LanguageManual Joins

Hive Joins

Join Syntax

Hive supports the following syntax for joining tables:

```
join_table:
    table_reference [INNER] JOIN table_factor [join_condition]
  | table_reference {LEFT|RIGHT|FULL} [OUTER] JOIN table_reference join_condition
  | table_reference LEFT SEMI JOIN table_reference join_condition
  | table_reference CROSS JOIN table_reference [join_condition] (as of Hive 0.10)

table_reference:
    table_factor
  | join_table

table_factor:
    tbl_name [alias]
  | table_subquery alias
  | ( table_references )

join_condition:
    ON expression
```

See Select Syntax for the context of this join syntax.

Version 0.13.0+: Implicit join notation

Implicit join notation is supported starting with Hive 0.13.0 (see HIVE-5558). This allows the FROM clause to join a comma-separated list of tables, omitting the JOIN keyword. For example:

```
SELECT *
FROM table1 t1, table2 t2, table3 t3
WHERE t1.id = t2.id AND t2.id = t3.id AND t1.zipcode = '02535';
```

Version 0.13.0+: Unqualified column references

Unqualified column references are supported in join conditions, starting with Hive 0.13.0 (see HIVE-6393). Hive attempts to resolve these against the inputs to a Join. If an unqualified column reference resolves to more than one table, Hive will flag it as an ambiguous reference.

For example:

```
CREATE TABLE a (k1 string, v1 string);
CREATE TABLE b (k2 string, v2 string);

SELECT k1, v1, k2, v2
FROM a JOIN b ON k1 = k2;
```

Version 2.2.0+: Complex expressions in ON clause

Complex expressions in ON clause are supported, starting with Hive 2.2.0 (see HIVE-15211, HIVE-15251). Prior to that, Hive did not support join conditions that are not equality conditions.

In particular, syntax for join conditions was restricted as follows:

```
join_condition:
    ON equality_expression ( AND equality_expression )*

equality_expression:
    expression = expression
```

Examples

Some salient points to consider when writing join queries are as follows:

- Complex join expressions are allowed e.g.

  ```
  SELECT a.* FROM a JOIN b ON (a.id = b.id)
  ```

  ```
  SELECT a.* FROM a JOIN b ON (a.id = b.id AND a.department = b.department)
  ```

  ```
  SELECT a.* FROM a LEFT OUTER JOIN b ON (a.id <> b.id)
  ```

  are valid joins.

- More than 2 tables can be joined in the same query e.g.

  ```
  SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
  ```

  is a valid join.

- Hive converts joins over multiple tables into a single map/reduce job if for every table the same column is used in the join clauses e.g.

  ```
  SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1)
  ```

  is converted into a single map/reduce job as only key1 column for b is involved in the join. On the other hand

  ```
  SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
  ```

  is converted into two map/reduce jobs because key1 column from b is used in the first join condition and key2 column from b is used in the second one. The first map/reduce job joins a with b and the results are then joined with c in the second map/reduce job.

- In every map/reduce stage of the join, the last table in the sequence is streamed through the reducers where as the others are buffered. Therefore, it helps to reduce the memory needed in the reducer for buffering the rows for a particular value of the join key by organizing the tables such that the largest tables appear last in the sequence. e.g. in

  ```
  SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key1)
  ```

  all the three tables are joined in a single map/reduce job and the values for a particular value of the key for tables a and b are buffered in the memory in the reducers. Then for each row retrieved from c, the join is computed with the buffered rows. Similarly for

  ```
  SELECT a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key = b.key2)
  ```

  there are two map/reduce jobs involved in computing the join. The first of these joins a with b and buffers the values of a while streaming the values of b in the reducers. The second of one of these jobs buffers the results of the first join while streaming the values of c through the reducers.

- In every map/reduce stage of the join, the table to be streamed can be specified via a hint. e.g. in

  ```
    SELECT /*+ STREAMTABLE(a) */ a.val, b.val, c.val FROM a JOIN b ON (a.key = b.key1) JOIN c ON (c.key =
  b.key1)
  ```

  all the three tables are joined in a single map/reduce job and the values for a particular value of the key for tables b and c are buffered in the memory in the reducers. Then for each row retrieved from a, the join is computed with the buffered rows. If the STREAMTABLE hint is omitted, Hive streams the rightmost table in the join.

- LEFT, RIGHT, and FULL OUTER joins exist in order to provide more control over ON clauses for which there is no match. For example, this query:

  ```
    SELECT a.val, b.val FROM a LEFT OUTER JOIN b ON (a.key=b.key)
  ```

  will return a row for every row in a. This output row will be a.val,b.val when there is a b.key that equals a.key, and the output row will be a.val, NULL when there is no corresponding b.key. Rows from b which have no corresponding a.key will be dropped. The syntax "FROM a LEFT OUTER JOIN b" must be written on one line in order to understand how it works--a is to the LEFT of b in this query, and so all rows from a are kept; a RIGHT OUTER JOIN will keep all rows from b, and a FULL OUTER JOIN will keep all rows from a and all rows from b. OUTER JOIN semantics should conform to standard SQL specs.

- Joins occur BEFORE WHERE CLAUSES. So, if you want to restrict the OUTPUT of a join, a requirement should be in the WHERE clause, otherwise it should be in the JOIN clause. A big point of confusion for this issue is partitioned tables:

  ```
    SELECT a.val, b.val FROM a LEFT OUTER JOIN b ON (a.key=b.key)
    WHERE a.ds='2009-07-07' AND b.ds='2009-07-07'
  ```

  will join a on b, producing a list of a.val and b.val. The WHERE clause, however, can also reference other columns of a and b that are in the output of the join, and then filter them out. However, whenever a row from the JOIN has found a key for a and no key for b, all of the columns of b will be NULL, **including the ds column**. This is to say, you will filter out all rows of join output for which there was no valid b.key, and thus you have outsmarted your LEFT OUTER requirement. In other words, the LEFT OUTER part of the join is irrelevant if you reference any column of b in the WHERE clause. Instead, when OUTER JOINing, use this syntax:

  ```
    SELECT a.val, b.val FROM a LEFT OUTER JOIN b
    ON (a.key=b.key AND b.ds='2009-07-07' AND a.ds='2009-07-07')
  ```

  ..the result is that the output of the join is pre-filtered, and you won't get post-filtering trouble for rows that have a valid a.key but no matching b. key. The same logic applies to RIGHT and FULL joins.

- Joins are NOT commutative! Joins are left-associative regardless of whether they are LEFT or RIGHT joins.

  ```
    SELECT a.val1, a.val2, b.val, c.val
    FROM a
    JOIN b ON (a.key = b.key)
    LEFT OUTER JOIN c ON (a.key = c.key)
  ```

  ...first joins a on b, throwing away everything in a or b that does not have a corresponding key in the other table. The reduced table is then joined on c. This provides unintuitive results if there is a key that exists in both a and c but not b: The whole row (including a.val1, a.val2, and a.key) is dropped in the "a JOIN b" step because it is not in b. The result does not have a.key in it, so when it is LEFT OUTER JOINed with c, c.val does not make it in because there is no c.key that matches an a.key (because that row from a was removed). Similarly, if this were a RIGHT OUTER JOIN (instead of LEFT), we would end up with an even weirder effect: NULL, NULL, NULL, c.val, because even though we specified a.key=c.key as the join key, we dropped all rows of a that did not match the first JOIN.
  To achieve the more intuitive effect, we should instead do FROM c LEFT OUTER JOIN a ON (c.key = a.key) LEFT OUTER JOIN b ON (c.key = b. key).

- LEFT SEMI JOIN implements the uncorrelated IN/EXISTS subquery semantics in an efficient way. As of Hive 0.13 the IN/NOT IN/EXISTS/NOT EXISTS operators are supported using subqueries so most of these JOINs don't have to be performed manually anymore. The restrictions of using LEFT SEMI JOIN are that the right-hand-side table should only be referenced in the join condition (ON-clause), but not in WHERE- or SELECT-clauses etc.

  ```
    SELECT a.key, a.value
    FROM a
    WHERE a.key in
     (SELECT b.key
      FROM B);
  ```

  can be rewritten to:

```
 SELECT a.key, a.val
 FROM a LEFT SEMI JOIN b ON (a.key = b.key)
```

- If all but one of the tables being joined are small, the join can be performed as a map only job. The query

  ```
  SELECT /*+ MAPJOIN(b) */ a.key, a.value
  FROM a JOIN b ON a.key = b.key
  ```

  does not need a reducer. For every mapper of A, B is read completely. The restriction is that **a FULL/RIGHT OUTER JOIN b** cannot be performed.

- If the tables being joined are bucketized on the join columns, and the number of buckets in one table is a multiple of the number of buckets in the other table, the buckets can be joined with each other. If table A has 4 buckets and table B has 4 buckets, the following join

  ```
  SELECT /*+ MAPJOIN(b) */ a.key, a.value
  FROM a JOIN b ON a.key = b.key
  ```

  can be done on the mapper only. Instead of fetching B completely for each mapper of A, only the required buckets are fetched. For the query above, the mapper processing bucket 1 for A will only fetch bucket 1 of B. It is not the default behavior, and is governed by the following parameter

  ```
  set hive.optimize.bucketmapjoin = true
  ```

- If the tables being joined are sorted and bucketized on the join columns, and they have the same number of buckets, a sort-merge join can be performed. The corresponding buckets are joined with each other at the mapper. If both A and B have 4 buckets,

  ```
  SELECT /*+ MAPJOIN(b) */ a.key, a.value
  FROM A a JOIN B b ON a.key = b.key
  ```

  can be done on the mapper only. The mapper for the bucket for A will traverse the corresponding bucket for B. This is not the default behavior, and the following parameters need to be set:

  ```
  set hive.input.format=org.apache.hadoop.hive.ql.io.BucketizedHiveInputFormat;
  set hive.optimize.bucketmapjoin = true;
  set hive.optimize.bucketmapjoin.sortedmerge = true;
  ```

MapJoin Restrictions

- If all but one of the tables being joined are small, the join can be performed as a map only job. The query

  ```
  SELECT /*+ MAPJOIN(b) */ a.key, a.value
  FROM a JOIN b ON a.key = b.key
  ```

  does not need a reducer. For every mapper of A, B is read completely.

- The following is not supported.
  - Union Followed by a MapJoin
  - Lateral View Followed by a MapJoin
  - Reduce Sink (Group By/Join/Sort By/Cluster By/Distribute By) Followed by MapJoin
  - MapJoin Followed by Union
  - MapJoin Followed by Join
  - MapJoin Followed by MapJoin

- The configuration variable hive.auto.convert.join (if set to true) automatically converts the joins to mapjoins at runtime if possible, and it should be used instead of the mapjoin hint. The mapjoin hint should only be used for the following query.
  - If all the inputs are bucketed or sorted, and the join should be converted to a bucketized map-side join or bucketized sort-merge join.

- Consider the possibility of multiple mapjoins on different keys:

```
select /*+MAPJOIN(smallTableTwo)*/ idOne, idTwo, value FROM
  ( select /*+MAPJOIN(smallTableOne)*/ idOne, idTwo, value FROM
    bigTable JOIN smallTableOne on (bigTable.idOne = smallTableOne.
idOne)
  ) firstjoin
  JOIN
  smallTableTwo ON (firstjoin.idTwo = smallTableTwo.idTwo)
```

The above query is not supported. Without the mapjoin hint, the above query would be executed as 2 map-only jobs. If the user knows in advance that the inputs are small enough to fit in memory, the following configurable parameters can be used to make sure that the query executes in a single map-reduce job.

- hive.auto.convert.join.noconditionaltask - Whether Hive enable the optimization about converting common join into mapjoin based on the input file size. If this paramater is on, and the sum of size for n-1 of the tables/partitions for a n-way join is smaller than the specified size, the join is directly converted to a mapjoin (there is no conditional task).
- hive.auto.convert.join.noconditionaltask.size - If hive.auto.convert.join.noconditionaltask is off, this parameter does not take affect. However, if it is on, and the sum of size for n-1 of the tables/partitions for a n-way join is smaller than this size, the join is directly converted to a mapjoin(there is no conditional task). The default is 10MB.

Join Optimization

Predicate Pushdown in Outer Joins

See Hive Outer Join Behavior for information about predicate pushdown in outer joins.

Enhancements in Hive Version 0.11

See Join Optimization for information about enhancements to join optimization introduced in Hive version 0.11.0. The use of hints is de-emphasized in the enhanced optimizations (HIVE-3784 and related JIRAs).