**Find the Hive version:** Type hive --version in command-line.

**SETTING UP THE ENVIRONMENT**

**Table users\_new:**

drop table users\_new;

create table users\_new

(

userid INT COMMENT 'USER ID',

Gender CHAR(1) COMMENT 'User Gender',

Age INT COMMENT 'User Age',

Occupation INT COMMENT 'USER OCCUPATION',

Zipcode INT COMMENT 'User Zip-code'

)

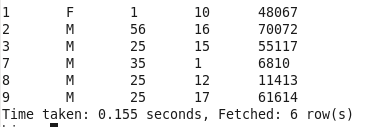
ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\;'

STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/input/users\_basicquery' INTO TABLE users\_new;

select \* from users\_new;



**Table movies\_new:**

drop table movies\_new;

create table movies\_new

(

movieid INT COMMENT 'Movie ID',

title STRING COMMENT 'Movie Title',

genres STRING COMMENT 'Movie Genre'

)

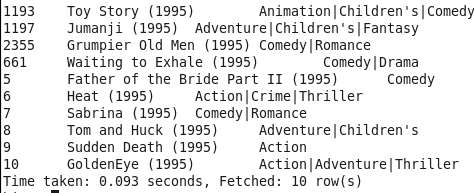
ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\;'

STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/input/movie\_basicquery' INTO TABLE movies\_new;

select \* from movies\_new;



**Table ratings\_new\_temp:**

drop table ratings\_new\_temp;

create table ratings\_new\_temp

(

userid INT COMMENT 'USER ID',

movieid INT COMMENT 'Movie ID',

rating INT COMMENT 'Rating',

timestamp\_rating Bigint COMMENT 'Rated time'

)

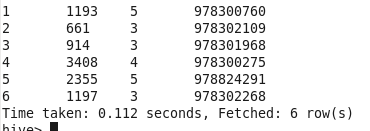
ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\;'

STORED AS TEXTFILE;

LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/input/rating\_basicquery' INTO TABLE ratings\_new\_temp;

select \* from ratings\_new\_temp;



**Table ratings\_new:**

**Table ratings\_new has a timestamp datatype which can contain value by converting a column value using from\_unixtime() method as shown below:**

drop table ratings\_new;

create table ratings\_new

(

userid INT COMMENT 'USER ID',

movieid INT COMMENT 'Movie ID',

rating INT COMMENT 'Rating',

timestamp\_rating Timestamp COMMENT 'Rated time'

)

ROW FORMAT DELIMITED

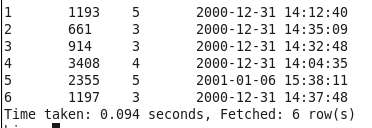
FIELDS TERMINATED BY '\;'

STORED AS TEXTFILE;

**Command to insert timestamp:**

**insert into table ratings\_new select userid, movieid, rating, from\_unixtime(timestamp\_rating) from ratings\_new\_temp;**

select \* from ratings\_new;



**Use GroupBy with aggregate functions:**

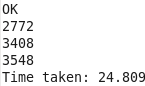
select timestamp\_rating, count(rating), sum(movieid) from ratings\_new;



select timestamp\_rating, sum(movieid) from ratings\_new group by rating;

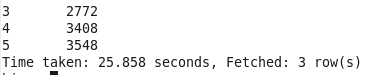


select sum(movieid) from ratings\_new group by rating;

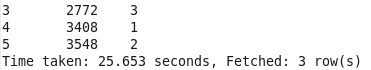


**Should be the same or receive error:**

select rating, sum(movieid) from ratings\_new group by rating;

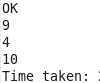


select rating, sum(movieid), count(rating) from ratings\_new group by rating;



**DISTINCT: Distinct keyword inside an aggregate function does the operation on columns specified. The columns values on which the operations is performed will be distinct.**

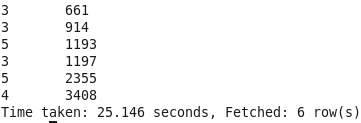
select sum(rating) from ratings\_new group by rating;



select sum(Distinct rating) from ratings\_new group by rating;



select movieid , count( DISTINCT rating), sum( DISTINCT userid) from ratings\_new group by rating;



select movieid , count( DISTINCT rating), sum( DISTINCT userid) from ratings\_new group by rating; #Illegal

**Limit Clause:**

SELECT \* FROM ratings\_new limit 50;

**Columns Alias:**

SELECT userid as UID, movieID as MID, rating as RAT, timestamp\_rating as TR from ratings\_new;

**Nested SELECT:**

SELECT column\_name FROM ( SELECT userid AS column\_name FROM ratings\_new ) users\_new;

SELECT column\_name FROM ( SELECT userid AS column\_name FROM ratings\_new UNION ALL SELECT userid AS column\_name FROM users\_new ) users\_new;

FROM ( SELECT movieid as MovieID, genres as Genre, title as Title from movies\_new ) e SELECT e.MovieID, e.Title WHERE e.Genre RLIKE '.\*Com\*.';

Supported by HIVE 0.13 and latest: (try these with HIVE 0.13)

SELECT \* FROM A WHERE A.a IN (SELECT foo FROM B);

SELECT A FROM T1 WHERE EXISTS (SELECT B FROM T2 WHERE T1.X = T2.Y);

There are a few limitations:

* These subqueries are only supported on the right-hand side of an expression.
* IN/NOT IN subqueries may only select a single column.
* EXISTS/NOT EXISTS must have one or more correlated predicates.
* References to the parent query are only supported in the WHERE clause of the subquery.

**Case ... When ... Else:**

SELECT userid, movieid,

CASE

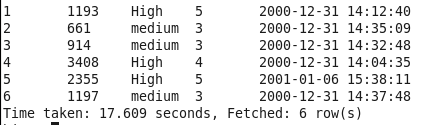
WHEN rating == 1 THEN 'low'

WHEN rating <= 3 THEN 'medium'

WHEN rating >= 5 OR rating == 4 THEN 'High'

ELSE 'very high'

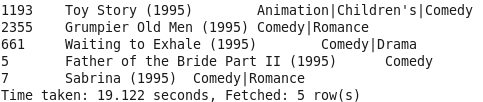
END AS word, rating, timestamp\_rating FROM ratings\_new;



**LIKE and RLIKE:**

LIKE matches the substring whereas RLIKE matches java expression.

**Returns rows that matches "Com" anywhere in genre:**

SELECT \* FROM movies\_new where genres like '%Com%'; 

**Returns rows that matches "dy" at the end of genre:**

SELECT \* FROM movies\_new where genres like '%dy';



**Returns rows that matches "dy" at the start of genre:**

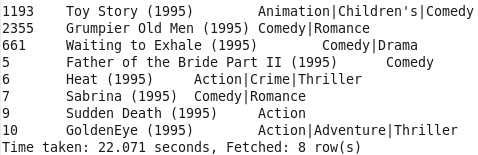
SELECT \* FROM movies\_new where genres like ' Ani%';



**Comedy and Drama:**

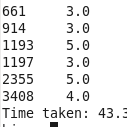
SELECT \* FROM movies\_new where genres RLIKE '.\*Comedy.\*Drama.\*'; 

**Comedy or Drama or Action:**

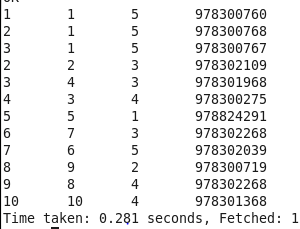
SELECT \* FROM movies\_new where genres RLIKE '.\*(Comedy|Drama|Action).\*'; 

**GroupBY:** GroupBy is mostly used in combination with aggregate functions, *controls how reducers receives rows for processing.*

SELECT movieid ,avg(rating) FROM ratings\_new GROUP BY movieid limit 50;



select \* from ratings\_decre;

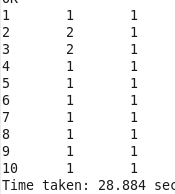


select count( DISTINCT rating), count( DISTINCT userid) from ratings\_decre;



select userid, count( DISTINCT rating), count( DISTINCT userid) from ratings\_decre; //Illegal

select userid, count( DISTINCT rating), count( DISTINCT userid) from ratings\_decre group by userid; //Should be same



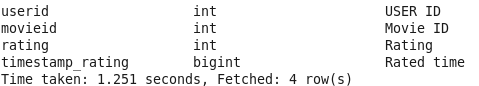
**Having:**

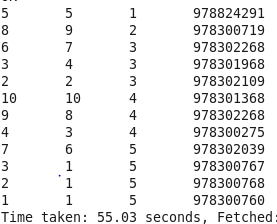
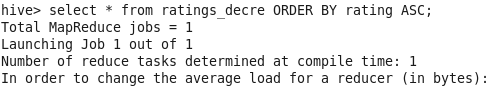
Having is mostly used to eliminate nested queries.

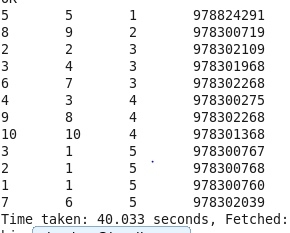
SELECT movieid ,avg(rating) FROM ratings\_new GROUP BY movieid HAVING avg(rating) >= 4 ;



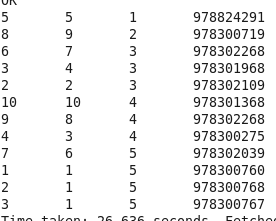
**ORDER BY:** ORDER BY performs a total ordering of the query result set. This means that all the data is passed through a single reducer, which may take an unacceptably long time to execute for larger data sets.



select \* from ratings\_decre ORDER BY rating ASC;

select \* from ratings\_decre ORDER BY rating ASC, movieid ASC;

select \* from ratings\_decre ORDER BY rating ASC, movieid DESC, USERID ASC ;

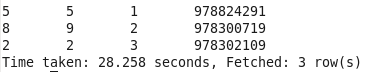


One can also set the optimization parameter as below. When hive.mapred.mode is set to strict ORDERBY should use LIMITBY Clause in the query.



SET hive.mapred.mode = strict;

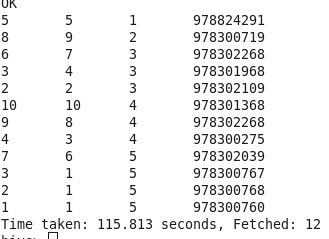
select \* from ratings\_decre ORDER BY rating ASC, movieid ASC LIMIT 10;



SET hive.mapred.mode = nonstrict;

**SORT BY**: SORT BY only guarantees ordering of the rows within a reducer. If there are more than one reducer, "sort by" may give partially ordered final results.

select \* from ratings\_decre SORT BY rating ASC;

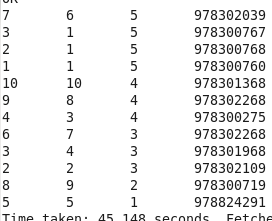


How the data will be sorted across multiple reducers while using SORT BY ?

**DISTRIBUTE BY: Distribute BY** controls how map output is divided among reducers. All data that flows through a MapReduce job is organized into key-value pairs. MapReduce computes a hash on the keys output by mappers and tries to evenly distribute the key-value pairs among the available reducers using the hash values. Unfortunately, this means that when we use SORT BY, the contents of one reducer’s output will overlap significantly with the output of the other reducers. **DISTRIBUTE BY ensures that the records ends in the same reducer based on a specific value in a column (for ex stock symbol) without over-lapping with the inputs of other reducers**, then use SORT BY to order the data the way we want.

ASC keywords could have been omitted as they are the defaults. DISTRIBUTE BY works similar to GROUP BY in the sense that it **controls how reducers receive rows for processing, while SORT BY controls the sorting of data inside the reducer**. Note that Hive requires that **the DISTRIBUTE BY clause come before the SORT BY clause.**

SELECT \* FROM ratings\_decre DISTRIBUTE BY userid SORT BY rating DESC;



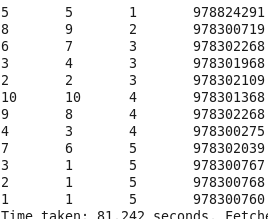
**Important Note:**

SELECT \* FROM ratings\_decre DISTRIBUTE BY userid Order BY rating DESC; Received an exception because distribute by the userid to number of reducers and order by is used for ordering entire data in one single reducer.

SELECT \* FROM ratings\_decre DISTRIBUTE BY userid SORT BY rating DESC; - works fine.

**Cluster BY** : The functionality of both Distribute By and sort by combined is denoted by Cluster BY. Cluster By delivers non-overlapping rows to reducers and also sort them. Distribute by columns and sort by column should be the same.

SELECT \* FROM ratings\_decre cluster BY rating;



SELECT \* FROM ratings\_decre cluster BY rating ASC; #Illegal - Not working

SELECT \* FROM ratings\_decre cluster BY rating DESC; #Illegal - Not Working

**Casting:**

cast(value AS TYPE) - The preferred way to convert float to integer is to use round() or floor()

SELECT userid,rating FROM ratings\_new WHERE cast(rating AS Float) >= 4.0;

SELECT userid,rating FROM ratings\_new WHERE cast(rating AS Int) >= cast(4.0 AS INT);

SELECT userid,rating FROM ratings\_new WHERE cast(rating AS Int) >= round(4.0);

SELECT userid,rating FROM ratings\_new WHERE cast(rating AS Int) >= floor(4.0);

**Hive V0.8 allows casting binary types into string. Therefore a binary type can be converted any data type by converting them into string first and use nested casting.**

SELECT (2.0\*cast(cast(b as string) as double)) from src; One can also cast STRING to BINARY

**Sampling Data:**

**Bucketing:** If partitioning the data results in large number of partitions one can use bucketing instead of sampling.

set hive.enforce.bucketing = true;

drop table ratings;

CREATE TABLE IF NOT EXISTS ratings (

userid INT COMMENT 'USER ID',

movieid INT COMMENT 'Movie ID',

rating INT COMMENT 'Rating',

timestamp TIMESTAMP COMMENT 'Rated time'

)

CLUSTERED BY(rating) INTO 3 BUCKETS

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\;'

STORED AS Textfile;

load data local inpath '/home/cloudera/Desktop/input/rating\_long' into table ratings;

Buckets on rating columns(should be a non-partitioned columns), ie) rating column is hashed on a function into 3(denominator) buckets(splits).

select count(\*) from ratings;

5001050

select count(\*) from ratings tablesample (bucket 1 out of 3 on rating);

1305990

select count(\*) from ratings tablesample (bucket 2 out of 3 on rating);

Receiving ArrayIndexOutofBounds Exception

set hive.enforce.bucketing = true;

drop table ratings\_less;

CREATE TABLE IF NOT EXISTS ratings\_less (

userid INT COMMENT 'USER ID',

movieid INT COMMENT 'Movie ID',

rating INT COMMENT 'Rating',

timestamp TIMESTAMP COMMENT 'Rated time'

)

CLUSTERED BY(rating) INTO 3 BUCKETS

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\;'

STORED AS Textfile;

load data local inpath '/home/cloudera/Desktop/input/rating\_join1' into table ratings\_less;

**rand()** indicating sampling on the entire row instead of an individual column. It returns random value, so the out may during each execution as the data is hashed on random number while selecting the same bucket. "s" is the alias name for the table.

SELECT \* from ratings\_less TABLESAMPLE(BUCKET 1 OUT OF 2 ON rand());

SELECT \* from ratings\_less TABLESAMPLE(BUCKET 2 OUT OF 2 ON rand());

SELECT s.movieid from ratings\_less TABLESAMPLE(BUCKET 1 OUT OF 5 ON rand()) s;

**Using Columns:** Returns the same result as the table is hashed on a specific column. The columns being hashed should be non-partitioned column.

SELECT \* from ratings\_less TABLESAMPLE(BUCKET 1 OUT OF 2 ON movieid);

SELECT \* from ratings\_less TABLESAMPLE(BUCKET 2 OUT OF 2 ON movieid);

MovieID should be a non-partitioned columns.

The TABLESAMPLE clause allows the users to write queries for samples of the data instead of the whole table. The TABLESAMPLE clause can be added to any table in the FROM clause. Sampling scans through the entire table and selects every Nth row. However, if the columns specified in the TABLESAMPLE clause match the columns in the CLUSTERED BY clause, TABLESAMPLE queries only scan the required hash partitions of the table.

SELECT \* FROM ratings\_less TABLESAMPLE(1 ROWS); //Returns 1 Row

select sum(movieid) from ratings\_less TABLESAMPLE(2 ROWS); //Adds the movieID of first two columns alone.

**Block Sampling:**

It will allow Hive to pick up at least n% data size. This type of sampling is not known to work with all file formats. CombineHiveInputFormat is supported. For tables less than the typical block size of 128 MB, all rows will be returned. Size of raring\_long is 215 MB.

SELECT count(\*) FROM ratings\_long;



SELECT count(\*) FROM ratings\_long TABLESAMPLE (0.1 PERCENT) s;

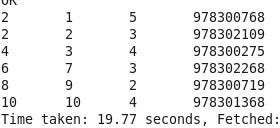


Sometimes you want to sample the same data with different blocks, you can change this seed number:

set hive.sample.seednumber = <INTEGER>; (sample from 2 different blocks if the seed number is 2)

**Input pruning:** One can also sample rows based on columns values.

SELECT \* FROM ratings\_decre WHERE userid % 2 = 0;

****

**Union All:** UNION ALL combines two or more tables. Each sub-query of the union query must produce the same number of columns, and for each column, its type must match all the column types in the same position. In SQL, UNION removes duplicate records, UNION ALL does not. Union is not working here in Hive.

drop table ratings1;

CREATE TABLE IF NOT EXISTS ratings1 (

userid INT COMMENT 'USER ID',

movieid INT COMMENT 'Movie ID',

rating INT COMMENT 'Rating',

timestamp TIMESTAMP COMMENT 'Rated time'

)

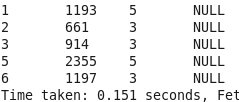
CLUSTERED BY(rating) INTO 3 BUCKETS

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\;'

STORED AS Textfile;

load data local inpath '/home/cloudera/Desktop/input/rating\_join1' into table ratings1;



drop table ratings2;

CREATE TABLE IF NOT EXISTS ratings2 (

userid INT COMMENT 'USER ID',

movieid INT COMMENT 'Movie ID',

rating INT COMMENT 'Rating',

timestamp TIMESTAMP COMMENT 'Rated time'

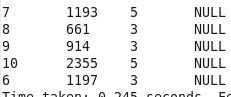
)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY '\;'

STORED AS Textfile;

load data local inpath '/home/cloudera/Desktop/input/rating\_join2' into table ratings2;



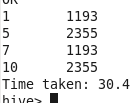
SELECT \* FROM (

SELECT r1.userid AS uid, r1.movieid AS mid FROM ratings1 r1 WHERE r1.rating >= 4

UNION ALL

SELECT r2.userid AS uid, r2.movieid AS mid FROM ratings2 r2 WHERE r2.rating >= 4

) combinedtable sort by combinedtable.uid ASC;



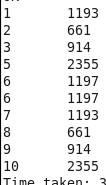
SELECT \* FROM (

SELECT r1.userid AS uid, r1.movieid AS mid FROM ratings1 r1

UNION ALL

SELECT r2.userid AS uid, r2.movieid AS mid FROM ratings2 r2

) combinedtable sort by combinedtable.uid ASC;



**Another format:**

This format increases the readability but makes multiple parses over the same table unless it is indexed.

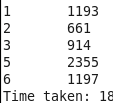
FROM (

FROM ratings1 SELECT ratings1.userid, ratings1.movieid WHERE ratings1.rating <= 4

UNION ALL

FROM ratings1 SELECT ratings1.userid, ratings1.movieid WHERE ratings1.rating > 4)

unioninput SELECT unioninput.\*;

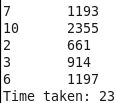


FROM

( FROM ratings1 SELECT ratings1.userid, ratings1.movieid WHERE ratings1.rating <= 4

UNION ALL

FROM ratings2 SELECT ratings2.userid, ratings2.movieid WHERE ratings2.rating > 4) unioninput SELECT unioninput.\*;



**MISC:**

select upper(name) from employees; // Display names in Upper case.

select lower(state) from employees; // Display names in Lower case.

select name,salary, ( salary - salary ) from employees; // Manipulation with columns names

SELECT explode(subordinates) AS sub FROM employees; //Explode with array columns

explode(ARRAY) - Returns one row for each element from the array.

explode(MAP) - Returns one row for each key-value pair from the input map with two columns in each row: one for the key and another for the value. (As of Hive 0.8.0.)

select key, value from employees lateral view explode(deductions) employees as key, value;

