

ISIT312 Big Data Management

Hive Programming

Dr Janusz R. Getta

School of Computing and Information Technology -
University of Wollongong

Hive Programming

Outline

Data Selection and Scope

Data Manipulation

Data Aggregation and Sampling

Data Selection and Scope

To query data **Hive** provides **SELECT** statement

Typically **SELECT** statement projects the rows satisfying the query conditions specified in the **WHERE** clause and returns the result set

SELECT statement is usually used with **FROM**, **DISTINCT**, **WHERE**, and **LIMIT** keywords

```
SELECT C_NAME, C_PHONE
FROM customer
WHERE C_ACCTBAL > 0
LIMIT 2;
```

SELECT statement with LIMIT clause

Data Selection and Scope

Multiple **SELECT** statements can be combined into complex queries using **nested queries** or **subqueries**

Subqueries can use **Common Table Expressions (CTE)** in the format of **WITH** clause

When using **subqueries**, an **alias** should be given for the subquery

```
WITH cord AS ( SELECT *  
                FROM customer JOIN orders  
                ON c_custkey = o_custkey)  
SELECT c_name, c_phone, o_orderkey, o_orderstatus  
FROM cord;
```

WITH clause

Data Selection and Scope

Multiple **SELECT** statements can be combined into complex queries using **nested queries** or **subqueries**

Nested queries can use **SELECT** statement wherever a table is expected or a scalar value is expected

```
SELECT c_name, c_phone, o_orderkey, o_orderstatus
FROM ( SELECT *
      FROM customer JOIN orders
           ON c_custkey = o_custkey) cord
```

Nested query

Data Selection and Scope

When **inner join** is performed between multiple tables the **MapReduce** jobs are created to process data in **HDFS**

It is recommended to put the big table right at the end for better because the last table in the sequence is streamed through the reducers where the others are buffered in the reducer by default

```
SELECT /*+ STREAMTABLE(lineitem) */ c_name, o_orderkey, l_linenumber
FROM customer JOIN orders
      ON c_custkey = o_custkey
      JOIN lineitem
      ON l_orderkey = o_orderkey;
```

Inner join

Data Selection and Scope

Outer join (left, right, and full) and cross join preserve their HQL semantics

Map join means that join is computed only by map job without reduce job

In map join all data are read from a small table to memory and broadcasted to all maps

During map phase each row in from a big table is compared with the rows in small tables against join conditions

Join performance is improved because there is no reduce phase

Data Selection and Scope

```
SELECT /*+ MAPJOIN(orders) */ c_name, c_phone, o_orderkey, o_orderstatus
FROM customer JOIN orders
      ON c_custkey = o_custkey;
```

Map join

Hive automatically converts the `JOIN` to `MAPJOIN` at runtime when `hive.auto.convert.join` setting is set to `true`

`Bucket map join` is a special type of `MAPJOIN` that uses bucket columns in join condition.

Then instead of fetching the whole table `bucket map join` only fetches the required bucket data.

A variable `hive.optimize.bucketmapjoin` must be set to `true` to enable `bucket map join`

Data Selection and Scope

Hive supports **LEFT SEMI JOIN**

```
SELECT c_name, c_phone
FROM customer LEFT SEMI JOIN orders
      ON c_custkey = o_custkey;
```

Left semi join

In **LEFT SEMI JOIN** the right-hand side table should only be referenced in the join condition and not in **WHERE** or **SELECT** clauses

Data Selection and Scope

Hive supports **UNION ALL** it does not support **INTERSECT** and **MINUS** operations

```
SELECT p_name
FROM PART
UNION ALL
SELECT c_name
FROM CUSTOMER;
```

UNION ALL operation

INTERSECT operation can be implemented as **JOIN** operation

MINUS operation can be implemented as **LEFT OUTER JOIN** operation with **IS NULL** condition in **WHERE** clause

Hive Programming

Outline

[Data Selection and Scope](#)

[Data Manipulation](#)

[Data Aggregation and Sampling](#)

Data Manipulation

LOAD statement can be used to load data to **Hive** tables from local file system or from **HDFS**

Load data to **Hive** table from a local file

Loading data from a local file

```
LOAD DATA LOCAL INPATH '/local/home/janusz/HIVE-EXAMPLES/TPCHR/part.txt'  
OVERWRITE INTO TABLE part;
```

Load data to **Hive** partitioned table from a local file

Loading data into partitioned table from a local file

```
LOAD DATA LOCAL INPATH '/local/home/janusz/HIVE-EXAMPLES/TPCHR/part.txt'  
OVERWRITE INTO TABLE part PARTITION  
(P_BRAND='GoldenBolts');
```

LOCAL keyword determines a location of the input files

Data Manipulation

Load **HDFS** data to the **Hive** table using the default system path

```
LOAD DATA INPATH '/user/janusz/part.txt'  
OVERWRITE INTO TABLE part;
```

Loading data from HDFS

Load **HDFS** data to the **Hive** table using using full **URI**

```
LOAD DATA INPATH 'hdfs://10.9.28.14:8020/user/janusz/part.txt'  
OVERWRITE INTO TABLE part;
```

Loading data from HDFS

If **LOCAL** keyword is not specified, the files are loaded from the full **URI** specified after **INPATH** or the value from the **fs.default**

OVERWRITE keyword decides whether to append or replace the existing data in the target table/partition

Data Manipulation

EXPORT and **IMPORT** statements are available to support the import and export of data in HDFS for data migration or backup/restore purposes

EXPORT statement exports both data and metadata from a table or partition

```
EXPORT TABLE part TO '/user/tpchr/part'
```

Exporting table to HDFS

Metadata is exported to a file called `__metadata`

```
-rwxr-xr-x 3 janusz supergroup 2739 2017-07-09
14:37 /user/tpchr/part/__metadata
drwxr-xr-x - janusz supergroup 0 2017-07-09
14:37 /user/tpchr/part/p_brand=GoldenBolts
```

Contents of HDFS

After **EXPORT** the exported files can be copied to other **Hive** instances or to other **HDFS** clusters

Data Manipulation

IMPORT statement imports files exported from other **HIVE** instances into an internal table

```
IMPORT table new_part FROM '/user/tpchr/part';
```

HQL

An imported table is located in a default **HIVE** location in **HDFS**

```
drwxrwxr-x - janusz supergroup 0 2017-07-09  
14:56 /user/hive/warehouse/new_part
```

Importing data from HDFS

IMPORT EXTERNAL statement imports a file exported from other **HIVE** instances into an external table

```
IMPORT EXTERNAL table new_extpart FROM '/user/tpchr/  
part';
```

Importing external table from HDFS

An imported table is located in a default **HIVE** location in **HDFS**

```
drwxrwxr-x - janusz supergroup 0 2017-07-09  
15:04 /user/hive/warehouse/new_extpart
```

Contents of HDFS

Data Manipulation

ORDER BY sorts the results of **SELECT** statement

An order is maintained across all of the output from every reducer and global sort is performed using only one reducer

```
SELECT p_partkey, p_name
FROM part
ORDER BY p_name ASC;
```

ORDER BY clause

SORT BY does the same job as **ORDER BY** and indicates which columns to sort when ordering the reducer input records

SORT BY completes sorting before sending data to the reducer

SORT BY statement does not perform a global sort and only makes sure data is locally sorted in each reducer

```
SET mapred.reduce.tasks = 2;
SELECT p_partkey, p_name
FROM part
SORT BY p_name ASC;
```

SORT BY clause

Data Manipulation

When **DISTRIBUTE BY** clause is applied rows with matching column values are partitioned by the same reducer

```
SELECT p_partkey, p_name FROM part
DISTRIBUTE BY p_partkey
SORT BY p_name;
```

DISTRIBUTE BY clause

DISTRIBUTE BY clause is similar to **GROUP BY** in relational systems in terms of deciding which reducer is used to distribute the mapper

When using with **SORT BY**, **DISTRIBUTE BY** must be specified before the **SORT BY** statement

Data Manipulation

CLUSTER BY clause is a shorthand operator to perform **DISTRIBUTE BY** and **SORT BY** operations on the same group of columns.

```
SELECT p_partkey, p_name
FROM part
CLUSTER BY p_name;
```

CLUSTER BY clause

ORDER BY performs a global sort, while **CLUSTER BY** sorts in each distributed group

To fully utilize all the available reducers we can do **CLUSTER BY** first and then **ORDER BY**

```
SELECT p_partkey, p_name
FROM part
CLUSTER BY p_name
ORDER BY p_name;
```

CLUSTER BY clause

Hive Programming

Outline

[Data Selection and Scope](#)

[Data Manipulation](#)

[Data Aggregation and Sampling](#)

Data Aggregation and Sampling

Hive supports several aggregation functions, analytic functions working with **GROUP BY** and **PARTITION BY**, and windowing clauses

Hive supports advanced aggregation by using **GROUPING SETS**, **ROLLUP**, **CUBE**, analytic functions, and windowing

Basic aggregation uses **GROUP BY** clause and aggregation functions

```
SELECT p_type, count(*)  
FROM part  
GROUP BY p_type;
```

GROUP BY clause

To aggregate into sets a function **collect_set** can be used

```
SELECT p_type, collect_set(p_name), count(*)  
FROM part  
GROUP BY p_type;
```

GROUP BY clause with collect_set function

Data Aggregation and Sampling

GROUPING SETS clause implements advanced multiple **GROUP BY** operations against the same set of data

```
SELECT p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name
GROUPING SETS ( (p_type), (p_name) );
```

GROUPING SETS clause

ROLLUP clause allows to calculate multiple levels of aggregations across a specified group of dimensions

```
SELECT p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name WITH ROLLUP;
```

ROLLUP clause

CUBE clause allows to create aggregations over all possible subsets of attributes in a given set

```
SELECT p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name WITH CUBE;
```

CUBE clause

Data Aggregation and Sampling

GROUPING__ID function works as an extension to distinguish entire rows from each other

```
SELECT GROUPING__ID, p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name WITH CUBE
ORDER BY grouping__id;
```

GROUPING__ID function

HAVING can be used for the conditional filtering of **GROUP BY** results

```
SELECT GROUPING__ID, p_type, p_name, count(*)
FROM part
GROUP BY p_type, p_name WITH CUBE
HAVING count(*) > 1
ORDER BY grouping__id;
```

GROUPING__ID function

Data Aggregation and Sampling

Analytic functions scan multiple input rows to compute each output value

Analytic functions are usually used with `OVER`, `PARTITION BY`, `ORDER BY`, and the windowing specification

Analytic functions operate on windows where the input rows are ordered and grouped using flexible conditions expressed through an `OVER PARTITION` clause

Syntax is the following

```
function (arg1,..., argn)
OVER ([PARTITION BY <...>]
[ORDER BY <....>] [])
```

Syntax of analytic functions

For standard aggregation `function (arg1,..., argn)` can be either `COUNT()`, `SUM()`, `MIN()`, `MAX()`, or `AVG()`

Data Aggregation and Sampling

Typical aggregations implemented as analytic functions in the following way

```
SELECT p_name,  
       COUNT(*) OVER (PARTITION BY p_name)  
FROM PART;
```

PARTITION BY clause

Other analytic functions are used as follows

```
SELECT l_orderkey, l_partkey, l_quantity,  
       RANK() OVER (ORDER BY l_quantity),  
       DENSE_RANK() OVER (ORDER BY l_quantity)  
FROM lineitem;
```

ORDER BY clause

```
SELECT l_orderkey, l_partkey, l_quantity,  
       RANK() OVER (PARTITION BY l_orderkey ORDER BY l_quantity),  
       DENSE_RANK() OVER (PARTITION BY l_orderkey ORDER BY l_quantity)  
FROM lineitem;
```

PARTITION BY clause

Data Aggregation and Sampling

More analytic functions ...

PARTITION BY clause

```
SELECT l_orderkey, l_partkey, l_quantity,  
       FIRST_VALUE(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_quantity),  
       LAST_VALUE(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_quantity)  
FROM lineitem;;
```

PARTITION BY clause

```
SELECT l_orderkey, l_partkey, l_quantity,  
       MAX(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_partkey  
                            ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)  
FROM lineitem;
```

PARTITION BY clause

```
SELECT l_orderkey, l_partkey, l_quantity,  
       MAX(l_quantity) OVER (PARTITION BY l_orderkey ORDER BY l_partkey  
                            ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)  
FROM lineitem;
```

Data Aggregation and Sampling

When data volume is extra large we can use a subset of data to speed up data analysis.

Random sampling uses the `RAND()` function and `LIMIT` clause to get the samples of data

```
SELECT *  
FROM lineitem DISTRIBUTE BY RAND() SORT BY RAND() LIMIT 5;
```

DISTRIBUTE BY clause

`DISTRIBUTE` and `SORT` clauses are used here to make sure the data is also randomly and efficiently distributed among mappers and reducers

Bucket table sampling is a special sampling optimized for [bucket tables](#)

```
SELECT *  
FROM customer TABLESAMPLE(BUCKET 3 OUT OF 8 ON rand());
```

Bucket sampling

Data Aggregation and Sampling

Block sampling allows to randomly pick up **N** rows of data, percentage (**n** percentage) of data size, or **N** byte size of data

```
SELECT *  
FROM lineitem TABLESAMPLE(4 ROWS);
```

Block sampling

```
SELECT *  
FROM lineitem TABLESAMPLE(50 PERCENT);
```

Block sampling

```
SELECT *  
FROM lineitem TABLESAMPLE(20B);
```

Block sampling

References

Gross C., Gupta A., Shaw S., Vermeulen A. F., Kjerrumgaard D., Practical Hive: A guide to Hadoop's Data Warehouse System, Apress 2016, Chapter 4 (Available through UOW library)

Lee D., Instant Apache Hive essentials how-to: leverage your knowledge of SQL to easily write distributed data processing applications on Hadoop using Apache Hive, Packt Publishing Ltd. 2013 (Available through UOW library)

Apache Hive TM, <https://hive.apache.org/>