# Materialized views



**Version information**

Materialized views support is introduced in Hive 3.0.0.

## Introduction

This page documents the work done for the supporting materialized views in Apache Hive.

### Objectives

Traditionally, one of the most powerful techniques used to accelerate query processing in data warehouses is the pre-computation of relevant summaries or materialized views.

The initial implementation introduced in Apache Hive 3.0.0 focuses on introducing materialized views and automatic query rewriting based on those materializations in the project. In particular, materialized views can be stored natively in Hive or in other systems such as Druid using custom storage handlers, and they can seamlessly exploit new exciting Hive features such as LLAP acceleration. Then the optimizer relies in Apache Calcite to automatically produce full and partial rewritings for a large set of query expressions comprising projections, filters, join, and aggregation operations.

In this document, we provide details about materialized view creation and management in Hive, describe the current coverage of the rewriting algorithm with some examples, and explain how Hive controls important aspects of the life cycle of the materialized views such as the freshness of their data.

## Management of materialized views in Hive

In this section, we present the main operations that are currently present in Hive for materialized views management.

### Materialized views creation

The syntax to create a materialized view in Hive is very similar to the CTAS statement syntax, supporting common features such as partition columns, custom storage handler, or passing table properties.

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [db_name.]materialized_view_name
  [DISABLE REWRITE]
  [COMMENT materialized_view_comment]
  [PARTITIONED ON (col_name, ...)]
  [CLUSTERED ON (col_name, ...) | DISTRIBUTED ON (col_name, ...) SORTED ON (col_name, ...)]
  [
    [ROW FORMAT row_format]
    [STORED AS file_format]
      | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]
  ]
  [LOCATION hdfs_path]
  [TBLPROPERTIES (property_name=property_value, ...)]
AS
<query>;
```

When a materialized view is created, its contents will be automatically populated by the results of executing the query in the statement. The materialized view creation statement is atomic, meaning that the materialized view is not seen by other users until all query results are populated.

By default, materialized views are usable for query rewriting by the optimizer, while the `DISABLE REWRITE` option can be used to alter this behavior at materialized view creation time.

The default values for SerDe and storage format when they are not specified in the materialized view creation statement (they are optional) are specified using the configuration properties `hive.materializedview.serde` and `hive.materializedview.fileformat`, respectively.

Materialized views can be stored in external systems, e.g., Druid, using custom storage handlers. For instance, the following statement creates a materialized view that is stored in Druid:

**Example:**

```
CREATE MATERIALIZED VIEW druid_wiki_mv
STORED AS 'org.apache.hadoop.hive.druid.DruidStorageHandler'
AS
SELECT __time, page, user, c_added, c_removed
FROM src;
```

### Other operations for materialized view management

Currently we support the following operations that aid at managing the materialized views in Hive:

```
-- Drops a materialized view
DROP MATERIALIZED VIEW [db_name.]materialized_view_name;
-- Shows materialized views (with optional filters)
SHOW MATERIALIZED VIEWS [IN database_name] ['identifier_with_wildcards'];
-- Shows information about a specific materialized view
DESCRIBE [EXTENDED | FORMATTED] [db_name.]materialized_view_name;
```

The functionality of these operations will be extended in the future and more operations may be added.

## Materialized view-based query rewriting

Once a materialized view has been created, the optimizer will be able to exploit its definition semantics to automatically rewrite incoming queries using materialized views, and hence, accelerate query execution.

The rewriting algorithm can be enabled and disabled globally using the `hive.materializedview.rewriting` configuration property (default value is `true`). In addition, users can selectively enable/disable materialized views for rewriting. Recall that, by default, materialized views are enabled for rewriting at creation time. To alter that behavior, the following statement can be used:

```
ALTER MATERIALIZED VIEW [db_name.]materialized_view_name ENABLE|DISABLE REWRITE;
```

The rewriting algorithm is part of Apache Calcite and it supports queries containing TableScan, Project, Filter, Join, and Aggregate operators. More information about the rewriting coverage can be found here. In the following, we include a few examples that briefly illustrate different rewritings.

### Example 1

Consider the database schema created by the following DDL statements:

```
CREATE TABLE emps (
  empid INT,
  deptno INT,
  name VARCHAR(256),
  salary FLOAT,
  hire_date TIMESTAMP)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');

CREATE TABLE depts (
  deptno INT,
  deptname VARCHAR(256),
  locationid INT)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');
```

Assume we want to obtain frequently information about employees that were hired in different period granularities after 2016 and their departments. We may create the following materialized view:

```
CREATE MATERIALIZED VIEW mv1
AS
SELECT empid, deptname, hire_date
FROM emps JOIN depts
  ON (emps.deptno = depts.deptno)
WHERE hire_date >= '2016-01-01';
```

Then, the following query extracting information about employees that were hired in Q1 2018 is issued to Hive:

```
SELECT empid, deptname
FROM emps
JOIN depts
  ON (emps.deptno = depts.deptno)
WHERE hire_date >= '2018-01-01'
    AND hire_date <= '2018-03-31';
```

Hive will be able to rewrite the incoming query using the materialized view, including a compensation predicate on top of the scan over the materialization. Though the rewriting happens at the algebraic level, to illustrate this example, we include the SQL statement equivalent to the rewriting using the `mv` used by Hive to answer the incoming query:

```
SELECT empid, deptname
FROM mv1
WHERE hire_date >= '2018-01-01'
    AND hire_date <= '2018-03-31';
```

## Example 2

For the second example, consider the star schema based on the SSB benchmark created by the following DDL statements:

```
CREATE TABLE `customer`(
  `c_custkey` BIGINT,
  `c_name` STRING,
  `c_address` STRING,
  `c_city` STRING,
  `c_nation` STRING,
  `c_region` STRING,
  `c_phone` STRING,
  `c_mktsegment` STRING,
  PRIMARY KEY (`c_custkey`) DISABLE RELY)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');

CREATE TABLE `dates`(
  `d_datekey` BIGINT,
  `d_date` STRING,
  `d_dayofweek` STRING,
  `d_month` STRING,
  `d_year` INT,
  `d_yearmonthnum` INT,
  `d_yearmonth` STRING,
  `d_daynuminweek` INT,
  `d_daynuminmonth` INT,
  `d_daynuminyear` INT,
  `d_monthnuminyear` INT,
  `d_weeknuminyear` INT,
  `d_sellingseason` STRING,
  `d_lastdayinweekfl` INT,
  `d_lastdayinmonthfl` INT,
  `d_holidayfl` INT,
  `d_weekdayfl`INT,
  PRIMARY KEY (`d_datekey`) DISABLE RELY)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');

CREATE TABLE `part`(
```

```
  `p_partkey` BIGINT,
  `p_name` STRING,
  `p_mfgr` STRING,
  `p_category` STRING,
  `p_brand1` STRING,
  `p_color` STRING,
  `p_type` STRING,
  `p_size` INT,
  `p_container` STRING,
  PRIMARY KEY (`p_partkey`) DISABLE RELY)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');

CREATE TABLE `supplier`(
  `s_suppkey` BIGINT,
  `s_name` STRING,
  `s_address` STRING,
  `s_city` STRING,
  `s_nation` STRING,
  `s_region` STRING,
  `s_phone` STRING,
  PRIMARY KEY (`s_suppkey`) DISABLE RELY)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');

CREATE TABLE `lineorder`(
  `lo_orderkey` BIGINT,
  `lo_linenumber` int,
  `lo_custkey` BIGINT not null DISABLE RELY,
  `lo_partkey` BIGINT not null DISABLE RELY,
  `lo_suppkey` BIGINT not null DISABLE RELY,
  `lo_orderdate` BIGINT not null DISABLE RELY,
  `lo_ordpriority` STRING,
  `lo_shippriority` STRING,
  `lo_quantity` DOUBLE,
  `lo_extendedprice` DOUBLE,
  `lo_ordtotalprice` DOUBLE,
  `lo_discount` DOUBLE,
  `lo_revenue` DOUBLE,
  `lo_supplycost` DOUBLE,
  `lo_tax` DOUBLE,
  `lo_commitdate` BIGINT,
  `lo_shipmode` STRING,
  PRIMARY KEY (`lo_orderkey`) DISABLE RELY,
  CONSTRAINT fk1 FOREIGN KEY (`lo_custkey`) REFERENCES `customer_n1`(`c_custkey`) DISABLE RELY,
  CONSTRAINT fk2 FOREIGN KEY (`lo_orderdate`) REFERENCES `dates_n0`(`d_datekey`) DISABLE RELY,
  CONSTRAINT fk3 FOREIGN KEY (`lo_partkey`) REFERENCES `ssb_part_n0`(`p_partkey`) DISABLE RELY,
  CONSTRAINT fk4 FOREIGN KEY (`lo_suppkey`) REFERENCES `supplier_n0`(`s_suppkey`) DISABLE RELY)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');
```

As you can observe, we declare multiple integrity constraints for the database, using the `RELY` keyword so they are visible to the optimizer. Now assume we want to create a materialization that denormalizes the database contents (consider `dims` to be the set of dimensions that we will be querying often):

```
CREATE MATERIALIZED VIEW mv2
AS
SELECT <dims>,
    lo_revenue,
    lo_extendedprice * lo_discount AS d_price,
    lo_revenue - lo_supplycost
FROM customer, dates, lineorder, part, supplier
WHERE lo_orderdate = d_datekey
    AND lo_partkey = p_partkey
    AND lo_suppkey = s_suppkey
    AND lo_custkey = c_custkey;
```

The materialized view above may accelerate queries that execute joins among the different tables in the database. For instance, consider the following query:

```
SELECT SUM(lo_extendedprice * lo_discount)
FROM lineorder, dates
WHERE lo_orderdate = d_datekey
  AND d_year = 2013
  AND lo_discount between 1 and 3;
```

Though the query does not use all tables present in the materialized view, it may be answered using the materialized view because the joins in `mv2` preserve all the rows in the `lineorder` table (we know this because of the integrity constraints). Hence, the materialized view-based rewriting produced by the algorithm would be the following:

```
SELECT SUM(d_price)
FROM mv2
WHERE d_year = 2013
  AND lo_discount between 1 and 3;
```

### Example 3

For the third example, consider the database schema with a single table that stores the edit events produced by a given website:

```
CREATE TABLE `wiki` (
  `time` TIMESTAMP,
  `page` STRING,
  `user` STRING,
  `characters_added` BIGINT,
  `characters_removed` BIGINT)
STORED AS ORC
TBLPROPERTIES ('transactional'='true');
```

For this example, we will use Druid to store the materialized view. Assume we want to execute queries over the table, however we are not interested on any information about the events at a higher time granularity level than a minute. We may create the following materialized view that rolls up the events by the minute:

```
CREATE MATERIALIZED VIEW mv3
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
AS
SELECT floor(time to minute) as `__time`, page,
    SUM(characters_added) AS c_added,
    SUM(characters_removed) AS c_removed
FROM wiki
GROUP BY floor(time to minute), page;
```

Then, assume we need to answer the following query that extracts the number of characters added per month:

```
SELECT floor(time to month),
    SUM(characters_added) AS c_added
FROM wiki
GROUP BY floor(time to month);
```

Hive will be able to rewrite the incoming query using `mv3` by rolling up the data of the materialized view to month granularity and projecting the information needed for the query result:

```
SELECT floor(time to month),
    SUM(c_added)
FROM mv3
GROUP BY floor(time to month);
```

## Materialized view maintenance

When data in the source tables used by a materialized view changes, e.g., new data is inserted or existing data is modified, we will need to refresh the contents of the materialized view to keep it up-to-date with those changes. Currently, the rebuild operation for a materialized view needs to be triggered by the user. In particular, the user should execute the following statement:

```
ALTER MATERIALIZED VIEW [db_name.]materialized_view_name REBUILD;
```

Hive supports incremental view maintenance, i.e., only refresh data that was affected by the changes in the original source tables. Incremental view maintenance will decrease the rebuild step execution time. In addition, it will preserve LLAP cache for existing data in the materialized view.

By default, Hive will attempt to rebuild a materialized view incrementally, falling back to full rebuild if it is not possible. Current implementation only supports incremental rebuild when there were `INSERT` operations over the source tables, while `UPDATE` and `DELETE` operations will force a full rebuild of the materialized view.

To execute incremental maintenance, following conditions should be met:

- The materialized view should only use transactional tables, either micromanaged or ACID.
- If the materialized view definition contains a Group By clause, the materialized view should be stored in an ACID table, since it needs to support MERGE operation. For materialized view definitions consisting of Scan-Project-Filter-Join, this restriction does not exist.

A rebuild operation acquires an exclusive write lock over the materialized view, i.e., for a given materialized view, only one rebuild operation can be executed at a given time.

## Materialized view lifecycle

By default, once a materialized view contents are stale, the materialized view will not be used for automatic query rewriting.

However, in some occasions it may be fine to accept stale data, e.g., if the materialized view uses non-transactional tables and hence we cannot verify whether its contents are outdated, however we still want to use the automatic rewriting. For those occasions, we can combine a rebuild operation run periodically, e.g., every 5minutes, and define the required freshness of the materialized view data using the `hive.materializedview.rewriting.time.window` configuration parameter, for instance:

```
SET hive.materializedview.rewriting.time.window=10min;
```

The parameter value can be also overridden by a concrete materialized view just by setting it as a table property when the materialization is created.

## Open issues (JIRA)

| key | summary | type | created | updated | due | assignee | reporter | priority | status | resolution |
|---|---|---|---|---|---|---|---|---|---|---|

⚠ JQL and issue key arguments for this macro require at least one Jira application link to be configured