Enhanced Aggregation, Cube, Grouping and Rollup

This document describes enhanced aggregation features for the GROUP BY clause of SELECT statements.

Version

Grouping sets, CUBE and ROLLUP operators, and the GROUPING__ID function were added in Hive 0.10.0.
See HIVE-2397, HIVE-3433, HIVE-3471, and HIVE-3613.
Also see HIVE-3552 for an improvement added in Hive 0.11.0.

Version

GROUPING__ID is compliant with semantics in other SQL engines starting in Hive 2.3.0 (see HIVE-16102).
Support for SQL *grouping* function was added in Hive 2.3.0 too (see HIVE-15409).

For general information about GROUP BY, see GroupBy in the Language Manual.

GROUPING SETS clause

The GROUPING SETS clause in GROUP BY allows us to specify more than one GROUP BY option in the same record set. All GROUPING SET clauses can be logically expressed in terms of several GROUP BY queries connected by UNION. Table-1 shows several such equivalent statements. This is helpful in forming the idea of the GROUPING SETS clause. A blank set ( ) in the GROUPING SETS clause calculates the overall aggregate.

Table 1 - GROUPING SET queries and the equivalent GROUP BY queries

Aggregate Query with GROUPING SETS	Equivalent Aggregate Query with GROUP BY
SELECT a, b, SUM(c) FROM tab1 GROUP BY a, b GROUPING SETS ((a,b))	SELECT a, b, SUM(c) FROM tab1 GROUP BY a, b
SELECT a, b, SUM(c) FROM tab1 GROUP BY a, b GROUPING SETS ((a,b), a)	SELECT a, b, SUM(c) FROM tab1 GROUP BY a, b *UNION* SELECT a, null, SUM(c) FROM tab1 GROUP BY a
SELECT a,b, SUM(c) FROM tab1 GROUP BY a, b GROUPING SETS (a,b)	SELECT a, null, SUM(c) FROM tab1 GROUP BY a *UNION* SELECT null, b, SUM(c) FROM tab1 GROUP BY b
SELECT a, b, SUM(c) FROM tab1 GROUP BY a, b GROUPING SETS ((a, b), a, b, ( ))	SELECT a, b, SUM(c) FROM tab1 GROUP BY a, b *UNION* SELECT a, null, SUM(c) FROM tab1 GROUP BY a, null *UNION* SELECT null, b, SUM(c) FROM tab1 GROUP BY null, b *UNION* SELECT null, null, SUM(c) FROM tab1

Grouping__ID function

When aggregates are displayed for a column its value is null. This may conflict in case the column itself has some null values. There needs to be some way to identify NULL in column, which means aggregate and NULL in column, which means value. GROUPING__ID function is the solution to that.

This function returns a bitvector corresponding to whether each column is present or not. For each column, a value of "1" is produced for a row in the result set if that column has been aggregated in that row, otherwise the value is "0". This can be used to differentiate when there are nulls in the data.

Consider the following example:

Column1 (key)	Column2 (value)
1	NULL
1	1
2	2
3	3
3	NULL
4	5

The following query:

```
SELECT key, value, GROUPING__ID, count(*)
FROM T1
GROUP BY key, value WITH ROLLUP;
```

will have the following results:

Column 1 (key)	Column 2 (value)	GROUPING__ID	count(*)
NULL	NULL	3	6
1	NULL	1	2
1	NULL	0	1
1	1	0	1
2	NULL	1	1
2	2	0	1
3	NULL	1	2
3	NULL	0	1
3	3	0	1
4	NULL	1	1
4	5	0	1

Note that the third column is a bitvector of columns being selected.
For the first row, none of the columns are being selected.
For the second row, only the first column is being selected, which explains the value 1.
For the third row, both the columns are being selected (and the second column happens to be null), which explains the value 0.

Grouping function

The grouping function indicates whether an expression in a GROUP BY clause is aggregated or not for a given row. The value 0 represents a column that is part of the grouping set, while the value 1 represents a column that is not part of the grouping set.

Going back to our example above, consider the following query:

```
SELECT key, value, GROUPING__ID,
  grouping(key, value), grouping(value, key), grouping(key), grouping(value),
  count(*)
FROM T1
GROUP BY key, value WITH ROLLUP;
```

This query will produce the following results.

Column 1 (key)	Column 2 (value)	GROUPING__ID	grouping(key, value)	grouping(value, key)	grouping(key)	grouping(value)	count(*)
NULL	NULL	3	3	3	1	1	6
1	NULL	1	1	2	0	1	2
1	NULL	0	0	0	0	0	1
1	1	0	0	0	0	0	1
2	NULL	1	1	2	0	1	1
2	2	0	0	0	0	0	1
3	NULL	1	1	2	0	1	2
3	NULL	0	0	0	0	0	1
3	3	0	0	0	0	0	1
4	NULL	1	1	2	0	1	1
4	5	0	0	0	0	0	1

Cubes and Rollups

The general syntax is WITH CUBE/ROLLUP. It is used with the GROUP BY only. CUBE creates a subtotal of all possible combinations of the set of column in its argument. Once we compute a CUBE on a set of dimension, we can get answer to all possible aggregation questions on those dimensions.

It might be also worth mentioning here that
GROUP BY a, b, c WITH CUBE is equivalent to
GROUP BY a, b, c GROUPING SETS ((a, b, c), (a, b), (b, c), (a, c), (a), (b), (c), ( )).

ROLLUP clause is used with GROUP BY to compute the aggregate at the hierarchy levels of a dimension.
GROUP BY a, b, c with ROLLUP assumes that the hierarchy is "a" drilling down to "b" drilling down to "c".

GROUP BY a, b, c, WITH ROLLUP is equivalent to GROUP BY a, b, c GROUPING SETS ((a, b, c), (a, b), (a), ( )).

hive.new.job.grouping.set.cardinality

Whether a new map-reduce job should be launched for grouping sets/rollups/cubes.
For a query like: select a, b, c, count(1) from T group by a, b, c with rollup;
4 rows are created per row: (a, b, c), (a, b, null), (a, null, null), (null, null, null)
This can lead to explosion across map-reduce boundary if the cardinality of T is very high
and map-side aggregation does not do a very good job.

This parameter decides if hive should add an additional map-reduce job. If the grouping set
cardinality (4 in the example above), is more than this value, a new MR job is added under the
assumption that the orginal group by will reduce the data size.

Grouping__ID function (before Hive 2.3.0)

Grouping__ID function was fixed in Hive 2.3.0, thus behavior before that release is different (this is expected). For each column, the function would return a value of "0" iif that column has been aggregated in that row, otherwise the value is "1".

Hence the following query:

```
SELECT key, value, GROUPING__ID, count(*)
FROM T1
GROUP BY key, value WITH ROLLUP;
```

will have the following results.

Column 1 (key)	Column 2 (value)	GROUPING__ID	count(*)
NULL	NULL	0	6
1	NULL	1	2
1	NULL	3	1
1	1	3	1
2	NULL	1	1
2	2	3	1
3	NULL	1	2

3	NULL	3	1
3	3	3	1
4	NULL	1	1
4	5	3	1

Note that the third column is a bitvector of columns being selected.
For the first row, none of the columns are being selected.
For the second row, only the first column is being selected, which explains the count of 2.
For the third row, both the columns are being selected (and the second column happens to be null), which explains the count of 1.