SQL Tips for Hive and Presto

For tips specifically concerning Presto performance, refer to Presto Performance Tuning.

Use of PIVOT / UNPIVOT

You can use the **PIVOT** and **UNPIVOT** operators in standard SQL, Hive, and Presto.

The **PIVOT** operator transforms rows into columns. The **UNPIVOT** operator transforms columns into rows.

Example Tables

- Example of vertical table (vtable)

uid	key	value
101	c1	11
101	c2	12
101	c3	13
102	c1	21
102	c2	22
102	c3	23

- Example of horizontal table (htable)

uid	c1	c2	c3
101	11	12	13
102	21	22	23

Standard SQL

The following examples show how you can use an SQL syntax query language.

PIVOT

```
SELECT
  uid,
  kv['c1'] AS c1,
  kv['c2'] AS c2,
  kv['c3'] AS c3
FROM (
    SELECT
      uid,
      to_map(
        KEY,
        VALUE
      ) kv
    FROM
      vtable
    GROUP BY
      uid
  ) t
```

uid	c1	c2	c3
101	11	12	13
102	21	22	23

UNPIVOT

```
SELECT uid, 'c1' AS key, c1 AS value FROM htable
UNION ALL
SELECT uid, 'c2' AS key, c2 AS value FROM htable
UNION ALL
SELECT uid, 'c3' AS key, c3 AS value FROM htable
```

uid	key	value
101	c1	11
102	c1	21
101	c2	12
102	c2	22
101	c3	13
102	c3	23

Hive

The following examples show how you can use Hive.

PIVOT

Hive on Treasure Data supports `to_map` UDAF, which can generate Map type, and then transforms rows into columns. The general Hive function doesn't offer the same support.

```
SELECT
  uid,
  kv['c1'] AS c1,
  kv['c2'] AS c2,
  kv['c3'] AS c3
FROM (
  SELECT uid, to_map(key, value) kv
  FROM vtable
  GROUP BY uid
) t
```

uid	c1	c2	c3
101	11	12	13
102	21	22	23

UNPIVOT

`LATERAL VIEW explode` function transforms columns into rows.

```
SELECT
  t1.uid,
  t2.key,
  t2.value
FROM
  htable t1
LATERAL VIEW
  explode(
    MAP(
      'c1',
      c1,
      'c2',
      c2,
      'c3',
      c3
    )
  ) t2 AS KEY,
  VALUE
```

uid	key	value
101	c1	11
101	c2	12
101	c3	13
102	c1	21
102	c2	22
102	c3	23

Presto

The following examples show how you can use Presto.

PIVOT

This SQL transforms rows into columns by `map_agg` function.

```
SELECT
  uid,
  kv['c1'] AS c1,
  kv['c2'] AS c2,
  kv['c3'] AS c3
FROM (
    SELECT
      uid,
      map_agg(
        KEY,
        VALUE
      ) kv
    FROM
      vtable
    GROUP BY
      uid
  ) t
```

uid	c1	c2	c3
101	11	12	13
102	21	22	23

UNPIVOT

`CROSS JOIN unnest` function is similar to `LATERAL VIEW explode` function. It also transforms columns into rows.

```
SELECT
  t1.uid,
  t2.key,
  t2.value
FROM
  htable t1 CROSS
JOIN
  unnest(
    array['c1',
    'c2',
    'c3'],
    array[c1,
    c2,
    c3]
  ) t2(
    KEY,
    VALUE
  )
```

uid	key	value
101	c1	11
101	c2	12
101	c3	13
102	c1	21
102	c2	22
102	c3	23

Getting Data from Multiple Databases

From within in Hive and Presto, you can create a single query to obtain data from several databases or analyze data in different databases.

Example Databases and Tables

- Example of animals table in zoo_a database

id	name	sex
1	Lion	m
2	Gorilla	m
3	Zebra	f
4	Giraffe	f

- Example of emp animals in zoo_b database

animal_id	name	sex
101	Lion	m
102	Bear	f
103	Elephant	f
104	Gorilla	f
105	Tiger	m
106	Monkey	m
107	Rhinoceros	f

Use of Fully Qualified Table Names

If you use the fully qualified name of a table in the format `database.table`, you can query and process data from multiple databases.

```
SELECT
  name,
  sex
FROM
  zoo_a.animals
UNION
SELECT
name,
sex
FROM
zoo_b.animals
```

name	sex
Lion	m
Giraffe	f
Bear	f
Monkey	m
Gorilla	m
Gorilla	f
Elephant	f
Rhinoceros	f
Tiger	m
Zebra	f

More Presto Options

Consider the following Presto features when creating queries to run in Treasure Data.

Use the WITH Clause for Nested Queries

The WITH clause is useful for nested queries as shown in this example query:

```
SELECT
  a,
  b,
  c
FROM (
    SELECT
      a,
      MAX(b) AS b,
      MIN(c) AS c
    FROM
      tbl
    GROUP BY
      a
  ) tbl_alias
```

The same query can be written with the WITH clause as follows:

```
WITH tbl_alias AS(
  SELECT
    a,
    MAX(b) AS b,
    MIN(c) AS c
  FROM
    tbl
  GROUP BY
    a
) SELECT
  a,
  b,
  c
FROM
  tbl_alias
```

Notice that the query has been “de-nested”.

The following example shows multiple subqueries use WITH. Each subquery is delimited by a comma (,).

```
WITH tbl1 AS(
  SELECT
    a,
    MAX(b) AS b,
    MIN(c) AS c
  FROM
    tbl
  GROUP BY
    a
),
tbl2 AS(
  SELECT
    a,
    AVG(d) AS d
  FROM
    another_tbl
  GROUP BY
    a
) SELECT
  tbl1. * ,
  tbl2. *
FROM
  tbl1
JOIN
  tbl2
  ON tbl1.a = tbl2.a
```

Using VALUES for Prototyping

If you want to quickly test Presto syntax, you can use VALUES to create a table immediately.

```
SELECT
  a,
  b,
  c
FROM (
  VALUES(
    1,
    2.0,
    'x'
  ),
  (
    3,
    4.0,
    'y'
  ),
  (
    5,
    6.0,
    'z'
  )
  ) tbl(
    a,
    b,
    c
  )
```

The above query returns the following table:

a	b	c
1	2.0	'x'
3	4.0	'y'
5	6.0	'z'

Clean up a Table

To clean up a table before using CREATE TABLE AS or INSERT INTO statements, use multiple statements split by semi-colon (;):

```
DROP
  TABLE
    IF EXISTS mytable
;

CREATE
  TABLE
    mytable AS SELECT
      . . .
;
```

ALTER TABLE - DROP COLUMN

You can include the SQL DDL statement ALTER TABLE...DROP COLUMN SQL in your Treasure Data queries to, for example, deduplicate data.

An example is as follows:

```
DROP
  TABLE
    IF EXISTS task_temp
;

CREATE
  TABLE
    task_temp AS SELECT
       *
    FROM
      (
        SELECT
           * ,
          row_number(
          ) over(
            partition BY id
          ORDER BY
            TD_TIME_PARSE(lastmodifieddate) DESC
          ) AS rnk
        FROM
          task_raw
      )
    WHERE
      rnk = 1
      AND isdeleted = 0
    ORDER BY
      lastmodifieddate DESC
;

ALTER TABLE
  task_temp DROP
    COLUMN rnk
;
```

Limitation in Presto on Multiple Updates

Multiple-statement execution is not guarded by a transaction, therefore never write multiple update operations in a single job. For example, if you write two or more INSERT INTO statements in a single job, it may produce duplicated records:

```
# I1 INSERT
  INTO
    mytable SELECT
      . . .
;

# I2 INSERT
  INTO
    mytable SELECT
      . . .
;
```

If the system finds a sporadic failure during the job execution, it will start the job from the scratch, so the actual execution could be I1 (success), I2 (failure), (retry from the scratch), I1 (success) and I2 (success). In this case, I1 was executed twice and I2 was executed only once. The result is duplicated records in the target table.