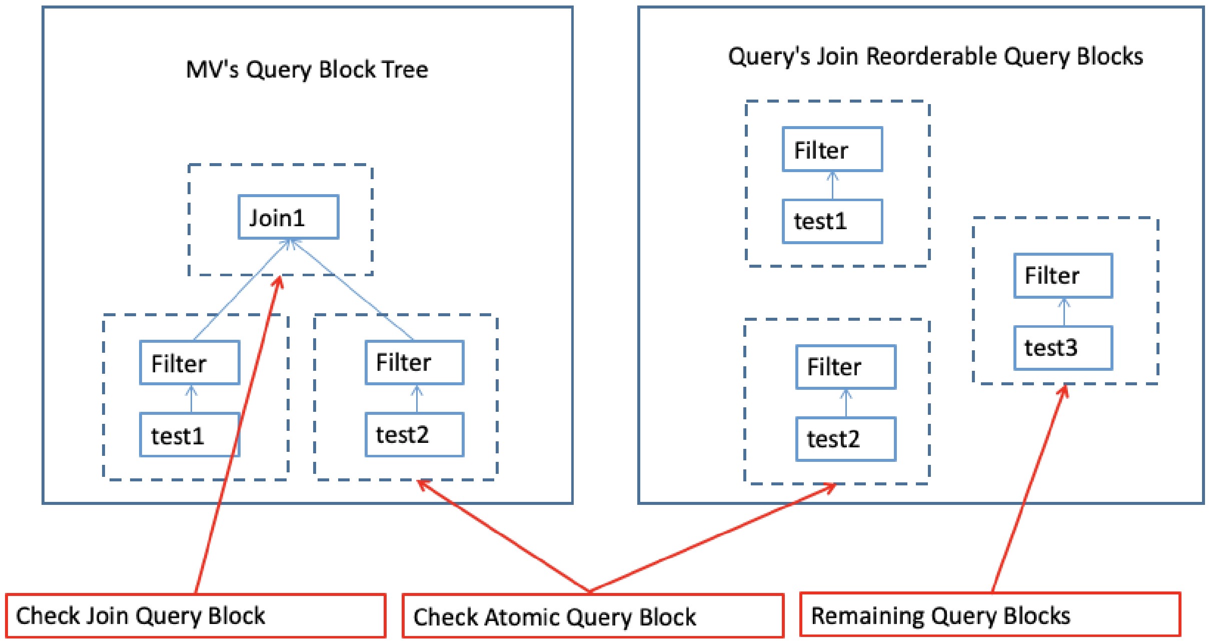
```
select * from test1, (select * from test2 where b =1) as t2, test3 where test1.a = t2.a and
t2.a = test3.a
```

1. Extract QueryBlockTree both for the query and each MaterializedView. There are two types of QB: JoinQueryBlock and LeafQueryBlock.

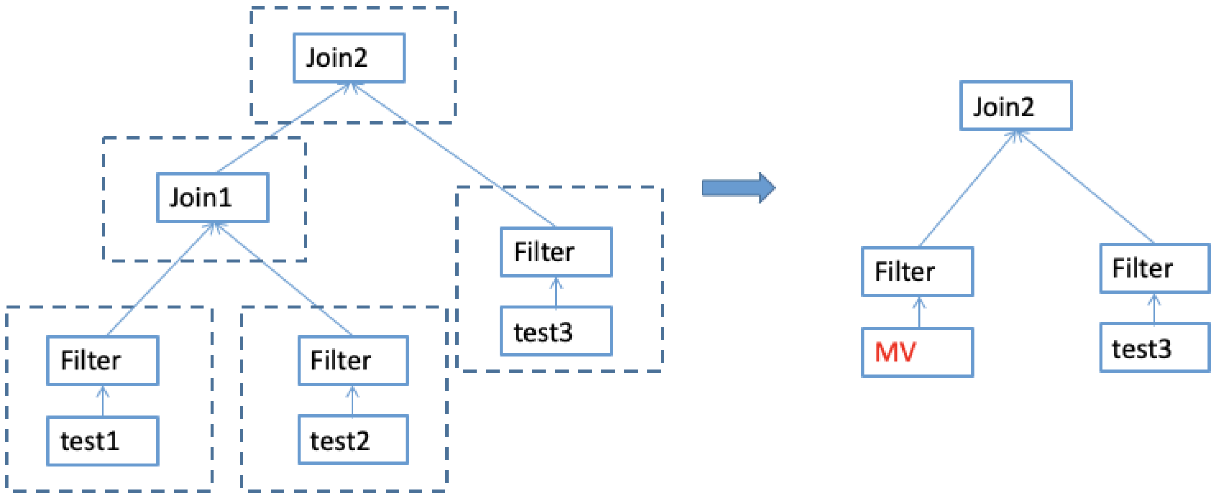




3. Check, build a RewriteContext.



4. Use the RewriteContext to rewrite the query. Use cost based comparator to pick the final result.



Performance Testing

The performance is based the rewrite successful rate and the construction of materialized views. Here is an example in TPCH.

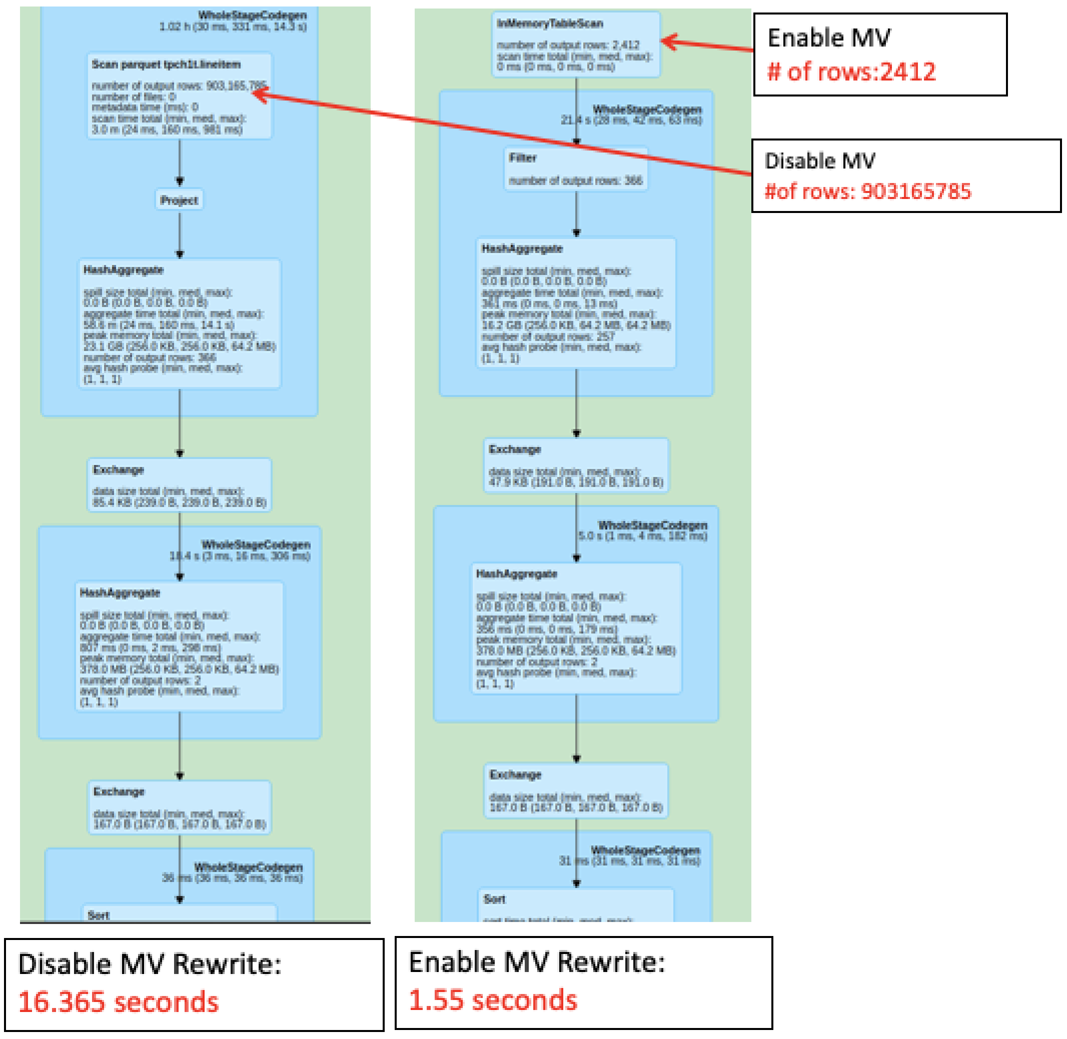
Materialized view:

```
create materialized view q1_mv as
select
        l_returnflag,
        l_linestatus, l_shipdate,
        sum(l_quantity) as sum_qty,
        sum(l_extendedprice) as sum_base_price,
        sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
            sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
            count(l_quantity) as cnt_qty,
            count(l_extendedprice) as cnt_price,
            count(l_discount) as cnt_disc,
            sum(l_discount) as sum_disc,
            count(*) as count_order
from
            lineitem
group by
            l_returnflag,
            l_linestatus,
            l_shipdate;
```

Query 1:

```
select
            l_returnflag,
            l_linestatus,
            sum(l_quantity) as sum_qty,
            sum(l_extendedprice) as sum_base_price,
            sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
            sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
            avg(l_quantity) as avg_qty,
            avg(l_extendedprice) as avg_price,
            avg(l_discount) as avg_disc,
            count(*) as count_order
from
            lineitem
where
            l_shipdate <= date '1998-09-02' and  l_shipdate >= date '1997-09-02'
group by
            l_returnflag,
            l_linestatus
order by
            l_returnflag,
            l_linestatus;
```

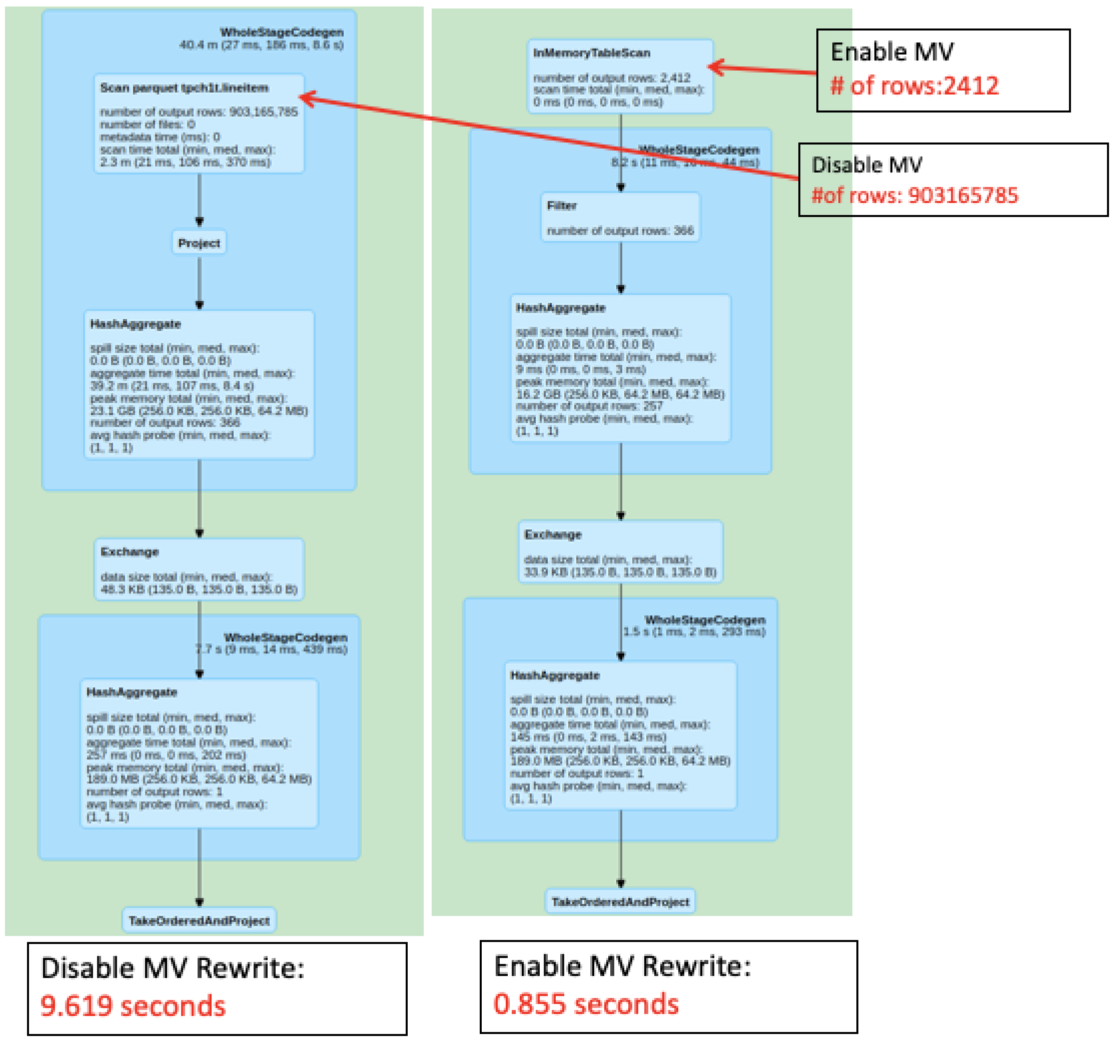
Result:



Query 2:

```
select
            l_returnflag,
            sum(l_quantity) as sum_qty,
            sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
            sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
            avg(l_extendedprice) as avg_price,
            count(*) as count_order
from
            lineitem
where
            l_shipdate <= date '1998-09-02' and  l_shipdate >= date '1997-09-02'
group by
            l_returnflag
order by
            l_returnflag
limit 5;
```

Result:



```
WholeStageCodegen
40.4 m (27 ms, 186 ms, 8.6 s)

Scan parquet tpch1t.lineitem

number of output rows: 903,165,785
number of files: 0
metadata time (ms): 0
scan time total (min, med, max):
2.3 m (21 ms, 106 ms, 370 ms)

Project

HashAggregate

spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
39.2 m (21 ms, 107 ms, 8.4 s)
peak memory total (min, med, max):
23.1 GB (256.0 KB, 256.0 KB, 64.2 MB)
number of output rows: 366
avg hash probe (min, med, max):
(1, 1, 1)

Exchange

data size total (min, med, max):
48.3 KB (135.0 B, 135.0 B, 135.0 B)

WholeStageCodegen
7.7 s (9 ms, 14 ms, 439 ms)

HashAggregate

spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
257 ms (0 ms, 0 ms, 202 ms)
peak memory total (min, med, max):
189.0 MB (256.0 KB, 256.0 KB, 64.2 MB)
number of output rows: 1
avg hash probe (min, med, max):
(1, 1, 1)

TakeOrderedAndProject
```

```
InMemoryTableScan

number of output rows: 2,412
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)

WholeStageCodegen
8.2 s (11 ms, 16 ms, 44 ms)

Filter

number of output rows: 366

HashAggregate

spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
9 ms (0 ms, 0 ms, 3 ms)
peak memory total (min, med, max):
16.2 GB (256.0 KB, 64.2 MB, 64.2 MB)
number of output rows: 257
avg hash probe (min, med, max):
(1, 1, 1)

Exchange

data size total (min, med, max):
33.9 KB (135.0 B, 135.0 B, 135.0 B)

WholeStageCodegen
1.5 s (1 ms, 2 ms, 293 ms)

HashAggregate

spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
145 ms (0 ms, 2 ms, 143 ms)
peak memory total (min, med, max):
189.0 MB (256.0 KB, 256.0 KB, 64.2 MB)
number of output rows: 1
avg hash probe (min, med, max):
(1, 1, 1)

TakeOrderedAndProject
```

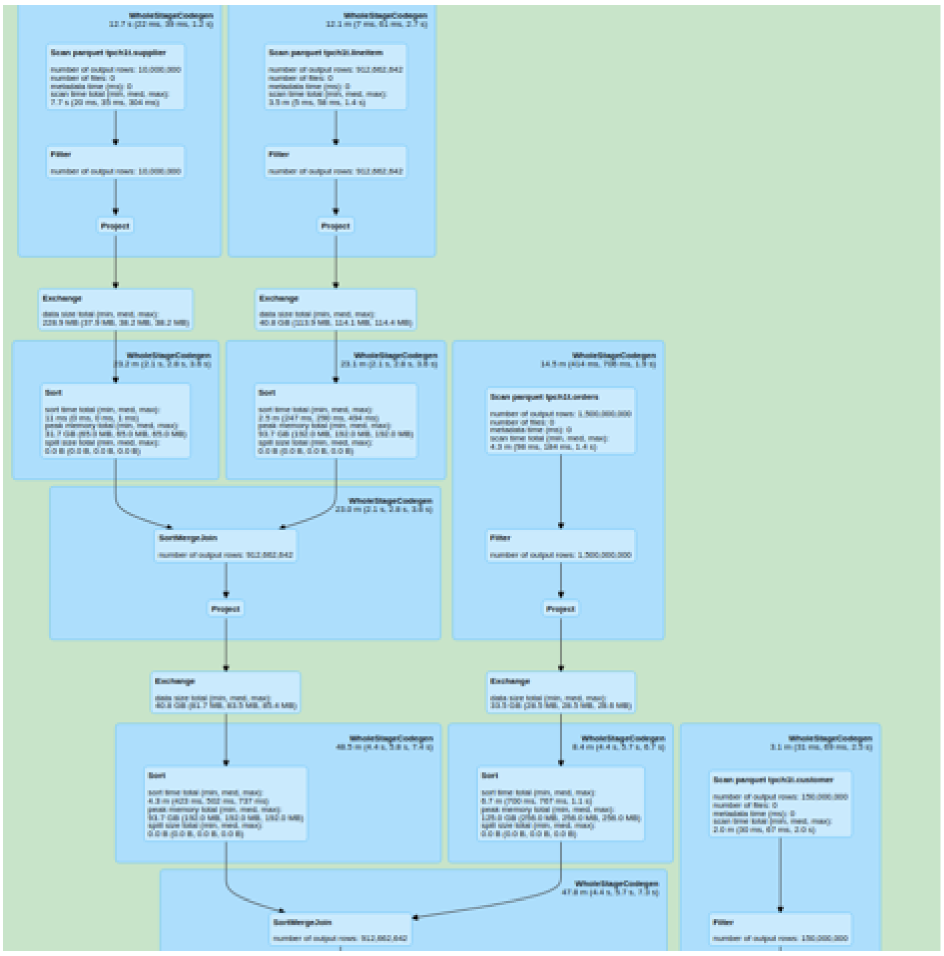
MV 2:

```
create materialized view q2_mv as
select *
from
        lineitem
        join orders on o_orderkey = l_orderkey
        join supplier on s_suppkey = l_suppkey
        join nation n1 on s_nationkey = n1.n_nationkey
where
        l_shipdate between date '1996-01-01' and date '1996-12-31';
```

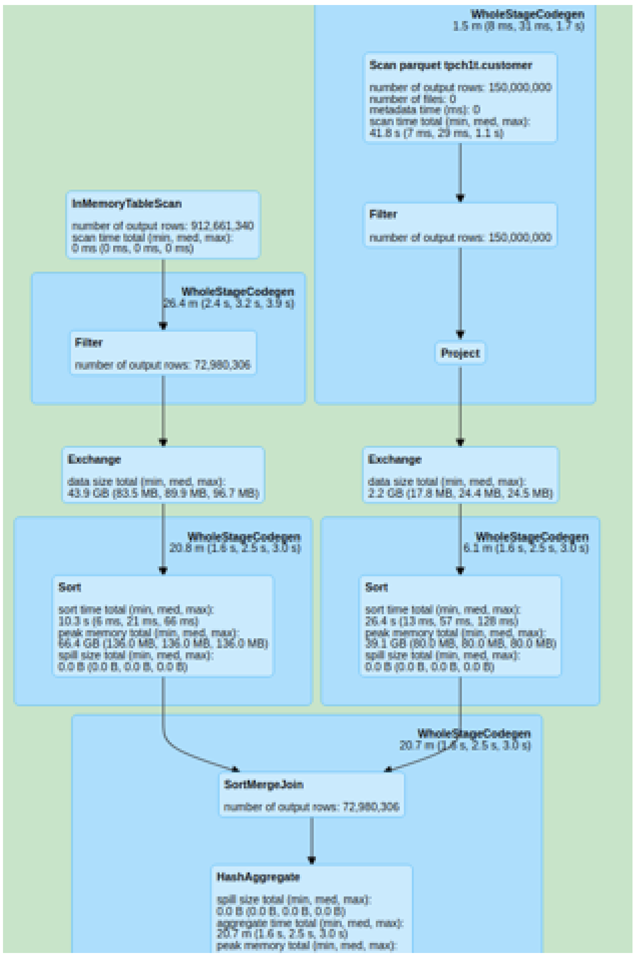
Query 3:

```
select
        n_name as supp_nation,
        l_shipdate as l_year,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        supplier,
        lineitem,
        orders,
        customer,
        nation
where
        s_suppkey = l_suppkey
        and o_orderkey = l_orderkey
        and c_custkey = o_custkey
        and s_nationkey = n_nationkey
        and l_shipdate between date '1996-01-01' and date '1996-12-31'
        and (
          (n_name = 'FRANCE')
          or (n_name = 'GERMANY')
        )
group by
        n_name,
        l_shipdate
order by
        supp_nation,
        l_year;
```

Result:



Disable MV Rewrite:
22.92 seconds


WholeStageCodegen
1.5 m (8 ms, 31 ms, 1.7 s)
Scan parquet tpch1t.customer
number of output rows: 150,000,000
number of files: 0
metadata time (ms): 0
scan time total (min, med, max):
41.8 s (7 ms, 29 ms, 1.1 s)
Filter
number of output rows: 150,000,000
Project
InMemoryTableScan
number of output rows: 912,661,340
scan time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
WholeStageCodegen
26.4 m (2.4 s, 3.2 s, 3.9 s)
Filter
number of output rows: 72,980,306
Exchange
data size total (min, med, max):
43.9 GB (83.5 MB, 89.9 MB, 96.7 MB)
Exchange
data size total (min, med, max):
2.2 GB (17.8 MB, 24.4 MB, 24.5 MB)
WholeStageCodegen
20.8 m (1.6 s, 2.5 s, 3.0 s)
Sort
sort time total (min, med, max):
10.3 s (6 ms, 21 ms, 66 ms)
peak memory total (min, med, max):
66.4 GB (136.0 MB, 136.0 MB, 136.0 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen
6.1 m (1.6 s, 2.5 s, 3.0 s)
Sort
sort time total (min, med, max):
26.4 s (13 ms, 57 ms, 128 ms)
peak memory total (min, med, max):
39.1 GB (80.0 MB, 80.0 MB, 80.0 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
WholeStageCodegen
20.7 m (1.6 s, 2.5 s, 3.0 s)
SortMergeJoin
number of output rows: 72,980,306
HashAggregate
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
aggregate time total (min, med, max):
20.7 m (1.6 s, 2.5 s, 3.0 s)
peak memory total (min, med, max):

Enable MV Rewrite:
10.844 seconds

Plan

We have already completed the code to address Goal 1, Goal 2.a, 2.b, Goal 3 and partial Goal 4. And relevant unit tests and partial TPCH benchmark.
After SPIP accepted, we will create tickets to post above codes. Fully address all goals will be in mid of Q42019.

Referance

[1] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.113
[2] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.97.6252