SQOOP FUNDAMENTALS

Sqoop Basics - mainly used for data ingestion/migration

- Sqoop is a tool used to transfer data from RDBMS to HDFS or HDFS to RDBMS.
- Ex. Employee table in your MySQL Db, you want to bring this in your HDFS.
- **Sqoop Import** is used to transfer data from **Db to HDFS**.
- **Sqoop Export** is used to transfer data from **HDFS to Db**.

Key features of Sqoop -->
- Full Load
- Incremental Load
- Parallel Import/Export
- Compression
- Security Integration
- Load data directly into Hive/HBase

= **Sqoop Import** imports individual tables from RDBMS to HDFS.
= Each row in a table is treated as a record in HDFS.
= Records can be stored as Text file format, Sequence file format, Avro & Parquet file format.

= **Sqoop Export** exports a set of files from HDFS to an RDBMS.
= The target table must already exist in the DB.
= Files are read & parsed into a set of records according to the user-specified delimiters.

Sqoop Eval - to run queries on database.

To enter MySql as root	Mysql -u root -p
As another user	Mysql -u retail_dba -p
To come out	exit
Password	cloudera
List of databases	Show databases;
To use a db	Use retail_db;
To get tables inside db	Show tables;
Create a database	Create database database_name;
Create a table	CREATE TABLE people (PersonID int, LastName varchar(255), FirstName varchar(255), Address varchar(255), City varchar(255));

Accessing MySQL Db from Hadoop using Sqoop --

```
sqoop-list-databases \
--connect "jdbc:mysql://quickstart.cloudera:3306" \
--username retail_dba \
--password cloudera
```

Accessing MySQL tables using the root user:

```
sqoop-list-tables \
--connect “jdbc:mysql://quickstart.cloudera:3306/retail_db” \
--username retail_dba \
--password cloudera
```

Displaying table data using sqoop-eval:

```
sqoop-eval \
--connect “jdbc:mysql://quickstart.cloudera:3306” \
--username retail_dba \
--password cloudera \
--query “select * from retail_db.customers limit 10”
```

To get IP address of machine -> ifconfig

SQOOP-IMPORT

Sqoop Import - Map Reduce job, only mappers work and no reducer.
- by default there are 4 mappers which do the work.
- we can change these number of mappers.
- these mappers divide the work based on primary key.

If there is no primary key then what will happen -- it will not import

Solution --

- ✓ You change the number of mappers to 1 -- not a good approach as no parallelism.
- ✓ Split by column where you will be mentioning column name that split after this column

```
sqoop-eval \
--connect “jdbc:mysql://10.0.2.15:3306/retail_db” \
--username retail_dba \
--password cloudera \
--query “describe retail_db.orders”
```

Import data from Mysql to Sqoop: if tables don’t have primary key it will not import.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--target-dir /queryresult
```

To display contents of queryresult dir in HDFS(use terminal):

```
Hadoop fs -ls /queryresult
```

= by default, number of mappers are 4, so 4 folders are created.

To get the content inside target dir/files.

```
Hadoop fs -cat /queryresult/*
```

If we don't have primary key in table:

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/trendytech \
--username root \
--password cloudera \
--table persons \
-m 1 \
--target-dir /peopleresult
```

=here, as we don't have primary key in table persons, so -m 1 is specifying that only 1 mapper will work instead of usual 4. otherwise if we have had primary key then mappers will divide content according to PK column.

To import all tables from Mysql to HDFS:

```
Sqoop-import-all-tables \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--as-sequencefile \        <---------------------------------------- file format
-m 4 \                     <----------------------------------------redundant line
--warehouse-dir /user/cloudera/sqoopdir
```

== no. Of mappers are 4 so 4 files. We can also mention file format while importing data as mentioned above. By default, it is Text format if we don't mention anything.

File formats :

- Text file format
- Sequence file format
- Avro file format
- Parquet file format

Target dir vs Warehouse dir.

Ex. Employee table that you are importing from mysql.

= In case of Target Dir the directory path mentioned is the final path where data is copied.
/data
= In case of Warehouse Dir, the system will create a Sub Directory with the table name. For each table a separate sub dir will be created.
/data/employee

= -warehouse-dir is used to specify a base dir within hdfs where sqoop will create a sub folder inside with the name of the source table, and import the data files into that folder.
= better choice is to use the warehouse dir, as it is production ready

= Dir structure for retail_db will be:

```
/user/cloudera/sqoopdir/employee
/user/cloudera/sqoopdir/customer
/user/cloudera/sqoopdir/table3
```

To display a list of all available tools: Sqoop help
To know version of sqoop : Sqoop version
Sqoop help with command alias : Ex. sqoop help eval, sqoop help import

Same commands:

--password	-P
--query	-e
--num-mappers 1	-m 1

Redirecting logs: to store logs/messages in a file to view later for debugging purposes.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--warehouse-dir /queryresult4 1>query.out 2>query.err
```

1 => all output files
Query.out =>file name, it can be anything
2 => all other messages other than output

= query.out and query.err files will be generated in present working dir.

To check the content of the queryresult4: Hadoop fs -ls /queryresult4/orders/
To check the content of query files : cat query.out and cat query.err

---

Sqoop Boundary query (very imp interview ques) -- this tells each mapper will work on that many records
Find the max of Pk and Min of Pk
Split size = (max_of_pk - min_of_pk)/Num_mappers

Ex. For 1 lakh records, split size = (100000 - 0)/4 --> ie 25000
Mapper1 0 - 25000
Mapper2 25001 - 50000
Mapper3 50001 - 75000
Mapper4 75001 - 100000

= whenever we run sqoop import with no. Of mappers >1, a boundary vals query will run.
= BVQ is run on Primary key field to get min and max value of it.
= we can customize the BVQ by using **--boundary-query.**
= Query should return 2 values ie min and max as one row. These values will be used to compute the split size.
= we can hardcode the min and max values with an intention to remove outliers(extra record).

Customize boundary query

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--boundary-query “select 1, 68883” \        <----------customize BVQ
--warehouse-dir /user/cloudera/ordersboundval
```

= with customize BVQ, all mappers will do the same amount of work, which is ideallt good.

Sqoop import execution flow --

How mappers divide their work when a query is fired:

1. Selects 1 record and by using that it gets the metadata and builds the java file.
   Select * from tablename limit 1
2. Build the POJO(Plain old java objects) class with appropriate getters and setters. Combine the POJO class into jar file (Using above java file it builds the jar file. Ie it will try to bundle it into a java archive.)
3. Runs **BoundingValsQuery** based on min and max on primary key.
4. Now it will calculate split size.
5. Submit map reduce job with number of mappers = 4 by default.
6. Each map task will run select query on the source table with where cond'n based on the splits to read the data.
7. Data will be written to the files in the location specified.

Compression techniques:

You can compress data by using the default (**gzip**) algorithm with the **-z** or **--compress** argument.

We will see **.gz** extension

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

We can specify any Hadoop compression codec using the **--compression-codec** argument.

To get compression result in other format other than .gz example. **BZip2Codec**. Extension is **.bz2**

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--compression-codec BZip2Codec \
--warehouse-dir /user/cloudera/bzipcompresult
```

Import the data with selected columns:

We can select a subset of columns and control their ordering by using the **--columns** argument.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table customers \
--columns customer_id,customer_fname,customer_city \
--warehouse-dir /user/cloudera/customerresult
```

Import with WHERE clause/condition :

You can control which rows are imported by adding a SQL where clause to the import statement.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--columns order_id,order_customer_id,order_status \
--where "order_status in ('complete','closed')" \
--warehouse-dir /user/cloudera/customimportresult
```

Another example of BVQ where we can even customize the min and max values using non primary key column.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table order_items \
--boundary-query 'select min(order_item_order_id), max(order_item_order_id) from order_items where order_item_order_id ge 10000" \      <---------- > instead of ge
--warehouse-dir /user/cloudera/bvqresult
```

Where clause also internally treated as BVQ:

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--columns order_id,order_customer_id,order_status \
--where "order_status in ('processing')" \
--warehouse-dir /user/cloudera/whereclauseresult
```

= Here, where clause is appended as part of BVQ.

Sqoop import using SPLIT-BY:

= *Sqoop--split-by* comes into picture when there is no primary key or the Pk column is not evenly distributed.

When to use Split-by:

- When there is no PK, we can use split-by to indicate the column based on which mappers should divide the work.
- Or when PK has lot of outliers, then we can use any other column as split-by column(to get a better performance.)

Understanding SQOOP SPLIT-BY:

- Let us understand what happens when we run typical sqoop import and when --split-by argument should be used.
- Whenever we run sqoop import with number of mappers >1, a bounding val query will run.
- BVQ is run on PK field to get min and max value of it.
- If there is no PK or unique key, import will fail unless number of mappers is set to 1 or specify a field using split-by.
- It is better practice to use an indexed field which do not continue null values as part of split-by.

Example:

- Create a table orders_no_pk in MYSQL without any PK:

```
CREATE TABLE orders_no_pk (
        Order_id int(11) NOT NULL,
        Order_date datetime NOT NULL,
        Order_customer_id int(11) NOT NULL,
        Order_status varchar(45) NOT NULL
);
```

- Copying data from orders table to orders_no_pk table:

```
Insert into orders_no_pk select order_id, order_date, order_customer_id, order_status from orders;
Commit;
```

- Import the orders_no_pk table into HDFS using Sqoop:

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders_no_pk \
--warehouse-dir /ordernopk
```

- Above import fails coz the order_no_pk table doesn't have PK and Sqoop doesn't know how to divide records among mappers.

Import the same orders_no_pk table into HDFS using split_by:
When there is no PK --split-by is the ideal situation for import control.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders_no_pk \
--split-by order_id \
--warehouse-dir /user/cloudera/ordernopksplit
```

= Split column should have numeric values. It is not recommended to use split-by on a text column.

Dealing with SPLIT-BY or PK on non numeric fields:
Sqoop import using non numeric field fail with hint to use
org.apache.sqoop.splitter.allow_text_splitter=true
It is also applicable when we use non numeric field as part of split-by clause.

```
Sqoop-import \
-D***org.apache.sqoop.splitter.allow_text_splitter=true*** \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table categories \
--split-by "category_name" \
--warehouse-dir /user/cloudera/splitonnonnumeric \
--delete-target-dir
```

= this is not recommended however due to poor performance.

Sqoop autoreset to one mapper: Sqoop import should not fail even if there no pk.

```
Sqoop import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders_no_pk \
--warehouse-dir /user/cloudera/npkresult \
--autoreset-to-one-mapper \
--num-mappers 8
```

= it mean autoreset to 1 mapper if theres no Pk, if theres Pk then use number of mappers = 8.
= if you don't specify --num-mappers 8, then by default mappers = 4

Import all table if some have PK and some don't have PK:

--autoreset-to-one-mapper uses one mapper if a table with no PK is encountered

```
Sqoop import-all-tables \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--warehouse-dir /user/cloudera/autoreset1mresult \
--autoreset-to-one-mapper \
--num-mappers 2
```

Delimiters may be specified by following arguments:

- ✓ --fields-terminated-by <char>
- ✓ --lines-terminated-by <char>

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--fields-terminated-by '|' \
--lines-terminated-by ';' \
--target-dir /user/cloudera/delimiterresult
```

Create a HIVE table based on a database table(mysql):

The *--create-hive-table* argument populates a hive metastore with a definition for a table based on a database. This will create a empty table in hive based on metadata in mysql.

```
Sqoop create-hive-table \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--hive-table emps \
--fields-terminated-by ','
```

Hive commands:

```
To Open hive - hive;
Show tables;
Describe  emps;
```

Sqoop verbose: run your sqoop job with the *--verbose* flag to generate more logs and debugging information. We can see the BVQ frames properly.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--verbose \
--target-dir /user/cloudera/verbosereult
```

Sqoop append: by default, imports go to a new target location. If the destination dir already exists in HDFS, sqoop will refuse to import.

The *--apppend* argument will append data to an existing dataset in HDFS.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
```

```
--username retail_dba \
--password cloudera \
--table orders \
--target-dir /user/cloudera/appendresult \
--append
```

= It will give double files in hdfs location.

Delete target dir if exists: it means overwrite that dir, ie whatever I am importing should be the only data in dir and delete previous one.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--verbose \
--target-dir /user/cloudera/appendresult \
--delete-target-dir
```

Controlling Parallelism : specify the number of map tasks (parallel processes) to import data by using the *-m* or *--num-mappers* argument.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--target-dir /user/cloudera/mapperresult \
--delete-target-dir \
--num-mappers 8
```

Displaying schema of mysql table from terminal:

```
Sqoop eval \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
-e “DESCRIBE customers”
```

Dealing with nulls while importing data:

When we import data into text files, we might have to explicitly deal with null values.

We can specify non string nulls using *--nulls-non-string* and string nulls using *--null-string.*

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username retail_dba \
--password cloudera \
--table orders \
--warehouse-dir /user/cloudera/nullstringresult \
--delete-target-dir \
--null-non-string “-1”
```

SQOOP EXPORT

Sqoop-export = is used to transfer data from HDFS to RDBMS
The table should be present in mysql.

= we have card_trans.csv on desktop locally in cloudera.
= we should be moving this file from local to HDFS.
= create a data dir inside hadoop

- In MySQL create a table in banking database.
  Create database banking;
  Use banking;

  CREATE TABLE card_transactions (
  Transaction_id INT,
  Card_id BIGINT,
  Member_id BIGINT,
  Amount INT,
  Postcode INT,
  Pos_id BIGINT,
  Transaction_dt varchar(255),
  Status varchar(255),
  PRIMARY KEY (transaction_id));

- Move the card_trans.csv file in /data folder in hdfs & execute the sqoop export command.
  Hadoop fs -mkdir /data
  Hadoop fs -ls /data
  Hadoop fs -put Desktop/card_trans.csv /data

  Sqoop export \
  --connect jdbc:mysql://quickstart.cloudera:3306/banking \
  --username root \
  --password cloudera \
  --table card_transactions \
  --export-dir /data/card_trans.csv \
  --fields-terminated-by ‘,’

SQOOP EXPORT FAILURE:

- ✓ Why the job failed?
  - ✧ To track the failed job:
  - ✧ Go to the log and find the url to track the job
  - ✧ open the given url
  - ✧ Click on logs of failed jobs.
  - ✧ It can be due to duplicate records.

- ✓ If a job fails how to make sure that target table is not impacted(should not be partial transfer)
  STAGING TABLE
  - ✧ Create a staging table card_transactions_stage in mysql with same schema:
    - ◆ Sqoop export using staging table, data will first go to staging table and if there’s any error, it will just remain at ST, & will not impact actual table & we may see partial records in staging table and no records in actual table.
    - ◆ If the sqoop export works fine then first all records all loaded to ST and if it is successful it will migrate all records to actual table

```
Sqoop export \
--connect jdbc:mysql://quickstart.cloudera:3306/banking \
--username root \
--password cloudera \
--table card_transactions \
--staging-table card_transactions_stage \
--export-dir /data/card_transactions.csv \
--fields-terminated-by ','
```

Sqoop Incremental Import - Delta Load

Sqoop povides an incremental import mode which can be used to retrieve only rows newer than some previously-imported set of rows.

Parameters:

- ✓ --check-column (col) char not supported
- ✓ --incremental (mode) append, lastmodified
- ✓ --last-value (value)

Sqoop supports two types of incremental imports:

- **Append mode** - used when there are no updates in data, but there are just new inserts.
- **Lastmodified -** when we need to capture the updates also. So in this case we will be using a date on basis of which we will try to fetch the day.

You can use the --incremental argument to specify the type of incremental import to perform.

Theory:
You should specify **append mode** when importing a table where new rows are continually being added with increasing row id values.
You specify the column containing the row's id with **--check-column**.
Sqoop imports rows where the check column has a value greater than the specified with **--last-value.**

An alternate table **update** strategy supported by sqoop is called **lastmodified mode.**
You should use this when rows of the source table may be updated, and each such update will set the value of a last-modified column to the current timestamp.
Rows where the check column holds a **timestamp** more recent than the timestamp specified with **--last-value** are imported.

Using Append Mode: only inserts, no updates.
Example: orders table have 68883 records in retail_db in mysql.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--warehouse-dir /data \
--incremental append \
--check-column order_id \
--last-value 0          <---------- last-value is 0 coz first we need to import all records in new dir
```

Here, in logs, Lower bound value = 0
Upper bound value = 68883

New records:
insert into orders values(68884,'2014-07-23 00:00:00',5522,'COMPLETE');
insert into orders values(68885,'2014-07-23 00:00:00',5522,'COMPLETE');
insert into orders values(68886,'2014-07-23 00:00:00',5522,'COMPLETE');
insert into orders values(68887,'2014-07-23 00:00:00',5522,'COMPLETE');
insert into orders values(68888,'2014-07-23 00:00:00',5522,'COMPLETE');
insert into orders values(68889,'2014-07-23 00:00:00',5522,'COMPLETE');
commit;

After inserting new records in orders table.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--warehouse-dir /data \
--incremental append \
--check-column order_id \
--last-value 68883
```

When import is running we can see boundary query, split size & the last value fetched.
= we have to manually keep track of the id of last record.

Counters: are stats related to the job. For example bytes read, bytes written etc.

Sqoop Incremental Last Modified: when there are updates as well.

Before proceeding delete the orders dir inside data: *Hadoop fs -rm -R /data/orders*

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--warehouse-dir /data \
--incremental lastmodified \
--check-column order_date \      <--------- check-col should be a date col or timestamp
--last-value 0 \                 <--------- o to bring all records initially.
--append                         <--------- if already have a dir then just append it.
```

To check current timestamp in mysql :
Select current_timestamp from dual;

More records:
insert into orders values(68890,current_timestamp,5523,'COMPLETE');
insert into orders values(68891,current_timestamp,5523,'COMPLETE');
insert into orders values(68892,current_timestamp,5523,'COMPLETE');
insert into orders values(68893,current_timestamp,5523,'COMPLETE');
insert into orders values(68894,current_timestamp,5523,'COMPLETE');
update orders set order_status='COMPLETE' and order_date = current_timestamp
WHERE ORDER_ID = 68862;
commit;

We have inserted some new records and modified some existing records.
Now we want all the updated records and new records to be pulled in next sqoop import.

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--warehouse-dir /data \
--incremental lastmodified \
--check-column order_date \          <--------- check-col should be a date col or timestamp
--last-value '2022-01-03 04:32:43.0' \   <---------previous date last timestamp after initial import
--append
```

Note: Sqoop import with last modified, supplied with --append will result in duplicate records in HDFS dir.
What if we just want the latest records in HDFS?
Then in that case instead of --append we need to use: **--merge-key <merge-column>**
Here, **merge-column can be PK**.

[If a record is updated in table and then we use incremental import + last modified, then we will get the updated record also.
50000 oldtimestamp in hdfs
50000 newtimestamp in hdfs.
You want that hdfs file should be always in sync with the table]

Incremental lastmodified merge-key
The merge tool allows you to combine two datasets where entries in one dataset should overwrite entries of an older dataset. For example, an incremental import run in last-modified mode will generate multiple datasets in HDFS where successively newer data appears in each dataset. The merge tool will 'flatten' two datasets into one, taking the newest available records for each PK.

The merge tool is typically run after an incremental import with the date-last-modified mode
(sqoop import --incremental lastmodified…).

```
Sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--warehouse-dir /data \
--incremental lastmodified \
--check-column order_date \
--last-value '2022-01-03 04:32:43.0' \
--merge-key order_id
```

Once the records are imported then a mapreduce job will run to merge new records with old ones.
All the files in output folder are now merged in just one single part-r file.
After merge, it will combined all part-m files into one part-r file. This will hold only unique order_id with latest timestamp.

Sqoop Job & Password Management:

Sqoop Job : to automate things

Why a sqoop job?
Saved jobs remember the parameters used to specify a job, so they can be re-executed by invoking the job.
If a saved job is configured to perform an incremental import, state regarding the most recently imported rows is updated in the saved job to allow the job to continually import only the newest rows.

What was the pain point?
Earlier in incremental import, we have to manually keep a track of last value and then supply that during the next run. This was a manual process and was a big pain point.

Creating a Sqoop Job(for incremental import)

```
Sqoop job \
--create job_orders \
-- import \                    <---------- note: there is a space here b/w -- & import
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--warehouse-dir /data \
--incremental append \
--check-column order_id \
--last-value 0
```

See the list of sqoop jobs: *sqoop job --list*
Executing the sqoop job: *sqoop job --exec <sqoop_job_name>*
Checking the saved state of the job: *sqoop job --show <sqoop_job_name>*
Deleting a sqoop job: *sqoop job --delete <sqoop_job_name>*

But the job is not completely automated yet. Why?
->Because it asks for password when we run the sqoop job.

Solution to password problem?
A **password file** can help in this case. This means that our password (to connect to db) will be stored in a file. We need to specify the file path so that we do not have to provide the password manually at runtime.

How to create password file? e*cho -n "cloudera" >>.password-file*

Note:
- If you try creating a file using normal ways then some special characters gets appended at the end of the password. Which will lead to incorrect password error.
- In the above command the name of file is password-file.
- Also "." before the filename indicates its a hidden file. So the output of echo (cloudera) is stored in a hidden file named as password-file.

Re-create the sqoop job which uses the password file.

```
Sqoop job \
--create job_orders \
-- import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password-file file:///home/cloudera/.password-file \
--table orders \
--warehouse-dir /data \
--incremental append \
--check-column order_id \
--last-value 0
```

Here, file:// indicates that the password file is in local, and not in HDFS. If you do not mention file:// then it will expect the file in HDFS.

Where is the state of Job stored?
The state of job is stored locally in a hidden folder named (.sqoop) which resides in home dir (/home/cloudera)
Command: ***ls -altr /home/cloudera*** (here a is to list all files including hidden files)
cd .sqoop/
cat metastore.db.script | grep incremental <------ we are piping the results to grep and searching for incremental keyword. Grep is used for searching.
It stores 3 things:
'Incremental.last.value' ---> '68894'
'Incremental.col' ----> 'order_id'
'Incremental.mode' -----> 'AppendRows'

What if someone gets hold of your password file?
The concept of **Password alias** using encrypted password file with JCEKS (java cryptography encryption key store)

How to create a password alias?
Hadoop credential create mysql.banking.password -provider jceks://hdfs/user/cloudera/mysql.password.jceks

mysql.banking.password ==> name of password alias
jceks://hdfs/user/cloudera ==> location where password is stored in encrypted form.

Check the content of your file
Hadoop fs -cat /user/cloudera/mysql.password.jceks

Use the password-alias in your sqoop command:
A sqoop eval job to count the number of rows in orders table.

```
Sqoop eval \
-Dhadoop.security.credential.provider.path=jceks://hdfs/user/cloudera/mysql.password.jceks \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password-alias mysql.banking.password \
--query "select count(*) from orders"
```

-D is to set a property, after -D property starts

4 ways to pass the password:
- ✓ In the command using --password
- ✓ Passing the parameter at runtime using -P
- ✓ Storing the password in password in password file
- ✓ Password alias with encrypted password

SQOOP Additional Reading

Default Import

By default, Sqoop will import a table named orders to a directory named orders inside your home directory in HDFS. For example, if your username is someuser, then the import tool will write to /user/someuser/orders/(files)

```
sqoop-import
\--connect jdbc:mysql://quickstart.cloudera:3306/retail_db
\--username root
\--password cloudera
\--table orders
```

Note: we are not specifying the target-dir or warehouse-dirAlso just like we import a table we can import a view also.

Free-Form Query Imports

Instead of using the --table, --columns and --where arguments, you can specify a SQL statement with the --query argument.

When importing a free-form query, you must specify a destination directory with --target-dir.

If you want to import the results of a query in parallel, then each map task will need to execute a copy of the query, with results partitioned by bounding conditions inferred by Sqoop. Your query must include the token $CONDITIONS which each Sqoop process will replace with a unique condition expression. You must also select a splitting column with --split-by.

Note: If you are issuing the query wrapped with double quotes ("), you will have to use \$CONDITIONS instead of just $CONDITIONS to disallow your shell from treating it as a shell variable.

Example 1:

```
sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--query 'select * from orders where $CONDITIONS AND order_id > 50000' \
--target-dir /data/orders3 \
--split-by order_id
```

Example 2:

```
sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--query "select * from orders where \$CONDITIONS AND order_id > 50000" \
--target-dir /data/orders2 \
--split-by order_id
```

Note: The facility of using free-form query in the current version of Sqoop is limited to simple queries where there are no ambiguous projections and no OR conditions in the WHERE clause. Use of complex queries such as queries that have sub-queries or joins leading to ambiguous projections can lead to unexpected results.

Direct Import

Controlling the Import ProcessBy default, the import process will use JDBC which provides a reasonable cross-vendor import channel. Some databases can perform imports in a more high-performance fashion by using database-specific data movement tools. For example, MySQL provides the mysqldump tool which can export data from MySQL to other systems very quickly. By supplying the --direct argument, you are specifying that Sqoop should attempt the direct import channel. This channel may be higher performance than using JDBC. But can be used for very basic things only.

example:

```
sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--target-dir /data/orders \
--direct
```

Importing Data into Hive:

Sqoop's import tool's main function is to upload your data into files in HDFS. If you have a Hive metastore associated with your HDFS cluster, Sqoop can also import the data into Hive by generating and executing a CREATE TABLE statement to define the data's layout in Hive. Importing data into Hive is as simple as adding the --hive-import option to your Sqoop command line.

If the Hive table already exists, you can specify the --hive-overwrite option to indicate that existing table in hive must be replaced. After your data is imported into HDFS or this step is omitted, Sqoop will generate a Hive script containing a CREATE TABLE operation defining your columns using Hive's types, and a LOAD DATA INPATH statement to move the data files into Hive's warehouse directory.

Sqoop will by default import NULL values as string null. Hive is however using string \N to denote NULL values and therefore predicates dealing with NULL (like IS NULL) will not work correctly. You should append parameters --null-string and --null-non-string in case of import job or --input-null-string and --input-null-non-string in case of an export job if you wish to properly preserve NULL values. Because sqoop is using those parameters in generated code, you need to properly escape value \N to \\N

```
sqoop import  ... --null-string '\\N' --null-non-string '\\N'
```

The table name used in Hive is, by default, the same as that of the source table. You can control the output table name with the --hive-table option.

example:

```
sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \
--hive-import \
--hive-table orders_new \
--verbose
```

Importing Data into HBase:

Sqoop supports additional import targets beyond HDFS and Hive. Sqoop can also import records into a table in HBase.

By specifying --hbase-table, you instruct Sqoop to import to a table in HBase rather than a directory in HDFS. Sqoop will import data to the table specified as the argument to --hbase-table. Each row of the input table will be transformed into an HBase Put operation to a row of the output table. The key for each row is taken from a column of the input. By default Sqoop will use the split-by column as the row key column. If that is not specified, it will try to identify the primary key column, if any, of the source table. You can manually specify the row key column with --hbase-row-key. Each output column will be placed in the same column family, which must be specified with --column-family.
If the target table and column family do not exist, the Sqoop job will exit with an error. You should create the target table and column family before running an import. If you specify --hbase-create-table, Sqoop will create the target table and column family if they do not exist, using the default parameters from your HBase configuration.

Validate:
Validate the data copied, either import or export by comparing the row counts from the source and the target post copy. Validation currently only validates data copied from a single table into HDFS and there are a lot of limitations.
example:

```
sqoop-import \
--connect jdbc:mysql://quickstart.cloudera:3306/retail_db \
--username root \
--password cloudera \
--table orders \-
-target-dir /data/orders \
--validate
```

Exports may fail for a number of reasons:
1. Loss of connectivity from the Hadoop cluster to the database (either due to hardware fault, or server software crashes)
2. Attempting to INSERT a row which violates a consistency constraint (for example, inserting a duplicate primary key value)
3. Attempting to parse an incomplete or malformed record from the HDFS source data
4. Attempting to parse records using incorrect delimiters
5. Capacity issues (such as insufficient RAM or disk space)

Note: If an export map task fails due to these or other reasons, it will cause the export job to fail. The results of a failed export are undefined. Each export map task operates in a separate transaction. Furthermore, individual map tasks commit their current transaction periodically. If a task fails, the current transaction will be rolled back. Any previously-committed transactions will remain durable in the database, leading to a partially-complete export.

To leave safe mode: sudo -u hdfs hdfs dfsadminp -safemode leave