Druid Integration

Version information

Druid integration is introduced in Hive 2.2.0 (HIVE-14217). Initially it was compatible with Druid 0.9.1.1, the latest stable release of Druid to that date.

Introduction

This page documents the work done for the integration between Druid and Hive, which was started in HIVE-14217.

Objectives

Our main **goal** is to be able to index data from Hive into Druid, and to be able to query Druid datasources from Hive. Completing this work will bring benefits to the Druid and Hive systems alike:

- *Efficient execution of OLAP queries in Hive.* Druid is a system specially well tailored towards the execution of OLAP queries on event data. Hive will be able to take advantage of its efficiency for the execution of this type of queries.
- *Introducing a SQL interface on top of Druid.* Druid queries are expressed in JSON, and Druid is queried through a REST API over HTTP. Once a user has declared a Hive table that is stored in Druid, we will be able to transparently generate Druid JSON queries from the input Hive SQL queries.
- *Being able to execute complex operations on Druid data.* There are multiple operations that Druid does not support natively yet, e.g. joins. Putting Hive on top of Druid will enable the execution of more complex queries on Druid data sources.
- *Indexing complex query results in Druid using Hive.* Currently, indexing in Druid is usually done through MapReduce jobs. We will enable Hive to index the results of a given query directly into Druid, e.g., as a new table or a materialized view (HIVE-10459), and start querying and using that dataset immediately.

The initial implementation, started in HIVE-14217, focused on 1) enabling the discovery of data that is already stored in Druid from Hive, and 2) being able to query that data, trying to make use of Druid advanced querying capabilities. For instance, we put special emphasis on pushing as much computation as possible to Druid, and being able to recognize the type of queries for which Druid is specially efficient, e.g. *timeseries* or *groupBy* queries.

Future work after the first step is completed is being listed in HIVE-14473. If you want to collaborate on this effort, a list of remaining issues can be found at the end of this document.

Preliminaries

Before going into further detail, we introduce some background that the reader needs to be aware of in order to understand this document.

Druid

Druid is an open-source analytics data store designed for business intelligence (OLAP) queries on event data. Druid provides low latency (real-time) data ingestion, flexible data exploration, and fast data aggregation. Existing Druid deployments have scaled to trillions of events and petabytes of data. Druid is most commonly used to power user-facing analytic applications. You can find more information about Druid here.

Storage Handlers

You can find an overview of Hive Storage Handlers here; the integration of Druid with Hive depends upon that framework.

Usage

For the running examples, we use the *wikiticker* dataset included in the quickstart tutorial of Druid.

Discovery and management of Druid datasources from Hive

First we focus on the discovery and management of Druid datasources from Hive.

Create tables linked to existing Druid datasources

Assume that we have already stored the *wikiticker* dataset mentioned previously in Druid, and the address of the Druid broker is *10.5.0.10:8082*.

First, you need to set the Hive property `hive.druid.broker.address.default` in your configuration to point to the broker address:

```
SET hive.druid.broker.address.default=10.5.0.10:8082;
```

Then, to create a table that we can query from Hive, we execute the following statement in Hive:

```
CREATE EXTERNAL TABLE druid_table_1
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
TBLPROPERTIES ("druid.datasource" = "wikiticker");
```

Observe that you need to specify the *datasource* as TBLPROPERTIES using the `druid.datasource` property. Further, observe that the table needs to be created as *EXTERNAL*, as data is stored in Druid. The table is just a logical entity that we will use to express our queries, but *there is no data movement when we create the table*. In fact, what happened under the hood when you execute that statement, is that Hive sends a *segment metadata* query to Druid in order to discover the schema (columns and their types) of the data source. Retrieval of other information that might be useful such as statistics e.g. number of rows, is in our roadmap, but it is not supported yet. Finally, note that if we change the Hive property value for the default broker address, queries on this table will automatically run against the new broker address, as the address is not stored with the table.

If we execute a *DESCRIBE* statement, we can actually see the information about the table:

```
hive> DESCRIBE FORMATTED druid_table_1;
OK
# col_name                  data_type                  comment
__time                      timestamp                  from deserializer
added                       bigint                     from deserializer
channel                     string                     from deserializer
cityname                    string                     from deserializer
comment                     string                     from deserializer
count                       bigint                     from deserializer
countryisocode              string                     from deserializer
countryname                 string                     from deserializer
deleted                     bigint                     from deserializer
delta                       bigint                     from deserializer
isanonymous                 string                     from deserializer
isminor                     string                     from deserializer
isnew                       string                     from deserializer
isrobot                     string                     from deserializer
isunpatrolled               string                     from deserializer
metrocode                   string                     from deserializer
namespace                   string                     from deserializer
page                        string                     from deserializer
regionisocode               string                     from deserializer
regionname                  string                     from deserializer
user                        string                     from deserializer
user_unique                 string                     from deserializer
# Detailed Table Information
Database:                   druid
Owner:                      user1
CreateTime:                 Thu Aug 18 19:09:10 BST 2016
LastAccessTime:             UNKNOWN
Retention:                  0
Location:                   hdfs:/tmp/user1/hive/warehouse/druid.db/druid_table_1
Table Type:                 EXTERNAL_TABLE
Table Parameters:
        COLUMN_STATS_ACCURATE       {\"BASIC_STATS\":\"true\"}
        EXTERNAL                    TRUE
        druid.datasource            wikiticker
        numFiles                    0
        numRows                     0
        rawDataSize                 0
        storage_handler             org.apache.hadoop.hive.druid.DruidStorageHandler
        totalSize                   0
        transient_lastDdlTime       1471543750
# Storage Information
SerDe Library:              org.apache.hadoop.hive.druid.serde.DruidSerDe
InputFormat:                null
OutputFormat:               null
Compressed:                 No
Num Buckets:                -1
Bucket Columns:             []
Sort Columns:               []
Storage Desc Params:
        serialization.format        1
Time taken: 0.111 seconds, Fetched: 55 row(s)
```

We can see there are three different groups of columns corresponding to the Druid categories: the **timestamp** column (`__time`) mandatory in Druid, the **di mension** columns (whose type is STRING), and the **metrics** columns (all the rest).

Create Druid datasources from Hive

If we want to manage the data in the Druid datasources from Hive, there are multiple possible scenarios.

For instance, we might want to create an empty table backed by Druid using a *CREATE TABLE* statement and then append and overwrite data using *INSE RT* and *INSERT OVERWRITE* Hive statements, respectively.

```
CREATE EXTERNAL TABLE druid_table_1
(`__time` TIMESTAMP, `dimension1` STRING, `dimension2` STRING, `metric1` INT, `metric2` FLOAT)
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler';
```

Another possible scenario is that our data is stored in Hive tables and we want to preprocess it and create Druid datasources from Hive to accelerate our SQL query workload. We can do that by executing a *Create Table As Select* (CTAS) statement. For example:

```
CREATE EXTERNAL TABLE druid_table_1
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
AS
<select `timecolumn` as `__time`, `dimension1`, `dimension2`, `metric1`, `metric2`....>;
```

Observe that we still create three different groups of columns corresponding to the Druid categories: the **timestamp** column (`__time`) mandatory in Druid, the **dimension** columns (whose type is STRING), and the **metrics** columns (all the rest).

In both statements, the column types (either specified statically for *CREATE TABLE* statements or inferred from the query result for *CTAS* statements) are used to infer the corresponding Druid column category.

Further, note that if we do not specify the value for the `druid.datasource` property, Hive automatically uses the fully qualified name of the table to create the corresponding datasource with the same name.

Version Info

Version 2.2.0: CREATE TABLE syntax when data is managed via hive.

```
CREATE TABLE druid_table_1
(`__time` TIMESTAMP, `dimension1` STRING, `dimension2` STRING, `metric1` INT, `metric2` FLOAT)
STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler';
```

NOTE - Before Hive 3.0.0, we do not use *EXTERNAL* tables and do not specify the value for the `druid.datasource` property.

For versions 3.0.0+, All Druid tables are EXTERNAL (HIVE-20085).

Druid kafka ingestion from Hive

Version Info

Integration with Druid Kafka Indexing Service is introduced in Hive 3.0.0 (HIVE-18976).

Druid Kafka Indexing Service supports exactly-once ingestion from Kafka topic by managing the creation and lifetime of Kafka indexing tasks. We can manage Druid Kafka Ingestion using Hive *CREATE TABLE* statement as shown below.

Druid Kafka Ingestion

```
CREATE EXTERNAL TABLE druid_kafka_table_1(`__time` timestamp,`dimension1` string, `dimension1` string,
`metric1` int, `metric2 double ....)
        STORED BY 'org.apache.hadoop.hive.druid.DruidStorageHandler'
        TBLPROPERTIES (
        "kafka.bootstrap.servers" = "localhost:9092",
        "kafka.topic" = "topic1",
        "druid.kafka.ingestion.useEarliestOffset" = "true",
        "druid.kafka.ingestion.maxRowsInMemory" = "5",
        "druid.kafka.ingestion.startDelay" = "PT1S",
        "druid.kafka.ingestion.period" = "PT1S",
        "druid.kafka.ingestion.consumer.retries" = "2"
        );
```

Observe that we specified kafka topic name and kafka bootstrap servers as part of the table properties. Other tunings for Druid Kafka Indexing Service can also be specified by prefixing them with *'druid.kafka.ingestion.' e.g.* to configure duration of druid ingestion tasks we can add *"druid.kafka.ingestion.taskDuration" = "PT60S"* as a table property.

Start/Stop/Reset Druid Kafka ingestion

We can Start/Stop/Reset druid kafka ingestion using sql statement shown below.

```
ALTER TABLE druid_kafka_test SET TBLPROPERTIES('druid.kafka.ingestion' = 'START');
ALTER TABLE druid_kafka_test SET TBLPROPERTIES('druid.kafka.ingestion' = 'STOP');
ALTER TABLE druid_kafka_test SET TBLPROPERTIES('druid.kafka.ingestion' = 'RESET');
```

Note: Reseting the ingestion will reset the kafka consumer offset maintained by druid to the next offset. The consumer offsets maintained by druid will be reset to either the earliest or latest offset depending on *druid.kafka.ingestion.useEarliestOffset*

table property. This can cause duplicate/missing events. We typically only need to reset kafka ingestion when messages in Kafka at the current consumer offsets are no longer available for consumption and therefore won't be ingested into Druid.

INSERT, INSERT OVERWRITE and DROP statements

Version Info

Version 2.2.0 : These statements are supported by Hive managed tables (not external) backed by Druid.

For versions 3.0.0+, All Druid tables are EXTERNAL (HIVE-20085) and these statements are supported for any table.

Querying Druid from Hive

Once we have created our first table stored in Druid using the `DruidStorageHandler`, we are ready to execute our queries against Druid.

When we express a query over a Druid table, Hive tries to *rewrite* the query to be executed efficiently by pushing as much computation as possible to Druid. This task is accomplished by the cost optimizer based in Apache Calcite, which identifies patterns in the plan and apply rules to rewrite the input query into a new equivalent query with (hopefully) more operations executed in Druid.

In particular, we implemented our extension to the optimizer in HIVE-14217, which builds upon the work initiated in CALCITE-1121, and extends its logic to identify more complex query patterns (*timeseries* queries), translate filters on the *time* dimension to Druid intervals, push limit into Druid *select* queries, etc.

Currently, we support the recognition of *timeseries*, *groupBy,* and s*elect* queries.

Once we have completed the optimization, the (sub)plan of operators that needs to be executed by Druid is translated into a valid Druid JSON query, and passed as a property to the Hive physical TableScan operator. The Druid query will be executed within the TableScan operator, which will generate the records out of the Druid query results.

We generate a single Hive split with the corresponding Druid query for *timeseries* and *groupBy*, from which we generate the records. Thus, the degree of parallelism is 1 in these cases. However, for simple *select* queries without limit (although they might still contain filters or projections), we partition the original query into *x* queries and generate one split for each of them, thus incrementing the degree of parallelism for these queries, which usually return a large number of results, to *x*.

Consider that depending on the query, it might not be possible to push any computation to Druid. However, *our contract is that the query should always be executed*. Thus, in those cases, Hive will send a *select* query to Druid, which basically will read all the segments from Druid, generate records, and then execute the rest of Hive operations on those records. This is also the approach that will be followed if the cost optimizer is disabled (***not recommended***).

Queries completely executed in Druid

We focus first on queries that can be pushed completely into Druid. In these cases, we end up with a simple plan consisting of a TableScan and a Fetch operator on top. Thus, there is no overhead related to launching containers for the execution.

Select queries

We start with the simplest type of Druid query: *select* queries. Basically, a *select* query will be equivalent to a scan operation on the data sources, although operations such as projection, filter, or limit can still be pushed into this type of query.

Consider the following query, a simple select query for 10 rows consisting of all the columns of the table:

```
SELECT * FROM druid_table_1 LIMIT 10;
```

The Hive plan for the query will be the following:

```
hive> EXPLAIN
    > SELECT * FROM druid_table_1 LIMIT 10;
OK
Plan optimized by CBO.
Stage-0
  Fetch Operator
    limit:-1
    Select Operator [SEL_1]
      Output:["_col0","_col1","_col2","_col3","_col4","_col5","_col6","_col7","_col8","_col9","_col10","
_col11","_col12","_col13","_col14","_col15","_col16","_col17","_col18","_col19","_col20","_col21"]
      TableScan [TS_0]
        Output:["__time","added","channel","cityname","comment","count","countryisocode","countryname","
deleted","delta","isanonymous","isminor","isnew","isrobot","isunpatrolled","metrocode","namespace","page","
regionisocode","regionname","user","user_unique"],properties:{"druid.query.json":"{\"queryType\":\"select\",\"
dataSource\":\"wikiticker\",\"descending\":\"false\",\"intervals\":[\"-146136543-09-08T08:22:17.096-00:01:15
/146140482-04-24T16:36:27.903+01:00\"],\"dimensions\":[\"channel\",\"cityname\",\"comment\",\"countryisocode\",
\"countryname\",\"isanonymous\",\"isminor\",\"isnew\",\"isrobot\",\"isunpatrolled\",\"metrocode\",\"namespace\",
\"page\",\"regionisocode\",\"regionname\",\"user\",\"user_unique\"],\"metrics\":[\"added\",\"count\",\"
deleted\",\"delta\"],\"pagingSpec\":{\"threshold\":10},\"context\":{\"druid.query.fetch\":true}}","druid.query.
type":"select"}
Time taken: 0.141 seconds, Fetched: 10 row(s)
```

Observe that the Druid query is in the properties attached to the TableScan. For readability, we format it properly:

```
{
  "queryType":"select",
  "dataSource":"wikiticker",
  "descending":"false",
  "intervals":["-146136543-09-08T08:22:17.096-00:01:15/146140482-04-24T16:36:27.903+01:00"],
  "dimensions":
    ["channel","cityname","comment","countryisocode",
     "countryname","isanonymous","isminor","isnew",
     "isrobot","isunpatrolled","metrocode","namespace",
     "page","regionisocode","regionname","user","user_unique"
    ],
  "metrics":["added","count","deleted","delta"],
  "pagingSpec":{"threshold":10}
}
```

Observe that we get to push the limit into the Druid query (`threshold`). Observe as well that as we do not specify a filter on the timestamp dimension for the data source, we generate an interval that covers the range (,+).

In Druid, the timestamp column plays a central role. In fact, Druid allows to filter on the time dimension using the `intervals` property for all those queries. This is very important, as the time intervals determine the nodes that store the Druid data. Thus, specifying a precise range minimizes the number of nodes hit by the broken for a certain query. Inspired by Druid PR-2880, we implemented the intervals extraction from the filter conditions in the logical plan of a query. For instance, consider the following query:

```
SELECT `__time`
FROM druid_table_1
WHERE `__time` >= '2010-01-01 00:00:00' AND `__time` <= '2011-01-01 00:00:00'
LIMIT 10;
```

The Druid query generated for the SQL query above is the following (we omit the plan, as it is a simple TableScan operator).

```
{
  "queryType":"select",
  "dataSource":"wikiticker",
  "descending":"false",
  "intervals":["2010-01-01T00:00:00.000Z/2011-01-01T00:00:00.001Z"],
  "dimensions":[],
  "metrics":[],
  "pagingSpec":{"threshold":10}
}
```

Observe that we infer correctly the interval for the specified dates, `2010-01-01T00:00:00.000Z/2011-01-01T00:00:00.001Z`, because in Druid the starting date of the interval is included, but the closing date is not. We also support recognition of multiple interval ranges, for instance in the following SQL query:

```
SELECT `__time`
FROM druid_table_1
WHERE (`__time` BETWEEN '2010-01-01 00:00:00' AND '2011-01-01 00:00:00')
    OR (`__time` BETWEEN '2012-01-01 00:00:00' AND '2013-01-01 00:00:00')
LIMIT 10;
```

Furthermore we can infer overlapping intervals too. Finally, the filters that are not specified on the time dimension will be translated into valid Druid filters and included within the query using the `filter` property.

Timeseries queries

Timeseries is one of the types of queries that Druid can execute very efficiently. The following SQL query translates directly into a Druid *timeseries* query:

```
-- GRANULARITY: MONTH
SELECT `floor_month`(`__time`), max(delta), sum(added)
FROM druid_table_1
GROUP BY `floor_month`(`__time`);
```

Basically, we group by a given time granularity and calculate the aggregation results for each resulting group. In particular, the `floor_month` function over the timestamp dimension `__time` represents the Druid month granularity format. Currently, we support `floor_year`, `floor_quarter`, `floor_month`, `floor_week`, `floor_day`, `floor_hour`, `floor_minute`, and `floor_second` granularities. In addition, we support two special types of granularities, `all` and `none`, which we describe below. We plan to extend our integration work to support other important Druid custom granularity constructs, such as *duration* and *period* granularities.

The Hive plan for the query will be the following:

```
hive> EXPLAIN
    > SELECT `floor_month`(`__time`), max(delta), sum(added)
    > FROM druid_table_1
    > GROUP BY `floor_month`(`__time`);
OK
Plan optimized by CBO.
Stage-0
  Fetch Operator
    limit:-1
    Select Operator [SEL_1]
      Output:["_col0","_col1","_col2"]
      TableScan [TS_0]
        Output:["__time","$f1","$f2"],
        properties:{"druid.query.json":"{\"queryType\":\"timeseries\",\"dataSource\":\"wikiticker\",\"
descending\":\"false\",\"granularity\":\"MONTH\",\"aggregations\":[{\"type\":\"longMax\",\"name\":\"$f1\",\"
fieldName\":\"delta\"},{\"type\":\"longSum\",\"name\":\"$f2\",\"fieldName\":\"added\"}],\"intervals\":[\"
-146136543-09-08T08:22:17.096-00:01:15/146140482-04-24T16:36:27.903+01:00\"]}","druid.query.type":"timeseries"}
Time taken: 0.116 seconds, Fetched: 10 row(s)
```

Observe that the Druid query is in the properties attached to the TableScan. For readability, we format it properly:

```
{
  "queryType":"timeseries",
  "dataSource":"wikiticker",
  "descending":"false",
  "granularity":"MONTH",
  "aggregations":[
    {"type":"longMax", "name":"$f1", "fieldName":"delta"},
    {"type":"longSum", "name":"$f2", "fieldName":"added"}
  ],
  "intervals":["-146136543-09-08T08:22:17.096-00:01:15/146140482-04-24T16:36:27.903+01:00"]
}
```

Observe that the granularity for the Druid query is `MONTH`.

One rather special case is `all` granularity, which we introduce by example below. Consider the following query:

```
-- GRANULARITY: ALL
SELECT max(delta), sum(added)
FROM druid_table_1;
```

As it will do an aggregation on the complete dataset, it translates into a *timeseries* query with granularity `all`. In particular, the equivalent Druid query attached to the TableScan operator is the following:

```
{
  "queryType":"timeseries",
  "dataSource":"wikiticker",
  "descending":"false",
  "granularity":"ALL",
  "aggregations":[
    {"type":"longMax", "name":"$f1", "fieldName":"delta"},
    {"type":"longSum", "name":"$f2", "fieldName":"added"}
  ],
  "intervals":["-146136543-09-08T08:22:17.096-00:01:15/146140482-04-24T16:36:27.903+01:00"]
}
```

GroupBy queries

The final type of queries we currently support is *groupBy*. This kind of query is more expressive than *timeseries* queries; however, they are less performant. Thus, we only fall back to *groupBy* queries when we cannot transform into *timeseries* queries.

For instance, the following SQL query will generate a Druid *groupBy* query:

```
SELECT max(delta), sum(added)
FROM druid_table_1
GROUP BY `channel`, `user`;
```

```
{
  "queryType":"groupBy",
  "dataSource":"wikiticker",
  "granularity":"ALL",
  "dimensions":["channel","user"],
  "aggregations":[
    {"type":"longMax","name":"$f2","fieldName":"delta"},
    {"type":"longSum","name":"$f3","fieldName":"added"}],
  "intervals":["-146136543-09-08T08:22:17.096-00:01:15/146140482-04-24T16:36:27.903+01:00"]
}
```

Queries across Druid and Hive

Finally, we provide an example of a query that runs across Druid and Hive. In particular, let us create a second table in Hive with some data:

```
CREATE TABLE hive_table_1 (col1 INT, col2 STRING);
INSERT INTO hive_table_1 VALUES(1, '#en.wikipedia');
```

Assume we want to execute the following query:

```
SELECT a.channel, b.col1
FROM
(
  SELECT `channel`, max(delta) as m, sum(added)
  FROM druid_table_1
  GROUP BY `channel`, `floor_year`(`__time`)
  ORDER BY m DESC
  LIMIT 1000
) a
JOIN
(
  SELECT col1, col2
  FROM hive_table_1
) b
ON a.channel = b.col2;
```

The query is a simple join on columns `channel` and `col2`. The subquery a is executed completely in Druid as a *groupBy* query. Then the results are joined in Hive with the results of results of subquery b. The query plan and execution in Tez is shown in the following:

```
hive> explain
    > SELECT a.channel, b.col1
    > FROM
    > (
    >   SELECT `channel`, max(delta) as m, sum(added)
    >   FROM druid_table_1
    >   GROUP BY `channel`, `floor_year`(`__time`)
    >   ORDER BY m DESC
    >   LIMIT 1000
    > ) a
    > JOIN
    > (
    >   SELECT col1, col2
    >   FROM hive_table_1
    > ) b
    > ON a.channel = b.col2;
OK
Plan optimized by CBO.
Vertex dependency in root stage
Map 2 <- Map 1 (BROADCAST_EDGE)
Stage-0
  Fetch Operator
    limit:-1
    Stage-1
      Map 2
      File Output Operator [FS_11]
        Select Operator [SEL_10] (rows=1 width=0)
          Output:["_col0","_col1"]
          Map Join Operator [MAPJOIN_16] (rows=1 width=0)
            Conds:RS_7._col0=SEL_6._col1(Inner),HybridGraceHashJoin:true,Output:["_col0","_col2"]
          <-Map 1 [BROADCAST_EDGE]
            BROADCAST [RS_7]
              PartitionCols:_col0
              Filter Operator [FIL_2] (rows=1 width=0)
                predicate:_col0 is not null
                Select Operator [SEL_1] (rows=1 width=0)
                  Output:["_col0"]
                  TableScan [TS_0] (rows=1 width=0)
                    druid@druid_table_1,druid_table_1,Tbl:PARTIAL,Col:NONE,Output:["channel"],properties:
{"druid.query.json":"{\"queryType\":\"groupBy\",\"dataSource\":\"wikiticker\",\"granularity\":\"all\",\"
dimensions\":[{\"type\":\"default\",\"dimension\":\"channel\"},{\"type\":\"extraction\",\"dimension\":\"
__time\",\"outputName\":\"floor_year\",\"extractionFn\":{\"type\":\"timeFormat\",\"format\":\"yyyy-MM-dd'T'HH:
mm:ss.SSS'Z'\",\"granularity\":\"year\",\"timeZone\":\"UTC\",\"locale\":\"en-US\"}}],\"limitSpec\":{\"type\":\"
default\",\"limit\":1000,\"columns\":[{\"dimension\":\"$f2\",\"direction\":\"descending\",\"dimensionOrder\":\"
numeric\"}]},\"aggregations\":[{\"type\":\"doubleMax\",\"name\":\"$f2\",\"fieldName\":\"delta\"},{\"type\":\"
doubleSum\",\"name\":\"$f3\",\"fieldName\":\"added\"}],\"intervals\":[\"1900-01-01T00:00:00.000/3000-01-01T00:
00:00.000\"]}","druid.query.type":"groupBy"}
          <-Select Operator [SEL_6] (rows=1 width=15)
              Output:["_col0","_col1"]
```

```
              Filter Operator [FIL_15] (rows=1 width=15)
                predicate:col2 is not null
                TableScan [TS_4] (rows=1 width=15)
                  druid@hive_table_1,hive_table_1,Tbl:COMPLETE,Col:NONE,Output:["col1","col2"]
Time taken: 0.924 seconds, Fetched: 31 row(s)
hive> SELECT a.channel, b.col1
    > FROM
    > (
    >   SELECT `channel`, max(delta) as m, sum(added)
    >   FROM druid_table_1
    >   GROUP BY `channel`, `floor_year`(`__time`)
    >   ORDER BY m DESC
    >   LIMIT 1000
    > ) a
    > JOIN
    > (
    >   SELECT col1, col2
    >   FROM hive_table_1
    > ) b
    > ON a.channel = b.col2;
Query ID = user1_20160818202329_e9a8b3e8-18d3-49c7-bfe0-99d38d2402d3
Total jobs = 1
Launching Job 1 out of 1
2016-08-18 20:23:30 Running Dag: dag_1471548210492_0001_1
2016-08-18 20:23:30 Starting to run new task attempt: attempt_1471548210492_0001_1_00_000000_0
Status: Running (Executing on YARN cluster with App id application_1471548210492_0001)
--------------------------------------------------------------------------------
        VERTICES      MODE        STATUS  TOTAL  COMPLETED  RUNNING  PENDING  FAILED  KILLED
--------------------------------------------------------------------------------
Map 1 .......... container     SUCCEEDED      1          1        0        0       0       0
Map 2 .......... container     SUCCEEDED      1          1        0        0       0       0
--------------------------------------------------------------------------------
VERTICES: 02/02  [==========================>>] 100%  ELAPSED TIME: 0.15 s
--------------------------------------------------------------------------------
2016-08-18 20:23:31 Completed running task attempt: attempt_1471548210492_0001_1_00_000000_0
OK
#en.wikipedia        1
Time taken: 1.835 seconds, Fetched: 2 row(s)
```

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