Defining Partitioning for Presto

All tables in Treasure Data are partitioned based on the time column. This is why queries that use TD_TIME_RANGE or similar predicates on the time column are efficient in Treasure Data. Presto can eliminate partitions that fall outside the specified time range without reading them.

User-defined partitioning (UDP) provides hash partitioning for a table on one or more columns in addition to the time column. A query that filters on the set of columns used as user-defined partitioning keys can be more efficient because Presto can skip scanning partitions that have matching values on that set of columns.

Limitations

The benefits of UDP can be limited when used with more complex queries. The query optimizer might not always apply UDP in cases where it can be beneficial.

The Maximum Number of Partitioning Keys is Less Than or Equal to Three Keys

If the limit is exceeded, Presto causes the following error message:

'bucketed_on' must be less than 4 columns

Imports Other than SQL

The import method provided by Treasure Data for the following does not support UDP tables:

- Streaming Import
- Data Connector
- BulkImport

If you try to use any of these import methods, you will get an error. As a workaround, you can use a workflow to copy data from a table that is receiving streaming imports to the UDP table.

Use Cases for UDP

Where the lookup and aggregations are based on one or more specific columns, UDP can lead to:

- efficient lookup and aggregation queries
- efficient join queries

UDP can add the most value when records are filtered or joined frequently by non-time attributes::

- a customer's ID, first name+last name+birth date, gender, or other profile values or flags
- a product's SKU number, bar code, manufacturer, or other exact-match attributes
- an address's country code; city, state, or province; or postal code

Performance benefits become more significant on tables with >100M rows.

UDP can help with these Presto query types:

- "Needle-in-a-Haystack" lookup on the partition key
- Aggregations on the partition key
- Very large joins on partition keys used in tables on both sides of the join

Basic UDP Usage

CREATE TABLE Syntax for UDP

To create a UDP table:

1. Use CREATE TABLE with the attributes bucketed_on to identify the bucketing keys and bucket_count for the number of buckets.
2. Optionally, define the max_file_size and max_time_range values.

```
CREATE
  TABLE
    table_name WITH(
      bucketed_on = array['col1' [,
      'col2']... ] [,
      bucket_count = n] [,
      max_file_size = 'nnnMB'] [,
      max_time_range = 'nnd']
    ) [AS SELECT
      . . . ]
;
```

For bucket_count the default value is 512. This should work for most use cases.

For example:

```
CREATE
  TABLE mytable_p(
    time bigint,
    col1 VARCHAR,
    col2 VARCHAR,
    col3 bigint
  ) WITH (
    bucketed_on = array['col1' [,
    'col2']... ]
  )
;If not specified:
```

- max_file_size will default to 256MB partitions
- max_time_range to 1d or 24 hours for time partitioning
- bucket_count = 512

Choosing Bucketing Columns for UDP

Supported TD data types for UDP partition keys include int, long, and string. These correspond to Presto data types as described in About TD Primitive Data Types.

Choose a set of one or more columns used widely to select data for analysis-- that is, one frequently used to look up results, drill down to details, or aggregate data. For example, depending on the most frequently used types, you might choose:

- customer_id
- Country + State/Province + City
- Postal_code
- Customer-first name + last name + date of birth

Choose a column or set of columns that have high cardinality (relative to the number of buckets), and are frequently used with equality predicates. For example:

- Unique values, for example, an email address or account number
- Non-unique but high-cardinality columns with relatively even distribution, for example, date of birth

Checking For and Addressing Data Skew

The performance is inconsistent if the number of rows in each bucket is not roughly equal. For example, if you partition on the US zip code, urban postal codes will have more customers than rural ones.

To help determine bucket count and partition size, you can run a SQL query that identifies distinct key column combinations and counts their occurrences. For example:

```
SELECT
  (
    concat(key_column1 [,
      '|',
      key_column2])
  ),
  COUNT(*)
FROM
  tbl. . .
GROUP BY
  1
;
```

If the counts across different buckets are roughly comparable, your data is not skewed.

For consistent results, choose a combination of columns where the distribution is roughly equal.

If you do decide to use partitioning keys that do not produce an even distribution, see Improving Performance with Skewed Data.

Using INSERT and INSERT OVERWRITE to Partitioned Tables

INSERT and INSERT OVERWRITE with partitioned tables work the same as with other tables. You can create an empty UDP table and then insert data into it the usual way. The resulting data is partitioned.

Partitioning an Existing Table

Tables must have partitioning specified when first created. For an existing table, you must create a copy of the table with UDP options configured and copy the rows over. To do this use a CTAS from the source table.

For example:

```
drop table if EXISTS customer_p
;

create table customer_p with (
  bucketed_on = array['customer_id'],
  bucket_count = 512
)  as select  *  from  customer
```

When partitioning an existing table:

- Creating a partitioned version of a very large table is likely to take hours or days. Consult with TD support to make sure you can complete this operation.
- If the source table is continuing to receive updates, you must update it further with SQL. For example:

  ```
  INSERT
    INTO
      customer_p SELECT
         *
      FROM
        customer WHERE...
  ```

Creating and Using UDP Tables: Examples

Create a partitioned copy of the customer table named customer_p, to speed up lookups by customer_id;

```
drop table if EXISTS customer_p
;

create table customer_p with (
  bucketed_on = array['customer_id'],
  bucket_count = 512
)   as select  *  from  customer
```

Create and populate a partitioned table customers_p to speed up lookups on "city+state" columns:

```
-- create partitioned table, bucketing on combination of city + state columns
-- create table customer_p with (bucketed_on = array['city','state'] , bucket_count=512, max_file_size
= '256MB', max_time_range='30d');
create table customer_p with (
  bucketed_on = array['city',
  'state'],
  bucket_count = 512
)
;

-- update for beta
-- Insert new records into the partitioned table
INSERT
  INTO
    customer_p
  VALUES(
    . . . . .
  )
;

INSERT
  overwrite INTO
    customer_p
  VALUES(
    . . . .
  )
;

-- accelerates queries that test equality on all bucketing columns
SELECT
   *
FROM
  customer_p
WHERE
  city = 'San Francisco'
  AND state = 'California'
;

SELECT
   *
FROM
  customer_p
WHERE
  city IN(
    'San Francisco',
    'San Jose'
  )
  AND state = 'California'
  AND monthly_income > 10000
;

-- NOT accelerated: filter predicate does not use all hash columns
SELECT
   *
FROM
  customer_p
WHERE
  state = 'California'
;

SELECT
   *
FROM
  customer_p
WHERE
  city IN(
    'San Francisco',
    'San Jose'
  )
;
```

UDP Advanced Use Case Details

Choosing Bucket Count, Partition Size in Storage, and Time Ranges for Partitions

Bucket counts must be in powers of two. A higher bucket count means dividing data among many smaller partitions, which can be less efficient to scan. TD suggests starting with 512 for most cases. If you aren't sure of the best bucket count, it is safer to err on the low side.

We recommend partitioning UDP tables on one-day or multiple-day time ranges, instead of the one-hour partitions most commonly used in TD. Otherwise, you might incur higher costs and slower data access because too many small partitions have to be fetched from storage.

Aggregations on the Hash Key

Using a GROUP BY key as the bucketing key, major improvements in performance and reduction in cluster load on aggregation queries were seen. For example, you can see the UDP version of this query on a 1TB table:

- used 10 Presto workers instead of 19
- ran in 45 seconds instead of 2 minutes 31 seconds

processing >3x as many rows per second. The total data processed in GB was greater because the UDP version of the table occupied more storage.

UDP version:

```
presto: udp_tpch_sf1000 > SELECT
  COUNT(*)
FROM
  (
    SELECT
      max_by(
        l_discount,
        time
      ),
      max_by(
        l_partkey,
        time
      )
    FROM
      lineitem
    GROUP BY
      l_orderkey
  )
;

_col0------------
1500000000(
  1 row
) Query 20171227_014452_14154_sjh9g,
FINISHED,
10 nodes Splits: 517 total,
517 done(
  100.00 %
) 0: 45 [6B ROWS,
25.5GB] [134M ROWS / s,
585MB / s]
```

non-UDP:

```
presto: udp_tpch_sf1000 > SELECT
  COUNT(*)
FROM
  (
    SELECT
      max_by(
        l_discount,
        time
      ),
      max_by(
        l_partkey,
        time
      )
    FROM
      tpch_sf1000.lineitem
    GROUP BY
      l_orderkey
  )
;

_col0------------
1500000000(
  1 row
) Query 20171227_014549_14273_sjh9g,
FINISHED,
19 nodes Splits: 175 total,
175 done(
  100.00 %
) 2: 31 [6B ROWS,
18.3GB] [39.7M ROWS / s,
124MB / s]
```

Needle-in-a-Haystack Lookup on the Hash Key

The largest improvements – 5x, 10x, or more – will be on lookup or filter operations where the partition key columns are tested for equality. Only partitions in the bucket from hashing the partition keys are scanned.

For example, consider:

- table `customers` is bucketed on `customer_id`
- table `contacts` is bucketed on `country_code` and `area_code`

These queries will improve:

```
SELECT... FROM customers WHERE customer_id = 10001;
```

Here UDP Presto scans only one bucket (the one that 10001 hashes to) if customer_id is the only bucketing key.

```
SELECT... FROM contacts WHERE country_code='1' and area_code = '650' and phone like'555-____';
```

Here UDP Presto scans only the bucket that matches the hash of country_code 1 + area_code 650.

These queries will not improve:

```
SELECT... FROM customers WHERE customer_id >= 10001;
```

Here UDP will not improve performance, because the predicate doesn't use '='.

```
SELECT... FROM contacts WHERE area_code = '650' ;
```

Here UDP will not improve performance, because the predicate does not include both bucketing keys.

Very Large Join Operations

Very large join operations can sometimes run out of memory. Such joins can benefit from UDP. Distributed and colocated joins will use less memory, CPU, and shuffle less data among Presto workers. This may enable you to finish queries that would otherwise run out of resources. To leverage these benefits, you must:

1. Make sure the two tables to be joined are partitioned on the same keys
2. Use equijoin across all the partitioning keys
3. Set the following options on your join using a magic comment:

```
-- set session join_distritbution_type = 'PARTITIONED'
-- set session colocated_join = 'true'
```

Improving Performance on Skewed Data

When processing a UDP query, Presto ordinarily creates one split of filtering work per bucket (typically 512 splits, for 512 buckets). But if data is not evenly distributed, filtering on skewed bucket could make performance worse -- one Presto worker node will handle the filtering of that skewed set of partitions, and the whole query lags.

To enable higher scan parallelism you can use:

```
-- set session distributed_bucket='true|false'
```

When set to true, multiple splits are used to scan the files in a bucket in parallel, increasing performance. The tradeoff is that colocated join is always disabled when distributed_bucket is true. As a result, some operations such as GROUP BY will require shuffling and more memory during execution.

This query hint is most effective with needle-in-a-haystack queries. Even if these queries perform well with the query hint, test performance with and without the query hint in other use cases on those tables to find the best performance tradeoffs.