Introduction

Syllabus(about cloudera cca training)

| **Category** | **Topic** | **Description** |
| --- | --- | --- |
| Data Ingest | Sqoop | Understand sqoop import and export in detail |
| Data Ingest | Flume | Understand ingesting data into HDFS using Flume |
| Data Ingest | HDFS | Understand HDFS commands to copy data back and forth from HDFS |
| Transform, state and store | Spark with scala | Core Spark API such as read/write data, joins, aggregations, filters as well as sorting and ranking |
| Transform, state and store | Spark with Python | Core Spark API such as read/write data, joins, aggregations, filters as well as sorting and ranking |
| Data Analysis | Hive | Create tables and load data into tables using Hive |
| Data Analysis | Impala | Create tables and load data into tables using Impala |
| Data Analysis | avro-tools | Using avro-tools to convert into readable formats such as json, schema evolution etc |

### **Hadoop and Spark Introduction**

Command to restart Is sudo init 6

Apache Hadoop is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and should be automatically handled by the framework.

Typically Hadoop cluster can have few hundreds to few thousand nodes/physical servers. Setting up plain vanilla or Apache Hadoop and monitoring it can be tedious task and hence there are several distributions which provide tools for setting up and managing the clusters.

HDFS which is foundation to Hadoop provide solution for distributed storage, Map Reduce provide solution for distributed computing. Map Reduce works effectively to process large batches at scale, it does not work very well for interactive applications as well as micro batches. Spark - an in-memory distributed computing framework, can process data for micro batches as well as interactive applications effectively at scale. Lately Spark is gaining lot of momentum in large enterprises where enterprise hardware is used for infrastructure.  
  
There are several distributions of Hadoop supported by respective vendors.

* Cloudera
* Hortonworks
* MapR
* Amazon EMR

These vendors provide training, support and services for the clients. As part of their distribution these vendors provide tools to simplify setup process of the cluster as well as to simplify the operations.  
  
Hadoop eco system can be divided into core components and other tools. HDFS - Hadoop Distributed File System is the foundation in Hadoop eco system, Map Reduce is Distributed Computing framework developed in tandem with HDFS and other tools can be categorized into Map Reduce based tools and non Map Reduce based tools

* Hadoop core components
  + HDFS
  + Map Reduce
* Map Reduce based tools
  + Hive - Logical database on top of HDFS with SQL based interface on top of Map Reduce to process the data
  + Pig - Data flow language based interface on top of Map Reduce to process the data in HDFS
  + Sqoop - Generic data movement tool to copy data between relational databases and HDFS using Map Reduce leveraging its distributed processing capabilities
  + Mahout - Machine learning library which uses Map Reduce framework to process the data
  + Oozie - Map Reduce based work flow tool
* Non Map Reduce based tools
  + Flume - Data integration tool using Flume agents which can collect streaming data from sources such as weblogs and load into target such as HDFS
  + Spark - In memory data processing tool which can accelerate data processing
  + Impala - Alternative to hive to process lower volumes of data in quicker and interactive fashion
  + HBase - A NoSQL database to build applications which are operational in nature at scale

### **Hadoop Certifications and CCA**

Vendors such as Cloudera, Hortonworks etc package all these tools as part of distributions and provide wizards and tools to set up and maintain on larger clusters

Vendors such as Cloudera, Hortonworks not only provide training and support, they also issue certifications which are highly recognized in the industry. Most of the certifications are practical oriented which tests level of understanding of test takers.

Why should one certify?

* Tests level of understanding of several Hadoop eco system tools
* Instill confidence in individuals while delivering projects
* Certifications can give some traction in job search process
* Instills confidence in taking the interviews
* Separate certifications for separate roles
* Tests breadth and depth of eco system tools

What are the certifications that are available?

* Administrators
  + CCAH - Cloudera Certified Administrator of Apache Hadoop
  + HDPCA - Hortonworks Data Platform Certified Administrator
* Developers
  + CCA - Cloudera Certified Associate Spark and Hadoop Developer (HDFS, Sqoop, Flume, Spark with Python, Spark with Scala, Hive, Impala and Avro tools)
  + HDPCD - Hortonworks Data Platform Certified Developer (Flume, Hive, Pig and Sqoop)
  + HDPCD:Java - Hortonworks Data Platform Certified Developer (Java Map Reduce APIs)
* Data Engineers
  + CCP DE - Cloudera Certified Professional Data Engineer (Sqoop, Flume, Hive and Oozie)

#### **CCA Spark and Hadoop Developer**

* Certification covers below skills
  + Hadoop Distributed File System
  + Sqoop
  + Flume
  + Spark with Scala
  + Spark with Python
  + HIve
  + Impala
  + Avro tools

### **Setup Cloudera Quickstart VM**

* Prerequisites: i7, Quad Core, 16GB RAM (as we need to allocate at least 8 GB to 10 GB for Cloudera Quickstart VM)
* Download [Virtualbox](http://download.virtualbox.org/virtualbox/5.0.24/VirtualBox-5.0.24-108355-Win.exe) or[VMWare Workstation](https://www.vmware.com/in/products/workstation/workstation-evaluation) for Windows
* Download [Virtualbox](http://download.virtualbox.org/virtualbox/5.0.24/VirtualBox-5.0.24-108355-OSX.dmg) or [VMWare Fusion](http://www.vmware.com/in/products/fusion) for Mac
* Install Virtualization software chosen for your platform (Windows or Mac)
* Download latest version of [Cloudera Quickstart VM](http://www.cloudera.com/downloads/quickstart_vms/5-7.html)
* Unzip/Extract the virtual image
* Open with Virtualbox or VMWare
* Change settings - memory 8 or 10 GB (mandatory) and processors 4 (recommended)
* Cloudera Quickstart VM contains
  + Eclipse with Maven plugin for local development
  + MySQL Database
  + Hadoop eco system (Hadoop, Map Reduce, Hive, Pig, Sqoop etc)
* Follow the video to setup Cloudera Quickstart VM (Download latest version) Username: cloudera (for mysql it is root/retail\_dba)
* Password: cloudera
* Open firefox and validate cloudera manager (bookmarks are available in toolbar of firefox)
* Open terminal and validate mysql mysql -u retail\_dba -p (password: cloudera)

show databases;

use retail\_db;

show tables; -- should list 6 tables

select \* from departments;

mysql –u hive -p

* Follow below videos and validate other tools as well
* Other commands to be used in the process :
* For connecting to sqoop use
* Mysql –u retail\_dba –p
* touch testing1
* hadoop fs -put testing1 /user/cloudera

#### hadoop fs -ls /user/cloudera/testing1

* spark-shell
* impala-shell
* show databases;

### **Setup single node lab on cloud**

#### **Why lab on cloud?**

* While setting up cloudera quickstart VM is the easiest way of setting up the lab, it requires expensive laptops with higher configuration (i7 Quad Core, 16 GB RAM)
* Some people, especially students might not be able to afford it. Hence it will be better to use cloud platforms such as Amazon Web Service (AWS), Google Cloud, Microsoft Azure etc. Azure and Google Cloud provide some credits which can be leveraged to set up the environment.
* People need to be diligent to understand the cost associate with cloud provider and make sure the instance or server is stopped when not in use

#### **Setup Process**

* Sign up for cloud account
* Setup prerequisites for cloud account
* Steps to provision host on cloud, for AWS
  + Create VPC, key pair and take care of other prerequisites
  + Provision ec2 instance with 15 GB RAM, 4 or 8 cores and centos 6.5
  + Open up ports in security group (open all for convenience, in production only necessary ports should be exposed).
  + Make sure instance is running, resize root file system using resize2fs /dev/xvde
* Install cloudera manager
  + Login to host using ssh command

wget http://archive.cloudera.com/cm5/installer/latest/cloudera-manager-installer.bin

chmod +x cloudera-manager-installer.bin

sudo ./cloudera-manager-installer.bin

* Launch cloudera manager and install CDH
* Make sure cloudera manager is successfully installed
* Get public DNS from AWS or any other cloud provider
* Go to browser and run http://<public\_dns>:7180
* Accept enterprise trial license for 60 days
* Run hostname -f command on the cloud host
* In the wizard use private dns retrieved from previous command and then follow the wizard
* Validate all the tools
* Setup retail\_db and get-logs to practice sqoop and flume

Here is the playlist which covers all the steps to set up single node lab on public cloud (eg: AWS). Setup process is almost same for cloud host of any vendor

### **Using cloudxlab for learning Hadoop and Spark**

#### **Cloudxlab Introduction**

**Limitations of virtual machines (Cloudera QuickStart VM or Hortonworks Sandbox)**

* Requires premium laptops
* Long setup time if internet speed is slow
* Performance of laptop can become slow (as virtual machine need to have up to 10 GB memory and 4 cores)
* At times practicing single node VM might lead into confusion at the time of taking exam (ip addresses, Gateway etc)

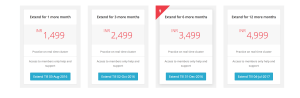
**Limitations of single node lab on AWS**

* Long setup time
* Can be expensive
* Monitor closely to stop the instance when not in use

Keeping the limitations of having virtual machines and single node lab, itversity, llc have partnered with [cloudxlab](https://cloudxlab.com/) to provide online cloud based lab for students.

**Advantages of cloudxlab**

* Accessible from any where over internet
* 6 node lab which might scale up
* Support from Hadoop experts and itversity
* Affordable pricing



#### **Signing up to cloudxlab**

Here is the [affiliation link](https://cloudxlab.com/affiliate/code/WOZZ13VCAPU9JS2VTTI8) and video for the instructions to use cloudxlab

### **Next Lesssons**

As we completed setting up the environment you need to go through below lessons to get confidence for the certification.

Certification curriculum talks about 3 major categories

* Data Ingest (Sqoop and Flume)
* Transform, Stage, Store (Spark with Python and Spark with Scala)
* Data Analysis (Hive, Impala and avro)

Based on the curriculum below lessons will be created to cover major topics

#### **Lesson 2 - Data Ingest using Sqoop**

* Import data from a MySQL database into HDFS using Sqoop
* Export data to a MySQL database from HDFS using Sqoop
* Change the delimiter and file format of data during import using Sqoop

#### **Lesson 3 - Data Ingest using Flume and HDFS**

* Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume
* Load data into and out of HDFS using the Hadoop File System (FS) commands

#### **Lesson 4 - Transform, Stage, Store using Spark with Python**

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

#### **Lesson 5 - Transform, Stage, Store using Spark with Scala**

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

#### **Lesson 6 - Data Analysis using Hive, Impala and avro**

* Read and/or create a table in the Hive metastore in a given schema
* Extract an Avro schema from a set of datafiles using avro-tools
* Create a table in the Hive metastore using the Avro file format and an external schema file
* Improve query performance by creating partitioned tables in the Hive metastore
* Evolve an Avro schema by changing JSON files

Data Ingest using Sqoop

Apache Sqoop is open source tool which can copy data between relational databases and Hadoop. Sqoop have several commands such as import, import-all-tables, export, eval, list-databases etc. For certification purpose sqoop import and sqoop export are important.

* Sqoop is open source and incubated under Apache
* It uses map reduce to connect to databases using JDBC and import/export data
* It is primarily developed using java
* Click [Sqoop Documentation (v1.4.5-cdh5.3.2)](http://archive-primary.cloudera.com/cdh5/cdh/5/sqoop-1.4.5-cdh5.3.2/) for documentation relevant to Certification

Before exploring the topics, make sure you have practice environment using one of the below:

* [Cloudera QuickStart VM](http://www.itversity.com/topic/setup-cloudera-quickstart-vm/)
* [Cloudera Single node lab on Cloud](http://www.itversity.com/topic/setup-single-node-lab-on-cloud/)
* [Cloudxlab](http://www.itversity.com/topic/using-cloudxlab-for-learning-hadoop-and-spark/) - Sign up below

### **Topics covered as part of this lesson**

### **Data Ingest**

The skills to transfer data between external systems and your cluster. This includes the following:

* Import data from a MySQL database into HDFS using Sqoop
* Export data to a MySQL database from HDFS using Sqoop
* Change the delimiter and file format of data during import using Sqoop

### **Apache Sqoop – Getting started**

* MySQL Database (user name: retail\_dba, password: cloudera). You can choose your own credentials if you use some thing else while setting up
* Database Name: retail\_db with tables departments, categories, products, order\_items, orders and customers
* Make sure HDFS and YARN are up and running
* Make sure you have user space /user/cloudera on QuickStart VM and single node lab. For cloudxlab, you have to use /user/<YOUR\_USER\_NAME> (in my case it is /user/training8634)

Make sure you run hostname -f in terminal to use in jdbc url. In exam Cloudera might give the hostname or they might say sample mysql database is in a specific host. If they do not specify, mysql might be running on gateway/client node

Here is the video to get started with sqoop commands such as list-databases, list-tables and eval

* sqoop list-databases connects to mysql and lists all the databases user have access to
* sqoop list-tables connects to mysql and lists all the tables in the database connected to
* sqoop eval connects to mysql and runs any query passed in the database user connected to

<https://www.youtube.com/watch?v=hY9nnU4PTFw>

flume-ng version

hive , avaro-tools , hbase-shell ,hbase for help ,list after spark 5.0 user has 755 privalages to run the program means

rwx = 111 in binary = 7

rw- = 110 in binary = 6

r-x = 101 in binary = 5

r-- = 100 in binary = 4

where r is read w is write and x is execute in the shell

as spark is owned by hdfs therefore we use the following commands in the shell

--sudo –u hdfs hadoop fs –chown –R 777 /user/spark

--sudo –u hdfs hadoop fs –chmod –R 777 /user/spark

spark -shell --master yarn

so the above fix for running the spark as yarn the master when hdfs was the default master of the given spark program and the permission of the shell were 755 for the spark context in the cloudera vm ware

cd /opt/examples/flume/

To check resource manager and node manager are running or not,then to check for hdfs and yarn,then finally check for mysql using the commands that were used earlier here ps stand for process status

ps -ef|grep -i manager , , ps -ef|grep -i node,ps -fu hdfs , ps -fu yarn

hadoop fs -ls /user/cloudera

Code snippets to list databases, list tables and run queries in mysql on quickstart VM

sqoop list-databases

--connect "jdbc:mysql://quickstart.cloudera:3306"

--username retail\_dba

--password cloudera

sqoop list-tables

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username retail\_dba

--password cloudera

sqoop eval

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username retail\_dba

--password cloudera

--query "select count(1) from order\_items"

Code snippets to list databases, list tables and run queries in single node lab  
<hostname> - output of hostname -f  
<password> - password you have chosen while setting up

sqoop list-databases

--connect "jdbc:mysql://<hostname>:3306"

--username retail\_dba

--password <password>

sqoop list-tables

--connect "jdbc:mysql://<hostname>:3306/retail\_db"

--username retail\_dba

--password <password>

sqoop eval

--connect "jdbc:mysql://<hostname>:3306/retail\_db"

--username retail\_dba

--password <password>

--query "select count(1) from order\_items"

Code snippets to list databases, list tables and run queries in cloudxlab

* [Click here](http://f.cloudxlab.com/) to launch cloudxlab web console
* Login to cloudxlab web console using your credential (signup required)

* Login to web console and run below sqoop commands to validate

sqoop list-databases

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306"

--username retail\_dba

--password itversity

sqoop list-tables

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username retail\_dba

--password itversity

sqoop eval

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username retail\_dba

--password itversity

--query "select count(1) from order\_items"

Available commands for the sqoop are as follows

Available commands:

codegen Generate code to interact with database records

create-hive-table Import a table definition into Hive

eval Evaluate a SQL statement and display the results

export Export an HDFS directory to a database table

help List available commands

import Import a table from a database to HDFS

import-all-tables Import tables from a database to HDFS

import-mainframe Import datasets from a mainframe server to HDFS

job Work with saved jobs

list-databases List available databases on a server

list-tables List available tables in a database

merge Merge results of incremental imports

metastore Run a standalone Sqoop metastore

version Display version information

### **Sqoop Import into Hadoop – Cloudera Quickstart VM or Single Node lab**

From CCA Spark and Hadoop Developer perspective, this topic covers highlighted task on Cloudera QuickStart VM. Most of the instructions are same for single node lab on cloud as well.

#### **Data Ingest**

The skills to transfer data between external systems and your cluster. This includes the following:

* **Import data from a MySQL database into HDFS using Sqoop**
* Export data to a MySQL database from HDFS using Sqoop
* Change the delimiter and file format of data during import using Sqoop
* Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume
* Load data into and out of HDFS using the Hadoop File System (FS) commands

#### **Credentials for single node lab on cloud**

* Hostname: run hostname -f and use in JDBC url
* Username: retail\_dba
* Password: what ever you have created while setting up the lab

Sqoop Import is the most important sqoop command. It uses map reduce framework to connect to database and copy data in parallel into HDFS.

Here are the videos to explore different options of import-all-tables and import to copy the data.also by default no of mapper is 4

Import all tables using sqoop-import

* Open terminal and run hadoop fs -mkdir /user/cloudera/sqoop\_import”
* Run sqoop import-all-tables as shown below

sqoop import-all-tables

-m 4

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--warehouse-dir=/user/cloudera/sqoop\_import

Or

sqoop import-all-tables

--num-mappers 4

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--warehouse-dir=/user/cloudera/sqoop\_import

Or working code for imports folder

sqoop import-all-tables \

-m 12 \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--warehouse-dir=/user/cloudera/sqoop\_imports

* Validate by running hadoop fs -ls -R /user/cloudera/sqoop\_import. It should list all the files that are imported
* Hadoop hadoop fs -ls  /user/cloudera/sqoop\_import to see all the databases
* hadoop fs -ls  /user/cloudera/sqoop\_import /orders to see all the tables in the order databases
* hadoop fs -tail  /user/cloudera/sqoop\_import /orders/part-m-00000 shows the records in text and comma format
* 5740,2013-08-29 00:00:00.0,554,PENDING
* 5741,2013-08-29 00:00:00.0,2571,COMPLETE
* "" at the end of each line is linux terminal line break to develop code in developer friendly manner
* Also you can use the cat command to see everything in details
* hadoop fs -cat /user/cloudera/sqoop\_imports/orders/part-m-00000
* hadoop fs -cat /user/cloudera/sqoop\_imports/orders/part-m-\*
* hadoop fs -cat /user/cloudera/sqoop\_imports/orders/part-m-\*|wc -l
* sqoop import-all-tables is the main command
* -m or --num-mappers signifies parallel threads
* --connect uses jdbc url to connect to remote database eg: mysql (use hostname -f to get proper dns name to be used)
* --username database user
* --password database password
* --warehouse-dir is base directory under which tables will be created

hadoop fs -rm -R /user/cloudera/sqoop\_import

sqoop import-all-tables

-m 12

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--as-avrodatafile

--warehouse-dir=/user/cloudera/sqoop\_import

* Make sure .avsc files created in the directory from where above command is run. ls -ltr \*.avsc
* Avro files generate metadata for the structure of data. It is required later.

**For using hive**

* Make sure hive is up and running using hive -e "create table testing (t int); insert into testing values (1); select count(1) from testing;"
* Use command to list all tables hadoop fs -ls /user/hive/warehouse
* In hive command line we use dfs -ls /user/hive/warehouse
* Run show databases , use default , show tables
* Run below sqoop import-all-tables command to load data into hive default database

sqoop import-all-tables

--num-mappers 1

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--hive-import

--hive-overwrite

--create-hive-table

--compress

--compression-codec org.apache.hadoop.io.compress.SnappyCodec

--outdir java\_files

* --hive-import to perform import in hive mode
* --hive-overwrite to delete existing data in hive tables and load into them
* --create-hive-table to create hive table if it does not exists
* --compress to load data in compression mode
* --compression-codec to use specific compression algorithm
* --outdir to redirect java files to a different directory

Import all tables using sqoop-import into existing hive database (retail\_db)

To validate wirte the following things in the protion describe formatted departments , dfs -du –s –h /user/cloudera/sqoop\_import/departments

* Create hive database hive -e "CREATE DATABASE IF NOT EXISTS retail\_stage"
* Run below sqoop import-all-tables command to load data into existing hive database

sqoop import-all-tables

--num-mappers 1

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--hive-import

--hive-overwrite

--create-hive-table

--outdir java\_files

--hive-database retail\_stage

Or

sqoop import-all-tables \

--num-mappers 1 \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--hive-import\

--hive-overwrite \

--create-hive-table \

--outdir java\_files \

--hive-database retail\_stage

* Validate by running this query hive -e "USE retail\_stage; SHOW TABLES; SELECT \* FROM departments;"

Import single table departments using sqoop-import into HDFS location /user/cloudera/departments

* Make sure directory /user/cloudera/departments does not exist. Run hadoop fs -rm -R/user/cloudera/deparments to delete the directory (if already exists)
* Run sqoop import command with --table and --target-directory options

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

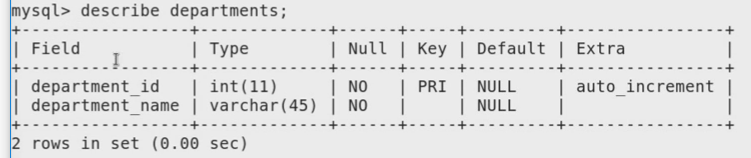
--password=cloudera

--table departments

--as-textfile

--target-dir=/user/cloudera/sqoop\_import/departments

Here describe departments will show the detail of the tables



* Now for checking use the following commands

hadoop fs -ls /user/cloudera/departments , hadoop fs -cat /user/cloudera/departments/part\*

* --as-textfile (default) to store data in HDFS using text file format. Other valid formats are

|  |  |
| --- | --- |
| --as-avrodatafile | Imports data to Avro Data Files |
| --as-sequencefile | Imports data to SequenceFiles |
| --as-textfile | Imports data as plain text (default) |
| --as-parquetfile | Imports data to Parquet Files (from 1.4.6) |

* --target-dir to specify target location where data need to be stored
* To check the data the following commands are used sqoop eval --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" --username=retail\_dba --password=cloudera --query "insert into departments value (8000,'TESTING')"
* To delete the file from the location hadoop fs -rm -R /user/cloudera/departments
* sqoop import --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" --username=retail\_dba --password=cloudera --table departments --target-dir /user/cloudera/departments -m 2 --boundary-query "select min(department\_id),max(department\_id) from departments where department\_id <> 8000" for evenly distrubiuting
* for removing the directory use hadoop fs -rm -R /user/cloudera/departments
* --warehouse-dir /user/cloudera/sqoop\_import (in place of --target-dir/user/cloudera/sqoop\_import/departments of above command) will have same impact as above command. When specified sqoop will create directory with the name of the table under directory passed with --warehouse-dir
* --boundary-query can be added to above sqoop command to handle out-lies. It is used to address skew and process data more evenly across multiple threads.
* --columns can be used to pull data only from sub set of the columns
* Here is the command which uses boundary-query and columns

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--target-dir /user/cloudera/departments

-m 2

--boundary-query "select 2, 8 from departments limit 1"

--columns department\_id,department\_name

* --split-by can be used to use multiple threads in case there is no primary key or unique key in the table from source database. If --split-by is not used we should pass --num-mappers 1
* For the purpose of testing these query use the following things in the section

mysql> create table departments\_nopk as select \* from departments; this creates a table with no primary key so if we imprort by using then it doesn’t work we need to split the table

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments\_nopk \

--target-dir /user/cloudera/order\_join

To run it one must specify the –split and the no of mappers to run smoothly

* --query can be used to pass custom query to import the data
* Here is the command which uses --split-by and --query to get the join output between orders and order\_items to HDFS. where $CONDITIONS is mandatory when --query is used.

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--query="select \* from orders join order\_items on orders.order\_id = order\_items.order\_item\_order\_id where $CONDITIONS"

--target-dir /user/cloudera/order\_join

--split-by order\_id

--num-mappers 4

Or

sqoop import \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--table departments \

--target-dir /user/hive/warehouse/retail\_ods.db/departments \

--append \

--fields-terminated-by '|' \

--lines-terminated-by '\n' \

--split-by department\_id \

--outdir java\_files

* --where can be used to apply conditions to import the data

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--target-dir /user/hive/warehouse/retail\_ods.db/departments

--append

--fields-terminated-by '|'

--lines-terminated-by 'n'

--split-by department\_id

--where "department\_id > 7"

--outdir java\_files

Or

sqoop import \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--query="select \* from orders join order\_items on orders.order\_id = order\_items.order\_item\_order\_id where \$CONDITIONS" \

--target-dir /user/cloudera/order\_join \

--split-by order\_id \

--num-mappers 4

Or

sqoop import \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--query="select \* from orders join order\_items on orders.order\_id = order\_items.order\_item\_order\_id where \$CONDITIONS" \

--target-dir /user/cloudera/order\_join \

--split-by order\_id \

--num-mappers 4

Import single table into Hive default database

* Create table in hive (launch hive and run this command)
* Creating a table command create database sqoop\_import , use sqoop\_import , show tables; ,
* create table departments (department\_id int, department\_name string);
* describe formatted departments;
* Location of hdfs is hdfs://quickstart.cloudera:8020/user/hive/warehouse/sqoop\_import.db/departments
* dfs -ls /user/hive/warehouse/sqoop\_import.db/departments
* select \* from departments
* exit
* and now import in this table using hive command
* sqoop import \
* --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \
* --username=retail\_dba \
* --password=cloudera \
* --table departments \
* --fields-terminated-by '|' \
* --lines-terminated-by '\n' \
* --hive-home /user/hive/warehouse/retail\_ods.db \
* --hive-import \
* --hive-overwrite \
* --hive-table departments \
* --outdir java\_files

CREATE TABLE departments (

department\_id INT,

department\_name STRING

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

SHOW TABLES;

* Here is the sample command to perform hive import into an existing table

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--fields-terminated-by '|'

--lines-terminated-by 'n'

--hive-home /user/hive/warehouse

--hive-import

--hive-overwrite

--hive-table departments

--outdir java\_files

Or we can use

sqoop import \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--table departments \

--hive-home /user/hive/warehouse \

--hive-import \

--hive-table sqoop\_import.departments\_test \

--outdir java\_files

Select \* from departments\_test

* --fields-terminated-by should match hive table field terminator specified in ROW FORMAT DELIMITED FIELDS TERMINATED BY
* --hive-table to specify the target table
* --create-hive-table can be used to create hive table at the time of import in case it does not exist.

Import single table into Hive retail\_stage database

* Create table in hive (launch hive and run this command)

CREATE DATABASE IF NOT EXISTS retail\_stage;

USE retail\_stage;

CREATE TABLE departments (

department\_id INT,

department\_name STRING

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

SHOW TABLES;

* Here is the sample command

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--fields-terminated-by '|'

--lines-terminated-by 'n'

--hive-home /user/hive/warehouse

--hive-import

--hive-overwrite

--hive-table departments

--hive-database retail\_stage

--outdir java\_files

* Database name can be prefixed to the table name (eg: --hive-table retail\_db.departments, when used --hive-database should be removed)
* Now for incremental loading of the data in the sqoop we do the following things
* sqoop eval --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" --username=retail\_dba --password=cloudera --query "select \* from departments"
* hadoop fs -ls /user/cloudera/sqoop\_import/departments
* hadoop fs -cat /user/cloudera/sqoop\_import/departments/part\*
* sqoop import \
* --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \
* --username=retail\_dba \
* --password=cloudera \
* --table departments \
* --target-dir /user/cloudera/sqoop\_import/departments \
* --append \
* --where "department\_id>7" \
* --outdir java\_files

Or

sqoop import \

* --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \
* --username=retail\_dba \
* --password=cloudera \
* --table departments \
* --target-dir /user/cloudera/sqoop\_import/departments \
* --append \
* --check-column "department\_id" \
* --incremental append \
* --last-value 7 \
* --outdir java\_files

**Here is the sample script to perform incremental load and merge data**

* --Merge process begins
* hadoop fs -mkdir /user/cloudera/sqoop\_merge
* --Initial load
* sqoop import
* --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"
* --username=retail\_dba
* --password=cloudera
* --table departments
* --as-textfile
* --target-dir=/user/cloudera/sqoop\_merge/departments
* --Validate
* sqoop eval --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"
* --username retail\_dba
* --password cloudera
* --query "select \* from departments"
* hadoop fs -cat /user/cloudera/sqoop\_merge/departments/part\*
* --update
* sqoop eval --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"
* --username retail\_dba
* --password cloudera
* --query "update departments set department\_name='Testing Merge' where department\_id = 9000"
* --Insert
* sqoop eval --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"
* --username retail\_dba
* --password cloudera
* --query "insert into departments values (10000, 'Inserting for merge')"
* sqoop eval --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"
* --username retail\_dba
* --password cloudera
* --query "select \* from departments"
* --New load
* sqoop import
* --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"
* --username=retail\_dba
* --password=cloudera
* --table departments
* --as-textfile
* --target-dir=/user/cloudera/sqoop\_merge/departments\_delta
* --where "department\_id >= 9000"
* hadoop fs -cat /user/cloudera/sqoop\_merge/departments\_delta/part\*
* --Merge
* sqoop merge --merge-key department\_id
* --new-data /user/cloudera/sqoop\_merge/departments\_delta
* --onto /user/cloudera/sqoop\_merge/departments
* --target-dir /user/cloudera/sqoop\_merge/departments\_stage
* --class-name departments
* --jar-file
* hadoop fs -cat /user/cloudera/sqoop\_merge/departments\_stage/part\*
* --Delete old directory
* hadoop fs -rm -R /user/cloudera/sqoop\_merge/departments
* --Move/rename stage directory to original directory
* hadoop fs -mv /user/cloudera/sqoop\_merge/departments\_stage /user/cloudera/sqoop\_merge/departments
* --Validate that original directory have merged data
* hadoop fs -cat /user/cloudera/sqoop\_merge/departments/part\*
* --Merge process ends

### **Sqoop Import into Hadoop – cloudxlab**

#### **Introduction**

From CCA Spark and Hadoop Developer perspective, this topic covers highlighted task on cloudxlab

#### **Data Ingest**

The skills to transfer data between external systems and your cluster. This includes the following:

* **Import data from a MySQL database into HDFS using Sqoop**
* Export data to a MySQL database from HDFS using Sqoop
* Change the delimiter and file format of data during import using Sqoop
* Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume
* Load data into and out of HDFS using the Hadoop File System (FS) commands

#### **Cloudxlab credentials (after**[**signing up**](https://cloudxlab.com/affiliate/code/WOZZ13VCAPU9JS2VTTI8)**using itversity affiliation code)**

#### **Importing data into Hadoop using Sqoop on cloudxlab**

Sqoop Import is the most important sqoop command. It uses map reduce framework to connect to database and copy data in parallel into HDFS.

Here are the videos to explore different options of import-all-tables and import to copy the data. **For HDFS directory location use your id in place of training8634**

Import all tables using sqoop-import

* Open terminal and run hadoop fs -mkdir /user/training8634/sqoop\_import
* Run sqoop import-all-tables as shown below

sqoop import-all-tables

-m 4

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--warehouse-dir=/user/training8634/sqoop\_import

* Validate by running hadoop fs -ls -R /user/training8634/sqoop\_import. It should list all the files that are imported
* "" at the end of each line is linux terminal line break to develop code in developer friendly manner
* sqoop import-all-tables is the main command
* -m or --num-mappers signifies parallel threads
* --connect uses jdbc url to connect to remote database eg: mysql (use hostname -f to get proper dns name to be used)
* --username database user
* --password database password
* --warehouse-dir is base directory under which tables will be created
* Run this command to load data into avro format

hadoop fs -rm -R /user/training8634/sqoop\_import

sqoop import-all-tables

-m 12

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--as-avrodatafile

--warehouse-dir=/user/training8634/sqoop\_import

* Make sure .avsc files created in the directory from where above command is run. ls -ltr \*.avsc
* Avro files generate metadata for the structure of data. It is required later.

Import all tables using sqoop-import into default hive database

* Make sure hive is up and running using hive -e "create table testing (t int); insert into t values (1); select count(1) from testing;"
* Run below sqoop import-all-tables command to load data into hive default database

sqoop import-all-tables

--num-mappers 1

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--hive-import

--hive-overwrite

--create-hive-table

--compress

--compression-codec org.apache.hadoop.io.compress.SnappyCodec

--outdir java\_files

* --hive-import to perform import in hive mode
* --hive-overwrite to delete existing data in hive tables and load into them
* --create-hive-table to create hive table if it does not exists
* --compress to load data in compression mode
* --compression-codec to use specific compression algorithm
* --outdir to redirect java files to a different directory

Import all tables using sqoop-import into existing hive database (retail\_db)

* Create hive database hive -e "CREATE DATABASE IF NOT EXISTS retail\_stage"
* Run below sqoop import-all-tables command to load data into existing hive database

sqoop import-all-tables

--num-mappers 1

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--hive-import

--hive-overwrite

--create-hive-table

--outdir java\_files

--hive-database retail\_stage

* Validate by running this query hive -e "USE retail\_stage; show tables; select \* from departments;"

Import single table departments using sqoop-import into HDFS location /user/training8634/departments

* Make sure directory /user/training8634/departments does not exist. Run hadoop fs -rm -R/user/cloudera/deparments to delete the directory (if already exists)
* Run sqoop import command with --table and --target-directory options

sqoop import

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--table departments

--as-textfile

--target-dir=/user/training8634/sqoop\_import/departments

* --as-textfile (default) to store data in HDFS using text file format. Other valid formats are

|  |  |
| --- | --- |
| --as-avrodatafile | Imports data to Avro Data Files |
| --as-sequencefile | Imports data to SequenceFiles |
| --as-textfile | Imports data as plain text (default) |
| --as-parquetfile | Imports data to Parquet Files (from 1.4.6) |

* --target-dir to specify target location where data need to be stored
* --warehouse-dir /user/training8634/sqoop\_import (in place of --target-dir/user/cloudera/sqoop\_import/departments of above command) will have same impact as above command. When specified sqoop will create directory with the name of the table under directory passed with --warehouse-dir
* --boundary-query can be added to above sqoop command to handle out-lies. It is used to address skew and process data more evenly across multiple threads.
* --columns can be used to pull data only from sub set of the columns
* Here is the command which uses boundary-query and columns

sqoop import

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--table departments

--target-dir /user/training8634/departments

-m 2

--boundary-query "select 2, 8 from departments limit 1"

--columns department\_id,department\_name

* --split-by can be used to use multiple threads in case there is no primary key or unique key in the table from source database. If --split-by is not used we should pass --num-mappers 1
* --query can be used to pass custom query to import the data
* Here is the command which uses --split-by and --query to get the join output between orders and order\_items to HDFS. where $CONDITIONS is mandatory when --query is used.

sqoop import

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--query="select \* from orders join order\_items on orders.order\_id = order\_items.order\_item\_order\_id where $CONDITIONS"

--target-dir /user/training8634/order\_join

--split-by order\_id

--num-mappers 4

* --where can be used to apply conditions to import the data
* cloudxlab uses hortonworks distribution. Default hive warehouse location in hortonworks is /apps/hive/warehouse

sqoop import

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--table departments

--target-dir /apps/hive/warehouse/retail\_stage.db/departments

--append

--fields-terminated-by '|'

--lines-terminated-by 'n'

--split-by department\_id

--where "department\_id > 7"

--outdir java\_files

Import single table into Hive default database

* Create table in hive (launch hive and run this command)

CREATE TABLE departments (

department\_id INT,

department\_name STRING

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

SHOW TABLES;

* Here is the sample command to perform hive import into an existing table

sqoop import

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--table departments

--fields-terminated-by '|'

--lines-terminated-by 'n'

--hive-home /apps/hive/warehouse

--hive-import

--hive-overwrite

--hive-table departments

--outdir java\_files

* --fields-terminated-by should match hive table field terminator specified in ROW FORMAT DELIMITED FIELDS TERMINATED BY
* --hive-table to specify the target table
* --create-hive-table can be used to create hive table at the time of import in case it does not exist.

Import single table into Hive retail\_stage database

* Create table in hive (launch hive and run this command)

CREATE DATABASE IF NOT EXISTS retail\_db;

USE retail\_db;

CREATE TABLE departments (

department\_id INT,

department\_name STRING

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

SHOW TABLES;

* Here is the sample command

sqoop import

--connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username=retail\_dba

--password=itversity

--table departments

--fields-terminated-by '|'

--lines-terminated-by 'n'

--hive-home /apps/hive/warehouse

--hive-import

--hive-overwrite

--hive-table departments

--hive-database retail\_stage

--outdir java\_files

* Database name can be prefixed to the table name (eg: --hive-table retail\_db.departments, when used --hive-database should be removed)
* **Here is the sample script to perform incremental load and merge data**
* --Merge process begins
* hadoop fs -mkdir /user/training8634/sqoop\_merge
* --Initial load
* sqoop import
* --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"
* --username=retail\_dba
* --password=itversity
* --table departments
* --as-textfile
* --target-dir=/user/training8634/sqoop\_merge/departments
* --Validate
* sqoop eval --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"
* --username retail\_dba
* --password itversity
* --query "select \* from departments"
* hadoop fs -cat /user/training8634/sqoop\_merge/departments/part\*
* --update
* sqoop eval --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"
* --username retail\_dba
* --password itversity
* --query "update departments set department\_name='Testing Merge' where department\_id = 9000"
* --Insert
* sqoop eval --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"
* --username retail\_dba
* --password itversity
* --query "insert into departments values (10000, 'Inserting for merge')"
* sqoop eval --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"
* --username retail\_dba
* --password itversity
* --query "select \* from departments"
* --New load
* sqoop import
* --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"
* --username=retail\_dba
* --password=itversity
* --table departments
* --as-textfile
* --target-dir=/user/training8634/sqoop\_merge/departments\_delta
* --where "department\_id >= 9000"
* hadoop fs -cat /user/training8634/sqoop\_merge/departments\_delta/part\*
* --Merge
* sqoop merge --merge-key department\_id
* --new-data /user/training8634/sqoop\_merge/departments\_delta
* --onto /user/training8634/sqoop\_merge/departments
* --target-dir /user/training8634/sqoop\_merge/departments\_stage
* --class-name departments
* --jar-file
* hadoop fs -cat /user/training8634/sqoop\_merge/departments\_stage/part\*
* --Delete old directory
* hadoop fs -rm -R /user/training8634/sqoop\_merge/departments
* --Move/rename stage directory to original directory
* hadoop fs -mv /user/training8634/sqoop\_merge/departments\_stage /user/cloudera/sqoop\_merge/departments
* --Validate that original directory have merged data
* hadoop fs -cat /user/training8634/sqoop\_merge/departments/part\*
* --Merge process ends

### **Sqoop export from hadoop cloudera quickstart vm or single node lab**

From CCA Spark and Hadoop Developer perspective, this topic covers highlighted task on Cloudera QuickStart VM

#### **Data Ingest**

The skills to transfer data between external systems and your cluster. This includes the following:

* Import data from a MySQL database into HDFS using Sqoop
* **Export data to a MySQL database from HDFS using Sqoop**
* Change the delimiter and file format of data during import using Sqoop
* Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume
* Load data into and out of HDFS using the Hadoop File System (FS) commands

#### **Credentials for single node lab on cloud**

* Hostname: run hostname -f and use in JDBC url
* Username: retail\_dba
* Password: what ever you have created while setting up the lab

Sqoop Export uses map reduce framework to connect to database and copy data in parallel into HDFS. Here are the videos for the reference which explain sqoop export in detail

Running sqoop-export command

* Create table in mysql (using DB credentials retail\_dba and cloudera

mysql -u retail\_dba -p

USE retail\_db;

CREATE TABLE departments\_export AS SELECT \* FROM retail\_db.departments WHERE 1=2;

* Run sqoop export command

sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username retail\_dba \

--password cloudera \

--table departments \

--export-dir /user/hive/warehouse/retail\_ods.db/departments \

--input-fields-terminated-by '|' \

--input-lines-terminated-by 'n' \

--num-mappers 2 \

--outdir java\_files

* --export-dir to specify the HDFS directory from which data needs to be exported
* --input-fields-terminated-by to specify the field delimiter used while storing data into HDFS
* --lines-terminated-by to specify the row delimiter used while storing data into HDFS
* Other parameters have the same meaning as in import time

Running sqoop-export in update or upsert mode

sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username retail\_dba \

--password cloudera \

--table departments \

--export-dir /user/cloudera/sqoop\_import/departments\_export \

--batch \

--outdir java\_files \

-m 1 \

--update-key department\_id \

--update-mode allowinsert

* --update-key is used to tell on what column data needs to be looked up in the relational database (mysql) table to perform update of already existing data
* --update-mode is used to insert non-existing data
* If --update-mode is not set to allowinsert then data will only be updating matched data, unmatched data will be ignored.

Running sqoop-export by specifying delimiters

sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username retail\_dba \

--password cloudera \

--table departments\_test \

--export-dir /user/hive/warehouse/departments\_test \

--input-fields-terminated-by '�01' \

--input-lines-terminated-by 'n' \

--num-mappers 2 \

--batch \

--outdir java\_files \

--input-null-string nvl \

--input-null-non-string -1

* --input-null-string is used to tell how nulls are stored in alpha-numeric fields in HDFS. If a field in a row in HDFS have string nvl, then null will be inserted in the database table
* --input-null-non-string is used to tell how nulls are stored in numeric fields in HDFS. If a field in a row in HDFS have numeric value -1, then null will be inserted in the database table
* --input-fields-terminated-by '�01' is example about non alpha-numeric delimiters for data in HDFS
* Sqoop Expor – merge upsert
  + vi departments\_export

inside that write 7,fanshop 9000,testing export

* + hadoop fs -mkdir /user/cloudera/sqoop\_import/departments\_export/
  + hadoop fs -put /home/cloudera/departments\_export /user/cloudera/sqoop\_import/departments\_export/
  + sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \
  + --username retail\_dba \
  + --password cloudera \
  + --table departments \
  + --export-dir /user/cloudera/sqoop\_import/departments\_export \
  + --batch \
  + --outdir java\_files \
  + -m 1 \
  + --update-key department\_id \
  + --update-mode allowinsert
* hadoop fs -rm -R sqoop\_import/departments\_export
* vi departments\_export,chnge the value to , Fan shop 10000
* mv departments\_export export.csv to change the filename
* hadoop fs -put /home/cloudera/export.csv /user/cloudera/sqoop\_import/departments\_export
* sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \
* --username retail\_dba \
* --password cloudera \
* --table departments \
* --export-dir /user/cloudera/sqoop\_import/departments\_export \
* --batch \
* --outdir java\_files \
* -m 1 \
* --update-key department\_id

create table departments\_export as select \* from departments;

sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" --username retail\_dba --password cloudera --table departments\_export --export-dir /user/cloudera/sqoop\_import/departments\_export --batch --outdir java\_files -m 1 --update-key department\_id --update-mode allowinsert

### **sqoop export from hadoop cloudxlab**

#### **Introduction**

From CCA Spark and Hadoop Developer perspective, this topic covers highlighted task on Cloudxlab

#### **Data Ingest**

The skills to transfer data between external systems and your cluster. This includes the following:

* Import data from a MySQL database into HDFS using Sqoop
* **Export data to a MySQL database from HDFS using Sqoop**
* Change the delimiter and file format of data during import using Sqoop
* Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume
* Load data into and out of HDFS using the Hadoop File System (FS) commands

#### **Cloudxlab credentials (after**[**signing up**](https://cloudxlab.com/affiliate/code/WOZZ13VCAPU9JS2VTTI8)**using itversity affiliation code)**

#### **Exporting data from Hadoop using Sqoop on Cloudxlab**

Sqoop Export uses map reduce framework to connect to database and copy data in parallel into HDFS. Here are the videos for the reference which explain sqoop export in detail

Running sqoop-export command

* Create table in mysql (using DB credentials retail\_dba and itversity)

mysql -u retail\_dba -p

USE retail\_db;

CREATE TABLE departments\_export AS SELECT \* FROM retail\_db.departments WHERE 1=2;

* Run sqoop export command

sqoop export --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username retail\_dba

--password itversity

--table departments

--export-dir /apps/hive/warehouse/retail\_stage.db/departments

--input-fields-terminated-by '|'

--input-lines-terminated-by 'n'

--num-mappers 2

--outdir java\_files

* --export-dir to specify the HDFS directory from which data needs to be exported
* --input-fields-terminated-by to specify the field delimiter used while storing data into HDFS
* --lines-terminated-by to specify the row delimiter used while storing data into HDFS
* Other parameters have the same meaning as in import time

Running sqoop-export in update or upsert mode

* Create data in HDFS under /user/trining8634/sqoop\_import/departments\_export as shown in the video
* Make sure there are additional as well as modified records in the HDFS directory. Use video for reference.

sqoop export --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username retail\_dba

--password cloudera

--table departments

--export-dir /user/training8634/sqoop\_import/departments\_export

--batch

--outdir java\_files

-m 1

--update-key department\_id

--update-mode allowinsert

* --update-key is used to tell on what column data needs to be looked up in the relational database (mysql) table to perform update of already existing data
* --update-mode is used to insert non-existing data
* If --update-mode is not set to allowinsert then data will only be updating matched data, unmatched data will be ignored.

Running sqoop-export by specifying delimiters

* Generate test data and load into /apps/hive/warehouse/departments\_test as shown in the video

sqoop export --connect "jdbc:mysql://ip-172-31-13-154.ec2.internal:3306/retail\_db"

--username retail\_dba

--password itversity

--table departments\_test

--export-dir /apps/hive/warehouse/departments\_test

--input-fields-terminated-by '01'

--input-lines-terminated-by 'n'

--num-mappers 2

--batch

--outdir java\_files

--input-null-string nvl

--input-null-non-string -1

* --input-null-string is used to tell how nulls are stored in alpha-numeric fields in HDFS. If a field in a row in HDFS have string nvl, then null will be inserted in the database table
* --input-null-non-string is used to tell how nulls are stored in numeric fields in HDFS. If a field in a row in HDFS have numeric value -1, then null will be inserted in the database table
* --input-fields-terminated-by '01' is example about non alpha-numeric delimiters for data in HDFS

### **Sqoop delimiters file formats**

#### **Introduction**

From CCA Spark and Hadoop Developer perspective, this topic covers highlighted task for QuickStart VM, Single node lab as well as Cloudxlab

#### **Data Ingest**

The skills to transfer data between external systems and your cluster. This includes the following:

* Import data from a MySQL database into HDFS using Sqoop
* Export data to a MySQL database from HDFS using Sqoop
* **Change the delimiter and file format of data during import using Sqoop**
* Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume
* Load data into and out of HDFS using the Hadoop File System (FS) commands

#### **Cloudxlab credentials (after**[**signing up**](https://cloudxlab.com/affiliate/code/WOZZ13VCAPU9JS2VTTI8)**using itversity affiliation code)**

#### **Handling delimiters**

**2 imp calls are fileds terminated by and lines terminated by**

**sqoop import \**

**--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \**

**--username=retail\_dba \**

**--password=cloudera \**

**--table departments \**

**--target-dir /user/cloudera/sqoop\_import/departments\_enclosedby \**

**--enclosed-by \"**

**Enclosed by used for giving a double couted value**



Or

**sqoop import \**

**--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \**

**--username=retail\_dba \**

**--password=cloudera \**

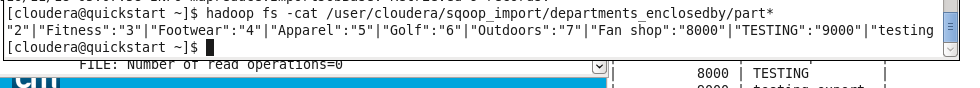
**--table departments \**

**--target-dir /user/cloudera/sqoop\_import/departments\_enclosedby \**

**--enclosed-by \" \**

**--fields-terminated-by \| \**

**--lines-terminated-by \:**



Or

**sqoop import \**

**--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \**

**--username=retail\_dba \**

**--password=cloudera \**

**--table departments \**

**--hive-home /user/hive/warehouse \**

**--hive-import \**

**--hive-table departments\_test \**

**--create-hive-table \**

**--outdir java\_files**

**describe formatted departments\_test;**

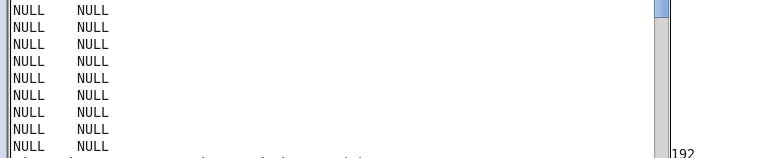
**hadoop fs -cat hdfs://quickstart.cloudera:8020/user/hive/warehouse/departments\_test/part\***

**drop table departments\_test; create table departments\_test(department\_id int,department\_name string); describe formatted departments\_test**

**sqoop import --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" --username=retail\_db --password=cloudera --table departments\_test--target-dir /user/hive/warehouse/departments\_test --append**

**select \* from departments\_test;**

**describe formatted departments\_test;**



For null string --null-string <null-string> and --null-non-string <null-string> ,now in mysql

**drop table departments\_test; create table departments\_test(department\_id integer ,department\_name varchar (30)); insert into departments\_test select \* from departments;, select \* from departments\_test;** **insert into departments\_test values(null,null);**

**sqoop import \**

**--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \**

**--username=retail\_dba \**

**--password=cloudera \**

**--table departments\_test \**

**--hive-home /user/hive/warehouse**

**--hive-import \**

**--hive-table departments\_test \**

**--create-hive-table \**

**--outdir java\_files \**

**--num-mappers 1 \**

**--split-by i need to figure out\**

**--null-string nvl \**

**--null-non-string -1**

Structured data in HDFS need to have delimiters defined for the columns/fields. We typically use Sqoop to export or import data between relational databases like Oracle, MySQL etc. Tables in relational databases have fixed schema/structure. While the data storage for typical relational database tables is proprietary to the underlying database, in Hadoop data storage and schema are decoupled. Hence we need to emphasize on delimiters while copying data between Relational Database and Hadoop using technologies like Sqoop. Also default delimiters between Sqoop and Hive are different.

Below are the videos which cover delimiters in detail.

Sqoop code to use delimiters while import

* Create table in hive (launch hive and run this command)

CREATE DATABASE IF NOT EXISTS retail\_stage;

USE retail\_stage;

CREATE TABLE departments (

department\_id INT,

department\_name STRING

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

SHOW TABLES;

()

* Here is the sample command

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--fields-terminated-by '|'

--lines-terminated-by 'n'

--hive-home /user/hive/warehouse

--hive-import

--hive-overwrite

--hive-table departments

--hive-database retail\_stage

--outdir java\_files

* Database name can be prefixed to the table name (eg: --hive-table retail\_db.departments, when used --hive-database should be removed)
* Fields for data that is imported to HDFS will be delimited by '|'
* Records or rows for data that is imported to HDFS will be delimited by 'n'. It is default
* Above example copies data to hive table in which fields are delimited by '|'

**Sqoop code to use delimiters while import**

* Create table in mysql (using DB credentials retail\_dba and cloudera

mysql -u retail\_dba -p

USE retail\_db;

CREATE TABLE departments\_export AS SELECT \* FROM retail\_db.departments WHERE 1=2;

* Run sqoop export command

sqoop export --connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username retail\_dba

--password cloudera

--table departments

--export-dir /user/hive/warehouse/retail\_ods.db/departments

--input-fields-terminated-by '|'

--input-lines-terminated-by 'n'

--num-mappers 2

--outdir java\_files

--input-null-non-string -1

* --export-dir to specify the HDFS directory from which data needs to be exported
* --input-fields-terminated-by to specify the field delimiter used while storing data into HDFS
* --lines-terminated-by to specify the row delimiter used while storing data into HDFS
* Other parameters have the same meaning as in import time

For viewing the file following Is used

view part –m-0000\*

cat part –m-0000\*

#### **Handling File Formats**

By default data is stored in text format, but at times we might have to use different file formats for variety of reasons such as security, performance, dealing with special characters etc. Sqoop supports text file, avro file, sequence file and in the latest version parquet file.

Below video covers how to deal with file formats.

Sample code to import data using different file formats

* Same code will run in cloudxlab by making necessary changes to jdbc url as well as directory path.
* Make sure to run hadoop fs -rm -R /user/cloudera/departments to clean up directory before rerunning the code with different file formats.
* Code to import data in text file format, it is not mandatory to specify --as-textfile as text file format is default

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--as-textfile

--target-dir=/user/cloudera/departments

We can also use –warehouse-dir but it will create a copy if departmets folder is already there

* Code to import data in sequence file format

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--as-sequencefile

--target-dir=/user/cloudera/departments

* Code to import data in avro file format

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--as-avrodatafile

--target-dir=/user/cloudera/departments

-- A file with extension avsc will be created under the directory from which sqoop import is executed

-- Copy avsc file to HDFS location

-- Create hive table with LOCATION to /user/cloudera/departments and TBLPROPERTIES pointing to avsc file

hadoop fs -put sqoop\_import\_departments.avsc /user/cloudera

CREATE EXTERNAL TABLE departments

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/cloudera/departments'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/sqoop\_import\_departments.avsc');

* Code to import data in parquet file format, available in only latest versions

sqoop import

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--table departments

--as-parquetfile

--target-dir=/user/cloudera/departments

* Creating a sqoop job

sqoop job --create sqoop\_job \

-- import \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--table departments \

--target-dir /user/hive/warehouse/retail\_ods.db/departments \

--append \

--fields-terminated-by '|' \

--lines-terminated-by '\n' \

--check-column "department\_id" \

--incremental append \

--last-value 7 \

--outdir java\_files

For listing and other stuff do the following in the terminal

sqoop job --list

sqoop job --show sqoop\_job

sqoop job --exec sqoop\_job

For sqoop merge

--Merge

sqoop merge --merge-key department\_id \

--new-data /user/cloudera/sqoop\_merge/departments\_delta \

--onto /user/cloudera/sqoop\_merge/departments \

--target-dir /user/cloudera/sqoop\_merge/departments\_stage \

--class-name departments \

--jar-file <get\_it\_from\_last\_import>

Cd java\_files to go in java files

For interview questions

ear readers, these **Sqoop Interview Questions** have been designed specially to get you acquainted with the nature of questions you may encounter during your interview for the subject of **Sqoop**. As per my experience good interviewers hardly plan to ask any particular question during your interview, normally questions start with some basic concept of the subject and later they continue based on further discussion and what you answer:

What is the role of JDBC driver in a Sqoop set up?

To connect to different relational databases sqoop needs a connector. Almost every DB vendor makes this connecter available as a JDBC driver which is specific to that DB. So Sqoop needs the JDBC driver of each of the database it needs to inetract with.

Is JDBC driver enough to connect sqoop to the databases?

No. Sqoop needs both JDBC and connector to connect to a database.

When to use --target-dir and when to use --warehouse-dir while importing data?

To specify a particular directory in HDFS use --target-dir but to specify the parent directory of all the sqoop jobs use --warehouse-dir. In this case under the parent directory sqoop will cerate a directory with the same name as th e table.

How can you import only a subset of rows form a table?

By using the WHERE clause in the sqoop import statement we can import only a subset of rows.

How can we import a subset of rows from a table without using the where clause?

We can run a filtering query on the database and save the result to a temporary table in database.

Then use the sqoop import command without using the --where clause

What is the advantage of using --password-file rather than -P option while preventing the display of password in the sqoop import statement?

The --password-file option can be used inside a sqoop script while the -P option reads from standard input , preventing automation.

What is the default extension of the files produced from a sqoop import using the --compress parameter?

.gz

What is the significance of using --compress-codec parameter?

To get the out file of a sqoop import in formats other than .gz like .bz2 we use the --compress -code parameter.

What is a disadvantage of using --direct parameter for faster data load by sqoop?

The native utilities used by databases to support faster laod do not work for binary data formats like SequenceFile

How can you control the number of mappers used by the sqoop command?

The Parameter --num-mapers is used to control the number of mappers executed by a sqoop command. We should start with choosing a small number of map tasks and then gradually scale up as choosing high number of mappers initially may slow down the performance on the database side.

How can you avoid importing tables one-by-one when importing a large number of tables from a database?

Using the command

sqoop import-all-tables

--connect

--usrename

--password

--exclude-tables table1,table2 ..

This will import all the tables except the ones mentioned in the exclude-tables clause.

When the source data keeps getting updated frequently, what is the approach to keep it in sync with the data in HDFS imported by sqoop?

sqoop can have 2 approaches.

**a** − To use the --incremental parameter with append option where value of some columns are checked and only in case of modified values the row is imported as a new row.

**b** − To use the --incremental parameter with lastmodified option where a date column in the source is checked for records which have been updated after the last import.

What is the usefulness of the options file in sqoop.

The options file is used in sqoop to specify the command line values in a file and use it in the sqoop commands.

For example the --connect parameter's value and --user name value scan be stored in a file and used again and again with different sqoop commands.

Is it possible to add a parameter while running a saved job?

Yes, we can add an argument to a saved job at runtime by using the --exec option

sqoop job --exec jobname -- -- newparameter

How do you fetch data which is the result of join between two tables?

By using the --query parameter in place of --table parameter we can specify a sql query. The result of the query will be imported

How can we slice the data to be imported to multiple parallel tasks?

Using the --split-by parameter we specify the column name based on which sqoop will divide the data to be imported into multiple chunks to be run in parallel.

How can you choose a name for the mapreduce job which is created on submitting a free-form query import?

By using the --mapreduce-job-name parameter. Below is a example of the command.

sqoop import \

--connect jdbc:mysql://mysql.example.com/sqoop \

--username sqoop \

--password sqoop \

--query 'SELECT normcities.id, \

countries.country, \

normcities.city \

FROM normcities \

JOIN countries USING(country\_id) \

WHERE $CONDITIONS' \

--split-by id \

--target-dir cities \

--mapreduce-job-name normcities

**Interview question on sqoop**

Before starting the data transfer using mapreduce job, sqoop takes a long time to retrieve the minimum and maximum values of columns mentioned in –split-by parameter. How can we make it efficient?

We can use the --boundary –query parameter in which we specify the min and max value for the column based on which the split can happen into multiple mapreduce tasks. This makes it faster as the query inside the –boundary-query parameter is executed first and the job is ready with the information on how many mapreduce tasks to create before executing the main query.

What is the difference between the parameters sqoop.export.records.per.statement and sqoop.export.statements.per.transaction

The parameter “sqoop.export.records.per.statement” specifies the number of records that will be used in each insert statement.

But the parameter “sqoop.export.statements.per.transaction” specifies how many insert statements can be processed parallel during a transaction.

How will you implement all-or-nothing load using sqoop?

Using the staging-table option we first load the data into a staging table and then load it to the final target table only if the staging load is successful.

How do you clear the data in a staging table before loading it by Sqoop?

By specifying the –clear-staging-table option we can clear the staging table before it is loaded. This can be done again and again till we get proper data in staging.

How will you update the rows that are already exported?

The parameter --update-key can be used to update existing rows. In it a comma-separated list of columns is used which uniquely identifies a row. All of these columns is used in the WHERE clause of the generated UPDATE query. All other table columns will be used in the SET part of the query.

How can you sync a exported table with HDFS data in which some rows are deleted?

Truncate the target table and load it again.

How can you export only a subset of columns to a relational table using sqoop?

By using the –column parameter in which we mention the required column names as a comma separated list of values.

How can we load to a column in a relational table which is not null but the incoming value from HDFS has a null value?

By using the –input-null-string parameter we can specify a default value and that will allow the row to be inserted into the target table.

How can you schedule a sqoop job using Oozie?

Oozie has in-built sqoop actions inside which we can mention the sqoop commands to be executed.

Sqoop imported a table successfully to HBase but it is found that the number of rows is fewer than expected. What can be the cause?

Some of the imported records might have null values in all the columns. As Hbase does not allow all null values in a row, those rows get dropped.

Give a sqoop command to show all the databases in a MySql server.

$ sqoop list-databases --connect jdbc:mysql://database.example.com/

What do you mean by Free Form Import in Sqoop?

Sqoop can import data form a relational database using any SQL query rather than only using table and column name parameters.

How can you force sqoop to execute a free form Sql query only once and import the rows serially.

By using the –m 1 clause in the import command, sqoop cerates only one mapreduce task which will import the rows sequentially.

In a sqoop import command you have mentioned to run 8 parallel Mapreduce task but sqoop runs only 4. What can be the reason?

The Mapreduce cluster is configured to run 4 parallel tasks. So the sqoop command must have number of parallel tasks less or equal to that of the MapReduce cluster.

What is the importance of --split-by clause in running parallel import tasks in sqoop?

The –split-by clause mentions the column name based on whose value the data will be divided into groups of records. These group of records will be read in parallel by the mapreduce tasks.

What does this sqoop command achieve?

$ sqoop import --connnect <connect-str> --table foo --target-dir /dest \

It imports data from a database to a HDFS file named foo located in the directory /dest

What happens when a table is imported into a HDFS directory which already exists using the –apend parameter?

Using the --append argument, Sqoop will import data to a temporary directory and then rename the files into the normal target directory in a manner that does not conflict with existing filenames in that directory.

How can you control the mapping between SQL data types and Java types?

By using the --map-column-java property we can configure the mapping between.

Below is an example

$ sqoop import ... --map-column-java id = String, value = Integer

How to import only the updated rows form a table into HDFS using sqoop assuming the source has last update timestamp details for each row?

By using the lastmodified mode. Rows where the check column holds a timestamp more recent than the timestamp specified with --last-value are imported.

What are the two file formats supported by sqoop for import?

Delimited text and Sequence Files.

Give a sqoop command to import the columns employee\_id,first\_name,last\_name from the MySql table Employee

$ sqoop import --connect jdbc:mysql://host/dbname --table EMPLOYEES \

--columns "employee\_id,first\_name,last\_name"

Give a sqoop command to run only 8 mapreduce tasks in parallel

$ sqoop import --connect jdbc:mysql://host/dbname --table table\_name\

-m 8

What does the following query do?

$ sqoop import --connect jdbc:mysql://host/dbname --table EMPLOYEES \

--where "start\_date > '2012-11-09'

It imports the employees who have joined after 9-NOv-2012.

Give a Sqoop command to import all the records from employee table divided into groups of records by the values in the column department\_id.

$ sqoop import --connect jdbc:mysql://db.foo.com/corp --table EMPLOYEES \

--split-by dept\_id

What does the following query do?

$ sqoop import --connect jdbc:mysql://db.foo.com/somedb --table sometable \

--where "id > 1000" --target-dir /incremental\_dataset --append

It performs an incremental import of new data, after having already imported the first 100,0rows of a table

Give a sqoop command to import data from all tables in the MySql DB DB1.

sqoop import-all-tables --connect jdbc:mysql://host/DB1

Give a command to execute a stored procedure named proc1 which exports data to from MySQL db named DB1 into a HDFS directory named Dir1.

$ sqoop export --connect jdbc:mysql://host/DB1 --call proc1 \

--export-dir /Dir1

What is a sqoop metastore?

It is a tool using which Sqoop hosts a shared metadata repository. Multiple users and/or remote users can define and execute saved jobs (created with sqoop job) defined in this metastore.

Clients must be configured to connect to the metastore in sqoop-site.xml or with the --meta-connect argument.

What is the purpose of sqoop-merge?

The merge tool combines two datasets where entries in one dataset should overwrite entries of an older dataset preserving only the newest version of the records between both the data sets.

How can you see the list of stored jobs in sqoop metastore?

sqoop job –list

Give the sqoop command to see the content of the job named myjob?

Sqoop job –show myjob

Which database the sqoop metastore runs on?

Running sqoop-metastore launches a shared HSQLDB database instance on the current machine.

Where can the metastore database be hosted?

The metastore database can be hosted anywhere within or outside of the Hadoop cluster..

//Following are the sqoop commands list that you should know and study

**Common arguments:**

--connect <jdbc-uri> Specify JDBC connect string

--connection-manager <class-name> Specify connection manage class name

--connection-param-file <properties-file> Specify connection parameters file

--driver <class-name> Manually specify JDBC driver class to use

--hadoop-home <hdir> Override $HADOOP\_MAPRED\_HOME\_ARG

--hadoop-mapred-home <dir> Override $HADOOP\_MAPRED\_HOME\_ARG

--help Print usage instructions

-P Read password from console

--password <password> Set authentication password

--password-alias <password-alias> Credential provider password alias

--password-file <password-file> Set authentication password file path

--relaxed-isolation Use read-uncommitted isolation for imports

--skip-dist-cache Skip copying jars to distributed cache

--temporary-rootdir <rootdir> Defines the temporary root directory for the import

--username <username> Set authentication username

--verbose Print more information while working

**Import control arguments:**

--append Imports data

in append

mode

--as-avrodatafile Imports data

to Avro data

files

--as-parquetfile Imports data

to Parquet

files

--as-sequencefile Imports data

to

SequenceFiles

--as-textfile Imports data

as plain

text

(default)

--autoreset-to-one-mapper Reset the

number of

mappers to

one mapper

if no split

key

available

--boundary-query <statement> Set boundary

query for

retrieving

max and min

value of the

primary key

--columns <col,col,col...> Columns to

import from

table

--compression-codec <codec> Compression

codec to use

for import

--delete-target-dir Imports data

in delete

mode

--direct Use direct

import fast

path

--direct-split-size <n> Split the

input stream

every 'n'

bytes when

importing in

direct mode

-e,--query <statement> Import

results of

SQL

'statement'

--fetch-size <n> Set number

'n' of rows

to fetch

from the

database

when more

rows are

needed

--inline-lob-limit <n> Set the

maximum size

for an

inline LOB

-m,--num-mappers <n> Use 'n' map

tasks to

import in

parallel

--mapreduce-job-name <name> Set name for

generated

mapreduce

job

--merge-key <column> Key column

to use to

join results

--split-by <column-name> Column of

the table

used to

split work

units

--split-limit <size> Upper Limit

of rows per

split for

split

columns of

Date/Time/Ti

mestamp and

integer

types. For

date or

timestamp

fields it is

calculated

in seconds.

split-limit

should be

greater than

0

--table <table-name> Table to

read

--target-dir <dir> HDFS plain

table

destination

--validate Validate the

copy using

the

configured

validator

--validation-failurehandler <validation-failurehandler> Fully

qualified

class name

for

ValidationFa

ilureHandler

--validation-threshold <validation-threshold> Fully

qualified

class name

for

ValidationTh

reshold

--validator <validator> Fully

qualified

class name

for the

Validator

--warehouse-dir <dir> HDFS parent

for table

destination

--where <where clause> WHERE clause

to use

during

import

-z,--compress Enable

compression

Incremental import arguments:

--check-column <column> Source column to check for incremental

change

--incremental <import-type> Define an incremental import of type

'append' or 'lastmodified'

--last-value <value> Last imported value in the incremental

check column

Output line formatting arguments:

--enclosed-by <char> Sets a required field enclosing

character

--escaped-by <char> Sets the escape character

--fields-terminated-by <char> Sets the field separator character

--lines-terminated-by <char> Sets the end-of-line character

--mysql-delimiters Uses MySQL's default delimiter set:

fields: , lines: \n escaped-by: \

optionally-enclosed-by: '

--optionally-enclosed-by <char> Sets a field enclosing character

Input parsing arguments:

--input-enclosed-by <char> Sets a required field encloser

--input-escaped-by <char> Sets the input escape

character

--input-fields-terminated-by <char> Sets the input field separator

--input-lines-terminated-by <char> Sets the input end-of-line

char

--input-optionally-enclosed-by <char> Sets a field enclosing

character

Hive arguments:

--create-hive-table Fail if the target hive

table exists

--hive-database <database-name> Sets the database name to

use when importing to hive

--hive-delims-replacement <arg> Replace Hive record \0x01

and row delimiters (\n\r)

from imported string fields

with user-defined string

--hive-drop-import-delims Drop Hive record \0x01 and

row delimiters (\n\r) from

imported string fields

--hive-home <dir> Override $HIVE\_HOME

--hive-import Import tables into Hive

(Uses Hive's default

delimiters if none are

set.)

--hive-overwrite Overwrite existing data in

the Hive table

--hive-partition-key <partition-key> Sets the partition key to

use when importing to hive

--hive-partition-value <partition-value> Sets the partition value to

use when importing to hive

--hive-table <table-name> Sets the table name to use

when importing to hive

--map-column-hive <arg> Override mapping for

specific column to hive

types.

HBase arguments:

--column-family <family> Sets the target column family for the

import

--hbase-bulkload Enables HBase bulk loading

--hbase-create-table If specified, create missing HBase tables

--hbase-row-key <col> Specifies which input column to use as the

row key

--hbase-table <table> Import to <table> in HBase

HCatalog arguments:

--hcatalog-database <arg> HCatalog database name

--hcatalog-home <hdir> Override $HCAT\_HOME

--hcatalog-partition-keys <partition-key> Sets the partition

keys to use when

importing to hive

--hcatalog-partition-values <partition-value> Sets the partition

values to use when

importing to hive

--hcatalog-table <arg> HCatalog table name

--hive-home <dir> Override $HIVE\_HOME

--hive-partition-key <partition-key> Sets the partition key

to use when importing

to hive

--hive-partition-value <partition-value> Sets the partition

value to use when

importing to hive

--map-column-hive <arg> Override mapping for

specific column to

hive types.

HCatalog import specific options:

--create-hcatalog-table Create HCatalog before import

--drop-and-create-hcatalog-table Drop and Create HCatalog before

import

--hcatalog-storage-stanza <arg> HCatalog storage stanza for table

creation

Accumulo arguments:

--accumulo-batch-size <size> Batch size in bytes

--accumulo-column-family <family> Sets the target column family for

the import

--accumulo-create-table If specified, create missing

Accumulo tables

--accumulo-instance <instance> Accumulo instance name.

--accumulo-max-latency <latency> Max write latency in milliseconds

--accumulo-password <password> Accumulo password.

--accumulo-row-key <col> Specifies which input column to

use as the row key

--accumulo-table <table> Import to <table> in Accumulo

--accumulo-user <user> Accumulo user name.

--accumulo-visibility <vis> Visibility token to be applied to

all rows imported

--accumulo-zookeepers <zookeepers> Comma-separated list of

zookeepers (host:port)

Code generation arguments:

--bindir <dir> Output directory for compiled

objects

--class-name <name> Sets the generated class name.

This overrides --package-name.

When combined with --jar-file,

sets the input class.

--input-null-non-string <null-str> Input null non-string

representation

--input-null-string <null-str> Input null string representation

--jar-file <file> Disable code generation; use

specified jar

--map-column-java <arg> Override mapping for specific

columns to java types

--null-non-string <null-str> Null non-string representation

--null-string <null-str> Null string representation

--outdir <dir> Output directory for generated

code

--package-name <name> Put auto-generated classes in

this package

Generic Hadoop command-line arguments:

(must preceed any tool-specific arguments)

Generic options supported are

-conf <configuration file> specify an application configuration file

-D <property=value> use value for given property

-fs <local|namenode:port> specify a namenode

-jt <local|resourcemanager:port> specify a ResourceManager

-files <comma separated list of files> specify comma separated files to be copied to the map reduce cluster

-libjars <comma separated list of jars> specify comma separated jar files to include in the classpath.

-archives <comma separated list of archives> specify comma separated archives to be unarchived on the compute machines.

The general command line syntax is

bin/hadoop command [genericOptions] [commandOptions]

At minimum, you must specify --connect and --table

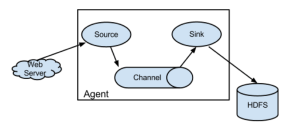
Arguments to mysqldump and other subprograms may be supplied

after a '--' on the command line.

Data Ingest using Flume and HDFS

Apache Flume is open source tool which can capture data generated by web logs in real time to Hadoop eco system.

* Distributed and reliable
* It can collect, aggregate and move large amounts of log data
* It is robust, fault tolerant and tunable
* Uses a simple extensible data model that allows for online analytic application



* [Flume User Guide](https://flume.apache.org/FlumeUserGuide.html) will be provided at the time of taking certification exam.

### **Topics covered as part of this lesson**

### **Data Ingest**

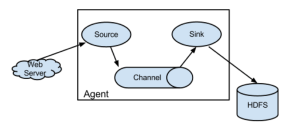
The skills to transfer data between external systems and your cluster. This includes the following:

* Import data from a MySQL database into HDFS using Sqoop
* Export data to a MySQL database from HDFS using Sqoop
* Change the delimiter and file format of data during import using Sqoop
* **Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume**
* **Load data into and out of HDFS using the Hadoop File System (FS) commands**

### **Apache Flume – Getting Started**

Apache Flume is open source tool which can capture data generated by web logs in real time to Hadoop eco system.

* Distributed and reliable
* It can collect, aggregate and move large amounts of log data
* It is robust, fault tolerant and tunable
* Uses a simple extensible data model that allows for online analytic application



* [Flume User Guide](https://flume.apache.org/FlumeUserGuide.html) will be provided at the time of taking certification exam.

#### **Topics covered as part of this lesson**

#### **Data Ingest**

The skills to transfer data between external systems and your cluster. This includes the following:

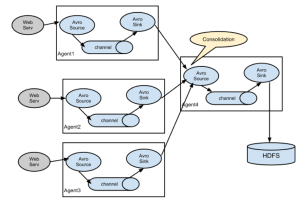
* Import data from a MySQL database into HDFS using Sqoop
* Export data to a MySQL database from HDFS using Sqoop
* Change the delimiter and file format of data during import using Sqoop
* **Ingest real-time and near-real time (NRT) streaming data into HDFS using Flume**
* **Load data into and out of HDFS using the Hadoop File System (FS) commands**

#### **Different Flume agent configurations**

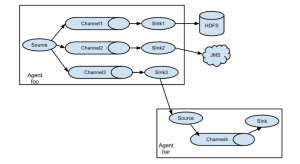
* Flume supports the following mechanisms to read data from popular log stream types, such as:
  1. Avro
  2. Thrift
  3. Syslog
  4. Netcat
* Agent needs to be installed to capture data from source into target and Flume supports several types of installations.
  + Multi-agent flow
    - In order to flow the data across multiple agents or hops, the sink of the previous agent and source of the current hop need to be avro type with the sink pointing to the hostname (or IP address) and port of the source.

02MultiagentSimpleFlow

* + Consolidation
    - A very common scenario in log collection is a large number of log producing clients sending data to a few consumer agents that are attached to the storage subsystem. For example, logs collected from hundreds of web servers sent to a dozen of agents that write to HDFS cluster.
    - This can be achieved in Flume by configuring a number of first tier agents with an avro sink, all pointing to an avro source of single agent (Again you could use the thrift sources/sinks/clients in such a scenario). This source on the second tier agent consolidates the received events into a single channel which is consumed by a sink to its final destination.



* + Multiplexing the flow
    - Flume supports multiplexing the event flow to one or more destinations. This is achieved by defining a flow multiplexer that can replicate or selectively route an event to one or more channels.
    - The below example shows a source from agent “foo” fanning out the flow to three different channels. This fan out can be replicating or multiplexing. In case of replicating flow, each event is sent to all three channels. For the multiplexing case, an event is delivered to a subset of available channels when an event’s attribute matches a preconfigured value. For example, if an event attribute called “txnType” is set to “customer”, then it should go to channel1 and channel3, if it’s “vendor” then it should go to channel2, otherwise channel3. The mapping can be set in the agent’s configuration file.



### **Ingest streaming data using Flume**

Flume need some streaming data to load into HDFS. To run simple flume agent we can use telnet to generate some test data by connecting to web server launched by flume.

If telnet is not available, install telnet following this command

For checking version use the command flume-ng version

* yum -y install telnet

#### **Running first flume job (telnet to logger)**

Steps to run first flume job

* Create directory for flume configuration files mkdir -p flume/conf
* To go inside the dir use the following cd /opt/examples , cd flumes ,vi flume.conf
* For installing and checking telent use telenet command and if not found command then use sudo yum –y install telnet ,ping localhost ,ping quickstart.cloudera ,ifconfig –a ,ps –ef|grep ssh ,telnet 192.168.247.128(ip address of vmware) 22(default port of ssh)

Here the loopback adapter address is 192.168.0.0.1 ,quickstart.cloudera is the hostname for this vm,ip address for vm we can get by ifconfig –a ,to list all the services running in vm we use

Finally for inserting a command in flue use vi example.conf to exit press :x and enter

* Create example.conf

# example.conf: A single-node Flume configuration

# Name the components on this agent

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = netcat

a1.sources.r1.bind = localhost

a1.sources.r1.port = 44444

# Describe the sink

a1.sinks.k1.type = logger

# Use a channel which buffers events in memory

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

* Agent name: a1
* Source name: r1
* Sink name: k1
* Channel name: c1
* Parameter format
  + Names: <agent\_name>.<component\_type> = <component\_name>
    - <agent\_name> is user defined name
    - <component\_type> is typically sources, sinks and channels
    - <component\_name> is user defined names to component\_type
    - eg: a1.sources=r1
  + Attributes: <agent\_name>.<component\_type>.<component\_name>.<attribute\_name> = <attribute\_value>
    - <agent\_name> is user defined name
    - <component\_type> is typically sources, sinks and channels for which attributes are being set
    - <component\_name> are names given as part of the parameter file
    - <attribute\_name> is name of attribute or property which needs to be changed
    - <attribute\_value> value of attribute
    - Below is the example for channel properties, one can review documentation for all properties or attributes related to channels

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

* Once configuration file is defined for sources, sinks and channels - flume agent can be run using

flume-ng agent --name a1

--conf /home/cloudera/flume/conf

--conf-file /home/cloudera/flume/conf/example.conf

-n or --name

-c or --conf

-f or --conf-file

flume-ng agent -n a1 -c /home/cloudera/flume/conf -f /home/cloudera/flume/example.conf

* So this is the first flume job which uses netstat for source,memory for channel and logger for sink
* Command will start web server with port number 44444. Output will show like this, all three components source, channel and sink will also be started.
* 16/07/07 05:44:46 INFO instrumentation.MonitoredCounterGroup: Component type: CHANNEL, name: c1 started
* 16/07/07 05:44:46 INFO node.Application: Starting Sink k1
* 16/07/07 05:44:46 INFO node.Application: Starting Source r1
* 16/07/07 05:44:46 INFO source.NetcatSource: Source starting
* 16/07/07 05:44:46 INFO source.NetcatSource: Created serverSocket:sun.nio.ch.ServerSocketChannelImpl[/127.0.0.1:44444]
* Open the terminal and run telnet localhost 44444
* It will connect to web server using port number 44444, now start typing random text
* Check other terminal where web server is running, as we are using logger as sink we will see text typed in telnet logged in web server started by flume agent
* This video covers how to store the text generated connecting to web server using telnet in HDFS
* We will be changing the sink to HDFS from logger in the configuration file
* Important parameter for sink of type HDFS

| **Name** | **Default** | **Description** |
| --- | --- | --- |
| channel | – |  |
| type | – | The component type name, needs to be hdfs |
| hdfs.path | – | HDFS directory path (eg hdfs://namenode/flume/webdata/) |
| hdfs.filePrefix | FlumeData | Name prefixed to files created by Flume in hdfs directory |
| hdfs.fileSuffix | – | Suffix to append to file (eg .avro - *NOTE: period is not automatically added*) |
| hdfs.inUsePrefix | – | Prefix that is used for temporal files that flume actively writes into |
| hdfs.inUseSuffix | .tmp | Suffix that is used for temporal files that flume actively writes into |
| hdfs.idleTimeout | 0 | Timeout after which inactive files get closed (0 = disable automatic closing of idle files) |
| hdfs.batchSize | 100 | number of events written to file before it is flushed to HDFS |
| hdfs.codeC | – | Compression codec. one of following : gzip, bzip2, lzo, lzop, snappy |
| hdfs.fileType | SequenceFile | File format: currently SequenceFile, DataStream or CompressedStream (1)DataStream will not compress output file and please don’t set codeC (2)CompressedStream requires set hdfs.codeC with an available codeC |

* Steps to integrate data to HDFS
  + Hit ctrl+c to kill flume agent and web server
  + Modify example.conf to add HDFS as sink
  + sinks.type: hdfs
  + hdfs.path: Valid location in HDFS
  + Set other relevant parameters as shown here

# example.conf: A single-node Flume configuration

# Name the components on this agent

a1.sources = r1

a1.sinks = k1

a1.channels = c1

# Describe/configure the source

a1.sources.r1.type = netcat

a1.sources.r1.bind = localhost

a1.sources.r1.port = 44444

# Describe the sink for hdfs

a1.sinks.k1.type = hdfs

a1.sinks.k1.hdfs.path = hdfs://quickstart.cloudera:8020/user/cloudera/flume

#a1.sinks.k1.hdfs.filePrefix: will make sure files are prefixed with netcat in location defined in a1.sinks.k1.hdfs.path

a1.sinks.k1.hdfs.filePrefix = netcat

# Use a channel which buffers events in memory

a1.channels.c1.type = memory

a1.channels.c1.capacity = 1000

a1.channels.c1.transactionCapacity = 100

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

* Save and exit from example.conf
* Run this command

flume-ng agent --name a1

--conf /home/cloudera/flume/conf

--conf-file /home/cloudera/flume/conf/example.conf

* Web server will be started along with source, sink and channel with the agent
* Open the terminal and run telnet localhost 44444
* It will connect to web server using port number 44444, now start typing random text
* Data will be streaming into HDFS to the path /user/cloudera/flume
* Kill the agent and validate by running hadoop fs -ls /user/cloudera/flume
* hadoop fs –cat  /user/cloudera/flume/netcat.1478304625035
* for the purpose of seeing the data
* Files with prefix netcat will be listed
* Sample folder for the flume path is **cd /opt/examples/flume , view flume-env.sh**
* To view the files , **cd /var/log , sudo tail –F messages ,sudo service mysqld restart,sudo service ntpd restart ,view /opt/gen\_logs/logs/access.log ,** **tail -F /opt/gen\_logs/logs/access.log**
* [cloudera@quickstart log]$ start\_logs
* [cloudera@quickstart log]$ stop\_logs to start and stop the logs show in the acces log
* By default files generated are of type sequencefile

This video covers how to integrate streaming data generated by log generator simulating web server generating logs. It uses a tool called gen\_logs.

Here are the steps to follow

* Run start\_logs command. It will generate logs simulating web server logs
* Validate by running tail\_logs command. You can see streaming logs generated
* Log file name: /opt/gen\_logs/logs/access.log
* Create flume.conf with
  + source is of type exec which execute tail -F on /opt/gent\_logs/logs/access.log

a1.sources = r1

a1.sources.r1.type = exec

a1.sources.r1.command = tail -F /opt/gen\_logs/logs/access.log

* + channel is of type FILE

a1.channels = c1

# Use a channel which buffers events to a file

# -- The component type name, needs to be FILE.

a1.channels.c1.type = FILE

# The maximum size of transaction supported by the channel

a1.channels.c1.capacity = 20000

a1.channels.c1.transactionCapacity = 1000

# Amount of time (in millis) between checkpoints

a1.channels.c1.checkpointInterval 3000

# Max size (in bytes) of a single log file

a1.channels.c1.maxFileSize = 2146435071

* + sink is of type HDFS

a1.sinks.k1.type = hdfs

a1.sinks.k1.channel = c1

a1.sinks.k1.hdfs.path = /user/cloudera/flume/%y-%m-%d

a1.sinks.k1.hdfs.filePrefix = flume-%y-%m-%d

#Roll properties

a1.sinks.k1.hdfs.rollSize = 1048576

a1.sinks.k1.hdfs.rollCount = 100

a1.sinks.k1.hdfs.rollInterval = 120

#files will be of type text when fileType is defined as DataStream (default was sequence file)

a1.sinks.k1.hdfs.fileType = DataStream

a1.sinks.k1.hdfs.idleTimeout = 10

a1.sinks.k1.hdfs.useLocalTimeStamp = true

* + Bind source (logs) and sink (hdfs) to channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

a1.sinks = k1

* + Final code in flume.conf

# Describe/configure r1

# agent name a1

a1.sources = r1

a1.sources.r1.type = exec

a1.sources.r1.command = tail -F /opt/gen\_logs/logs/access.log

a1.channels = c1

# Use a channel which buffers events to a file

# -- The component type name, needs to be FILE.

a1.channels.c1.type = FILE

# The maximum size of transaction supported by the channel

a1.channels.c1.capacity = 20000

a1.channels.c1.transactionCapacity = 1000

# Amount of time (in millis) between checkpoints

a1.channels.c1.checkpointInterval 3000

# Max size (in bytes) of a single log file

a1.channels.c1.maxFileSize = 2146435071

# Describe the sink

a1.sinks.k1.type = hdfs

a1.sinks.k1.channel = c1

a1.sinks.k1.hdfs.path = /user/cloudera/flume/%y-%m-%d

a1.sinks.k1.hdfs.filePrefix = flume-%y-%m-%d

a1.sinks.k1.hdfs.rollSize = 1048576

a1.sinks.k1.hdfs.rollCount = 100

a1.sinks.k1.hdfs.rollInterval = 120

a1.sinks.k1.hdfs.fileType = DataStream

a1.sinks.k1.hdfs.idleTimeout = 10

a1.sinks.k1.hdfs.useLocalTimeStamp = true

# Bind the source and sink to the channel

a1.sources.r1.channels = c1

a1.sinks.k1.channel = c1

a1.sinks = k1

* + Run the command using flume-ng

flume-ng agent --name a1

--conf /home/cloudera/flume/conf

--conf-file /home/cloudera/flume/conf/flume.conf

* + Validate by running hadoop fs -ls -R /user/cloudera/flume
  + You will find directory with date and will have files loaded by flume using the logs generated.

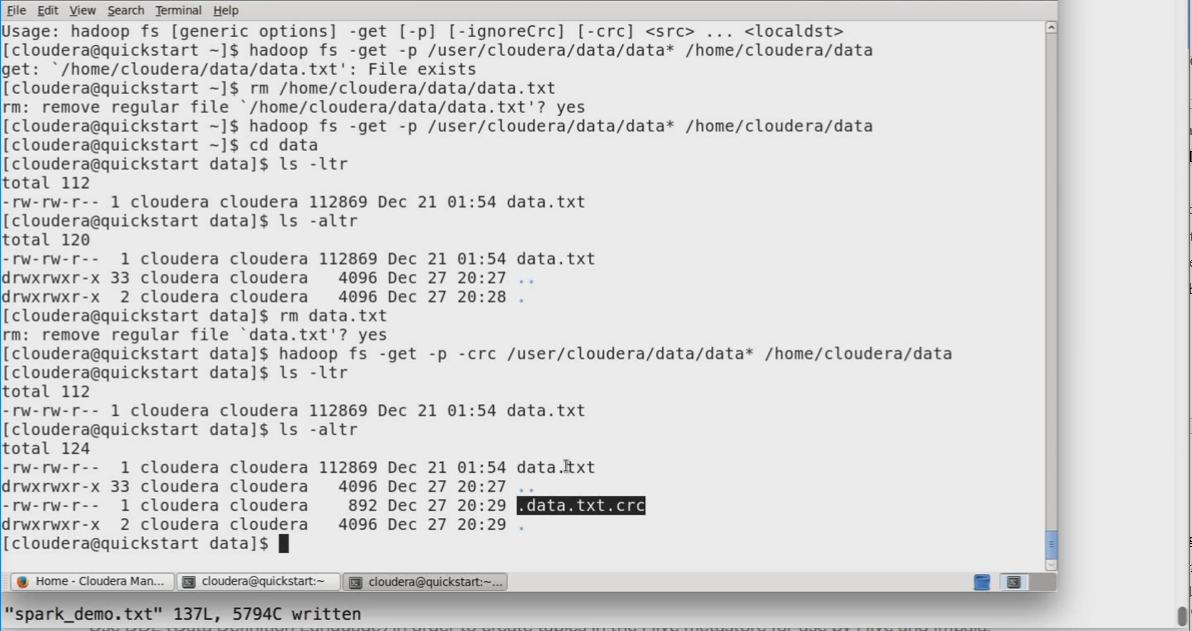
### **Ingest data using HDFS Commands**

HDFS commands are used to interact with files stored in HDFS.

* HDFS stands for Hadoop Distributed File System
* There are several commands in HDFS, but scope for this topic will be only data ingestion commands
* When files which contain data are copied to HDFS, files will be divided into 128 MB blocks and those blocks will be stored physically in Hadoop cluster. The component which manages storage of these physical files (for blocks) is called as Datanode. dfs.blocksize is the parameter which controls the size of each block.
* Each block will be copied on to multiple nodes for fault tolerance. By default it is 3 and defined by paramter called dfs.replication (replication factor)
* With 128 MB block size and replication factor 3, 1 GB file will be divided into 8 blocks, cloned into 3 copies (hence 8 \* 3 = 24 records for 1 GB file).
* Mapping between file name, block names and block locations are stored in in-memory namespace and managed by Namenode.

Here is the video which cover relevant topics to store files in HDFS

* hadoop fs and hit enter will show list of the commands
* Make sure to copy the data set to the VM or the Gateway node
* Run hadoop fs -copyFromLocal command to copy the data into HDFS as shown in the video
* It take 2 parameters
  + first one is the path of the data to be copied in the local file system of VM or Gateway node
  + second parameter is the path in HDFS to which data has to be copied
  + For using help use the following hadoop fs -help copyToLocal
  + hadoop fs -help get
  + hadoop fs -get /user/cloudera/data/data\* /home/cloudera/data
  + hadoop fs –get –p –crc /user/cloudera/data/data\* /home/cloudera/data
  + hadoop movefromlocal is also a good command
  + –R is for directory
  + here second parameter adds a checksum in the given code and gives the solution for the process ie



Transform,stage,store using Spark with Python

Apache Spark is open source cluster computing framework. This lesson will have all the topics related to Spark using Python.

* It works with any file system (s3, HDFS etc)
* Processing will be done in-memory
* It is effective in processing streaming data loads
* It is primarily distributed by databricks
* There are many components in Spark eco system, such as Core Spark or Transformations and Actions, Streaming, MLLib, Graphx, Data Frames (from 1.3.x) etc.

For CCA Spark and Hadoop Developer certification, we just need to focus on Core Spark which means core set of Transformations and Actions. While other components change with versions, core spark syntax remained almost same.

Here is the [documentation](http://spark.apache.org/docs/1.2.1/) that will be provided at the time of taking certification exam. One need to use Spark programming guide.

Here is the video about the introduction of Transformation, Stage and Store. One need to be knowledgeable enough of using Spark APIs with both the programming languages - Scala and Python. **Optionally** one can set up Spark 1.2.1 on **Cloudera Quickstart VM**. **It is not required as long as one want to invoke hive queries as part of the certification.**

Here are the topics that are covered as part of this lesson using pyspark

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

### **Pyspark – Getting Started**

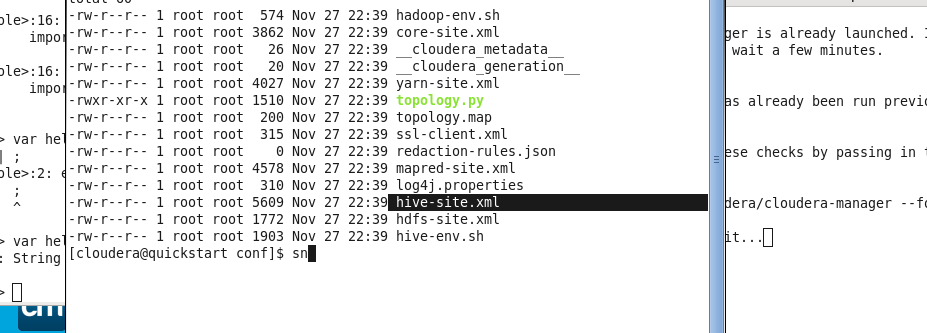
As part of this topic we will see how to get started with Python and submit sample applications.

Here is the video which talks about getting started using python. Note that using spark with JDBC is not relevant for CCA certification.

Here is the video which talks about submitting pyspark applications

Here are the steps to submit spark applications using pyspark

For creating the **soft link** for hive-site.xml ie sql context for hive support



1.cd /etc/hive/conf

2.sudo ln -s /etc/hive/conf/hive-site.xml /etc/spark/conf/hive-site.xml

To validate the link

ls -ltr /etc/spark/conf if the link is not valid then the color will be red

now if the link is ,

3.view /etc/spark/conf/hive-site.xml

now we can run spark in hive context

4.sqlContext.sql("select \* from customers").collect().foreach(println)

5.sqlContext.sql("select \* from customers").count()

* Other steps to get the new spark installed are as follows :-

Copy on desktop

Then use the following commands

tar xzf spark-1.6.0-bin-hadoop2.6.tgz

cd / spark-1.6.0-bin-hadoop2.6.tgz

bin/spark-shell

* Now to accesss the hdfs we need to give the confg to the spark
* ls -ltr /etc/spark/conf/
* cp -rf /etc/spark/conf/ .
* rm -rf conf
* ls –ltr
* vi spark-defaults.conf
* now you need to do bunch of other stuffs to manage the spark shell to see this look at tut no 27 lastly use the following command
* bin/spark-shell --master local
* bin/pyspark --master local
* sc.textFile("/user/cloudera/sqoop\_import/departments").collect().foreach(println)
* to see the records
* Also for running the pyspark in either local mode or normal we use the following commands
* pyspark --master local to run pyspark locally
* pyspark to run normally

using hive in spark

from pyspark.sql import SQLContext

from pyspark.sql import HiveContext

sqlContext=HiveContext(sc)

depts=sqlContext.sql("select \* from departments")

for rec in depts.collect():

println(rec):

* For listing all collection jars use

sudo find / -name "mysql-connector\*.jar"

* Or for info use the following in the spark context
* os.environ['SPARK\_CLASSPATH'] = "/usr/share/java/mysql-connector-java.jar"
* from pyspark.sql import SQLContext
* sqlContext = SQLContext(sc)
* jdbcurl = "jdbc:mysql://quickstart.cloudera:3306/retail\_db?user=retail\_dba&password=cloudera"
* df = sqlContext.load(source="jdbc", url=jdbcurl, dbtable="departments")
* for rec in df.collect():
* ... print(rec)
* df.count()
* Open a new file and name it as saveFile.py
* vi saveFile.py
* As part of the program we will just try to read data from HDFS and write it back to HDFS
* Enter below code in saveFile.py

from pyspark import SparkContext, SparkConf

conf = SparkConf().setAppName("pyspark")

sc = SparkContext(conf=conf)

dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")

for line in dataRDD.collect():

print(line)

dataRDD.saveAsTextFile("/user/cloudera/pyspark/departmentsTesting")

* Save saveFile.py
* Run it in spark native mode spark-submit --master local saveFile.py
* Run it in yarn mode spark-submit --master yarn saveFile.py
* Also to check some of the things
* data = sc.textFile("/user/cloudera/sqoop\_import/departments")
* for i in data.collect():
* print(i)
* data=sc.textFile("file:///home/cloudera/departments.jason")
* hadoop fs -ls hdfs://quickstart.cloudera:8020/user/cloudera/sqoop\_import/departments
* view /etc/hadoop/conf/core-site.xml for seeing the port no

### **Move data between HDFS and Spark – pyspark**

As part of this topic we will cover the highlighted certification topic

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Here is the video to copy data from text files between HDFS and Spark

from pyspark import SparkContext

dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")

for line in dataRDD.collect():

print(line)

print(dataRDD.count())

dataRDD.saveAsTextFile("/user/cloudera/pyspark/departments")

* Reading data dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")
* Validating (printing contents in pyspark console) - for line in dataRDD.collect():  
  print(line)
* Getting count - print(dataRDD.count())
* Writing back to HDFS - dataRDD.saveAsTextFile("/user/cloudera/pyspark/departments")
* Code snippet to write sequence files to HDFS

#saveAsSequenceFile

dataRDD.map(lambda x: (None, x)).saveAsSequenceFile("/user/cloudera/pyspark/departmentsSeq")

dataRDD.map(lambda x: tuple(x.split(",", 1))).saveAsSequenceFile("/user/cloudera/pyspark/departmentsSeq")

dataRDD.map(lambda x: tuple(x.split(",", 1))).saveAsSequenceFile("/user/cloudera/pyspark/orders")

* Sequence files store data in key and value format.
* Writing sequence files to HDFS with null key - dataRDD.map(lambda x: (None,x)).saveAsSequenceFile("/user/cloudera/pyspark/departmentsSeq")
* map is spark based API function to apply row level transformations. In this case we are saying None is the key and each line is the value from input text
* lambda is function with out name - lambda x: (None, x)
* Writing sequence files to HDFS with first column as key - dataRDD.map(lambda x:tuple(x.split(",",1))).saveAsSequenceFile("/user/cloudera/pyspark/departmentsSeq")
* lambda is function with out name, in this case it is tokenizing/splitting each record with delimiter and emitting value before first , as key and rest as value - lambda x: tuple(x.split(",", 1))
* Code snippet to use saveAsNewAPIHadoopFile for saving data in HDFS with file format of our choice (eg: sequence file)

path="/user/cloudera/pyspark/departmentsSeq"

dataRDD.map(lambda x: tuple(x.split(",", 1))).saveAsNewAPIHadoopFile(path,"org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat",keyClass="org.apache.hadoop.io.Text",valueClass="org.apache.hadoop.io.Text")

* Code snippet to read data from sequence files with out key

#reading sequence file

data = sc.sequenceFile("/user/cloudera/pyspark/departmentsSeq")

for rec in data.collect():

print(rec)

* Code snippet to read data from sequence files with key

data = sc.sequenceFile("/user/cloudera/pyspark/departmentsSeq", "org.apache.hadoop.io.IntWritable", "org.apache.hadoop.io.Text")

for rec in data.collect():

print(rec)

Here is the video to copy data from json files between HDFS and Spark. This is not very important for the certification.

* Code snippet to read data from hive tables in hive context. In 1.2.x, it might not run.

from pyspark.sql import HiveContext

sqlContext = HiveContext(sc)

depts = sqlContext.sql("select \* from departments")

for rec in depts.collect():

print(rec)

if you with to get some specific information then you write

depts = sqlContext.sql("select department\_id from departments")

for i in depts.collect():

print(i.department\_id)

trying to read another hive data

depts = sqlContext.sql("select \* from sqoop\_import.departments")

for i in depts.collect():

print(i.department\_id)

trying to read another hive data

depts = sqlContext.sql("create table departmentsTest as select \* from departments")

for i in depts.collect():

print(i.department\_id)

* Code snippet to deal with JSON files

#Make sure you copy departments.json to HDFS

#create departments.json on Linux file system

{"department\_id":2, "department\_name":"Fitness"}

{"department\_id":3, "department\_name":"Footwear"}

{"department\_id":4, "department\_name":"Apparel"}

{"department\_id":5, "department\_name":"Golf"}

{"department\_id":6, "department\_name":"Outdoors"}

{"department\_id":7, "department\_name":"Fan Shop"}

{"department\_id":8, "department\_name":"TESTING"}

{"department\_id":8000, "department\_name":"TESTING"}

#copying to HDFS (using linux command line)

vi departments.json

hadoop fs -mkdir -p /user/cloudera/pyspark

hadoop fs -put departments.json /user/cloudera/pyspark

from pyspark import SQLContext

sqlContext = SQLContext(sc)

departmentsJson = sqlContext.jsonFile("/user/cloudera/pyspark/departments.json")

departmentsJson.registerTempTable("departmentsTable")

departmentsData = sqlContext.sql("select \* from departmentsTable")

for rec in departmentsData.collect():

print(rec)

#Writing data in json format

departmentsData.toJSON().saveAsTextFile("/user/cloudera/pyspark/departmentsJson")

#Validating the data

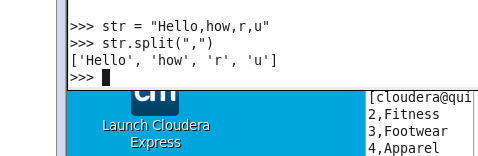
hadoop fs -cat /user/cloudera/pyspark/departmentsJson/part\*

For other things in the section like reading code is as follows

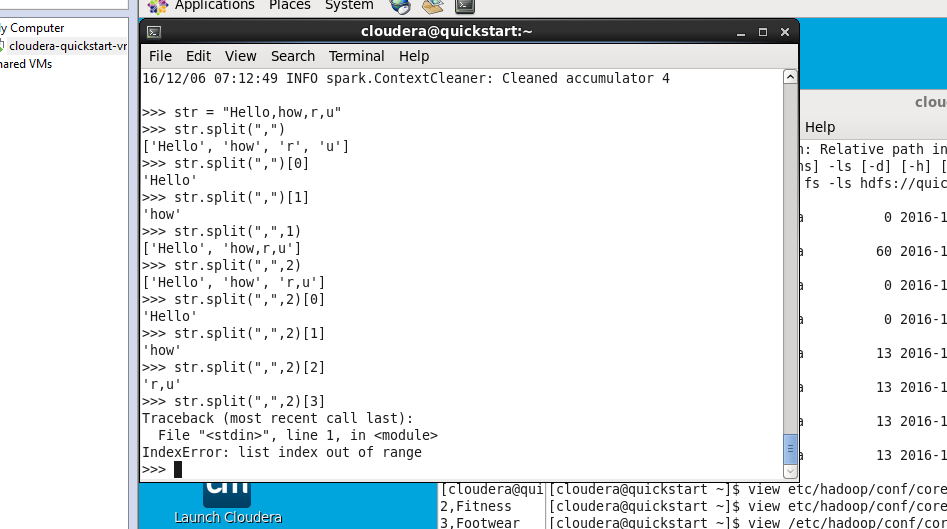
dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")

for i in dataRDD.collect()

print(i)



- The split function and it’s meaning can be shown below



* Reading data dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")
* for i in dataRDD.map(lambda x :tuple (x.split(",",1))).collect():
* **#saveAsSequenceFile**
* dataRDD.map(lambda x: (None, x)).saveAsSequenceFile("/user/cloudera/pyspark/departmentsSeq")
* dataRDD.map(lambda x: tuple(x.split(",", 1))).saveAsSequenceFile("/user/cloudera/pyspark/departmentsSeq")
* dataRDD.map(lambda x: tuple(x.split(",", 1))).saveAsSequenceFile("/user/cloudera/pyspark/orders")
* path="/user/cloudera/pyspark/departmentsSeq"
* **#For reading the file form the saved location**
* data = sc.sequenceFile("/user/cloudera/pyspark/departmentsSeq")
* for i in data.collect():
* print(i)
* #For using sequence data
* path="/user/cloudera/pyspark/departmentsSeq"
* dataRDD.map(lambda x: tuple(x.split(",", 1))).saveAsNewAPIHadoopFile(path,"org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat",keyClass="org.apache.hadoop.io.Text",valueClass="org.apache.hadoop.io.Text")
* data = sc.sequenceFile("/user/cloudera/pyspark/departmentsSeq", "org.apache.hadoop.io.IntWritable", "org.apache.hadoop.io.Text")
* or rec in data.collect():
* print(rec)

### **Word Count using pyspark**

In this topic we will try to understand how to write word count program.

Here is the video to get word count by using flatMap, map and reduceByKey functions

Here is the code snippet and details about the logic. One can run one line at a time and understand what is going on.

Two general commands are as follows vi wordcount.txt ,hadoop fs -put /user/cloudera

* Reading data from HDFS location

data = sc.textFile("/user/cloudera/wordcount.txt")

* Flatten each line into multiple words using " " (space) as delimiter.

dataFlatMap = data.flatMap(lambda x: x.split(" "))

* Associate value 1 for each of the input word to map function.

dataMap = dataFlatMap.map(lambda x: (x, 1))

* Aggregating using key (which are nothing but all unique words)

dataReduceByKey = dataMap.reduceByKey(lambda x,y: x + y)

* Saving to HDFS

dataReduceByKey.saveAsTextFile("/user/cloudera/wordcountoutput")

* Validating the data set

for i in dataReduceByKey.collect():

print(i)

Here is the video which explain the details of word count program

### **Joining disparate data sets using pyspark**

As part of this topic we will cover the highlighted certification topic

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Typically we end up joining data sets from multiple tables to get insights. In transaction based applications, tables in database are normalized and related data will be stored in multiple tables following normalization rules. Hence we need to join data from related tables to get insights.

Earlier we have copied data from mysql into HDFS using sqoop to /user/cloudera/sqoop\_import directory (the username can be different for non virtual machines). Data set have 6 tables

* departments
* categories
* products
* order\_items
* orders
* customers

As part of joining data sets we will try to explore APIs using pyspark and apply it on orders and order\_items tables.

* orders is parent table for which order\_id is primary key. Each record will store order level information such as order state, order date etc.

+-------------------+-------------+------+-----+---------+

| Field | Type | Null | Key | Default |

+-------------------+-------------+------+-----+---------+

| order\_id | int(11) | NO | PRI | NULL |

| order\_date | datetime | NO | | NULL |

| order\_customer\_id | int(11) | NO | | NULL |

| order\_status | varchar(45) | NO | | NULL |

+-------------------+-------------+------+-----+---------+

* order\_items is child table to orders. order\_item\_id is primary key and order\_item\_order\_id  is foreign key to orders.order\_id. There will be multiple records in order\_items for each order\_id in orders table (as we can typically check out multiple order items per order)

+--------------------------+------------+------+-----+---------+

| Field | Type | Null | Key | Default |

+--------------------------+------------+------+-----+---------+

| order\_item\_id | int(11) | NO | PRI | NULL |

| order\_item\_order\_id | int(11) | NO | | NULL |

| order\_item\_product\_id | int(11) | NO | | NULL |

| order\_item\_quantity | tinyint(4) | NO | | NULL |

| order\_item\_subtotal | float | NO | | NULL |

| order\_item\_product\_price | float | NO | | NULL |

+--------------------------+------------+------+-----+---------+

* **#Problem statement: get the revenue and number of orders from order\_items on daily basis. Here are the steps**.
  + Read the data from orders and order\_items
  + Extract the key from orders and order\_items (using map)
  + Join the orders and order\_items
  + Get revenue per order item per day
  + Get order count per date from order\_items (aggregation). As there are orders which do not have corresponding records in order\_items, we cannot get count using order table. We need to join order\_items with orders to get total number of orders per day.
  + Get revenue per day from joined data
  + Steps befor this thing are
  + Hive
  + show tables;
  + describe orders;
  + describe order\_items;
  + hadoop fs -ls /user/cloudera/sqoop\_import
  + hadoop fs -tail /user/cloudera/sqoop\_imports/orders/part-m-00000
  + ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")
  + orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")
  + ordersParsedRDD = ordersRDD.map(lambda rec: (int(rec.split(",")[0]), rec))
  + orderItemsParsedRDD = orderItemsRDD.map(lambda rec: (int(rec.split(",")[1]), rec))
  + ordersJoinOrderItems = orderItemsParsedRDD.join(ordersParsedRDD)
  + revenuePerOrderPerDay = ordersJoinOrderItems.map(lambda t: (t[1][1].split(",")[1], float(t[1][0].split(",")[4])))
  + # Get order count per day
  + ordersPerDay = ordersJoinOrderItems.map(lambda rec: rec[1][1].split(",")[1] + "," + str(rec[0])).distinct()
  + ordersPerDayParsedRDD = ordersPerDay.map(lambda rec: (rec.split(",")[0], 1))
  + totalOrdersPerDay = ordersPerDayParsedRDD.reduceByKey(lambda x, y: x + y)
  + # Get revenue per day from joined data
  + totalRevenuePerDay = revenuePerOrderPerDay.reduceByKey( \
  + lambda total1, total2: total1 + total2 \
  + )
  + , # Get order count per day
  + ordersPerDay = ordersJoinOrderItems.map(lambda rec: rec[1][1].split(",")[1] + "," + str(rec[0])).distinct()
  + ordersPerDayParsedRDD = ordersPerDay.map(lambda rec: (rec.split(",")[0], 1))
  + totalOrdersPerDay = ordersPerDayParsedRDD.reduceByKey(lambda x, y: x + y)
  + # Get revenue per day from joined data
  + totalRevenuePerDay = revenuePerOrderPerDay.reduceByKey( \
  + lambda total1, total2: total1 + total2 \
  + )
  + for data in totalRevenuePerDay.collect():
  + print(data)
  + # Joining order count per day and revenue per day
  + finalJoinRDD = totalOrdersPerDay.join(totalRevenuePerDay)
  + for data in finalJoinRDD.take(5):
  + print(data)

Here are the videos to perform join operations on orders and order\_items. Joining the data set will be emphasized and aggregations will be discussed later.

* Reading the data from both orders and order\_items

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

* Apply map function to get order\_id as key from orders table and whole record as value as map output. order\_id is the first column in the orders table.ordersRDD.map(lambda rec:(int(rec.split(",")[0]), rec))
* Apply map function to get order\_item\_order\_id as key from order\_items table and whole record as value as map output. order\_item\_order\_id is the 2nd column in the order table.orderItemsRDD.map(lambda rec: (int(rec.split(",")[1]), rec))

ordersParsedRDD = ordersRDD.map(lambda rec: (int(rec.split(",")[0]), rec))

orderItemsParsedRDD = orderItemsRDD.map(lambda rec: (int(rec.split(",")[1]), rec))

* Join data sets using spark transformation join

ordersJoinOrderItems = orderItemsParsedRDD.join(ordersParsedRDD)

* ordersJoinOrderItems will represent a tuple. Key is join column (order\_id) value is a tuple with corresponding order\_items record as first element and orders record as second element
* Get revenue for each order\_item per day (output will be only order\_date and order\_item\_subtotal from each record)

revenuePerOrderPerDay = ordersJoinOrderItems.map(lambda t: (t[1][1].split(",")[1], float(t[1][0].split(",")[4])))

Here is the continuation video to perform join and then aggregate the data as per our problem statement.

* Here is the code snippet which uses actions such as reduceByKey to get count per day. Aggregations will be covered in detail later.

ordersPerDay = ordersJoinOrderItems.map(lambda rec: rec[1][1].split(",")[1] + "," + str(rec[0])).distinct()

ordersPerDayParsedRDD = ordersPerDay.map(lambda rec: (rec.split(",")[0], 1))

totalOrdersPerDay = ordersPerDayParsedRDD.reduceByKey(lambda x, y: x + y)

* Get revenue per day from joined data using aggregation transformation reduceByKey.

totalRevenuePerDay = revenuePerOrderPerDay.reduceByKey(

lambda total1, total2: total1 + total2

)

* Validate totalRevenuePerDay by looping through and printing the data as part of standard output.

for data in totalRevenuePerDay.collect():

print(data)

* Joining order count per day and revenue per day

finalJoinRDD = totalOrdersPerDay.join(totalRevenuePerDay)

for data in finalJoinRDD.take(5):

print(data)

Here is the video to get the same results using SQL embedded in pyspark. This is not highly relevant for certification exam. SQL tested might fail in some of the spark versions.

* + Using Hive

from pyspark.sql import HiveContext

sqlContext = HiveContext(sc)

sqlContext.sql("set spark.sql.shuffle.partitions=10");

joinAggData = sqlContext.sql("select o.order\_date, round(sum(oi.order\_item\_subtotal), 2),

count(distinct o.order\_id) from orders o join order\_items oi

on o.order\_id = oi.order\_item\_order\_id

group by o.order\_date order by o.order\_date")

for data in joinAggData.collect():

print(data)

* Using spark native sql

from pyspark.sql import SQLContext, Row

sqlContext = SQLContext(sc)

sqlContext.sql("set spark.sql.shuffle.partitions=10");

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersMap = ordersRDD.map(lambda o: o.split(","))

orders = ordersMap.map(lambda o: Row(order\_id=int(o[0]), order\_date=o[1], \

order\_customer\_id=int(o[2]), order\_status=o[3]))

ordersSchema = sqlContext.inferSchema(orders)

ordersSchema.registerTempTable("orders")

orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

orderItemsMap = orderItemsRDD.map(lambda oi: oi.split(","))

orderItems = orderItemsMap.map(lambda oi: Row(order\_item\_id=int(oi[0]), order\_item\_order\_id=int(oi[1]),

order\_item\_product\_id=int(oi[2]), order\_item\_quantity=int(oi[3]), order\_item\_subtotal=float(oi[4]),

order\_item\_product\_price=float(oi[5])))

orderItemsSchema = sqlContext.inferSchema(orderItems)

orderItemsSchema.registerTempTable("order\_items")

joinAggData = sqlContext.sql("select o.order\_date, sum(oi.order\_item\_subtotal),

count(distinct o.order\_id) from orders o join order\_items oi

on o.order\_id = oi.order\_item\_order\_id

group by o.order\_date order by o.order\_date")

for data in joinAggData.collect():

print(data)

### **Aggregating data sets using pyspark-totals**

Aggregations can be broadly categorized into totals and by key. As part of this topic we will covering aggregations - totals.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Types of totals aggregations

* sum, average
* min, max

Spark provides actions such as count, total to compute sums. To compute average, we should be able to perform necessary operations by leveraging spark transformations and actions.

This video covers computing totals using actions such as count and reduce to get count and sum.

* Get total number of records in a data set (eg: orders)

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.count()

* Get total revenue from order\_items

orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

orderItemsMap = orderItemsRDD.map(lambda rec: float(rec.split(",")[4]))

for i in orderItemsMap.take(5):

print i

orderItemsReduce = orderItemsMap.reduce(lambda rev1, rev2: rev1 + rev2)

* Get max priced product in products table. Cleanup that one product with product\_id 685. We will look into filtering later.

hadoop fs -get /user/cloudera/sqoop\_import/products

#Delete the record with product\_id 685

hadoop fs -put -f products/part\* /user/cloudera/sqoop\_import/products

* Get max priced product using reduce, see below for the implementation in the lambda function

productsRDD = sc.textFile("/user/cloudera/sqoop\_import/products")

productsMap = productsRDD.map(lambda rec: rec)

productsMap.reduce(

lambda rec1, rec2:

  (rec1

  if((rec1.split(",")[4] != "" and rec2.split(",")[4] != "")

  and float(rec1.split(",")[4]) >= float(rec2.split(",")[4]))

  else rec2)

)

Computing average revenue using total revenue divided by total distinct orders.

* Computing average revenue (total revenue/total distinct orders from order\_items)

revenue = sc.textFile("/user/cloudera/sqoop\_import/order\_items").

map(lambda rec: float(rec.split(",")[4])).

reduce(lambda rev1, rev2: rev1 + rev2)

totalOrders = sc.textFile("/user/cloudera/sqoop\_import/order\_items").

map(lambda rec: int(rec.split(",")[1])).

distinct().

count()

revenue/totalOrders

or

revenue = sc.textFile("/user/cloudera/sqoop\_imports/order\_items").map(lambda rec: float(rec.split(",")[4])).reduce(lambda rev1, rev2: rev1 + rev2)

totalOrders = sc.textFile("/user/cloudera/sqoop\_imports/order\_items").map(lambda rec: int(rec.split(",")[1])).distinct().count()

revenue/totalOrders

revenue/totalOrders will give average revenue.

### **Aggregating data sets using pyspark- by key**

Aggregations can be broadly categorized into totals and by key. As part of this topic we will covering aggregations - by key.

**totalRevenue = orderItemsRDD.map(lambda rec : float(rec.split(",")[4])).reduce(lambda acc,val acc+val)**

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

By key aggregations are extensively used to group data by a key and get insights. Key could be time, category, department etc.

* 1. **Here is the introduction video about by key operations. In this all the by key transformations such as groupByKey, reduceByKey, aggregateByKey are extensively covered**.
* **Number of orders by status using different by key operations**
* Using countByKey

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersMap = ordersRDD.map(lambda rec: (rec.split(",")[3], 1))

for i in ordersMap.countByKey().items(): print(i)

* **Using groupByKey**
* groupByKey is not efficient in this case, as the size of the output is very small compared to size of input data.

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersByStatus = ordersMap.

groupByKey().

map(lambda t: (t[0], sum(t[1])))

for recs in ordersByStatus.collect(): print(recs)

* Using reduceByKey. It uses combiner internally.
* Input data and output data for reduceByKey need to be of same type.
* Combiner is implicit and uses the reduce logic.

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersByStatus = ordersMap.

reduceByKey(lambda acc, value: acc + value)

for recs in ordersByStatus.collect(): print(recs)

* Using aggregateByKey. It also uses combiner internally.
* Input data and output data for reduceByKey can be of different type.
* Also combiner logic can be different from reduce logic.

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersMap = ordersRDD.

map(lambda rec: (rec.split(",")[3], rec))

ordersByStatus = ordersMap.

aggregateByKey(0,

lambda acc, value: acc+1,

lambda acc, value: acc+value

)

for recs in ordersByStatus.collect(): print(recs)

* Using combineByKey. It also uses combiner internally.
* Input data and output data for reduceByKey can be of different type.
* It is almost same as aggregateByKey and less frequently used.

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersByStatus = ordersMap.

combineByKey(lambda value:

1,

lambda acc, value: acc+1,

lambda acc, value: acc+value

)

for recs in ordersByStatus.collect(): print(recs)

Here is the video which talks about combiner. Combiner is the main difference between groupByKey and other by key transformations such as reduceByKey, aggregateByKey etc.

* For all the by key aggregations where the input data volume is significantly higher than output data volume, then we should not use groupByKey
* groupByKey is the only option where we can extend functionality for other operations such as sorting.

Here is the video which talks about combiner in the context of reduceByKey, aggregateByKey etc

* Once we understand differences between different by key transformations, we need to apply for new requirements
* Requirement: Number of orders by order date and order status
* For this requirement, key is order\_date and order\_status
* We are doing aggregation, hence groupByKey is eliminated
* Combiner logic and reducer logic can be same, hence aggregateByKey and combineByKey are eliminated
* We are left with countByKey and reduceByKey, we can use either of them.
* Here is the logic implementing reduceByKey

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

ordersMapRDD = ordersRDD.

map(lambda rec: ((rec.split(",")[1], rec.split(",")[3]), 1))

ordersByStatusPerDay = ordersMapRDD.

reduceByKey(lambda v1, v2: v1+v2)

for recs in ordersByStatusPerDay.collect(): print(recs)

**Here is the video which explains aggregation such as average by key**

* Requirement: Generate average revenue per day
* Parse Orders (key order\_id)
* Parse Order items (key order\_item\_order\_id)
* Join Orders and Order item on the key
* Parse joined data and get (order\_date and order\_id) as key and order\_item\_subtotal as value. Here number of input records and output records will be same.
* Apply aggregate function (reduceByKey) to get revenue per order. Here number of output records will be number of distinct orders from order\_items table.
* Parse revenue per order and remove order\_id from the key. Here number of output records will be number of distinct orders from order\_items table.
* Apply aggregate function (combineByKey or aggregateByKey) to get total revenue per day. In this input type and output type for aggregate functions are different and also we need to have custom combiner logic, hence reduceByKey cannot be used.
* Now data will have order\_date as key and (revenue\_per\_day and total\_number\_of\_orders) as value. Apply map function to divide revenue with total number of orders per day.

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

ordersParsedRDD = ordersRDD.map(lambda rec: (rec.split(",")[0], rec))

orderItemsParsedRDD = orderItemsRDD.map(lambda rec: (rec.split(",")[1], rec))

ordersJoinOrderItems = orderItemsParsedRDD.join(ordersParsedRDD)

ordersJoinOrderItemsMap = ordersJoinOrderItems.

map(lambda t: ((t[1][1].split(",")[1], t[0]), float(t[1][0].split(",")[4])))

revenuePerDayPerOrder = ordersJoinOrderItemsMap.

reduceByKey(lambda acc, value: acc + value)

revenuePerDayPerOrderMap = revenuePerDayPerOrder.

map(lambda rec: (rec[0][0], rec[1]))

revenuePerDay = revenuePerDayPerOrderMap.combineByKey(

lambda x: (x, 1),

lambda acc, revenue: (acc[0] + revenue, acc[1] + 1),

lambda total1, total2: (round(total1[0] + total2[0], 2), total1[1] + total2[1])

)

revenuePerDay = revenuePerDayPerOrderMap.aggregateByKey(

(0, 0),

lambda acc, revenue: (acc[0] + revenue, acc[1] + 1),

lambda total1, total2: (round(total1[0] + total2[0], 2), total1[1] + total2[1])

)

for data in revenuePerDay.collect():

print(data)

avgRevenuePerDay = revenuePerDay.map(lambda x: (x[0], x[1][0]/x[1][1]))

for data in avgRevenuePerDay.collect():

print(data)

Here is the video which explains implementation of max by key. It includes SQL (but SQL is not relevant for the CCA certification)

* Requirement: Get customer\_id with max revenue for each day
* customer\_id is in orders table and revenue need to be derived from order\_items table
* Apply join between orders and order\_items with order\_id as key
* Apply map function to get (order\_date and customer\_id) as key and order\_item\_subtotal as value
* Apply aggregate function to generate revenue per customer per day
* Apply another aggregate function using reduceByKey to get customer with maximum revenue per day. Logic is implemented using named function and invoked from lambda function of reduceByKey

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

ordersParsedRDD = ordersRDD.map(lambda rec: (rec.split(",")[0], rec))

orderItemsParsedRDD = orderItemsRDD.map(lambda rec: (rec.split(",")[1], rec))

ordersJoinOrderItems = orderItemsParsedRDD.join(ordersParsedRDD)

ordersPerDayPerCustomer = ordersJoinOrderItems.

map(lambda rec:

((rec[1][1].split(",")[1], rec[1][1].split(",")[2]),

float(rec[1][0].split(",")[4])))

revenuePerDayPerCustomer = ordersPerDayPerCustomer.

reduceByKey(lambda x, y: x + y)

revenuePerDayPerCustomerMap = revenuePerDayPerCustomer.

map(lambda rec: (rec[0][0], (rec[0][1], rec[1])))

topCustomerPerDaybyRevenue = revenuePerDayPerCustomerMap.

reduceByKey(lambda x, y: (x if x[1] >= y[1] else y))

#Using regular function

def findMax(x, y):

if(x[1] >= y[1]):

return x

else:

return y

topCustomerPerDaybyRevenue = revenuePerDayPerCustomerMap.

reduceByKey(lambda x, y: findMax(x, y))

### **Filtering data using pyspark**

As part of this topic, filtering of data sets using pyspark is covered.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Spark have filter transformation to apply different types of filtering. If the underlying logic in the lambda function returns false then the record will be discarded.

* Applying simple filter using filter transformation
* Get all the orders with status COMPLETE

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in ordersRDD.

filter(lambda line: line.split(",")[3] == "COMPLETE").

take(5):

print(i)

* Get all the orders where status contains the word PENDING

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in ordersRDD.

filter(lambda line: "PENDING" in line.split(",")[3]).

take(5):

print(i)

* Get all the orders where order\_id is greater than 100

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in ordersRDD.

filter(lambda line: int(line.split(",")[0]) > 100).

take(5):

print(i)

* Boolean operation - or
* Get all the orders where order\_id > 100 or order\_status is in one of the pending states

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in ordersRDD.

filter(lambda line:

int(line.split(",")[0]) > 100

or line.split(",")[3] in "PENDING").

take(5):

print(i)

* Combination of boolean operators and as well as or
* Make sure when combined, the boolean operations are properly bracketed
* For eg: (a and b or c) results in different than (a and (b or c)). (a and (b or c)) is correct way in most of the scenarios.

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in ordersRDD.

filter(lambda line:

int(line.split(",")[0]) > 1000

and ("PENDING" in line.split(",")[3]

or line.split(",")[3] == ("CANCELLED"))).

take(5):

print(i)

* Another example

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in ordersRDD.

filter(lambda line:

int(line.split(",")[0]) > 1000

and line.split(",")[3] != ("COMPLETE")).

take(5):

print(i)

* Requirement: Check if there are cancelled orders with amount greater than 1000$
* Get cancelled orders from orders (filter)
* Join orders (filtered) and order\_items
* Generate revenue per order on the joined data set (reduceByKey)
* Discard all the cancelled orders whose revenue is less than 1000$

ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

ordersParsedRDD = ordersRDD.

filter(lambda rec: rec.split(",")[3] in "CANCELED").

map(lambda rec: (int(rec.split(",")[0]), rec))

orderItemsParsedRDD = orderItemsRDD.

map(lambda rec: (int(rec.split(",")[1]), float(rec.split(",")[4])))

orderItemsAgg = orderItemsParsedRDD.

reduceByKey(lambda acc, value: (acc + value))

ordersJoinOrderItems = orderItemsAgg.join(ordersParsedRDD)

for i in ordersJoinOrderItems.

filter(lambda rec: rec[1][0] >= 1000).take(5):

print(i)

also another command

for i in ordersRDD.filter(lambda line: line.split(",")[3]).distinct().collect():

print(i)

### **Sorting and Ranking using pyspark – global**

Sorting can be broadly categorized into global and by key. As part of this topic we will covering sorting - global.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark
* Global sorting and ranking
* Here are few examples of using different transformations and actions such as sortByKey, top

orders = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in orders.

map(lambda rec: (int(rec.split(",")[0]), rec)).

sortByKey().

collect():

print(i)

for i in orders.

map(lambda rec: (int(rec.split(",")[0]), rec)).

sortByKey(False).

take(5):

print(i)

for i in orders.

map(lambda rec: (int(rec.split(",")[0]), rec)).

top(5):

print(i)

* Global sorting using custom logic
* Here is the implementation of takeOrdered in which data is sorted by using field of our choice in our delimited data set.

orders = sc.textFile("/user/cloudera/sqoop\_import/orders")

for i in orders.

map(lambda rec: (int(rec.split(",")[0]), rec)).

takeOrdered(5, lambda x: x[0]):

print(i)

for i in orders.

map(lambda rec: (int(rec.split(",")[0]), rec)).

takeOrdered(5, lambda x: -x[0]):

print(i)

for i in orders.

takeOrdered(5, lambda x: int(x.split(",")[0])):

print(i)

for i in orders.

takeOrdered(5, lambda x: -int(x.split(",")[0])):

print(i)

### **Sorting and Ranking using pyspark – by key**

Sorting can be broadly categorized into global and by key. As part of this topic we will covering sorting - by key.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Here is the video which covers basic python list operations as well as Spark groupByKey transformation

* Requirement: Sort products by price with in each category
* Read data from HDFS and apply map function to define key which is category
* Apply groupByKey to group all the products in the category

products = sc.textFile("/user/cloudera/sqoop\_import/products")

productsMap = products.map(lambda rec: (rec.split(",")[1], rec))

productsGroupBy = productsMap.groupByKey()

for i in productsGroupBy.collect(): print(i)

* Get data sorted by product price per category
* With in each group use sorted function to sort the data in ascending or descending order

for i in productsGroupBy.

map(lambda rec: sorted(rec[1], key=lambda k: float(k.split(",")[4]))).

take(100):

print(i)

for i in productsGroupBy.

map(lambda rec: sorted(rec[1], key=lambda k: float(k.split(",")[4]), reverse=True)).

take(100):

print(i)

* Requirement: Get top 3 priced products in each category
* Develop python function which get RDD and topN as parameters
* Compute top 3 priced products (if there are 10 products with top 3 prices, the RDD should give us all 10 products)

def getTopDenseN(rec, topN):

x = [ ]

topNPrices = [ ]

prodPrices = [ ]

prodPricesDesc = [ ]

for i in rec[1]:

prodPrices.append(float(i.split(",")[4]))

prodPricesDesc = list(sorted(set(prodPrices), reverse=True))

import itertools

topNPrices = list(itertools.islice(prodPricesDesc, 0, topN))

for j in sorted(rec[1], key=lambda k: float(k.split(",")[4]), reverse=True):

if(float(j.split(",")[4]) in topNPrices):

x.append(j)

return (y for y in x)

* Here is the code which invokes getTopDenseN

products = sc.textFile("/user/cloudera/sqoop\_import/products")

productsMap = products.map(lambda rec: (rec.split(",")[1], rec))

for i in productsMap.

groupByKey().

flatMap(lambda x: getTopDenseN(x, 2)).

collect():

print(i)

Here is the video which covers ranking using Spark SQL using windowing functions. This might not be very important for CCA certification.

* Different queries to perform sorting and ranking using Spark SQL.

#By key sorting

#Using order by is not efficient, it serializes

select \* from products order by product\_category\_id, product\_price desc;

#Using distribute by sort by (to distribute sorting and scale it up)

select \* from products distribute by product\_category\_id sort by product\_price desc;

#By key ranking (in Hive we can use windowing/analytic functions)

select \* from

(select p.\*,

dense\_rank() over (partition by product\_category\_id order by product\_price desc) dr

from products p

distribute by product\_category\_id) q

where dr <= 2 order by product\_category\_id, dr;

**Basic python**

l =[2,3,4,88,7,46564,144,45,536,12,1222]

len(l) , sorted(l) , sorted(l,reverse=True)

Transform,Stage,Store using Spark with Scala

Apache Spark is open source cluster computing framework. This lesson will have all the topics related to Spark using Scala.

* It works with any file system (s3, HDFS etc)
* Processing will be done in-memory
* It is effective in processing streaming data loads
* It is primarily distributed by databricks
* There are many components in Spark eco system, such as Core Spark or Transformations and Actions, Streaming, MLLib, Graphx, Data Frames (from 1.3.x) etc.

For CCA Spark and Hadoop Developer certification, we just need to focus on Core Spark which means core set of Transformations and Actions. While other components change with versions, core spark syntax remained almost same.

Here is the [documentation](http://spark.apache.org/docs/1.2.1/) that will be provided at the time of taking certification exam. One need to use Spark programming guide.

Here is the video about the introduction of Transformation, Stage and Store. One need to be knowledgeable enough of using Spark APIs with both the programming languages - Scala and Python.

### **Spark with Scala – Getting Started**

#### **Getting Started with spark-shell - Spark with Scala**

As part of this topic we will see how to get started with Scala and submit sample applications.

Here is the video which talks about getting started using Scala. Note that using spark with JDBC is not relevant for CCA certification.

Here are the steps to submit spark applications using scala(sbt installation tutoriLA **50**)

* Install sbt
  + Download the tar file of sbt from [here](http://www.scala-sbt.org/download.html), untar and set environment variable SBT\_HOME
  + Update PATH export PATH=$PATH:$SBT\_HOME/bin
* Developing simple scala based applications for spark
* Create necessary directories

cd

mkdir scala

cd scala; mkdir -p src/main/scala

other command

(sudo yum –y install sbt)

* create a file with sbt extension (eg: simple.sbt)
* Define dependencies with below 4 lines

name := "Simple Project"

version := "1.0"

scalaVersion := "2.10.4"

libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.1"

* As part of the program we will just try to read data from HDFS and write it back to HDFS

import org.apache.spark.SparkContext, org.apache.spark.SparkConf

object SimpleApp {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("scala spark")

val sc = new SparkContext(conf)

val dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")

dataRDD.saveAsTextFile("/user/cloudera/scalaspark/departmentsTesting")

}

}

* Save this to a file with scala extension
* Compile into jar file

sbt package

* Run it using spark-submit in native mode

spark-submit --class "SimpleApp" --master local /home/cloudera/scala/target/scala-2.10/simple-project\_2.10-1.0.jar

* Run the application using spark-submit in yarn mode

spark-submit --class "SimpleApp" --master yarn /home/cloudera/scala/target/scala-2.10/simple-project\_2.10-1.0.jar

* Validate that files are copied to HDFS location

hadoop fs -ls /user/cloudera/scalaspark/departmentsTesting

SBT INSTALLATION

sbt installation

1.Extract sbt to cloudera folder

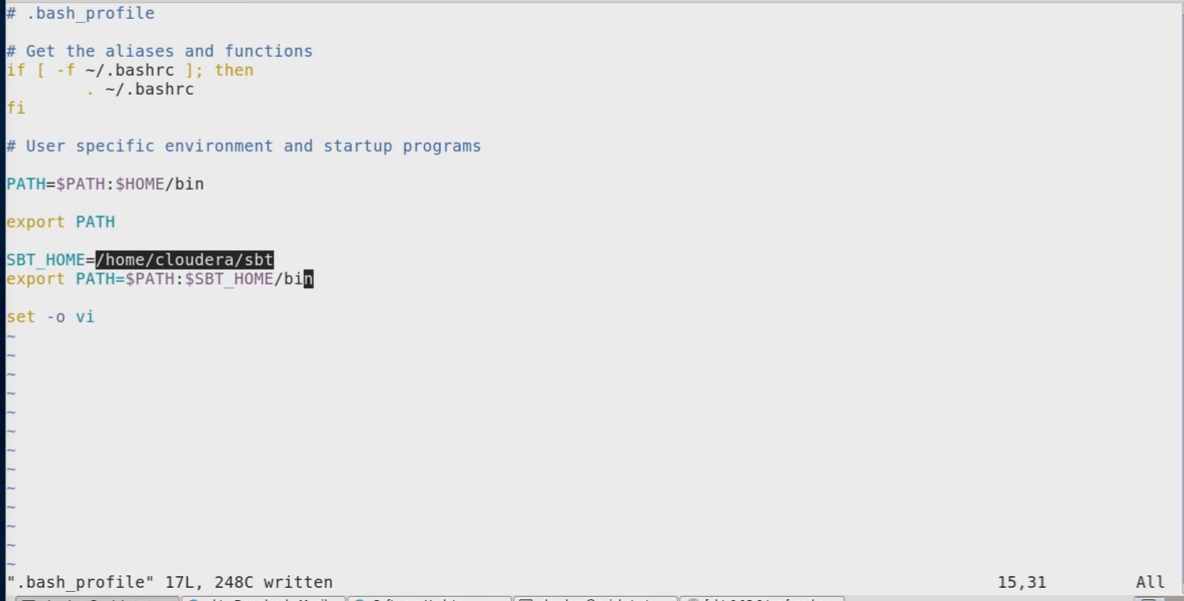
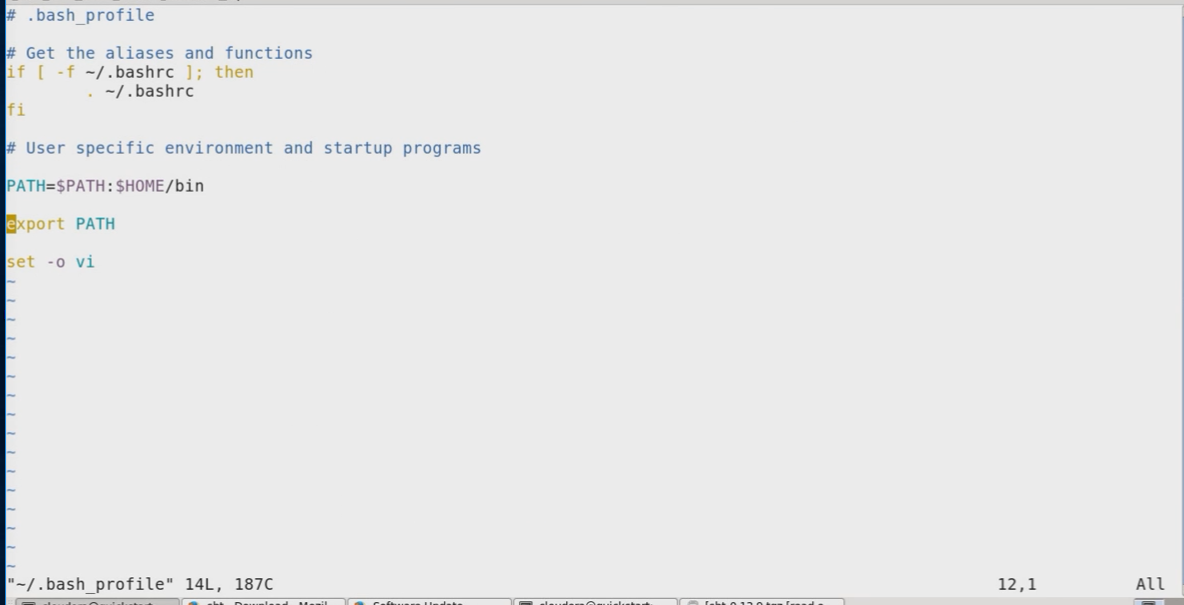
cd sbt

ls -ltr

pwd (copy the path /home/cloudera/sbt/bin)

cd

vi ~/.bash\_profile



sbt

. ~/.bash\_profile

sbt

cat ~/.bash\_profile

. ~/.bash\_profile

env

env|grep -i sbt

cat ~/.bash\_profile

mkdir -p scala

cd scala

ls -ltr

vi simple.sbt

name := "Simple Project"

version := "1.0"

scalaVersion := "2.11.7"

libraryDependencies += "org.apache.spark" %% "spark-core" % "2.1.0"

mkdir -p ./src/main/scala

ls -ltr

vi src/main/scala/simpleApp.scala

import org.apache.spark.SparkContext, org.apache.spark.SparkConf

object SimpleApp {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("scala spark")

val sc = new SparkContext(conf)

val dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")

dataRDD.saveAsTextFile("/user/cloudera/scalaspark/departmentsTesting")

}

}

cat src/main/scala/simpleApp.scala

ls -ltr

sbt package

du -sh\*

ls -ltr ~/.sbt

du -sh ~/.sbt

vi src/main/scala/simpleApp.scala

import org.apache.spark.SparkContext, org.apache.spark.SparkConf

object SimpleApp {

def main(args: Array[String]) {

val conf = new SparkConf().setAppName("scala spark")

val sc = new SparkContext(conf)

val dataRDD = sc.textFile("/user/cloudera/sqoop\_import/departments")

dataRDD.saveAsTextFile("/user/cloudera/scalaspark/departmentsTesting")

}

}

/sbt package

For running the file...

spark-submit --class "SimpleApp" \

--master local \

/home/cloudera/scala/target/scala-2.10/simple-project\_2.10-1.0.jar

hadoop fs -ls /user/cloudera/scalaspark/departmentsTesting

### **Move data between HDFS and Spark – scala**

As part of this topic we will cover the highlighted certification topic

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Here is the video to copy data from text files between HDFS and Spark

import org.apache.spark.SparkContext

val dataRDD = sc.textFile(

"/user/cloudera/sqoop\_import/departments"

)

dataRDD.collect().foreach(println)

dataRDD.count()

dataRDD.saveAsTextFile("/user/cloudera/scalaspark/departments")

* Reading data sc.textFile("/user/cloudera/sqoop\_import/departments")
* Validating (printing contents in spark-shell console) - dataRDD.collect().foreach(println)
* Getting count - dataRDD.count()
* Writing back to HDFS - dataRDD.saveAsTextFile("/user/cloudera/scalaspark/departments")

Here is the video to copy data from sequence files between HDFS and Spark

* Code snippet to write sequence files to HDFS
* Sequence files store data in key and value format
* Writing sequence files to HDFS with null key

import org.apache.hadoop.io.\_

import org.apache.hadoop.mapreduce.lib.output.\_

dataRDD.

map(x => (NullWritable.get(), x)).

saveAsSequenceFile(

"/user/cloudera/scalaspark/departmentsSeq"

)

* map is spark based API function to apply row level transformations. In this case we are saying NullWritable.get() is the key and each line is the value from input text
* lambda is function with out name - x => (NullWritable.get(), x)
* Writing sequence files to HDFS with first column as key

dataRDD.

map(x => (x.split(",")(0), x.split(",")(1))).

saveAsSequenceFile(

"/user/cloudera/scalaspark/departmentsSeq"

)

* lambda is function with out name, in this case it is tokenizing/splitting each record with delimiter and emitting value before first , as key and rest as value

x => (x.split(",")(0), x.split(",")(1))

* Code snippet to use saveAsNewAPIHadoopFile for saving data in HDFS with file format of our choice (eg: sequence file)

import org.apache.hadoop.io.\_

import org.apache.hadoop.mapreduce.lib.output.\_

val path="/user/cloudera/scalaspark/departmentsSeq"

dataRDD.

map(x => (new Text(x.split(",")(0)),

new Text(x.split(",")(1)))).

saveAsNewAPIHadoopFile(

path,

classOf[Text],

classOf[Text],

classOf[SequenceFileOutputFormat[Text, Text]]

)

* Code snippet to read data from sequence files with key

import org.apache.hadoop.io.\_

import org.apache.hadoop.mapreduce.lib.output.\_

sc.sequenceFile(

"/user/cloudera/spark/departmentsSeq",

classOf[IntWritable], classOf[Text]).

map(rec => rec.toString()).

collect().

foreach(println)

Here is the video to copy data from json files between HDFS and Spark. This is not very important for the certification

* Code snippet to read data from hive tables in hive context. In 1.2.x, it might not run.

import org.apache.spark.sql.hive.HiveContext

val sqlContext = new HiveContext(sc)

val depts = sqlContext.

sql("select \* from departments")

depts.collect().foreach(println)

or

import org.apache.spark.sql.hive.HiveContext

val sqlContext = new HiveContext(sc)

val depts = sqlContext.sql("use retail\_ods")

val depts = sqlContext.sql("select \* from departments")

depts.collect().foreach(println)

Here is the video to copy data from json files between HDFS and Spark. This is not very important for the certification.

* Code snippet to deal with JSON files

#Make sure you copy departments.json to HDFS

#create departments.json on Linux file system

<record>

<department\_id>2</department\_id>

<department\_name>Fitness</department\_name>

</record>

{"department\_id":2, "department\_name":"Fitness"}

{"department\_id":3, "department\_name":"Footwear"}

{"department\_id":4, "department\_name":"Apparel"}

{"department\_id":5, "department\_name":"Golf"}

{"department\_id":6, "department\_name":"Outdoors"}

{"department\_id":7, "department\_name":"Fan Shop"}

{"department\_id":8, "department\_name":"TESTING"}

{"department\_id":8000, "department\_name":"TESTING"}

#copying to HDFS (using linux command line)

hadoop fs -put departments.json /user/cloudera/scalaspark

import org.apache.spark.sql.SQLContext

val sqlContext = new SQLContext(sc)

val departmentsJson = sqlContext.

jsonFile("/user/cloudera/scalaspark/departments.json")

departmentsJson.registerTempTable("departmentsTable")

val departmentsData = sqlContext.

sql("select \* from departmentsTable")

departmentsData.collect().foreach(println)

#Writing data in json format

departmentsData.

toJSON.

saveAsTextFile("/user/cloudera/scalaspark/departmentsJson")

#Validating the data

hadoop fs -cat /user/cloudera/scalaspark/departmentsJson/part\*

### **Word count using spark scala**

In this topic we will try to understand how to write word count program.

Here is the video to get word count by using flatMap, map and reduceByKey functions

Here is the code snippet and details about the logic. One can run one line at a time and understand what is going on.

* Developing word count program
* Create a file and type few lines and save it as wordcount.txt and copy to HDFS to /user/cloudera/wordcount.txt
* Reading data from HDFS location

val data = sc.textFile("/user/cloudera/wordcount.txt")

* Flatten each line into multiple words using " " (space) as delimiter.

val dataFlatMap = data.flatMap(x => x.split(" "))

* Associate value 1 for each of the input word to map function.

val dataMap = dataFlatMap.map(x => (x, 1))

* Aggregating using key (which are nothing but all unique words)

val dataReduceByKey = dataMap.reduceByKey((x,y) => x + y)

* Saving to HDFS

dataReduceByKey.

saveAsTextFile("/user/cloudera/wordcountoutput")

* Validating the data set

dataReduceByKey.collect().foreach(println)

Here is the video which explain the details of word count program

### **Joining disparate data sets using scala**

As part of this topic we will cover the highlighted certification topic

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Typically we end up joining data sets from multiple tables to get insights. In transaction based applications, tables in database are normalized and related data will be stored in multiple tables following normalization rules. Hence we need to join data from related tables to get insights.

Earlier we have copied data from mysql into HDFS using sqoop to /user/cloudera/sqoop\_import directory (the username can be different for non virtual machines). Data set have 6 tables

* departments
* categories
* products
* order\_items
* orders
* customers

As part of joining data sets we will try to explore APIs using spark-shell or scala and apply it on orders and order\_items tables.

* orders is parent table for which order\_id is primary key. Each record will store order level information such as order state, order date etc.

+-------------------+-------------+------+-----+---------+

| Field | Type | Null | Key | Default |

+-------------------+-------------+------+-----+---------+

| order\_id | int(11) | NO | PRI | NULL |

| order\_date | datetime | NO | | NULL |

| order\_customer\_id | int(11) | NO | | NULL |

| order\_status | varchar(45) | NO | | NULL |

+-------------------+-------------+------+-----+---------+

* order\_items is child table to orders. order\_item\_id is primary key and order\_item\_order\_id is foreign key to orders.order\_id. There will be multiple records in order\_items for each order\_id in orders table (as we can typically check out multiple order items per order)

+--------------------------+------------+------+-----+---------+

| Field | Type | Null | Key | Default |

+--------------------------+------------+------+-----+---------+

| order\_item\_id | int(11) | NO | PRI | NULL |

| order\_item\_order\_id | int(11) | NO | | NULL |

| order\_item\_product\_id | int(11) | NO | | NULL |

| order\_item\_quantity | tinyint(4) | NO | | NULL |

| order\_item\_subtotal | float | NO | | NULL |

| order\_item\_product\_price | float | NO | | NULL |

+--------------------------+------------+------+-----+---------+

order\_id join order\_item\_order\_id

add all the order\_item\_subtotal

* Problem statement: get the revenue and number of orders from order\_items on daily basis. Here are the steps.
  + Read the data from orders and order\_items
  + Extract the key from orders and order\_items (using map)
  + Join the orders and order\_items
  + Get revenue per orde per day
  + Get order count per date from order\_items (aggregation). As there are orders which do not have corresponding records in order\_items, we cannot get count using order table. We need to join order\_items with orders to get total number of orders per day.
  + Get revenue per day from joined data

Here are the videos to perform join operations on orders and order\_items. Joining the data set will be emphasized and aggregations will be discussed later.

* Reading the data from both orders and order\_items

val ordersRDD = sc.textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.textFile("/user/cloudera/sqoop\_import/order\_items")

* Apply map function to get order\_id as key from orders table and whole record as value as map output. order\_id is the first column in the orders table.ordersRDD.map(rec => (rec.split(",")(0).toInt, rec))
* Apply map function to get order\_item\_order\_id as key from order\_items table and whole record as value as map output. order\_item\_order\_id is the 2nd column in the order table.orderItemsRDD.map(rec=> (rec.split(",")(1).toInt, rec))

val ordersParsedRDD = ordersRDD.

map(rec => (rec.split(",")(0).toInt, rec))

val orderItemsParsedRDD = orderItemsRDD.

map(rec => (rec.split(",")(1).toInt, rec))

* Join data sets using spark transformation join

val ordersJoinOrderItems = orderItemsParsedRDD.

join(ordersParsedRDD)

* ordersJoinOrderItems will represent a tuple. Key is join column (order\_id) value is a tuple with corresponding order\_items record as first element and orders record as second element
* Get revenue for each order\_item per day (output will be only order\_date and order\_item\_subtotal from each record)

val revenuePerOrderPerDay = ordersJoinOrderItems.

map(t =>

(t.\_2.\_2.split(",")(1), t.\_2.\_1.split(",")(4).toFloat)

)

* Here is the code snippet which uses actions such as reduceByKey to get count per day. Aggregations will be covered in detail later.

val ordersPerDay = ordersJoinOrderItems.

map(rec =>

rec.\_2.\_2.split(",")(1) + "," + rec.\_1

).distinct()

val ordersPerDayParsedRDD = ordersPerDay.

map(rec => (rec.split(",")(0), 1))

val totalOrdersPerDay = ordersPerDayParsedRDD.

reduceByKey((x, y) => x + y)

* Get revenue per day from joined data using aggregation transformation reduceByKey.

val totalRevenuePerDay = revenuePerOrderPerDay.reduceByKey(

(total1, total2) => total1 + total2

)

* Validate totalRevenuePerDay by looping through and printing the data as part of standard output.

totalRevenuePerDay.

sortByKey().

collect().

foreach(println)

* Joining order count per day and revenue per day

val finalJoinRDD = totalOrdersPerDay.

join(totalRevenuePerDay)

finalJoinRDD.collect().foreach(println)

Here is the video to get the same results using SQL embedded in spark-shell or scala. This is not highly relevant for certification exam. SQL tested might fail in some of the spark versions.

* Using Hive

import org.apache.spark.sql.hive.HiveContext

val sqlContext = new HiveContext(sc)

sqlContext.sql("set spark.sql.shuffle.partitions=10"); (by default it uses too many process 200 so we control by using it 10)

val joinAggData = sqlContext.sql("select " +

"o.order\_date, " +

"round(sum(oi.order\_item\_subtotal), 2), " +

"count(distinct o.order\_id) " +

"from orders o " +

"join order\_items oi " +

"on o.order\_id = oi.order\_item\_order\_id " +

"group by o.order\_date " +

"order by o.order\_date")

joinAggData.collect().foreach(println)

* Using spark native sql or data frames not relevant to spark 1.2.x or certification

import org.apache.spark.sql.SQLContext, org.apache.spark.sql.Row

val sqlContext = new SQLContext(sc)

sqlContext.sql("set spark.sql.shuffle.partitions=10");

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val ordersMap = ordersRDD.

map(o => o.split(","))

case class Orders(

order\_id: Int,

order\_date: String,

order\_customer\_id: Int,

order\_status: String

)

val orders = ordersMap.

map(o => Orders(o(0).toInt, o(1), o(2).toInt, o(3)))

import sqlContext.createSchemaRDD(this is depricated)

orders.registerTempTable("orders")

val orderItemsRDD = sc.

textFile("/user/cloudera/sqoop\_import/order\_items")

val orderItemsMap = orderItemsRDD.map(oi => oi.split(","))

case class OrderItems

  (order\_item\_id: Int,

   order\_item\_order\_id: Int,

   order\_item\_product\_id: Int,

   order\_item\_quantity: Int,

   order\_item\_subtotal: Float,

   order\_item\_product\_price: Float

  )

val orderItems = sc.textFile("/user/cloudera/sqoop\_import/order\_items").

  map(rec => rec.split(",")).

  map(oi => OrderItems(oi(0).toInt, oi(1).toInt, oi(2).toInt, oi(3).toInt, oi(4).toFloat, oi(5).toFloat))

orderItems.registerTempTable("order\_items")

val joinAggData = sqlContext.sql("select o.order\_date, sum(oi.order\_item\_subtotal), " +

  "count(distinct o.order\_id) from orders o join order\_items oi " +

  "on o.order\_id = oi.order\_item\_order\_id " +

  "group by o.order\_date order by o.order\_date")

joinAggData.collect().foreach(println)

### **Aggregating data sets using scala – totals**

Aggregations can be broadly categorized into totals and by key. As part of this topic we will covering aggregations - totals.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Types of totals aggregations

* sum, average
* min, max

Spark provides actions such as count, total to compute sums. To compute average, we should be able to perform necessary operations by leveraging spark transformations and actions.

This video covers computing totals using actions such as count and reduce to get count and sum.

* Get total number of records in a data set (eg: orders)(for total we use count or reduce)

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.count()

* Get total revenue from order\_items

val orderItemsRDD = sc.

textFile("/user/cloudera/sqoop\_import/order\_items")

val orderItemsMap = orderItemsRDD.

map(rec => (rec.split(",")(4).toDouble))

orderItemsMap.take(5).foreach(println)

val orderItemsReduce = orderItemsMap.

reduce((acc, value) => acc + value)

* Get max priced product in products table. Cleanup that one product with product\_id 685. We will look into filtering later.

hadoop fs -get /user/cloudera/sqoop\_import/products

#Delete the record with product\_id 685

hadoop fs -put -f products/part\* /user/cloudera/sqoop\_import/products

* Get max priced product using reduce, see below for the implementation in the lambda function

val productsRDD = sc.

textFile("/user/cloudera/sqoop\_import/products")

val productsMap = productsRDD.map(rec => rec)

productsMap.reduce((rec1, rec2) => (

  if(rec1.split(",")(4).toFloat >= rec2.split(",")(4).toFloat)

    rec1

  else

    rec2)

)

* Computing average revenue (total revenue/total distinct orders from order\_items)

val revenue = sc.

textFile("/user/cloudera/sqoop\_import/order\_items").

  map(rec => rec.split(",")(4).toDouble).

  reduce((rev1, rev2) => rev1 + rev2)

val totalOrders = sc.

textFile("/user/cloudera/sqoop\_import/order\_items").

  map(rec => rec.split(",")(1).toInt).

  distinct().

count()

revenue/totalOrders will give average revenue.

Also in mysql validate to see the data

select order\_item\_order\_id , sum(order\_item\_subtotal) from order\_items group by order\_item\_order\_id order by 2;

### **Aggregating data sets using scala – by key**

Aggregations can be broadly categorized into totals and by key. As part of this topic we will covering aggregations - by key.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

By key aggregations are extensively used to group data by a key and get insights. Key could be time, category, department etc.

Here is the introduction video about by key operations. In this all the by key transformations such as groupByKey, reduceByKey, aggregateByKey are extensively covered.

* Number of orders by status using different by key operations
* Using countByKey

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val ordersMap = ordersRDD.

map(rec => (rec.split(",")(3), 1))

ordersMap.countByKey().foreach(println)

mysql> select order\_status, count(1) from orders group by order\_status;

* Using groupByKey
* groupByKey is not efficient in this case, as the size of the output is very small compared to size of input data.

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val ordersMap = ordersRDD.

  map(rec =>  (rec.split(",")(3), 1))

val ordersByStatus = ordersMap.

groupByKey().

map(t => (t.\_1, t.\_2.sum))

ordersMap.countByKey().foreach(println)

* Using reduceByKey. It uses combiner internally.
* Input data and output data for reduceByKey need to be of same type.
* + - **Combiner is implicit and uses the reduce logic.**

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val ordersMap = ordersRDD.

  map(rec =>  (rec.split(",")(3), 1))

val ordersByStatus = ordersMap.

reduceByKey((acc, value) => acc + value)

ordersMap.countByKey().foreach(println)

* Using aggregateByKey. It also uses combiner internally.
* Input data and output data for reduceByKey can be of different type.
* Also combiner logic can be different from reduce logic.

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val ordersMap = ordersRDD.

map(rec =>  (rec.split(",")(3), 1))

val ordersByStatus = ordersMap.

aggregateByKey(0),

(acc, value) => acc+1,

(acc, value) => acc+value

)

ordersByStatus.collect().foreach(println)

* Using combineByKey. It also uses combiner internally.
* Input data and output data for reduceByKey can be of different type.
* It is almost same as aggregateByKey and less frequently used.

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val ordersMap = ordersRDD.

map(rec =>  (rec.split(",")(3), 1))

val ordersByStatus = ordersMap.

combineByKey(

value => 1,

(acc: Int, value: Int) => acc+value,

(acc: Int, value: Int) => acc+value

)

ordersByStatus.collect().foreach(println)

Here is the video which talks about combiner. Combiner is the main difference between groupByKey and other by key transformations such as reduceByKey, aggregateByKey etc.

* For all the by key aggregations where the input data volume is significantly higher than output data volume, then we should not use groupByKey
* groupByKey is the only option where we can extend functionality for other operations such as sorting.
* Once we understand differences between different by key transformations, we need to apply for new requirements
* Requirement: Number of orders by order date and order status
* For this requirement, key is order\_date and order\_status
* We are doing aggregation, hence groupByKey is eliminated
* Combiner logic and reducer logic can be same, hence aggregateByKey and combineByKey are eliminated
* We are left with countByKey and reduceByKey, we can use either of them.
* Here is the logic implementing reduceByKey

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val ordersMapRDD = ordersRDD.

map(rec => ((rec.split(",")(1), rec.split(",")(3)), 1))

val ordersByStatusPerDay = ordersMapRDD.

reduceByKey((v1, v2) => v1+v2)

ordersByStatusPerDay.collect().foreach(println)

Here is the video which explains aggregation such as average by key

* Requirement: Generate average revenue per day
* Parse Orders (key order\_id)
* Parse Order items (key order\_item\_order\_id)
* Join Orders and Order item on the key
* Parse joined data and get (order\_date and order\_id) as key and order\_item\_subtotal as value. Here number of input records and output records will be same.
* Apply aggregate function (reduceByKey) to get revenue per order. Here number of output records will be number of distinct orders from order\_items table.
* Parse revenue per order and remove order\_id from the key. Here number of output records will be number of distinct orders from order\_items table.
* Apply aggregate function (combineByKey or aggregateByKey) to get total revenue per day. In this input type and output type for aggregate functions are different and also we need to have custom combiner logic, hence reduceByKey cannot be used.
* Now data will have order\_date as key and (revenue\_per\_day and total\_number\_of\_orders) as value. Apply map function to divide revenue with total number of orders per day.

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.

textFile("/user/cloudera/sqoop\_import/order\_items")

val ordersParsedRDD = ordersRDD.

map(rec => (rec.split(",")(0), rec))

val orderItemsParsedRDD = orderItemsRDD.

map(rec => (rec.split(",")(1), rec))

val ordersJoinOrderItems = orderItemsParsedRDD.

join(ordersParsedRDD)

val ordersJoinOrderItemsMap = ordersJoinOrderItems.

map(t =>

((t.\_2.\_2.split(",")(1), t.\_1), t.\_2.\_1.split(",")(4).

toFloat))

val revenuePerDayPerOrder = ordersJoinOrderItemsMap.

reduceByKey((acc, value) => acc + value)

val revenuePerDayPerOrderMap = revenuePerDayPerOrder.

map(rec => (rec.\_1.\_1, rec.\_2))

val revenuePerDay = revenuePerDayPerOrderMap.

aggregateByKey((0.0, 0))(

(acc, revenue) => (acc.\_1 + revenue, acc.\_2 + 1),

(total1, total2) =>

(total1.\_1 + total2.\_1, total1.\_2 + total2.\_2)

)

revenuePerDay.collect().foreach(println)

val avgRevenuePerDay = revenuePerDay.

map(x => (x.\_1, x.\_2.\_1/x.\_2.\_2))

**Here is the video which explains implementation of max by key. It includes SQL (but SQL is not relevant for the CCA certification)**

**max by key**

* Requirement: Get customer\_id with max revenue for each day
* customer\_id is in orders table and revenue need to be derived from order\_items table
* Apply join between orders and order\_items with order\_id as key
* Apply map function to get (order\_date and customer\_id) as key and order\_item\_subtotal as value
* Apply aggregate function to generate revenue per customer per day
* Apply another aggregate function using reduceByKey to get customer with maximum revenue per day. Logic is implemented using named function and invoked from lambda function of reduceByKey

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.

textFile("/user/cloudera/sqoop\_import/order\_items")

val ordersParsedRDD = ordersRDD.

map(rec => (rec.split(",")(0), rec))

val orderItemsParsedRDD = orderItemsRDD.

map(rec => (rec.split(",")(1), rec))

val ordersJoinOrderItems = orderItemsParsedRDD.

join(ordersParsedRDD)

val ordersPerDayPerCustomer = ordersJoinOrderItems.

map(rec => (

(rec.\_2.\_2.split(",")(1), rec.\_2.\_2.split(",")(2)),

rec.\_2.\_1.split(",")(4).toFloat)

)

val revenuePerDayPerCustomer = ordersPerDayPerCustomer.

reduceByKey((x, y) => x + y)

val revenuePerDayPerCustomerMap = revenuePerDayPerCustomer.

map(rec => (rec.\_1.\_1, (rec.\_1.\_2, rec.\_2)))

val topCustomerPerDaybyRevenue = revenuePerDayPerCustomerMap.

reduceByKey((x, y) => (if(x.\_2 >= y.\_2) x else y))

#Using regular function

def findMax(x: (String, Float), y: (String, Float)): (String, Float) = {

  if(x.\_2 >= y.\_2)

    return x

  else

    return y

}

val topCustomerPerDaybyRevenue = revenuePerDayPerCustomerMap.

reduceByKey((x, y) => findMax(x, y))

/////////////////////////////////////////////////////////////////////

For using hive

import org.apache.spark.sql.hive.HiveContext

val hiveContext = new HiveContext(sc);

hiveContext.sql("select o.order\_date , sum(oi.order\_item,subtotal)/

count(distinct oi.order\_item\_order\_id)) from orders o orders

join order\_items oi on o.order\_id = oi.order\_item\_order\_id group

by o.order\_date order by o.order\_date").collect().foreach(println)

hiveContext.sql("set spark.sql.shuffle.partitions=10")

### **Filtering data using scala**

As part of this topic, filtering of data sets using pyspark is covered.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Spark have filter transformation to apply different types of filtering. If the underlying logic in the lambda function returns false then the record will be discarded.

* Applying simple filter using filter transformation
* Get all the orders with status COMPLETE
* **Select distinct order\_status form orders;**

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.

filter(line => line.split(",")(3).

equals("COMPLETE")).

take(5).

foreach(println)

or

ordersRDD.

filter(\_.split(",")(3).

equals("COMPLETE")).

collect().

foreach(println)

* Get all the orders where status contains the word PENDING

val ordersRDD = sc.

  textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.

filter(line => line.split(",")(3).

contains("PENDING")).

take(5).

foreach(println)

* Get all the orders where order\_id is greater than 100

val ordersRDD = sc.

  textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.

filter(line => line.split(",")(0).toInt > 100).

take(5).

foreach(println)

* Boolean operation - or
* Get all the orders where order\_id > 100 or order\_status is in one of the pending states

val ordersRDD = sc.

  textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.

filter(line => line.split(",")(0).toInt > 100

|| line.split(",")(3).

contains("PENDING")).

take(5).

foreach(println)

* Combination of boolean operators and as well as or
* Make sure when combined, the boolean operations are properly bracketed
* For eg: (a and b or c) results in different than (a and (b or c)). (a and (b or c)) is correct way in most of the scenarios.

val ordersRDD = sc.

  textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.filter(line =>

line.split(",")(0).toInt > 1000 &&

    (line.split(",")(3).contains("PENDING")

|| line.split(",")(3).equals("CANCELLED"))).

  take(5).

foreach(println)

* Another example

val ordersRDD = sc.

  textFile("/user/cloudera/sqoop\_import/orders")

ordersRDD.filter(

line => line.split(",")(0).toInt > 1000 &&

    !line.split(",")(3).equals("COMPLETE")).

  take(5).

  foreach(println)

* Requirement: Check if there are cancelled orders with amount greater than 1000$
* Get cancelled orders from orders (filter)
* Join orders (filtered) and order\_items
* Generate revenue per order on the joined data set (reduceByKey)
* Discard all the cancelled orders whose revenue is less than 1000$

val ordersRDD = sc.

textFile("/user/cloudera/sqoop\_import/orders")

val orderItemsRDD = sc.

textFile("/user/cloudera/sqoop\_import/order\_items")

val ordersParsedRDD = ordersRDD.

filter(rec => rec.split(",")(3).contains("CANCELED")).

  map(rec => (rec.split(",")(0).toInt, rec))

val orderItemsParsedRDD = orderItemsRDD.

  map(rec =>

(rec.split(",")(1).toInt, rec.split(",")(4).toFloat))

val orderItemsAgg = orderItemsParsedRDD.

reduceByKey((acc, value) => (acc + value))

val ordersJoinOrderItems = orderItemsAgg.

join(ordersParsedRDD)

ordersJoinOrderItems.

filter(rec => rec.\_2.\_1 >= 1000).

take(5).

foreach(println)

* Using SQL

import org.apache.spark.sql.hive.HiveContext

val sqlContext = new HiveContext(sc)

sqlContext.sql("select \* from " +

"(select o.order\_id, " +

"sum(oi.order\_item\_subtotal) as order\_item\_revenue " +

"from orders o join order\_items oi " +

"on o.order\_id = oi.order\_item\_order\_id " +

"where o.order\_status = 'CANCELED' " +

"group by o.order\_id) q " +

"where order\_item\_revenue >= 1000")

.count()

### **Sorting and Ranking using scala - global**

Sorting can be broadly categorized into global and by key. As part of this topic we will covering sorting - global.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark
* Global sorting and ranking
* Here are few examples of using different transformations and actions such as sortByKey, top

val orders = sc.

textFile("/user/cloudera/sqoop\_import/orders")

orders.

map(rec => (rec.split(",")(0).toInt, rec)).

sortByKey().

collect().

foreach(println)

//for desending order sort

orders.

map(rec => (rec.split(",")(0).toInt, rec)).

sortByKey(false).

take(5).

foreach(println)

orders.

map(rec => (rec.split(",")(0).toInt, rec)).

top(5).

foreach(println)

* Global sorting using custom logic
* Here is the implementation of takeOrdered in which data is sorted by using field of our choice in our delimited data set.
* map function can be avoided

orders.map(rec => (rec.split(",")(0).toInt, rec)).

takeOrdered(5).

foreach(println)

orders.map(rec => (rec.split(",")(0).toInt, rec)).

takeOrdered(5)(Ordering[Int].reverse.on(x => x.\_1)).

foreach(println)

orders.

takeOrdered(5)(

Ordering[Int].on(x => x.split(",")(0).toInt)

).

foreach(println)

orders.

takeOrdered(5)(

Ordering[Int].reverse.on(x => x.split(",")(0).toInt)

).

foreach(println)

### **Sorting and Ranking using scala – by key**

Sorting can be broadly categorized into global and by key. As part of this topic we will covering sorting - by key.

* Load data from HDFS and store results back to HDFS using Spark
* Join disparate datasets together using Spark
* Calculate aggregate statistics (e.g., average or sum) using Spark
* Filter data into a smaller dataset using Spark
* Write a query that produces ranked or sorted data using Spark

Here is the video which covers basic scala list operations as well as Spark groupByKey transformation.

* Requirement: Sort products by price with in each category
* Read data from HDFS and apply map function to define key which is category
* Apply groupByKey to group all the products in the category

val products = sc.

textFile("/user/cloudera/sqoop\_import/products")

val productsMap = products.

map(rec => (rec.split(",")(1), rec))

val productsGroupBy = productsMap.

groupByKey()

productsGroupBy.

collect().

foreach(println)

* Get data sorted by product price per category
* With in each group use sorted function to sort the data in ascending or descending order

productsGroupBy.

  map(rec => (

    rec.\_2.toList.sortBy(k => k.split(",")(4).toFloat))

  ).

  take(100).

  foreach(println)

productsGroupBy.(only map deosen’t work flat map works)

  map(rec => (

    rec.\_2.toList.sortBy(k => -k.split(",")(4).toFloat))

  ).

  take(100).

  foreach(println)

productsGroupBy.

  flatMap(rec => (

rec.\_2.toList.sortBy(k => -k.split(",")(4).toFloat))

  ).

  take(100).

foreach(println)

* Requirement: Get top 3 priced products in each category
* Develop python function which get RDD and topN as parameters
* Compute top 3 priced products (if there are 10 products with top 3 prices, the RDD should give us all 10 products)

In mysql use the following shortcut…..

select max(product\_category\_id) from products where product\_category\_id = 53;

select max(product\_price) from products where product\_category\_id = 53;

def getTopDenseN(rec: (String, Iterable[String]), topN: Int): Iterable[String] = {

  var prodPrices: List[Float] = List()

  var topNPrices: List[Float] = List()

  var sortedRecs: List[String] = List()

  for(i <- rec.\_2) {

    prodPrices = prodPrices:+ i.split(",")(4).toFloat

  }

  topNPrices = prodPrices.distinct.sortBy(k => -k).take(topN)

  sortedRecs = rec.\_2.toList.sortBy(k => -k.split(",")(4).toFloat)

  var x: List[String] = List()

  for(i <- sortedRecs) {

    if(topNPrices.contains(i.split(",")(4).toFloat))

      x = x:+ i

  }

  return x

}

//for invoking the funcn

productsMap.groupByKey().flatMap(x => getTopDenseN(x,2)).collect().foreach(println)

* Here is the code which invokes getTopDenseN

Here is the video which covers ranking using Spark SQL using windowing functions. This might not be very important for CCA certification.

* Different queries to perform sorting and ranking using Spark SQL.

#Sorting using queries

#Global sorting and ranking

select \* from products order by product\_price desc;

select \* from products order by product\_price desc limit 10;

#By key sorting

#Using order by is not efficient, it serializes

select \* from products order by product\_category\_id, product\_price desc;

#Using distribute by sort by (to distribute sorting and scale it up)

select \* from products distribute by product\_category\_id sort by product\_price desc;

#By key ranking (in Hive we can use windowing/analytic functions)

select \* from (select p.\*,

dense\_rank() over (partition by product\_category\_id order by product\_price desc) dr

from products p

distribute by product\_category\_id) q

where dr <= 2 order by product\_category\_id, dr;

Data Analysis using Hive,Impala and avro

As per the certification curriculum we need to work on DDL aspects of Hive and Impala as well as concepts around avro-tools.

Use Data Definition Language (DDL) to create tables in the Hive metastore for use by Hive and Impala.

* Read and/or create a table in the Hive metastore in a given schema
* Extract an Avro schema from a set of datafiles using avro-tools
* Create a table in the Hive metastore using the Avro file format and an external schema file
* Improve query performance by creating partitioned tables in the Hive metastore
* Evolve an Avro schema by changing JSON files

 There will be separate topics for each of these items highlighted as part of the certification

Some code that needs to implemented before the section are as follows :

other steps in video 70 cd /etc/hadoop/conf view core-site.xml hadoop fs -ls /user/hive/warehouse mysql –u hive –p ,use metastore.db,show tables;

create table cca\_demo (t int, s string)

hadoop fs -ls /user/hive/warehouse/cca\_demo

insert into table cca\_demo values(1, 'testing') ,

hadoop fs -cat /user/hive/warehouse/cca\_demo/000000\_0

select \* from cca\_demo;

select t from cca\_demo

describe cca\_demo

describe formatted cca\_demo

mysql –u hive –p

scala> import org.apache.spark.sql.hive.HiveContext;

val hiveContext = new HiveContext(sc)

hivecontext.sql(“select count(1) from order\_items”).foreach(println)

impala-shell

showtables;

create table impala\_cca\_hive\_demo(t int, s string)

show tables;

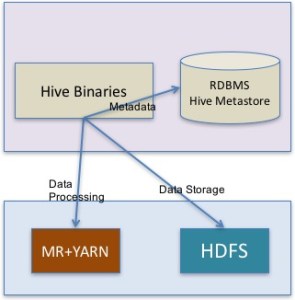
select count(1) form order\_items

for warehouse directory I will use this to check hive default metastore database.

### **Getting Started – Apache Hive**

#### **Introduction**

Apache Hive is open source tool to provide an interface to create logical databases for structured data and then provide interface to load/insert data as well as process data using Hive Query Language (similar to SQL). Data will be stored in HDFS. If data is structured then tables can be created. Metadata generated with table creation commands will be stored in RDBMS database such as MySQL.



As depicted in the picture, Hive requires

* Hive binaries (a bunch of jar files which have compiled java code)
* Hadoop cluster (with HDFS and YARN+MR, to store data using HDFS and YARN+MR to process the data)
* RDBMS database (to store metadata)
* Most of the hive queries will be compiled into map reduce programs and submitted as Map Reduce jobs.
* As technology stack evolve, hive is supported with other execution frameworks such as Tez, Spark etc
* For certification point of view, only Hive DDL as well as data load commands are important

Here is the introduction video about the plan for the preparation for "Data Analysis" category of the certification.

### **Creating tables In Hive/Impala**

#### **Introduction**

For the certification perspective Hive/Impala are covered only to create tables and read data from tables. Actual processing is covered as part of Transform, Stage and Store.

Highlighted certification item is covered as part of this topic.

Use Data Definition Language (DDL) to create tables in the Hive metastore for use by Hive and Impala.

* **Read and/or create a table in the Hive metastore in a given schema**
* Extract an Avro schema from a set of datafiles using avro-tools
* Create a table in the Hive metastore using the Avro file format and an external schema file
* Improve query performance by creating partitioned tables in the Hive metastore
* Evolve an Avro schema by changing JSON files

Here is the video to understand concepts behind hive databases and tables.

* Creating necessary databases in hive

CREATE DATABASE IF NOT EXISTS cards;

CREATE DATABASE IF NOT EXISTS retail\_ods;

CREATE DATABASE retail\_edw;

CREATE DATABASE retail\_stage;

cd /etc/hive/conf

dfs -ls /user/hive/warehouse;

view hive-site.xml

grep warehouse \*

use cards;

describe formatted deck\_of\_cards

--which was the table name..

drop table deck\_of\_cards

dfs -ls /user/hive/warehouse/cards.db/deck\_of\_cards;

* Creating necessary tables in hive

CREATE TABLE deck\_of\_cards (

COLOR string,

SUIT string,

PIP string)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

describe formatted deck\_of\_cards;

LOAD DATA LOCAL INPATH '/home/cloudera/Desktop/deckofcards.txt' INTO TABLE deck\_of\_cards;

CREATE EXTERNAL TABLE deck\_of\_cards\_external (

COLOR string,

SUIT string,

PIP string)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE

LOCATION '/user/hive/warehouse/cards.db/deck\_of\_cards';

//for mysql only needs correction for joining the tables;

select d.department\_name, round(sum(order\_item\_subtotal),2) from departm

ents d right outer join categories c on d.department\_id = category\_department\_id

join products p on c.category\_id = p.product\_category\_id join order\_items o on

p.product\_id = o.order\_item\_product\_id group by department\_name;

use retail\_ods;

CREATE TABLE categories (

category\_id int,

category\_department\_id int,

category\_name string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

describe formatted categories;

CREATE 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;

CREATE TABLE departments (

department\_id int,

department\_name string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

CREATE TABLE orders (

order\_id int,

order\_date string,

order\_customer\_id int,

order\_status string

)

PARTITIONED BY (order\_month string)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

CREATE TABLE order\_items (

order\_item\_id int,

order\_item\_order\_id int,

order\_item\_order\_date string,

order\_item\_product\_id int,

order\_item\_quantity smallint,

order\_item\_subtotal float,

order\_item\_product\_price float

)

PARTITIONED BY (order\_month string)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

CREATE TABLE orders\_bucket (

order\_id int,

order\_date string,

order\_customer\_id int,

order\_status string

)

CLUSTERED BY (order\_id) INTO 16 BUCKETS

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

CREATE TABLE order\_items\_bucket (

order\_item\_id int,

order\_item\_order\_id int,

order\_item\_order\_date string,

order\_item\_product\_id int,

order\_item\_quantity smallint,

order\_item\_subtotal float,

order\_item\_product\_price float

)

CLUSTERED BY (order\_item\_order\_id) INTO 16 BUCKETS

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

CREATE TABLE products (

product\_id int,

product\_category\_id int,

product\_name string,

product\_description string,

product\_price float,

product\_image string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

-- Create edw tables (following dimension model)

use retail\_edw;

CREATE TABLE products\_dimension (

product\_id int,

product\_name string,

product\_description string,

product\_price float,

product\_category\_name string,

product\_department\_name string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

CREATE TABLE order\_fact (

order\_item\_order\_id int,

order\_item\_order\_date string,

order\_item\_product\_id int,

order\_item\_quantity smallint,

order\_item\_subtotal float,

order\_item\_product\_price float

)

PARTITIONED BY (product\_category\_department string)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

Here is the video to recap Sqoop theoretically

* Use sqoop hive-import to import data into hive tables
* All tables will be created in hive default database, data will be compressed

sqoop import-all-tables

--num-mappers 1

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--hive-import

--hive-overwrite

--create-hive-table

--compress

--compression-codec org.apache.hadoop.io.compress.SnappyCodec

--outdir java\_files

Here is the video to understand loading data into hive

* Copy data from MySQL to local file system

Some detour

Vi test.txt

Interal 1,2,3,4,5,6,7,8,9,10

--hive

Create table test(t string);

LOAD DATA LOCAL INPATH ‘/home/cloudera/test.txt’ OVERWRITE INTO TABLE test;

Select \* from test;

* Enable file\_priv to retail\_dba

mysql -u root -p #if password enabled, else "mysql -u root"

select user from mysql.user;

describe mysql.user;

select user ,file\_priv from mysql.user;

update mysql.user set file\_priv = 'Y' where user = 'retail\_dba';

commit;

exit;

use retail\_db;

select \* from orders into outfile '/tmp/categories01.psv' fields terminated by '|' lines terminated by '\n';

select \* from orders into outfile '/tmp/categories02.psv' fields terminated by ',' lines terminated by '\n';

show table;

select \* form categories limit 10;

load data local inpath '/tmp/categories01.psv' overwrite into table categories;

load data local inpath '/tmp/categories01.psv' into table categories; (for not overwirite previous data)

dfs -cat /user/hive/warehouse/categories/\*;

hadoop fs –mkdir /user/cloudera/categories

hadoop fs -put /tmp/categories01.psv /user/cloudera/categoires

* Restart mysql database service mysql restart
* Make sure you understand table structure, delimiter, partition etc, run mysql export command
* Launch mysql cli mysql -u retail\_dba -p
* Run this code to generate pipe "|" delimited files

select \* from categories into outfile '/tmp/categories01.psv' fields terminated by '|' lines terminated by 'n';

select \* from customers into outfile '/tmp/customers.psv' fields terminated by '|' lines terminated by 'n';

select \* from departments into outfile '/tmp/departments.psv' fields terminated by '|' lines terminated by 'n';

select \* from products into outfile '/tmp/products.psv' fields terminated by '|' lines terminated by 'n';

* orders and order\_items cannot be loaded as tables created above are partitioned and structure is different between mysql and hive
* Code to LOAD data from local file system to hive table

load data local inpath '/tmp/categories01.psv' overwrite into table categories;

load data local inpath '/tmp/customers.psv' overwrite into table customers;

load data local inpath '/tmp/departments.psv' overwrite into table departments;

load data local inpath '/tmp/products.psv' overwrite into table products;

* Removing overwrite will append to existing tables

Here is the video to understand hive insert in detail

* Prepare orders on mysql database

select \* from orders into outfile '/tmp/orders.psv' fields terminated by '|' lines terminated by 'n';

* Create orders\_stage under hive database retail\_stage

use retail\_stage;

CREATE TABLE orders\_stage (

order\_id int,

order\_date string,

order\_customer\_id int,

order\_status string

)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '|'

STORED AS TEXTFILE;

* Load data to orders\_stage table in retail\_stage database

load data local inpath '/tmp/orders.psv' overwrite into table orders\_stage;

* Insert data into orders by selecting from orders\_stage table

set hive.exec.dynamic.partition;

set hive.exec.dynamic.partition.mode;

insert overwrite table retail\_ods.orders partition (order\_month)

select order\_id, order\_date, order\_customer\_id, order\_status,

substr(order\_date, 1, 7) order\_month from retail\_stage.orders\_stage;

* Similarly create stage table for order\_items, generate data in mysql, load into hive table in retail\_stage and then insert into partitioned table in retail\_ods using join
* We have included order\_month in order\_items to partition the table and hence we need to join with orders table to derive order\_month while inserting data into order\_items partitioned table

insert overwrite table order\_items partition (order\_month)

select oi.order\_item\_id, oi.order\_item\_order\_id, o.order\_date,

oi.order\_item\_product\_id, oi.order\_item\_quantity, oi.order\_item\_subtotal,

oi.order\_item\_product\_price, substr(o.order\_date, 1, 7)

order\_month from retail\_stage.order\_items oi join retail\_stage.orders\_stage o

on oi.order\_item\_order\_id = o.order\_id;

For checking partion and other things

set hive.exec.dynamic.partition;

set hive.exec.dynamic.partition.mode;

### **Using avro-tools**

#### **Introduction**

Avro file format is one of the popular file formats in Hadoop based applications. Avro is an Apache™ open source project that provides data serialization and data exchange services for Hadoop®. These services can be used together or independently. Using Avro, big data can be exchanged between programs written in any language

Highlighted certification item is covered as part of this topic.

Use Data Definition Language (DDL) to create tables in the Hive metastore for use by Hive and Impala.

* Read and/or create a table in the Hive metastore in a given schema
* Extract an Avro schema from a set of datafiles using avro-tools
* Create a table in the Hive metastore using the Avro file format and an external schema file
* Improve query performance by creating partitioned tables in the Hive metastore
* Evolve an Avro schema by changing JSON files

### **Using Avro in Hive**

Avro file format is one of the popular file formats in Hadoop based applications. Avro is an Apache™ open source project that provides data serialization and data exchange services for Hadoop®. These services can be used together or independently. Using Avro, big data can be exchanged between programs written in any language

Highlighted certification item is covered as part of this topic.

Use Data Definition Language (DDL) to create tables in the Hive metastore for use by Hive and Impala.

* Read and/or create a table in the Hive metastore in a given schema
* Extract an Avro schema from a set of datafiles using avro-tools
* Create a table in the Hive metastore using the Avro file format and an external schema file
* Improve query performance by creating partitioned tables in the Hive metastore
* Evolve an Avro schema by changing JSON files

Here is the video which demonstrates creation of hive tables using avro file format and external schema file

Here are the code snippets to create tables with avro format on the retail\_db data

* Run sqoop import-all-tables command with --as-avrodatafile

sqoop import-all-tables

-m 4

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db"

--username=retail\_dba

--password=cloudera

--as-avrodatafile

--warehouse-dir=/user/cloudera/sqoop\_import

sqoop import-all-tables \

-m 4 \

--connect "jdbc:mysql://quickstart.cloudera:3306/retail\_db" \

--username=retail\_dba \

--password=cloudera \

--as-avrodatafile \

--warehouse-dir=/user/cloudera/sqoop\_import

hadoop fs -ls /user/hive/warehouse/retail\_stage.db

/user/hive/warehouse/retail\_stage.db/departments

hadoop fs -get /user/hive/warehouse/retail\_stage.db/departments

ls -ltr \*.java

ls -ltr \*.avsc

cd departments

ls -ltr

view part-m-00000.avro

vi departments.txt

2,Fitness

3,Testing

<departments>

<record>

<department\_id>2</department\_id>

<department\_name>Testing</department\_name>

</record>

<record>

<department\_id>3</department\_id>

<department\_name>Testing</department\_name>

</record>

</departments>

avro-tools

avro-tools getmeta part-m-00000.avro

avro-tools getschema part-m-00000.avro

avro-tools tojson part-m-00000.avro

avro-tools tojson part-m-00000.avro > departments.json

cat departments.json

avro-tools fromjson departments.json

avro-tools getschema part-m-00000.avro >departments.avsc

cat departments.avsc

avro-tools fromjson departments.json --schema-file departments.avsc

avro-tools totext part-m-00000.avro departments.txt

ls -ltr

avro-tools totext

///lecture 80

hadoop fs -ls /user/hive/warehouse/retail\_stage.db

vi sqoop\_import\_departments.avsc

hadoop fs -mkdir /user/cloudera/retail\_stage

hadoop fs -copyFromLocal \*.avsc /user/cloudera/retail\_stage

hadoop fs -ls /user/hive/warehouse/retail\_stage.db/departments

hadoop fs -ls /user/cloudera/retail\_stage

drop table categories;(in hive)

hadoop fs -mv /user/cloudera/retial\_stage/categories.avsc /user/cloudera/retail\_stage/sqoop\_import\_categories.avsc

CREATE EXTERNAL TABLE categories

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/categories'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_categories.avsc');

select \* from categories

CREATE EXTERNAL TABLE categories

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/categories'

TBLPROPERTIES ('avro.schema.literal' = '{

"type" : "record"

"name" : "sqoop\_import\_departments",

"doc" : "Sqoop import of departments",

"fields" : [ {

"name" : department\_id,

"type" : ["int","null"],

"columnName" : "department\_id",

"sqlType" : "4"

},{

"name" : "department\_name",

"type" : ["string","null"],

"columnName" : "department\_name",

"sqlType" : "12"

}],

"tableName" : "departments"

}

');

////for the impala shell command line

impala-shell

show databases;

show tables;

> CREATE EXTERNAL TABLE orders

STORED AS AVRO

LOCATION 'hdfs:///user/hive/warehouse/retail\_db/orders'

TBLPROPERTIES('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_orders.avsc');

//for the other section of the avaro tools

impala-shell

show databases;

show tables;

> CREATE EXTERNAL TABLE orders

STORED AS AVRO

LOCATION 'hdfs:///user/hive/warehouse/retail\_db/orders'

TBLPROPERTIES('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_orders.avsc');

* It will copy data to HDFS location /user/cloudera/sqoop\_import and generate metadata files locally using avsc extension in the location where ever sqoop import-all-tables is ran
* Copy avsc files to HDFS location

hadoop fs -mkdir /user/cloudera/avsc\_files

hadoop fs -put ~/\*.avsc /user/cloudera/avsc\_files

* Create hive external tables
* CREATE DATABASE IF NOT EXISTS retail\_stage;
* USE retail\_stage;
* CREATE EXTERNAL TABLE categories
* STORED AS AVRO
* LOCATION 'hdfs:///user/cloudera/sqoop\_import/categories'
* TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/avsc\_files/sqoop\_import\_categories.avsc');
* CREATE EXTERNAL TABLE customers
* STORED AS AVRO
* LOCATION 'hdfs:///user/cloudera/sqoop\_import/customers'
* TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/avsc\_files/sqoop\_import\_customers.avsc');
* CREATE EXTERNAL TABLE departments
* STORED AS AVRO
* LOCATION 'hdfs:///user/cloudera/sqoop\_import/departments'
* TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/avsc\_files/sqoop\_import\_departments.avsc');
* CREATE EXTERNAL TABLE orders
* STORED AS AVRO
* LOCATION 'hdfs:///user/cloudera/sqoop\_import/orders'
* TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/avsc\_files/sqoop\_import\_orders.avsc');
* CREATE EXTERNAL TABLE order\_items
* STORED AS AVRO
* LOCATION 'hdfs:///user/cloudera/sqoop\_import/order\_items'
* TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/avsc\_files/sqoop\_import\_order\_items.avsc');
* CREATE EXTERNAL TABLE products
* STORED AS AVRO
* LOCATION 'hdfs:///user/cloudera/sqoop\_import/products'
* TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/avsc\_files/sqoop\_import\_products.avsc');

Some other data

select \* from orders limit 10

select order\_status, count(1) from orders

where from\_unixtime(cast(substr(order\_date, 1, 10) as int)) like '2014-01%' group by order\_status;

cd /tmp/

ls -ltr

cd cloudera

ls -ltr

vi hive.log

date

mapred job -counter job\_1484043401127\_0007( that was the job id of the hive query)

mkdir avro

cd avro

cp soop\_import\_orders.avsc orders\_part\_avro.avsc

cp: overwrite 'orders\_part\_avro.avsc'?yes

vi orders\_part\_avro.avsc

"type" : "record"

"name" : "orders\_part\_avro",

"doc" : "Hive Avro Partitioned table orders\_part\_avro",

"fields" : [ {

"name" : "order\_id",

"type" : ["int","null"],

},{

"name" : "order\_date",

"type" : ["long","null"],

},{

"name" : "order\_customer\_id",

"type" : ["int","null"],

},{

"name" : "order\_status",

"type" : ["string","null"],

}],

"tableName" : "orders\_part\_avro"

### **Hive Paritioning**

Avro file format is one of the popular file formats in Hadoop based applications. Avro is an Apache™ open source project that provides data serialization and data exchange services for Hadoop®. These services can be used together or independently. Using Avro, big data can be exchanged between programs written in any language

Highlighted certification item is covered as part of this topic.

Use Data Definition Language (DDL) to create tables in the Hive metastore for use by Hive and Impala.

* Read and/or create a table in the Hive metastore in a given schema
* Extract an Avro schema from a set of datafiles using avro-tools
* Create a table in the Hive metastore using the Avro file format and an external schema file
* Improve query performance by creating partitioned tables in the Hive metastore
* Evolve an Avro schema by changing JSON files

Here is the video which explains the process of creating hive partitioned table using avro file format

* Here is the example for creating tables with different names than column names in avsc files

CREATE TABLE orders\_part\_avro (

order\_id int,

order\_date bigint,

order\_customer\_id int,

order\_status string

)

PARTITIONED BY (order\_month string)

STORED AS AVRO

LOCATION 'hdfs:///user/cloudera/sqoop\_import/orders\_part\_avro'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/avsc\_files/orders\_part\_avro.avsc');

dfs -ls hdfs:///user/hive/warehouse/retail\_stage.db/orders\_part\_avro;

select \* from orders\_part\_avro limit 10;

drop table orders\_part\_avro;

select \* from orders limit 10

select order\_status, count(1) from orders

where from\_unixtime(cast(substr(order\_date, 1, 10) as int)) like '2014-01%' group by order\_status;

cd /tmp/

ls -ltr

cd cloudera

ls -ltr

vi hive.log

date

mapred job -counter job\_1484043401127\_0007( that was the job id of the hive query)

mkdir avro

cd avro

cp soop\_import\_orders.avsc orders\_part\_avro.avsc

cp: overwrite 'orders\_part\_avro.avsc'?yes

vi orders\_part\_avro.avsc

select order\_status count(1) from order\_part\_avro where order\_month = '2014-01';

### **Evolve Avro Schema(81 nd 82)**

Avro file format is one of the popular file formats in Hadoop based applications. Avro is an Apache™ open source project that provides data serialization and data exchange services for Hadoop®. These services can be used together or independently. Using Avro, big data can be exchanged between programs written in any language.

Highlighted certification item is covered as part of this topic.

Use Data Definition Language (DDL) to create tables in the Hive metastore for use by Hive and Impala.

* Read and/or create a table in the Hive metastore in a given schema
* Extract an Avro schema from a set of datafiles using avro-tools
* Create a table in the Hive metastore using the Avro file format and an external schema file
* Improve query performance by creating partitioned tables in the Hive metastore
* Evolve an Avro schema by changing JSON files

Here is the video which explains the concepts of evolving Avro schema

dfs -ls hdfs:///user/hive/warehouse/retail\_stage.db/orders\_part\_avro;

select count(1) from order\_part\_avro where order\_month = '2014-01';

hadoop fs -ls /user/hive/warehouse/retail\_stage.db;

cd avro;

ls -ltr

mv \*avsc avro

cd avro

hadoop fs -put \*.avsc /user/cloudera/retail\_stage/.

In hive

use retail\_stage;

create all tables;

CREATE EXTERNAL TABLE categories

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/categories'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_categories.avsc');

CREATE EXTERNAL TABLE customers

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/customers'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_customers.avsc');

CREATE EXTERNAL TABLE departments

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/departments'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_departments.avsc');

CREATE EXTERNAL TABLE orders

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/orders'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_orders.avsc');

CREATE EXTERNAL TABLE order\_items

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/order\_items'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_order\_items.avsc');

CREATE EXTERNAL TABLE products

ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.avro.AvroSerDe'

STORED AS INPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat'

OUTPUTFORMAT 'org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat'

LOCATION 'hdfs:///user/hive/warehouse/retail\_stage.db/products'

TBLPROPERTIES ('avro.schema.url'='hdfs://quickstart.cloudera/user/cloudera/retail\_stage/sqoop\_import\_products.avsc');

//Start impala serivice....

invalidate matadata;,

General workarounds

**To restart the cloudera manger**

ls -ltr|grep cloudera

ls -ltr|grep manager

sudo ./cloudera-manager --enterprise –force

cat /etc/password

jps

whoami

sudo -u hdfs jps

sudo -u yarn jps

cat /etc/passwrd

sudo -u spark jps

ps -ef|grep -i history

ps -fu hdfs

**For using shell commands**

cd shell\_demo

mkdir shell\_demo

cd shell\_demo

vi sqoop.sh

ls -ltr

chmod 764 sqoop.sh

/sqoop.sh

sqoop list-databases \

--connect "jdbc:mysql://quickstart.cloudera:3306" \

--username retail\_dba \

--password cloudera

export PATH=$PATH:/home/cloudera/shell\_demo

echo $PATH

sqoop.sh , /home/cloudera/shell\_demo/sqoop.sh , hive -e "show databases" , vi hive\_demo.sh ,

hive -e "show databases" , chmod 764 hive\_demo.sh , hive << EOF

show databases;

show tables;

EOF>>