Cmp-divisioning

NOTE: The features described on this page are Technology Preview (Complete But Not Tested) features in the current Trafodion release.

Support for multi-temperature data

Description of Trafodion Blueprint cmp-divisioning

Databases that maintain current and historical data often suffer from the problem of decreasing performance as the amount of historical data grows. Most database systems have features that avoid this performance and scalability problem, typically by ensuring that the recent (or "hot") data is stored separately from the older ("cold") data.

The most common way to do that is range-partitioning: For each time interval (e.g. a week, month or year), a separate partition is created. This ensures that all the hot data is found in the most recent partition or, in some cases, the most recent two partitions. This type of range partitioning can be found in Oracle, Hive, DB2 and many other systems.

In Trafodion, we want to implement multi-temperature data in a slightly different way, by creating logical, not physical range partitions. We do this by adding the range partition number as a prefix to the key, so that hot data is stored together in a key range and that cold data is stored in a separate key range. This is somewhat similar to a Teradata Partitioned Primary Index (PPI). We expect that these logical range partitions will exist within each region, and that this is achieved by salting. The range partition number is a computed system column, meaning that it is not normally user-visible and that it is automatically maintained by Trafodion.

To illustrate this, here is an example:

Let’s say we are storing “call home” data. We have 100,000 devices and these devices send a status update every 10 seconds, resulting in 10,000 rows inserted into the table every second. The table has this DDL:

```
create table call_home_data(
  id largeint not null,
  ts timestamp(6) not null,
  device_status varchar(200),
  primary key (id, ts));
```

Using (id, ts) as the HBase row key, inserts would be randomly spread across the regions (assuming there is no skew in the devices) and hot and cold data would be mixed. To segregate hot and cold data, we could put the timestamp first:

```
create table call_home_data(
  id largeint not null,
  ts timestamp(6) not null,
  device_status varchar(200),
  primary key (ts, id));
```

However, this would create a hot spot in the last region, and querying a single device for a given time range would require scanning all the data for the other devices. The usual solution is to range-partition the table by time, for example one partition per day. The proposed Trafodion syntax will look like this:

```
create table call_home_data(
  id largeint not null,
  ts timestamp(6) not null,
  device_status varchar(200),
  primary key (id, ts))
division by (date_trunc('day', ts));
```

The feature is called “divisioning” rather than “partitioning” for historical reasons. It will create a new column, “_DIVISION_0_” that always has the day of the transaction timestamp. The HBase row key will be (“_DIVISION_0_”, id, ts). However, this re-introduces a hot spot on the last region, which would receive all the inserts into the table. Therefore we need to use salting:

```
create table call_home_data(
  id largeint not null,
  ts timestamp(6) not null,
  device_status varchar(200),
  primary key (id, ts))
salt using 16 partitions on (id)
division by (date_trunc('day', ts));
```

Now, our HBase row key will consist of four columns: (“_SALT_”, “_DIVISION_0_”, id, ts).

“_SALT_” is computed as a hash value of id and it will distribute the data evenly across 16 regions. “_DIVISION_0_” is the day of the transaction timestamp and it will keep the data for one day together in a compact key range for more efficient insert and select operations.
A typical query could look like this:

```sql
select *
from call_home_data
where id = 123456
  and ts between timestamp '2014-11-11 11:11:11.111111'
  and timestamp '2014-11-12 01:00:00.000000';
```

Trafodion will automatically generate predicates for the computed columns "_SALT_" and "DIVISION_0_". Since the query has an equals predicate on id, we can generate an equals predicate on "_SALT_" as well: "_SALT_" = HASH2PARTFUNC(123456 for 16). The range predicate on ts translates into a range predicate on "DIVISION_0_": "DIVISION_0_" between timestamp '2014-11-11 00:00:00.000000' and timestamp '2014-11-12 00:00:00.000000'. We can then use MDAM to access only those id and ts values that the query specifies.

Comparison of the four DDL versions:

<table>
<thead>
<tr>
<th>Property</th>
<th>DDL 1</th>
<th>DDL 2</th>
<th>DDL 3</th>
<th>DDL 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inserts are spread over all regions</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Balances data across regions</td>
<td>maybe</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Keeps hot and cold data separate</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Optimizes queries on item id</td>
<td>yes</td>
<td>no</td>
<td>MDAM</td>
<td>MDAM</td>
</tr>
<tr>
<td>Optimizes queries on time range</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>MDAM</td>
</tr>
<tr>
<td>Optimizes queries on both item and time</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Syntax for DIVISION BY:**

```
DIVISION BY (div_expr_list)
```

div_expr_list is a list of expressions that can only refer to primary key or STORE BY columns.

List of allowed expressions in the DIVISION BY clause (note that all these are monotonically increasing expressions):

- `cast((exact_numeric_column[ + const1 ] ) / const2 as numeric_data_type)`
- `DATE_PART('YEAR', date_part_arg )`
- `DATE_PART('YEARQUARTER', date_part_arg )`
- `DATE_PART('YEARMONTH', date_part_arg )`
- `DATE_PART('YEARWEEK', date_part_arg )`
- `DATE_TRUNC(const, datetime_col)`
- `DATEDIFF('YEAR', const, datetime_col)`
- `DATEDIFF('QUARTER', const, datetime_col)`
- `DATEDIFF('MONTH', const, datetime_col)`
- `DATEDIFF('WEEK', const, datetime_col)`
- `LEFT(character_col, const)`
- `SUBSTR(ING)(character_col, 1, const)`
- `SUBSTR(ING)(character_col FROM 1 FOR const)`

`date_part_arg` is one of the following:

- `datetime_col`
- `datetime_col + const`
- `datetime_col - const`
- `ADD_MONTHS(datetime_col, const [, 0])`
- `DATE_ADD(datetime_col, const)`
- `DATE_SUB(datetime_col, const)`