ADVANCED HIVE

Optimizing Queries on Big Data:

- ✓ Design tables to optimize queries:
- ✓ Structure query so they run faster
- ✓ Simplify query expressions so they're easy to maintain.

- ➢ Table structure level optimization.
  - ■ Partitioning
  - ■ Bucketing

  That is, divide the data into smaller parts.
  40000 rows - scan all 40k records
  4 * (10000) rows - scan just 10k

- ➢ Optimizing the query that we are writing.
  - ■ Join optimization
- ➢ Simplify query expressions so that we can do it easily.
  - ■ Windowing function.

PARTITIONING:
split data into smaller subsets.
Separate records into manageable parts based on a column value.

Scenario:
Consider an e-commerce site with order data from across the united states
500mb data and a 4 node cluster, we can divide into 4 machines using default block size.
Query: select * from customers where state = "WA";
We can easily run on it.
But what if the data is very huge on each node ? And it is a frequent activity
We do partitioning, data can be logically segregated in this case based on states.

If US has 50 states then there will be 50 directories created using partition on state.
Data in customers table is logically divided into 50 smaller parts and each part will hold data for each state.
State specific queries will run only on data on one directory.

Managed table with customers table:
When there is no partitioning of data then the data will be directly inside customers folder.
/user/hive/warehouse/trendytech.db/customers

When partitioning wrt states:
/user/hive/warehouse/trendytech.db/customers/state=CA
/user/hive/warehouse/trendytech.db/customers/state=WA
/user/hive/warehouse/trendytech.db/customers/state=NY

Note: Partition should be based on the most common queries run.
Column should be based on select query.
Ex. Select * from customer where state = ? (10000 times in a day) ---------- Partition on this is okay.
Select * from customer where customer_type = ? (5 times in a day)

PARTITIONING TRADE-OFF (ISSUES):
Scenario:
Customer_id from 1 to 1 million.
Select * from customers where customer_id = ? (1000 times in a day)
That means we have lot of distinct values(or cardinality of the customer_id is very high)
If we try to do partitioning on such a column where cardinality is too high then we will get a lot of folders created.
Conclusion: if the cardinality of column is very high, then we should not go with partitioning on that column.
Solution is **bucketing** for this scenario.

- ✧ Large no. Of partitions:
  - Partition on customer_id on the orders table?
  - Millions of partitions, an HDFS dir for each partition.
  - Huge overhead for the NameNode in Hadoop.
  - May optimize some queries but be detrimental for others.
- ✧ Small no. Of partitions:
  - Partition on product quantity on the orders table?
  - Vary few partitions, few logical grouping in our data
  - No real optimizations on queries run.
  - No reduction in the data scanned by queries.

---

2 types of Partitioning:
- ✓ **Static -**
  - in static partitioning we should have an idea of our data and,
  - We should be loading each partition manually
  - This is a tedious process.
  - Managing partitions manually is not a scalable process.
  - Static partitioning is faster than dynamic

- ✓ **Dynamic -**
  - Partitions are created automatically, without any manual loading.
  - When we do not know the data in advance, then we go with dynamic partitioning.
  - Slower than static

BUCKETING tables in Hive:
Bucketing scenario: e-comm selling millions of products
Select * from product_table where product_id =?
In HDFS:
Each partition is a folder.
Each bucket is a file.

Note: In case of bucketing, we have to define a fixed number of buckets during table creation beforehand.
Trial and error works in this case, depends on data and experience. Divide based on some random function for ex. Modular function.
Consider 3 buckets that is 3 files(0-2) and product_id(1 - 8)
1%3=1
2%3=2
3%3=0
.
.
8%3=2
Knowing while file each record is stored in makes certain operations very fast.

Bucketing vs Partitioning:

- Partitioning on a column such as product_id can result in millions of directories.
- Hive **restricts** the maximum no of partitions allowed to avoid overwhelming the NameNode.
- A default number(may be 1,000) is set to ensure that if number of partitions exceed that number then hive will throw error.
- Solution: fixed number of buckets and an easy way to know which bucket holds a record.

Partitioning	Bucketing
Unknown number of partitions	Fixed number of buckets
Based on actual column values	Based on a hash value of a column
Each partition stored as a directory/folder	Each bucket stored as files under a directory
Partitions can very a lot in size	Buckets are almost the same size
Optimizes queries which retrieve or scan data in the logical group	Optimizes queries such as lookup, joins, sampling.
Provides a logical division of data.	Divides the data based on hash values.

When we have to give a sample data to someone?
So bucket can be a good sample of a sample data.
For ex. 10 buckets.
If someone ask to give 10% sample data ---> give 1 bucket
If someone ask to give 20% sample data ---> give 2 buckets

PARTITIONED + BUCKETED TABLES - COMBINATION OF BOTH:
A same table can be both partitioned and bucketed.
Ex. Customer table.
We can **partition** on **state** column
We can **bucket** this on **customer_id** column (4 buckets)

Directory structure
/user/hive/warehouse/trendytech.db/customers/state=CA
    Bucket_0_file 4, 8, 12
    Bucket_1_file 1, 5, 9
    Bucket_2_file 2, 6, 10, 14
    Bucket_3_file 3, 7, 11, 15
/user/hive/warehouse/trendytech.db/customers/state=WA
    Bucket_0_file
    Bucket_1_file
    Bucket_2_file
    Bucket_3_file
/user/hive/warehouse/trendytech.db/customers/state=CT
    Bucket_0_file (only this file will be scanned)
    Bucket_1_file
    Bucket_2_file
    Bucket_3_file

Select * from customers where state = CT and customer_id = 1012
That is CT dir and 1012%4 = 0 that means Bucket_0_file

Note: partitioning + bucketing = right
    Bucketing + partitioning = x (this is not right)

Advantages of bucketing:

- **Faster query response-** given a record, it is possible to figure out where exactly the record is stored.
- **Join optimization -** faster joins.

Practical:

Both P&B tries to optimize by dividing the data into manageable parts or chunks.
And finally the intention is to just scan one chunk of data and ignore rest of them.

Partitioning:

- Partitioning enables performance optimizations
- Split data into smaller subsets, separating records into manageable parts based on a column value.
- Data splits may not be of the same size per partition.
- Each of these units will be stored in a different directory.

Static partition:

Create table with partition:

```
create table orders_w_partition(
id string,
customer_id string,
product_id string,
quantity int,
amount double,
zipcode char(5))
partitioned by (state char(2))
row format delimited fields terminated by ',';
```

Load data:

```
load data local inpath
'/home/cloudera/Desktop/shared/week5_datasets/order_ca.csv'
into table orders_w_partition
partition (state="CA");
```

Dynamic partition:

Setting Hive properties to allow Dynamic partitioning: Note: by default DP is disabled.

```
SET hive.exec.dynamic.partition=true;
SET hive.exec.dynamic.partition.mode=nonstrict;
```

Generally DP is a 3 step process.
Step 1: create a normal table without any partition and have the data loaded in it. Table1
Step 2: create the partitioned table with partition on a column name. Table2
Step 3: transfer the data from normal table to partitioned table. Transfer from table1 to table2.

Step 1

Create a non partitioned table:

```
create table orders_no_partition(
id string,
customer_id string,
product_id string,
quantity int,
amount double,
zipcode char(5),
state char(2)
)
row format delimited fields terminated by ',';
```

Load data into non partitioned table: CA, CT, NY

```
load data local inpath
```

'/home/cloudera/Desktop/shared/week5_datasets/orders_CA_with_state.csv'
into table orders_no_partition;

Dir structure in HDFS:
hadoop fs -ls /user/hive/warehouse/trendytech.db/orders_no_partition

-rwxrwxrwx 1 cloudera supergroup 120 2022-01-23 17:07 /user/hive/warehouse/trendytech.db/orders_no_partition/**orders_CA_with_state.csv**
-rwxrwxrwx 1 cloudera supergroup 144 2022-01-23 17:07 /user/hive/warehouse/trendytech.db/orders_no_partition/**orders_CT_with_state.csv**
-rwxrwxrwx 1 cloudera supergroup 152 2022-01-23 17:08 /user/hive/warehouse/trendytech.db/orders_no_partition/**orders_NY_with_state.csv**

Step 2

Create a dynamic partition table:

```
create table orders_d_w_partition(
id string,
customer_id string,
product_id string,
quantity int,
amount double,
zipcode char(5))
partitioned by (state char(2))
row format delimited fields terminated by ',';
```

Step 3

Load the data from non partition table to the partitioned table:

```
insert into table orders_d_w_partition
partition (state)
select * from orders_no_partition;
```

Note: It is invoking a map reduce job.

3 distinct values ie CT, CA & NY so 3 partitions automatically

Number of partitions:
show partitions orders_d_w_partition;

Dir structure in HDFS:

hadoop fs -ls /user/hive/warehouse/trendytech.db/orders_d_w_partition

drwxrwxrwx - cloudera supergroup 0 2022-01-23 17:17 /user/hive/warehouse/trendytech.db/orders_d_w_partition/**state=CA**
drwxrwxrwx - cloudera supergroup 0 2022-01-23 17:17 /user/hive/warehouse/trendytech.db/orders_d_w_partition/**state=CT**
drwxrwxrwx - cloudera supergroup 0 2022-01-23 17:17 /user/hive/warehouse/trendytech.db/orders_d_w_partition/**state=NY**

Hive Bucketing Practical:

We should not use partitioning when the cardinality of column is high.
System will take care of hash function.

- Split data into smaller subsets, separating records into manageable parts by applying an hash function(automatically system defined-no need to worry about it) based on column value.

- Each bucket is a separate file under the table dir(if the table has no partitions) or under the table partition dir.
- A common hash function for integer column values is the modulo(%) operator.

Step 1: create a normal table with some data
Step 2: create a bucketed table on some column
Step 3: loading data from normal table to bucketed table.

Step 1
Create a normal hive table(w/o buckets):

```
create table products_no_buckets(
id int,
name string,
cost double,
category string
)
row format delimited fields terminated by',';
```

Load data into normal table:

```
load data local inpath
'/home/cloudera/Desktop/shared/week5_datasets/newproducts.csv'
into table products_no_buckets;
```

Step 2
Set the bucketing property in Hive:

```
SET hive.enforce.bucketing=true;
```

Create a Hive table with buckets:

```
create table products_w_buckets(
id int,
name string,
cost double,
category string
)
CLUSTERED BY (id) INTO 4 BUCKETS;
```

Step 3
Load data from non bucketed table into bucketed table:

```
FROM products_no_buckets
insert into table products_w_buckets
select id, name, cost, category;
```

Mappers = 1
Reducers = 4
Note: no. Of buckets = no. Of reducers

Dir structure in HDFS:

```
hadoop fs -ls /user/hive/warehouse/trendytech.db/products_w_buckets
```

-rwxrwxrwx 1 cloudera supergroup 23 2022-01-23 18:02 /user/hive/warehouse/trendytech.db/products_w_buckets/**000000_0**
-rwxrwxrwx 1 cloudera supergroup 71 2022-01-23 18:02 /user/hive/warehouse/trendytech.db/products_w_buckets/**000001_0**
-rwxrwxrwx 1 cloudera supergroup 60 2022-01-23 18:02 /user/hive/warehouse/trendytech.db/products_w_buckets/**000002_0**

-rwxrwxrwx 1 cloudera supergroup 71 2022-01-23 18:02 /user/hive/warehouse/trendytech.db/products_w_buckets/**000003_0**

Display records from bucketed table in bucket by bucket in Hive:

```
select * from products_w_buckets
TABLESAMPLE(bucket 1 out of 4);
```

Display bucketed table folder structure in hdfs using GUI;

quickstart.cloudera:50070/explorer.html#/user/hive/warehouse/trendytech.db/products_w_buckets

---

Hive partitioning with 2 columns practical:

Scenario:
Country=India
- State = UP
- state = Delhi

Dynamic partitioning with 2 columns:
Step 1:
Create normal table:

```
create table orders_no_partition1(
id string,
customer_id string,
product_id string,
quantity int,
amount double,
zipcode char(5),
country char(2),
state char(2)
)
row format delimited fields terminated by',';
```

Load data into normal table:

```
load data local inpath
'/home/cloudera/Desktop/shared/week5_datasets/orders_country_w_states.csv'
into table orders_no_partition1;
```

Step 2
Create a partition table based on 2 columns:

```
create table all_orders(
id string,
customer_id string,
product_id string,
quantity int,
amount double,
postalcode string
)
partitioned by (country string, state string)     <-------- (country->state) order cant be reversed.
row format delimited fields terminated by',';
```

Load data into partitioned table:

```
insert into table all_orders
partition (country, state)
select * from orders_no_partition1;
```

It will invoke a mapreduce and will take time because of dynamic partition.

Show partitions all_orders;
Output:

```
OK
country=UK/state=JI
country=UK/state=JJ
country=US/state=CA
country=US/state=TX
Time taken: 0.46 seconds, Fetched: 4 row(s)
```

PARTITIONING WITH BUCKETING: [order: partition ->bucketing]

Each partition is a folder and Each bucket is a file
Scenario: customer table
We can have **state** as the **partition** column and on **customer_id** we are **bucketing** into 4 buckets
State = CA
4 files <---- this one file is scanned only
State = NY
4 files
State = NJ
4 files
Query: select * from customers where state = CA and customer_id = 4?
In above scenario, we skipped 11 and only scanned 1

- Hive partition can be subdivided into buckets.
- Each partition will be created as directory
- But each bucket will be created as a file

Step 1: create a normal table with some data (products_no_buckets)
Step 2: create a partitioned & bucketed table on some column (products_partitioned_buckets)
Step 3: loading data from normal table to bucketed table.

Step2

```
create table products_partitioned_buckets(
id int,
name string,
cost double
)
partitioned by (category string)
clustered by (id) into 4 buckets
row format delimited fields terminated by',';
```

Set partitioning & bucketing properties in hive:

```
SET hive.exec.dynamic.partition=true;
SET hive.exec.dynamic.partition.mode=nonstrict;
SET hive.enforce.bucketing=true;
```

Step3
Load data into the table:

```
insert into table products_partitioned_buckets
partition (category)
select id, name, cost, category
from products_no_buckets;
```

Display results in hive:

```
select * from products_partitioned_buckets
TABLESAMPLE(bucket 1 out of 4);
```

Hive Join Optimization Theory

3 categories of optimization

- Table structure level optimization [partitioning & bucketing]
- Query level optimization [Join Optimization]
- Simplifying the query expressions [windowing functions]

Hive Join Optimizations

3 important kind of join optimizations:

- ✓ Map side join [very imp]
- ✓ Bucket map join
- ✓ SMB (sort merge bucket join)

Join Optimization

- Try to minimize the number of join columns if possible. Coz if we minimize the join columns then there will be less mapreduce jobs.
- Bucketing & Partitioning helps you to optimize the joins. For this your tables should be bucketed & partitioned on the joins columns. (we want to minimize the dataset to scan)
- By converting the joins to be Map-side joins whenever possible.

Select * from Names join Trades on Names.symbol = Trades.symbol
Or
Select * from Names join Trades on (Names.**symbol** = Trades.symbol) join Revenues on (Names.**symbol** = Revenues.symbol)
Here, join column = 1 ie symbol = 1 MR job

No of MapReduce jobs = No of Join columns

Select * from Names join Trades on (Names.**symbol** = Trades.symbol) join Revenues on (Names.**name** = Trades.name)
Here, mapreduce job = no of join column = 2

Note:

- ✧ Internally even a join is executed as a map reduce job.
- ✧ Also, mapper give o/p as (key,value) pair. So,
  - **Key - the join column**
  - **Value - all the remaining column**
- ✧ The reducer combines all the columns which have the same key.

Table1 (product_id, all the other columns list1 3) Table2 (id, all the other columns list2 5)	---> Reducer --->	(id, list1 + list2 8 columns)

- ✧ We have 2 stages for this map reduce job
  - Map (faster than reduce) - shuffling & sorting
  - Reduce (heavy, slow) - aggregation
- ✧ Certain queries/joins can be structured to have no reduce phase

Map-side joins - A join where mapper can give you the final output and no reducer is required. The entire joining activity will happen at the mapper end.

- Better performance
- Improves processing time
- Reduces data transfer b/w machines in the cluster.
- Reduces operations such as shuffle and sort b/w map and reduce phases
- Whenever feasible, we'd like joins to be map-side joins

Conditions for a map-side joins

- Except one table all the other tables should be small.
- If we are talking about 2 tables. Then one of them should be small enough to fit in memory.

Each machine/node holds full small table + parts of larger table.

Different kind of JOINs:

- **INNER JOIN (Natural Join) -** shows only the matching records from both tables.
- **LEFT OUTER JOIN** - shows all the matching records + show all the non matching records from the left table with nulls on the right side.
- **RIGHT OUTER JOIN -** shows all the matching records + show all the non matching records from the right table with nulls on the left side.
- **FULL OUTER JOIN -** union of left outer + right outer

Question: Can a inner join be treated as a Map Side join provided the left table is small enough to fit in memory and the right table is large.?
Ans: **Yes**, when the left table is smaller table then inner joins can be treated as map-side joins

Small table: Names & Large table: Trades
Parts of the larger table are distributed to each mapper.
Mappers run on entire Names table and parts of Trades table.
Combine those rows which are available on the mapper.
The output of all the mappers forms the final output.
No reducer needed.

Question: Can a left outer join be treated as a Map Side join provided the left table is small enough to fit in memory and the right table is large.?
Ans: **No**, when the left table is smaller table then left outer join cannot be treated as map-side joins

Coz there's no way to tell if the unmatched record is not present in this one chunk or not present in the entire table.

Question: Can a right outer join be treated as a Map Side join provided the left table is small enough to fit in memory and the right table is large.?
Ans: Yes, when left table is small

Question: full outer join as map-side join?
Ans: No, as left outer is not working so full outer will not work also.

<u>When Left table is small:</u>

- ✓ **Inner** join **can** be map-side join
- ✓ **Left outer** join **cannot** be map-side join
- ✓ **Right outer** join **can** be map-side join
- ✓ **Full outer** join **cannot** be map-side join

<u>When Right table is small:</u>

- ✓ **Inne**r join **can** be map-side join
- ✓ **Left outer** join **can** be map-side join
- ✓ **Right outer** join **cannot** be map-side join
- ✓ **Full outer** join **cannot** be map-side join

Bucket Map Join & SMB (Sort Merge Bucket Join)

<u>Bucket Map Join</u>

- This can work on 2 big tables also.

- Both the tables should be bucketed on join columns. (Ie department_id)
- The number of buckets in one table should be an integral multiple of number of buckets in other table.

That is 2 2,4,6,8
3 3,6,9,12

Table 1	Table 2
Department_id	Department_id
4	8

Map Side Join:

- Before the mapreduce job is executed, there is local task which is executed.
- This local task will take the small table and create a hashtable for this small table.
- Once the hashtable is created it will be put on hdfs.
- From hdfs this hashtable is broadcasted to all the nodes.
- When this hashtable is copied on the nodes, it will be existing in the local disk of each node. This also is called as distributed cache.
- Hashtable is loaded in memory in each of the node. So from local disk on each node it is loaded in memory.

After these steps, then only mapreduce job starts.
Constraint is entire small table has to be loaded into the memory. That's why it can't work on large tables.

Bucket Map Join:

Table1 4 buckets
Table2 8 buckets

In table 1 - bucket 0 corresponds to
Bucket 0 and 4 in table 2

Table 1	Table 2
4 buckets	8 buckets
0 - 4,8,12,16,20 1 - 1,5,9,13,17 2 - 2,6,10,14 3 - 3,7,11,15,19	0 - 8,16,24,32 1 - 1,9,17,25 2 - 2,10,19,27 3 - 3,11,19,27 4 - 4,12,20,28 . 8

Before running the mapreduce job it will execute a local task.

- A hashtable is created and then it is pushed to HDFS.
- From HDFS this hashtable is broadcasted to all the nodes.
- This will reside on local disk on each machine called as distributed caching.
- **Only 1 bucket has to be loaded into the memory and not the whole data.**
- This will make sure your memory utilization is not going high.

TT notes BMJ:

- Unlike Map side join it can be done on 2 big tables also.
- Both the tables should be bucketed on join column.
- Number of buckets in one table should be integral multiple of other table.
- Only the required buckets are loaded in memory that's the advantage over map side join.

SMB (Sort Merge bucket join): can work on 2 big tables.

Constraints:

- Both the tables should be bucketed on join column.
- Number of buckets in one table should exactly match number of buckets in other table.
- Both the tables should be sorted based on join column.

One-one mapping b/w tables:

- B1 in T1 will be joined easily with B1 in T2.
- There is no memory constraint and we donot have to hold anything in memory
- Mapper1 will work on B1 of T1 and B1 of T2.
- Mapper2 will work on B2 of T1 and B2 of T2

Table1	Table2
1	1
2	2
3	3
4	4

TT Notes:

- Unlike map side join it can be done on **2 big table** also.
- Both the tables should be sorted on join column.
- Both the tables should be bucketed on join column.
- **Number of buckets** in both tables should be exactly **equal**.
- There will be a one to one mapping b/w buckets in both tables.
- And a quick joining can be performed as both the buckets have sorted data.

Question: can we always have a join as map side join?
Ans: not always, however if certain criterias match then we can go for map side join.

We want to join 2 tables
=> out of 2 tables, one table should be smaller so that it can fit in memory of datanode.
If both are small, then no need for hadoop, traditional systems can be used.
If both are big, then we cannot use map-side join.

We want to join 3 or more tables
=> All the tables except one should be smaller to fit in memory.

Practical: Map-side join
Scenario:
Orders & customers table, one is small and other is big.
In mysql in cloudera, there is db called as retail_db. It has orders and customers table.
We need to get the data from these 2 tables which are available in mysql and we want to import it to hdfs.

Steps:
1. Sqoop import to get orders and customers data in hdfs.

```
sqoop import \
--connect
jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--warehouse-dir /user/cloudera
```

```
sqoop import \
--connect
jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table customers \
--warehouse-dir /user/cloudera
```

2. Create hive external tables on top of this hdfs data.
   create a database in Hive:

```
Create database bigdatabysumit;
Use bigdatabysumit;
```

```
create external table orders(
order_id int,
order_date string,
order_customer_id int,
order_status string )
row format delimited fields terminated by','
stored as textfile
location '/user/cloudera/orders';
```

```
create external table customers(
customer_id int,
customer_fname string,
customer_lname string,
customer_email string,
customer_password string,
customer_street string,
customer_city string,
customer_state string,
customer_zipcode string)
row format delimited fields terminated by','
stored as textfile
location '/user/cloudera/customers';
```

3. Check the value of property **hive.auto.convert.join**
   set hive.auto.convert.join;
   set hive.auto.convert.join=false; <------ if true, it will try to have map-side join whenever possible.
4. Try running the join query now: normal inner join **(number of mappers: 2; number of reducers: 1)**

```
select c.customer_id, c.customer_fname, c.customer_lname, o.order_id, o.order_date
from orders o join customers c on
(o.order_customer_id = c.customer_id) limit 10;
```

Check the runtime of the query
It took 23s approx for the job to run. This is to process the join without any optimization however we should not make any inference with this as data is quite small and we are doing it locally.

With Map-side join:
Now make set hive.auto.convert.join=true;
Now with same query, (**number of mappers: 1; number of reducers: 0**)

```
select c.customer_id, c.customer_fname, c.customer_lname, o.order_id, o.order_date
from orders o join customers c on
(o.order_customer_id = c.customer_id) limit 10;
```

In case of Map side join before starting the mapreduce job it will do a local task first.
What is the local task?
Getting the smaller table and putting it in the memory of each node as hashtable.

Note:
When the property hive.auto.convert.join=true; then in this case hive is treating it as map side join coz the conditions are met.
That is one of the table is small enough to fit in memory.

Definition of small table?
By default, any table which is < 25mb is considered small.

Que: What if we do not want to go with automatic conversion? Rather we want to specify using some hints.
Ans: set the hint manually to invoke map side join.
set hive.ignore.mapjoin.hint;
> hive.ignore.mapjoin.hint=true
hive> set hive.ignore.mapjoin.hint = false; <----- we are disabling the auto join feature but we will use hint.

Now try to execute inner join as map side join using hints indicating left table is small

```
select /*+ MAPJOIN(o) */ c.customer_id, c.customer_fname, c.customer_lname,
o.order_id, o.order_date
from orders o join customers c on
(o.order_customer_id = c.customer_id) limit 5;
```

Now try to execute inner join as map side join using hints indicating right table is small

```
select /*+ MAPJOIN(c) */ c.customer_id, c.customer_fname, c.customer_lname,
o.order_id, o.order_date
from orders o join customers c on
(o.order_customer_id = c.customer_id) limit 5;
```

Summary:
- Set the auto map join property to false and we saw that the reducer is coming into play.
- We set the property to true, and then we can see that there were 0 reducers.
- We set the auto map join property to false.
- We said that hints should not be ignored.
- In hints we mentioned left table in first run

- In hints we mentioned right table in second run.
- Inner joins are possible as map side join when one table is small enough to fit in memory.

In both the cases
1. Left table is small.
2. Right table is small

Inner join was treated as map side join

Conclusion: inner joins can be treated as map side join when left table/right table is small enough to fit in memory.

Que: Left outer join as map side join when left table is small?

Ans: NO, this gives error

```
select /*+ MAPJOIN(o) */ c.customer_id, c.customer_fname, c.customer_lname,
o.order_id, o.order_date
from orders o LEFT OUTER JOIN customers c on
(o.order_customer_id = c.customer_id) limit 5;
```

Que: Left outer join as map side join when right table is small?

Ans: YES.

```
select /*+ MAPJOIN(c) */ c.customer_id, c.customer_fname, c.customer_lname,
o.order_id, o.order_date
from orders o LEFT OUTER JOIN customers c on
(o.order_customer_id = c.customer_id) limit 5;
```

Que: Right outer join as map side join when left table is small?

Ans: YES, this works fine.

```
select /*+ MAPJOIN(o) */ c.customer_id, c.customer_fname, c.customer_lname,
o.order_id, o.order_date
from orders o RIGHT OUTER JOIN customers c on
(o.order_customer_id = c.customer_id) limit 5;
```

Que: Right outer join as map side join when right table is small?

Ans: NO.

```
select /*+ MAPJOIN(o) */ c.customer_id, c.customer_fname, c.customer_lname,
o.order_id, o.order_date
from orders o RIGHT OUTER JOIN customers c on
(o.order_customer_id = c.customer_id) limit 5;
```

Conclusion:

If left table is small:

Left outer join will fail
Right outer join will work
Full outer join will fail

If right table is small:

Left outer join will work
Right outer join will fail.
Full outer join will fail.

Full outer join can never be treated as map side join in any case.

What is small table as per hive?

There is a property: Hive.mapjoin.smalltable.filesize;

hive.mapjoin.smalltable.filesize=25000000 ie 25mb (default)

The value of this property indicates the size. If the file is having lesser size than this then it is considered as small file.

Bucket Map Join

What if we have 2 big tables?
Map side join wont work.
In this case you can go for bucket map join.

Constraints to BMJ:

- Both the tables should be bucketed on join column.
  - Customers table should be bucketed on customers_id.
  - Orders table should be bucketed on order_customer_id.
- Number of buckets in bigger table should be integral multiple of number of buckets in smaller table.

Example:
We need to create 2 tables bucked on join column. We need to set below property.

```
set hive.enforce.bucketing;
set hive.enforce.bucketing=true;
```

For customers table:

```
create external table customers_bucketed(
customer_id int,
customer_fname string,
customer_lname string,
customer_email string,
customer_password string,
customer_street string,
customer_city string,
customer_state string,
customer_zipcode string)
clustered by(customer_id) into 4 buckets row format delimited fields terminated by ',';
insert into customers_bucketed select * from customers;
```

Note: number of mappers: 1; number of reducers: 4

For orders table:

```
create external table orders_bucketed(
order_id int,
order_date string,
order_customer_id int,
order_status string )
clustered by (order_customer_id) into 8 buckets row format delimited fields terminated by ',';
insert into orders_bucketed select * from orders;
```

Note: number of mappers: 1; number of reducers: 8

When we created the external table we didn't specify any location.
That is why the data should be in default directory.

```
hadoop fs -ls /user/hive/warehouse/bigdatabysumit.db/customers_bucketed
hadoop fs -ls /user/hive/warehouse/bigdatabysumit.db/orders_bucketed
```

Now set one more property: **set hive.optimize.bucketmapjoin = true;**
Now try running the join query on bucketed tables:

```
select c.customer_id, c.customer_fname, c.customer_lname, o.order_id, o.order_date
from customers_bucketed c JOIN orders_bucketed o ON
(c.customer_id = o.order_customer_id) limit 10;
```

Here, number of mappers: 2; number of reducers: 1
So we need to set a property to auto mapside join.
Set hive.auto.convert.join = true;
Now, number of mappers: 1; number of reducers: 0

Sort Bucket Map Join (SMB)

Conditions for sort merge bucket join (SMB):99

- Both the tables should be bucketed on the join column.
- Number of buckets in larger table should be exactly equal to number of buckets in the smaller tables.
- Data in both tables should be sorted based on join column.
- Apart from this, we need to set certain properties.

Note: here both the tables can be large as well.

Example: we need to create 2 tables bucked on join column. We will have same number of buckets in both tables.
Let us drop and recreate the bucketed tables. The reason is that we need equal no. Of buckets in both tables & that too sorting based on join column.

```
drop table orders_bucketed;
drop table customers_bucketed;
```

Que: what do you expect? After dropping the table is the data deleted or not?
Ans: These were external tables so data will still stay. As only metadata is dropped.

So now recursively remove data using hadoop command manually.

```
hadoop fs -rm -r /user/hive/warehouse/bigdatabysumit.db/orders_bucketed
hadoop fs -rm -r /user/hive/warehouse/bigdatabysumit.db/customers_bucketed
```

We need to set the below properties:

```
set hive.auto.convert.sortmerge.join =true;
set hive.auto.convert.sortmerge.join.nonconditionaltask =true;
set hive.auto.optimize.bucketmapjoin =true;
set hive.auto.optimize.bucketmapjoin.sortedmerge =true;
set hive.auto.enforce.bucketing =true;
set hive.auto.enforce.sorting =true;
set hive.auto.convert.join =true;
```

Create first bucketed & sorted table:

```
create table customers_bucketed(
customer_id int,
customer_fname string,
customer_lname string,
customer_email string,
customer_password string,
customer_street string,
customer_city string,
customer_state string,
customer_zipcode string)
clustered by(customer_id) sorted by(customer_id asc) into 4 buckets
row format delimited fields terminated by ',';

insert into customers_bucketed select * from customers;
```

```
create table orders_bucketed(
order_id int,
order_date string,
order_customer_id int,
order_status string )
clustered by (order_customer_id) sorted by (order_customer_id) into 4 buckets row format delimited fields terminated by ',';

insert into orders_bucketed select * from orders;
```

Let us perform the join now:

```
select c.customer_id, c.customer_fname, c.customer_lname, o.order_id, o.order_date
from customers_bucketed c JOIN orders_bucketed o ON
(c.customer_id = o.order_customer_id) limit 10;
```

Question: Validating whether it actually happened
With the logs as seen in previously you cannot infer if actually SMB join took place or not. To get surety we can check the explain plan of the query. We should clearly see there.

> **EXPLAIN EXTENDED** select c.customer_id, c.customer_fname, c.customer_lname, o.order_id, o.order_date from customers_bucketed c JOIN orders_bucketed o ON (c.customer_id = o.order_customer_id) limit 10;

Windowing Functions:

A suite of functions which are syntactic sugar for complex queries.
Makes complex operations simple without needing many intermediate calculations.
It reduces the need of intermediate tables to store temporary data.

Example: what were the top selling N products in last week's sale?
Window = one week
Operation = ranking product sales

Practical:

Create table

```
CREATE table groceries(
id string,
store string,
product string,
day date,
revenue double
)
row format delimited fields terminated by ',';
```

Load data into groceries table:

```
load data local inpath '/home/cloudera/Desktop/shared/week5_datasets/groceries.csv'
into table groceries;
```

Display all records of table:

```
Select * from groceries;
```

To get running sum of groceries table - order by id: apply windowing function
Unbounded preceding means = the first row

```
from groceries
select id, revenue, day,
sum(revenue)                                              <-------- operation
over ( order by id
rows between unbounded preceding and current row )  <------ window
as running_total;
```

to get more meaningful data ----> sort order by day

Note: we can also avoid "rows between unbounded preceding and current row" as this is default settings.

To get running average of groceries table:

```
from groceries
select id, revenue, day,
avg(revenue)                                              <-------- operation
over ( order by id
rows between unbounded preceding and current row )  <------ window
as running_average;
```

Calculate aggregations over a window on blocks of records: now we want to do group wise on each day

Create another groceries1 table inside TT db:

```
CREATE table groceries1(
id string,
store string,
product string,
day date,
revenue double
)
row format delimited fields terminated by ',';
```

Load data from groceries1.csv to groceries1

To get revenue running total for each day:

```
from groceries1 select id, revenue, day,
sum(revenue) over (
partition by day
order by id )
as running_total;
```

To get running count of no of records for each day:

```
from groceries1 select id, revenue, day,
Count(id) over (
partition by day
order by id )
as running_count;
```

Revenue running total for each day without order by key specified:

```
from groceries1 select id, revenue, day,
sum(revenue) over (
partition by day )
as running_sum;
```

Here, we will get the final total not the incremental total with every id

Note: if we do not give order by clause then the behaviour will be window size is from 1st row to the last row. "Rows between unbounded preceding to unbounded following."

Moving averages using a window function:

Calculating aggregations over a window of a specific size:

Here, window size is 4 rows.

```
from groceries1
select id, revenue, day,
avg(revenue) over (
order by id
rows between 3 preceding and current row )
as running_average;
```

Ranking in Hive: Rank, dense rank, row number, partition by

Lets create the table and load the data:

Dataset: rank_dataset.csv

```
create table rank_test(name string, score int)
row format delimited fields terminated by ','
lines terminated by '\n'
stored as textfile;
```

```
load data local inpath
'/home/cloudera/Desktop/shared/week5_datasets/rank_dataset.csv'
into table rank_test;
```

Rank function on table:

```
select name, score, rank() over
  > (order by score desc)
  > as ranking from rank_test;
```

Meaning, ranking based on desc order of score(col 2). highest score will be rank 1
Ties are assigned same rank with next ranking skipped.

Dense rank:

Dense rank gives ranking. Ranks are consecutive, no ranks are skipped even in case of ties.

```
select name, score, dense_rank() over
(order by score desc)
as ranking from rank_test;
```

Row number:

Row number assigns unique numbers to each row given the order by clause.

```
select name, score, row_number() over
(order by score desc)
as ranking from rank_test;
```

Row number with partition by clause:

This query will first groups based on names and then do the ranking for each group.

```
select name, score, row_number() over(
partition by name
order by score desc)
as ranking from rank_test;
```

Sorting in Hive: order by, sort by, distribute by, cluster by

Order by:

- Order by guarantees total ordering
- Only 1 reducer is used.
- Limitations:
  - In strict mode **limit** clause is compulsory. So that it reduces load on single reducer.
  - Hive.mapred.mode=strict
- Order of Null's by default is Null's are shown first when sorting in asc order.

Create table and load data:

```
select name, score, row_number() over(
partition by name
order by score desc)
as ranking from rank_test;

load data local inpath
'/home/cloudera/Desktop/shared/week5_datasets/order.txt'
into table order;
```

Order by clause: sorted in asc order

```
Select order_value from order
Order by order_value;
```

number of mappers: 1; number of reducers: 1
As reducer = 1 so it will give global sort in one go

Sort by

- Sort by can use multiple reducers.
- Local sorting will happen on each reducer.
- We cannot guarantee a total ordering
- Below 2 properties define how many reducers will be there:
  - **Set hive.exec.reducers.bytes.per.reducer** (250mb) --- each reducer will hold 250mb
  - **Set mapred.reduce.tasks** ---- number of reducers (default is -1) which means above prop will hold true. You can change it to any no you want ie to num of reducers.

Sort by clause with 2 reducers: means 2 sorted lists

```
Select order_value from order
Sort by order_value;
```

Distribute by

Distribute by ensures each of the reducer gets **non-overlapping** set of values.
It will not sort, it will just distribute the values.\
Hash function will come into play.

```
select order_value from order
distribute by order_value;
```

Order by - uses one reducer, and it does complete sorting
Sort by - can use more than 1 reducer, total ordering is not guaranteed.
Also, same value is going to multiple reducers

Distribute by sort by:

if we need non overlapping set of values on each reducer and if we need sorting as well then we should use distribute by sort by.

```
select order_value from order
distribute by order_value
sort by order_value;
```

Cluster by:

It is an alternative to doing distribute by sort by.

Distribute by + Sort by = Cluster by

```
select order_value from order
cluster by order_value;
```