Lookup UDF Join in Pinot

dharakk@uber.com, balci@uber.com, yupeng@uber.com

This document is shared externally

Motivation

With ease of Presto-Pinot connector for querying and offline ingestion support, users are looking to build an analytical portfolio directly (without a presentation layer) on the Pinot-ecosystem which involves a fairly rich data model with a lot of entities and relationships. Because of this there are join use-cases emerging. These are currently being worked around via denormalizing data but concerns are developing on fast changing data and also increasing scale.

Most of the use cases are fact-to-dim join where the dimension table is a small offline table and fact is a realtime or an offline table The use case of join we are addressing in this doc is decoration (other use cases can be found here) where we need to decorate the result (before or after aggregation) from a dim table by joining on a primary key. An example of such decoration is as below:

```
Table factTable:
    -   string    uuid
    -   int       metric
    -   timestamp event_time
    -   string    status
```

```
Table dimTable:
    -   string uuid
    -   string name
    -   string country
```

```
SELECT
  f.uuid,
  d.name,
  d.country,
  abs(sum(m.metric)) as sum_metric
FROM
  factTable f join dimTable d on f.uuid = d.uuid
WHERE
  f.event_time > CAST(to_unixtime(date_trunc('day', now()))
AS BIGINT)
  AND f.status = 'OPEN'
GROUP BY
  1,
  2,
  3
ORDER BY
  2
```

Problem Analysis

In the usual case we would be solving this via broadcast joining approach where we would broadcast the dim table to all the servers of the fact table working on the query and add a join operation after filter operation in the execution tree (see the alternative approaches figure)

But if we look at the decoration use case without looking at the join query, we see that we are simply doing a lookup for a given key value from a data source that is different than the one being queried. This lookup can be done before or after aggregation. With this view of the problem we can also use UDFs to solve this problem.

For example the above query can be re-written using a UDF lookUp as below:

```
SELECT
  f.uuid,
  lookUp('dimTable', 'name', 'uuid', f.uuid) as name,
  lookUp('dimTable', 'country', 'uuid', f.uuid) as country,
  abs(sum(m.metric)) as sum_metric
FROM
  factTable f
WHERE
  f.event_time > CAST(to_unixtime(date_trunc('day', now())) AS
BIGINT)
  AND f.status = 'OPEN'
GROUP BY
  1,
  2,
  3
ORDER BY
  2
```

Architecture

Overview

As from the Problem Analysis we can see that if we model the problem as a UDF, it can be easily implemented in Pinot. For users to still use this functionality as a Join we can add a Join pushdown in Presto-Pinot connector and translate the join query to a one with UDF. Example as below:

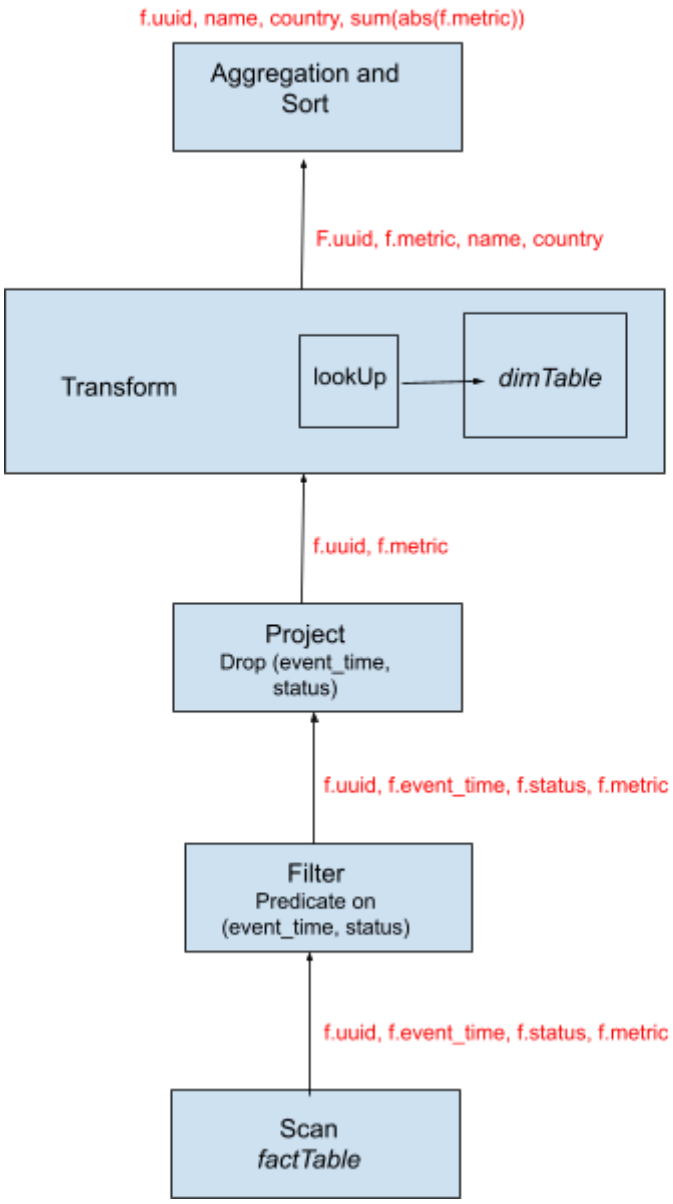
Query in Presto	Query to Pinot

`SELECT`
`  f.uuid,`
`  d.name,`
`  d.country,`
`  abs(sum(m.metric)) as sum_metric`
`FROM`
	`SELECT`
	`  f.uuid,`
	`  lookUp('dimTable', 'name', 'uuid', f.uuid)`
	`as name,`
	`  lookUp('dimTable', 'country', 'uuid',`
	`f.uuid) as country,`

```
  factTable f join dimTable d on f.uuid =
d.uuid
WHERE
  f.event_time >
CAST(to_unixtime(date_trunc('day', now())) AS
BIGINT)
  AND f.status = 'OPEN'
GROUP BY
  1,
  2,
  3
ORDER BY
  2
```

```
  abs(sum(m.metric)) as sum_metric
FROM
  factTable f
WHERE
  f.event_time >
CAST(to_unixtime(date_trunc('day', now())) AS
BIGINT)
  AND f.status = 'OPEN'
GROUP BY
  1,
  2,
  3
ORDER BY
  2
```

The overview of the operator tree for the example mentioned above will be as below:



The main aspects of this design are the lookUp transform function and recognizing a dim table and making it available locally to make the transform evaluation faster. Other than that there are no changes in query planning. More details on both these aspects as below.

lookUp UDF Signature

The lookUp UDF signature will be as below:

```
FieldSpec.DataType lookUp(string dimTableName, string dimColToLookUp, string dimJoinKey1,
DataType factJoinKeyVal1, string dimJoinKey2, .. )
```

UDF parameter description:

- **dimTableName**: Name of the dim table to perform the lookup on. *dimTable* in our example
- **dimColToLookUp**: The column name of the dim table to be retrieved to decorate our result. *name* and *country* in our example
- **dimJoinKey**: The column name on which we want to perform the lookup i.e. the join column name for dim table. *uuid* in our case. There can be multiple primary keys and corresponding values.
- **factJoinKeyVal:** The value of the dim table join column for which we will retrieve the *dimColToLookUp* for the scope and invocation.
- **return type:** Return type of the UDF will be that of the *dimColToLookUp* column type. IN our case *string* for both the invocations.

The UDF has two parts of implementation, one is shared which involves loading the segments from a dim table and then per row execution of looking up the column name.

We can achieve the shared part using a singleton class *DimTableManager,* which will help us load the dim table. This will help us achieve our goals without changing the transform function interface.

Here the disadvantage is that we are loading the dim table per segment as the plan tree is generated in Pinot for each segment. To improve this we can keep the dim table in an in-memory Cache of sorts which gets loaded once and then other segments just read from it.

If the *DimTableManager* finds necessary info in the cache then it will use it otherwise it will load information on it. Life cycle of this cache will be tied to the Query execution lifecycle so that we do not have to think about invalidation of the same.

Dimension table constraint and assignment

How we handle the dim table is at the core of this design. For a dim table, the following characteristics must be true for this design to work :

1. Dim table must be OFFLINE
2. Dim table should be small in size
3. Dim table should be available in all the server hosts

To enforce these requirements, we need a way to mark a table as a dim table. So for that we propose a new table config *isDimTable* to identify a dim table.

Once we mark a table as a dim table, the constraint (1) and (2) are easy to enforce by adding validation rules on quota and the table type.

For constraint (3) we propose a new assignment strategy for segments which essentially makes the table available on all the hosts. We can base this strategy off of the primitives already present for the *ReplicaGroupAssignmentStrategy*.

DimensionTableManager

DimTableManager will be implemented as an extension to OfflineTableManager and will host a static Map to store instances per table.

Each instance will load its table contents (all segments) into Java heap and will expose a ‘lookupRowByPrimaryKey’ method for the lookupUdf to fetch data from. In-heap cache will be re-built on every ‘addSegment’ call from HelixInstanceManager to make available the latest version of the table.

Alternative Design

Alternative for this design is to add Join clause support to Sql and add a new operator for Project which can handle a lookup from the dim table.

This is definitely a cleaner approach but this involves the overhead of implementing the planning of the join clause and optimizing it into the project operator mentioned above. And after this the physical aspects of the implementation are still the same as lookup.

Thus with this approach we are solidifying the physical aspects of a broadcast join first by making it available via a UDF. We can add a join pushdown in Presto connector and still use the functionality as a Join.

As future work we can also add the Join clause support to Pinot and reuse the lookup implementation.

Limitations

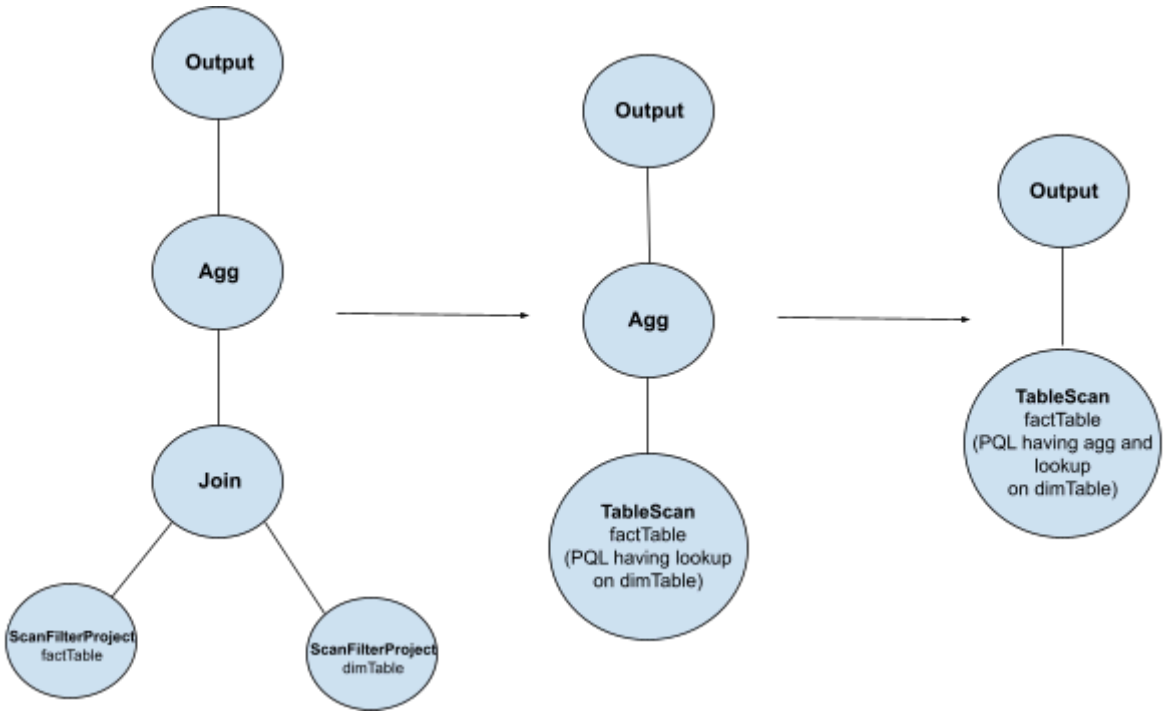
Following are the limitations of this approach:

- Subquery lookup from a fact table can not be done. Lookup has to be a dim-fact one
- There has to be a concept of primary key in the dim table for the lookup to work
- Dim table has to be small in size.

Presto Connector Changes

We will be using Presto as a syntax sugar for the join operation as the user will use a join clause in Presto and we will rewrite the query to use UDF in the Presto-Pinot connector. This work will involve mainly two major components:

1. Moving JoinNode to SPI framework to allow the new connector pushdown framework to push down Join operations as well.
2. Implement a simple join push-down logic where we push the TableScanNode of the dim table into the TableScanNode of the fact table. A visual illustration of this is as below:



Future Work

The major shortcoming of this approach is the size restrictions on the dim table because we require the dim table to be available on each host. Future work for this will involve lifting this restriction by implementing a broadcast join operator, which will enable larger and partitioned dim tables and help solve more generic join use cases.

Reference:
Join use cases
https://docs.google.com/document/d/1SFTbmH2Z86_bociZX8WIoVU7YQ4dnvWerguG8e2lszE/edit