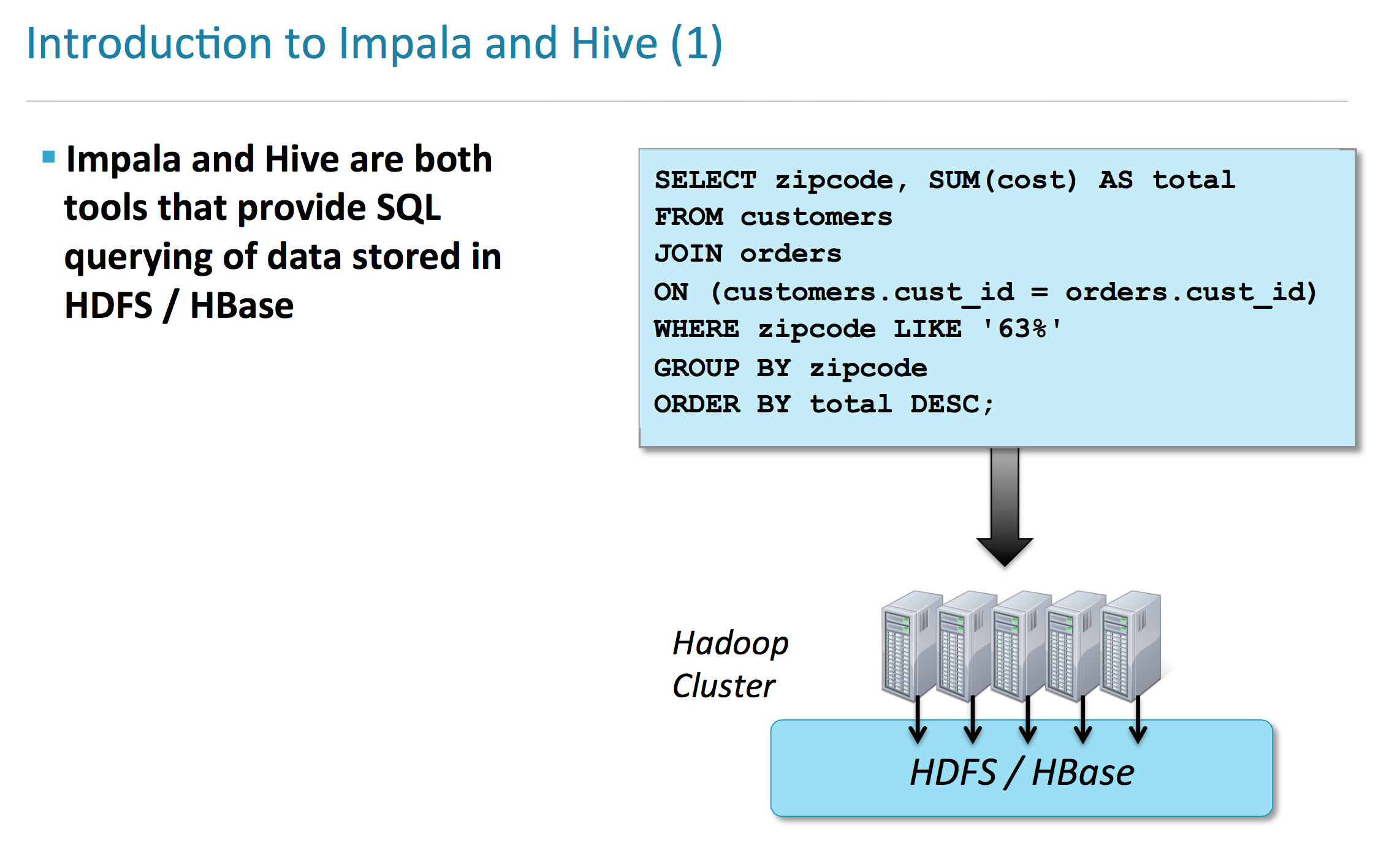
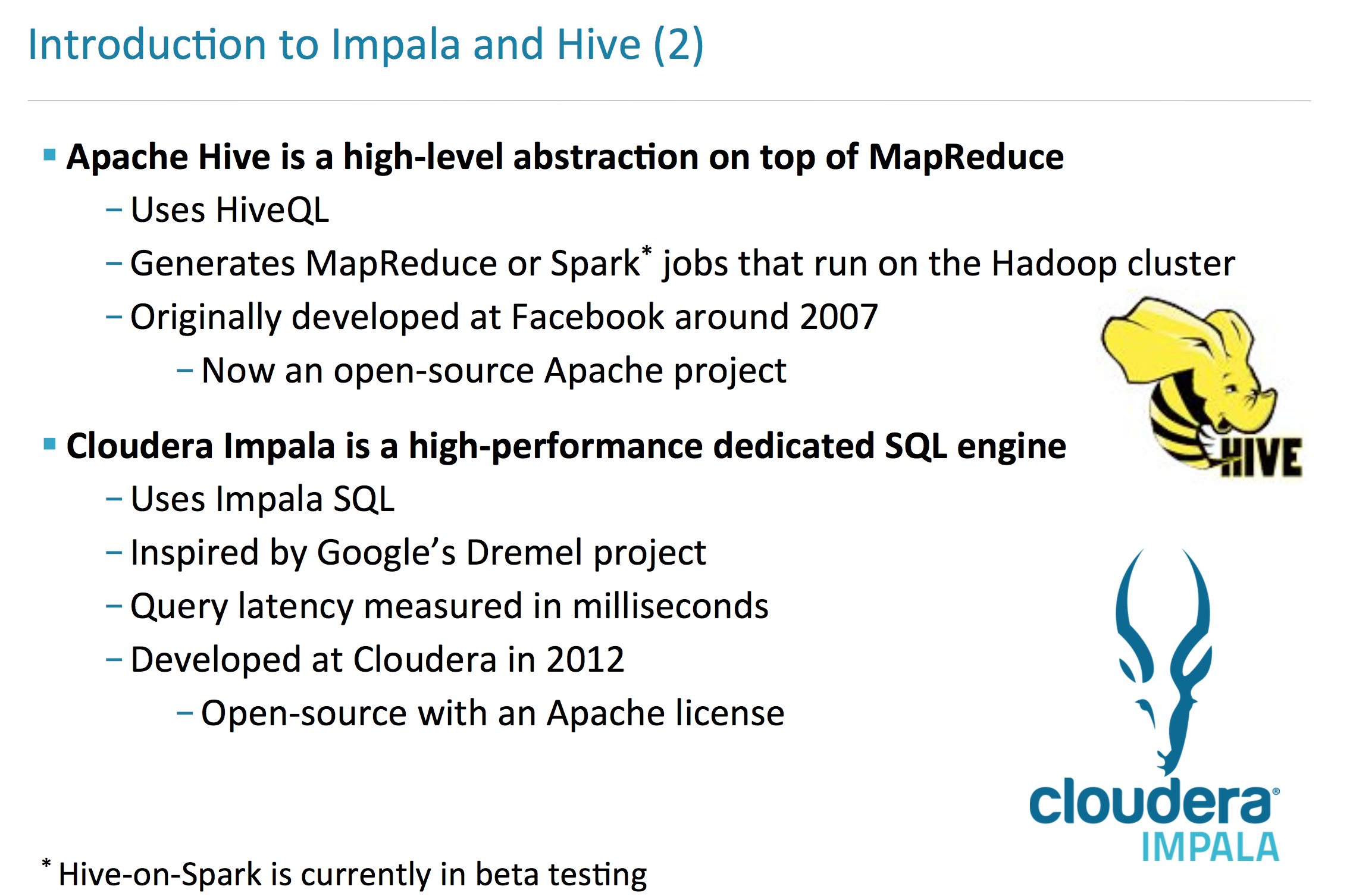
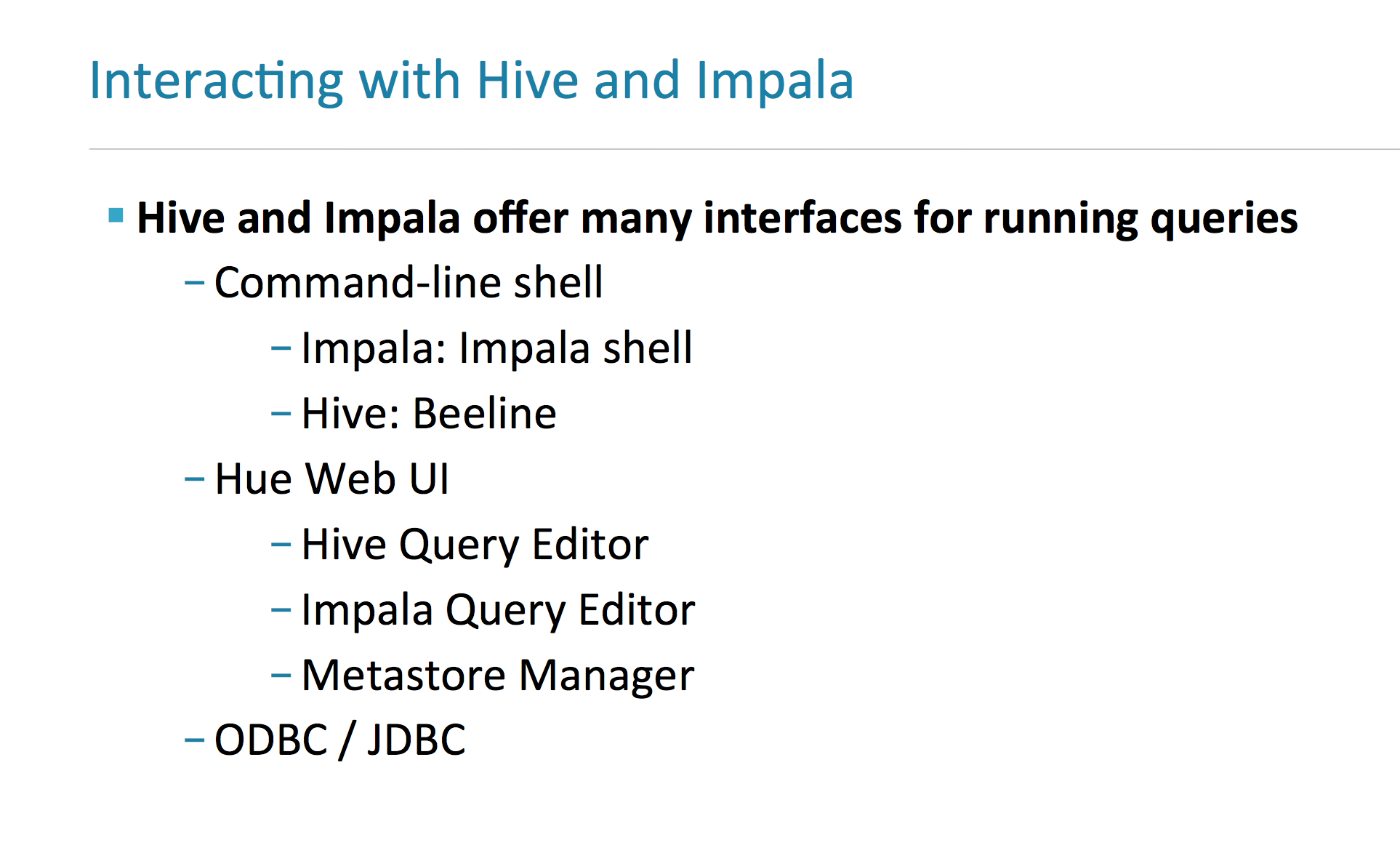
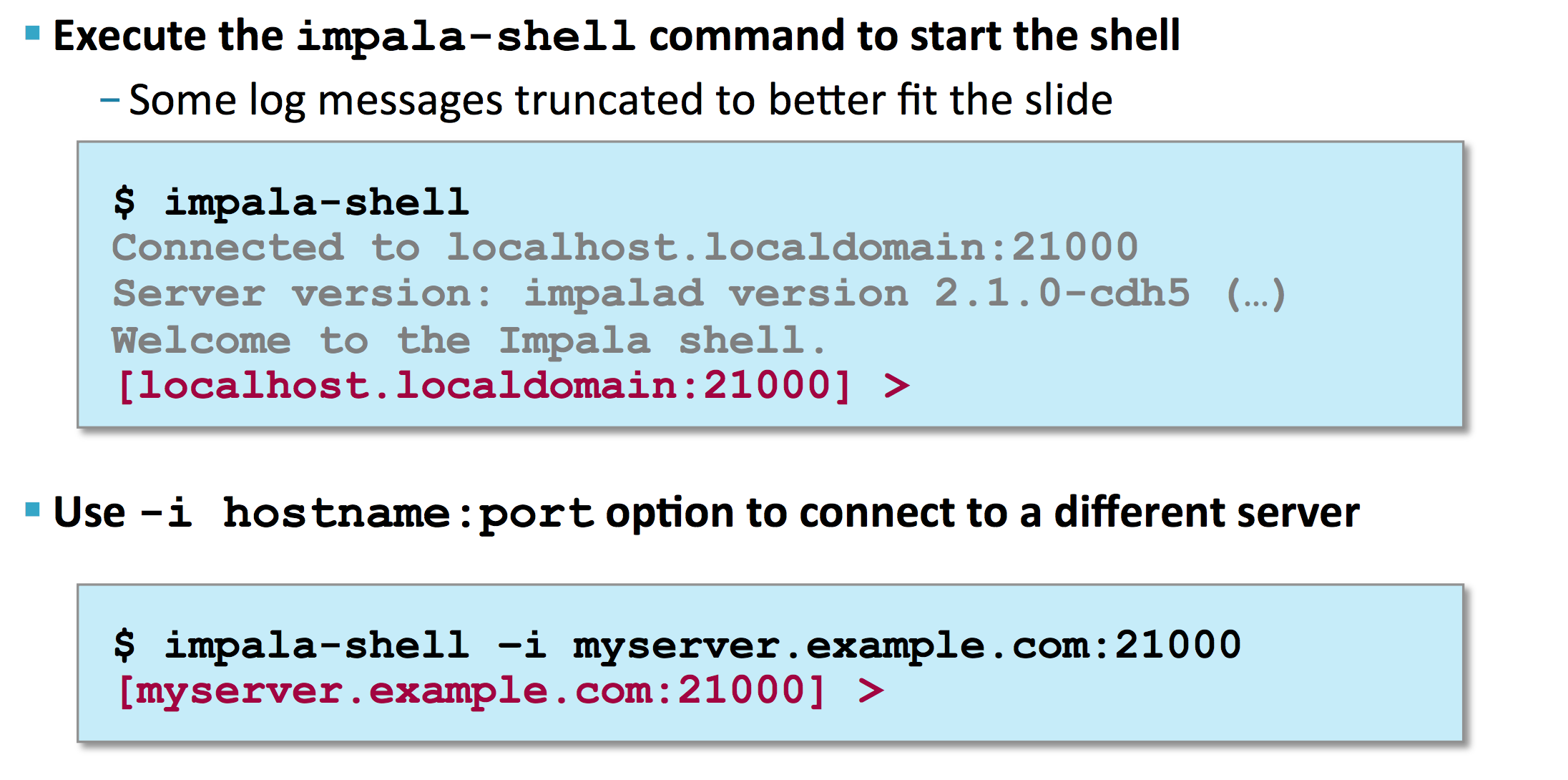
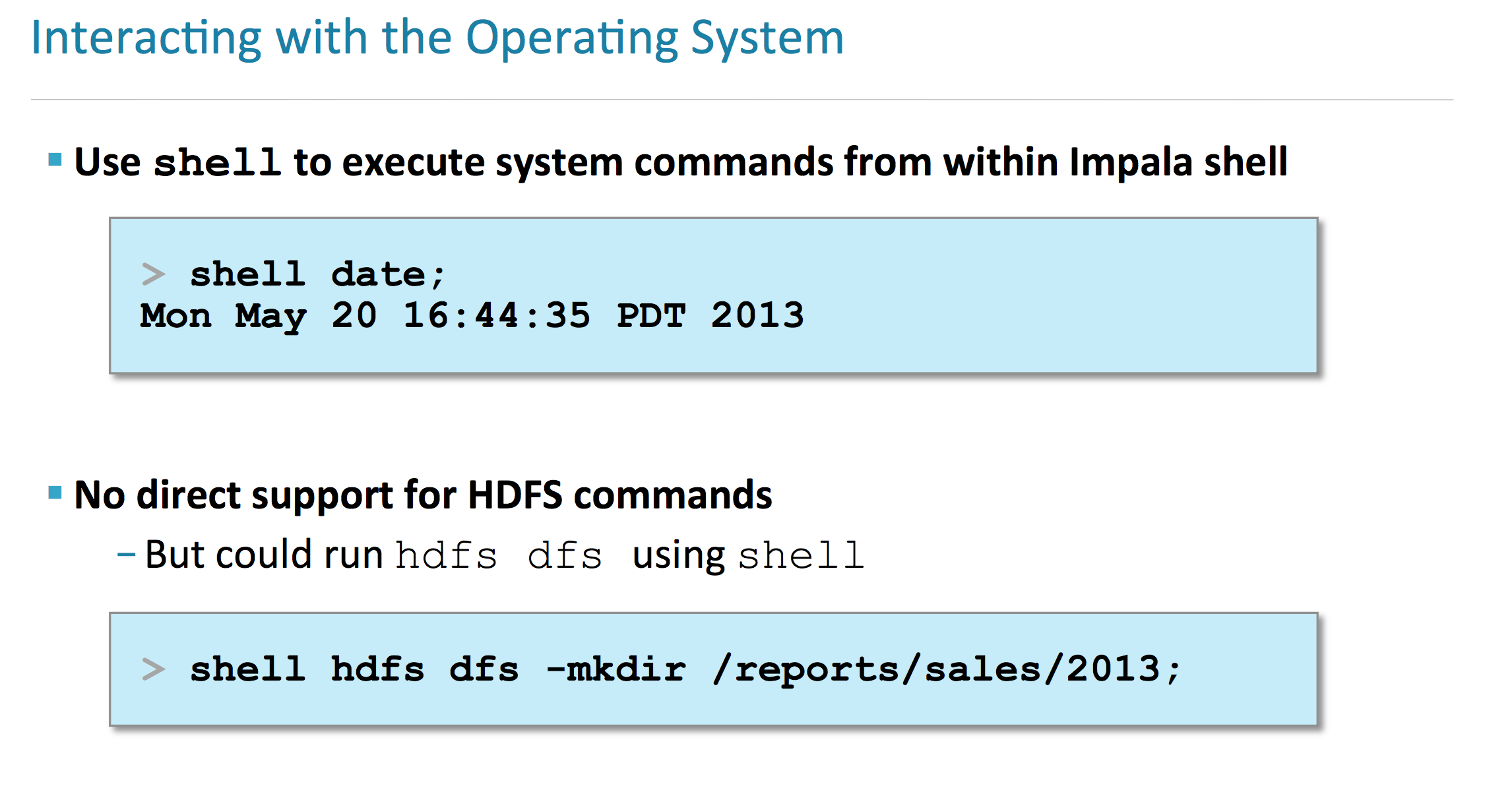
Hive and Impala











**Run queries from shell prompt**

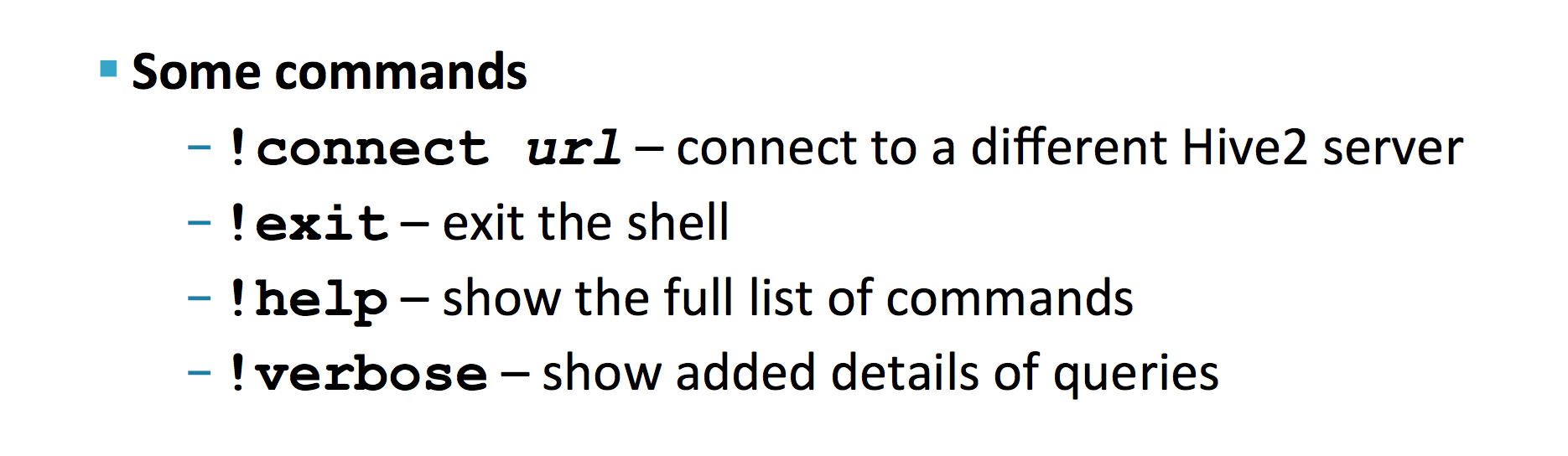
[cloudera@quickstart ~]$ impala-shell -f 'select \* from t1';

store results in a file with delimiter

[cloudera@quickstart ~]$ impala-shell -q 'select \* from t1' -o result\_t1.txt --delimited --output\_delimiter=','

**Connect to Beeline**

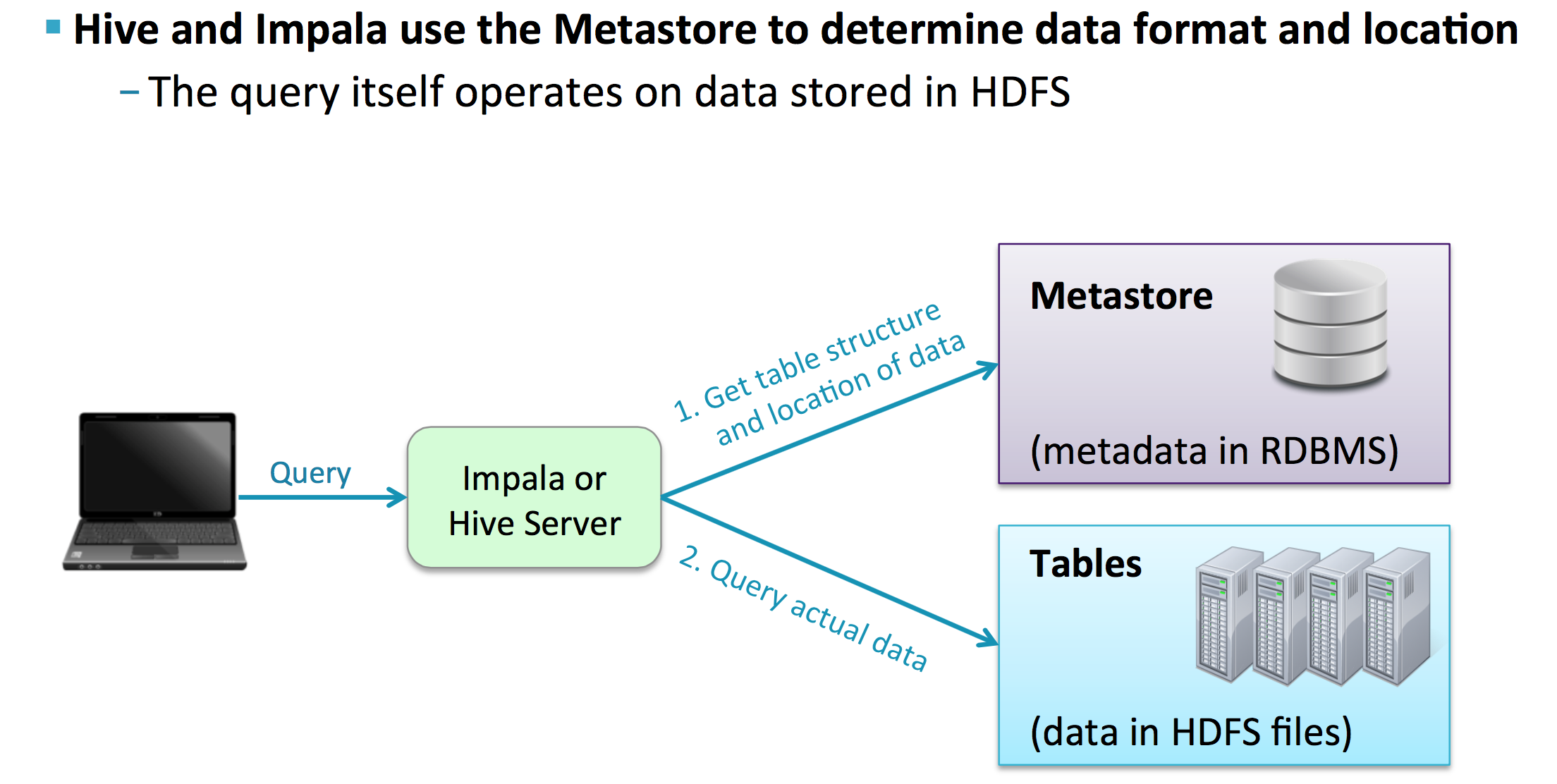
[cloudera@quickstart ~]$ beeline -u jdbc:hive2://localhost:10000 -n cloudera -p cloudera



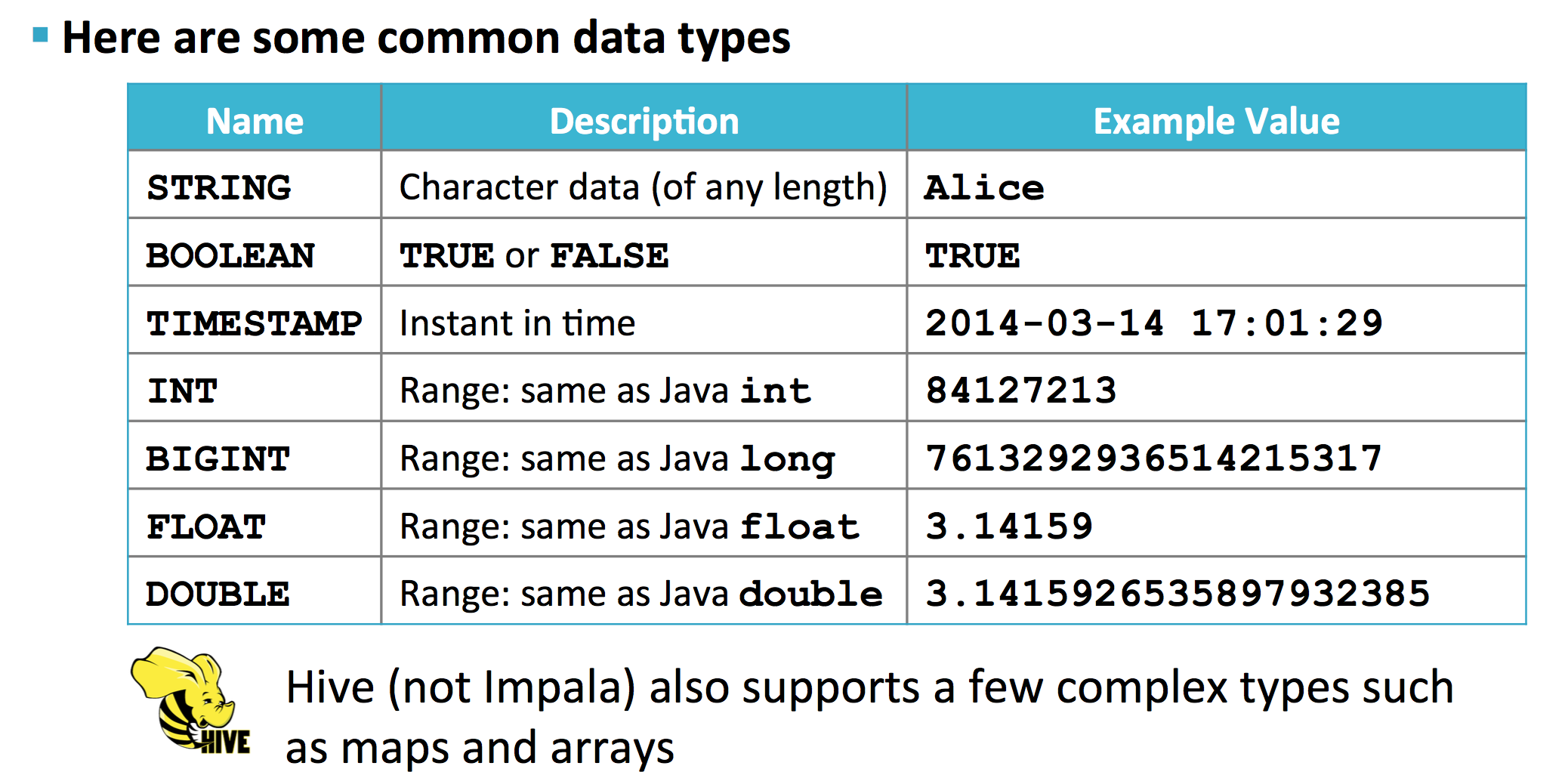
**Running beeline query from shell**

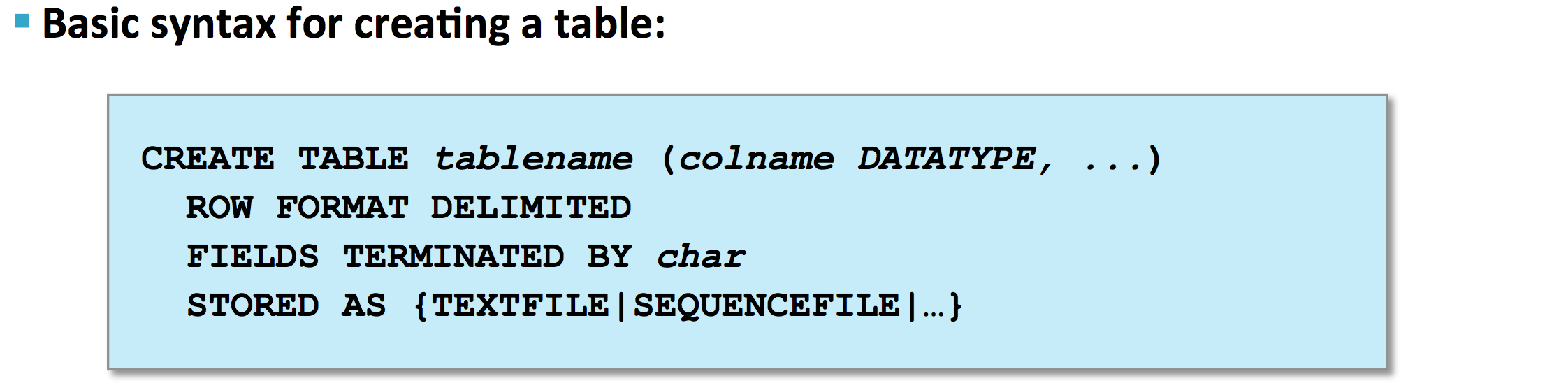
[cloudera@quickstart ~]$ beeline -u jdbc:hive2://localhost:10000 -n cloudera -p cloudera -e 'select \* from t1'

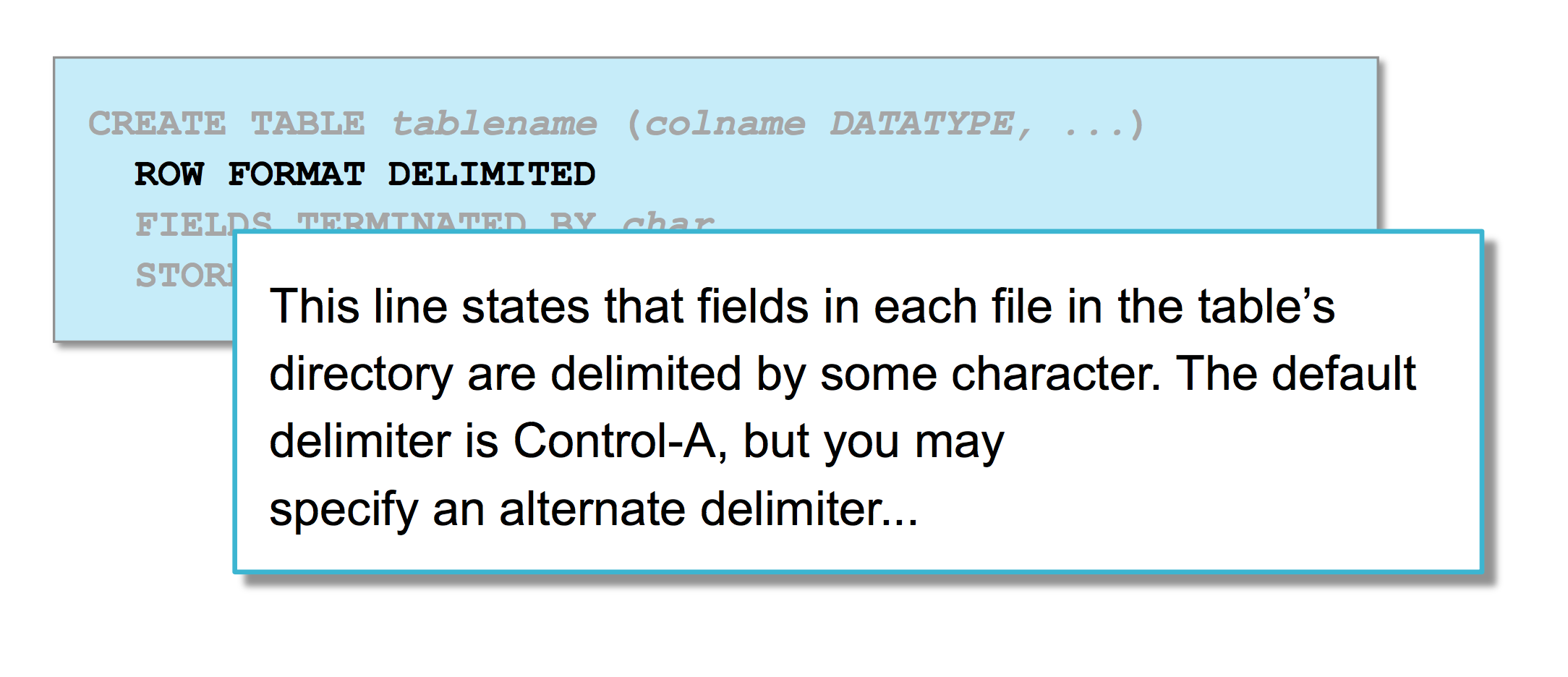
* Tables in Hive/Impala are simply one or more files in HDFS
* Default path of files in hive /user/hive/warehouse/<table\_name>
* Hive and Impala work with same data i.e. tables in HDFS and metadata in metastore

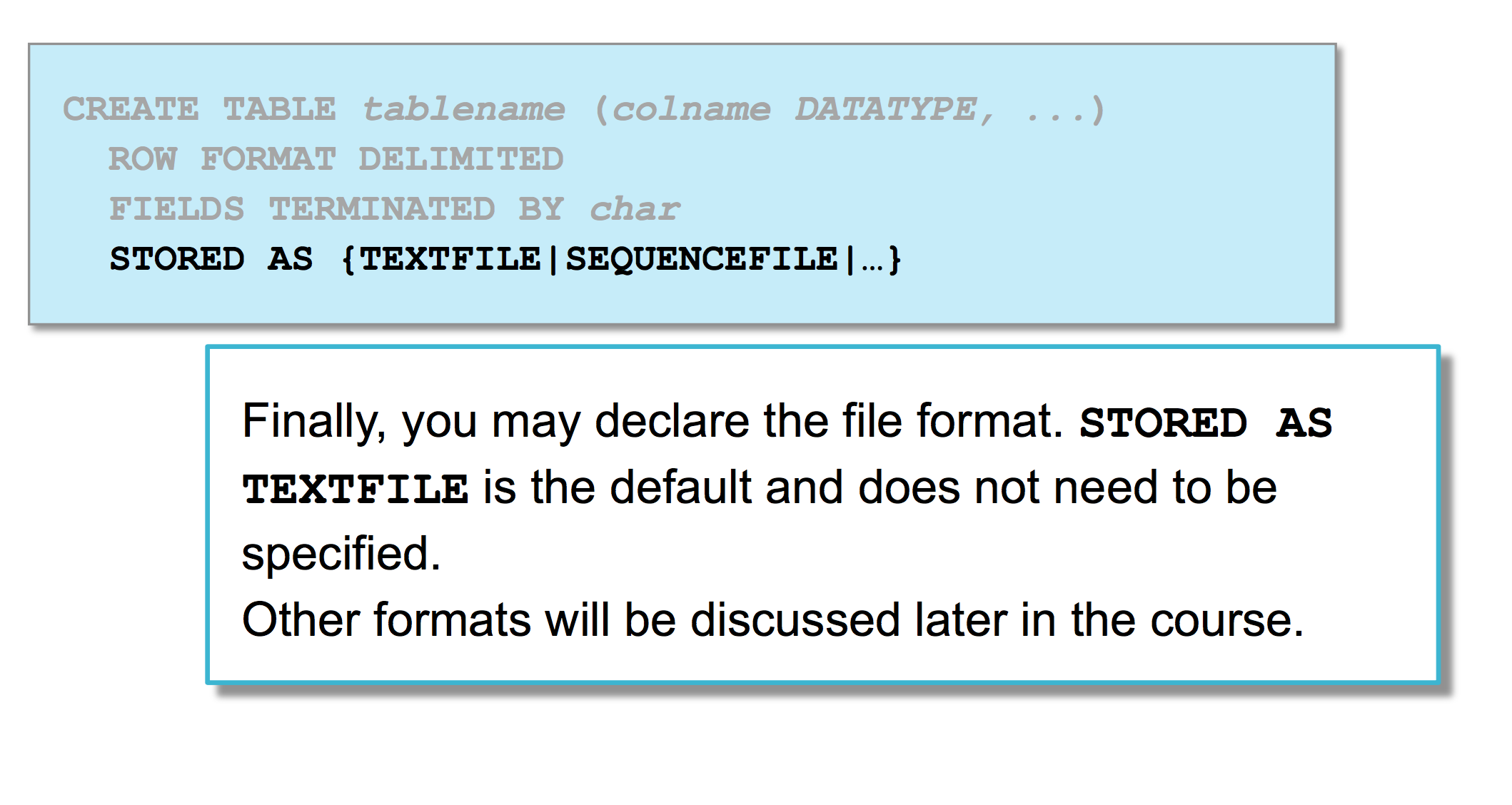


* Drop database command will fail if tables exists in database
* Add CASCADE keyword to forcefully remove the database if it contains tables









hive> create table jobs(id int, title string, salary int, posted timestamp) row format delimited fields terminated by ',' stored as textfile;

**hive> show create table jobs;**

OK

CREATE TABLE `jobs`(

`id` int,

`title` string,

`salary` int,

`posted` timestamp)

ROW FORMAT DELIMITED

FIELDS TERMINATED BY ','

STORED AS INPUTFORMAT

'org.apache.hadoop.mapred.TextInputFormat'

OUTPUTFORMAT

'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'

LOCATION

'hdfs://quickstart.cloudera:8020/user/hive/warehouse/jobs'

TBLPROPERTIES (

'transient\_lastDdlTime'='1470521762')

Time taken: 0.31 seconds, Fetched: 15 row(s)

hive> insert into jobs values(12,'Data Engineer', 130000, '2016-11-20 18:00:00');



**hive> create table jobs\_copy like jobs;**

create only table not data

**SPARK**

scala> val mydata = sc.textFile("file:/home/cloudera/emp.csv")

scala> mydata.count()

scala> for (line <- mydata.take(2)) println(line)

scala> for (line <- mydata.take(2)) println(line.toUpperCase)

**Applying transformation on RDD**

scala> val mydata\_ch = mydata.map(line => line.toUpperCase())

scala> for (line <- mydata\_ch) println(line)

**If a line contains a word**

scala> val mydata\_filt = mydata.map(line => line.contains("Anand"))

scala> for (line <- mydata\_filt) println(line)

false

true

false

false

true

false

false

true

false

false

true

false

false

true

false

…

**Applying a filter on RDD**

scala> val mydata\_t = mydata.map(line => line.toUpperCase()).filter(line => line.contains("ANAND"))

scala> for (line <- mydata\_t) println(line)

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

12,ANAND,130000,90

**RDD Lineage**

scala> mydata\_t.toDebugString

res14: String =

(1) MapPartitionsRDD[8] at filter at <console>:29 []

| MapPartitionsRDD[7] at map at <console>:29 []

| file:/home/cloudera/emp.csv MapPartitionsRDD[3] at textFile at <console>:27 []

| file:/home/cloudera/emp.csv HadoopRDD[2] at textFile at <console>:27 []

**Define in-line function**

scala> def toUpper(s: String) : String = {s.toUpperCase}

toUpper: (s: String)String

scala> mydata.map(toUpper).take(2)

res15: Array[String] = Array(11,SHIVA,120000,90, 12,ANAND,130000,90)

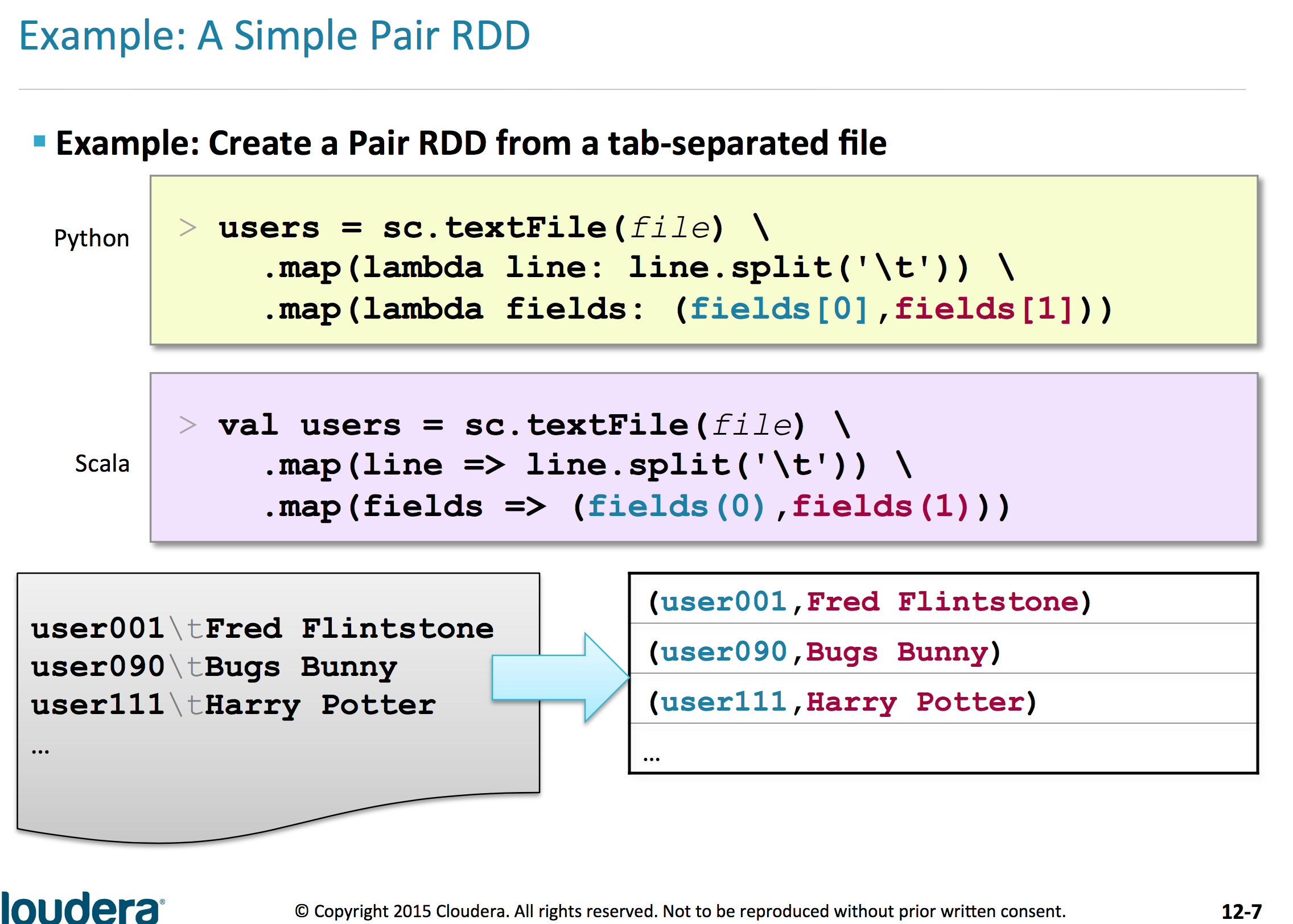
Here toUpper is an inline function

**Command to check how many cores we are running**

scala> sc.master

res17: String = local[\*]

Pair RDDs



[cloudera@quickstart ~]$ cat pair\_rdd

12,Ankit

14,Anand

15,Shiva

16,Nitin

18,Ankit

19,Anil

20,Anand

We will create a pair RDD out of file pair\_rdd shown above

scala> val pair\_rdd = sc.textFile("file:/home/cloudera/pair\_rdd").map(line => line.split(',')).map(fields => (fields(0),fields(1)))

pair\_rdd: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[3] at map at <console>:27

scala> pair\_rdd.collect.foreach(println)

(12,Ankit)

(14,Anand)

(15,Shiva)

(16,Nitin)

(18,Ankit)

(19,Anil)

(20,Anand)

keyBy

We are telling Spark to make the value key-value pair by selecting a key in the string. And we pass that value to keyBy function.

[cloudera@quickstart ~]$ cat keyby\_rdd

Ankit,Data Engineer,18,75 Liberty Ave

Anand,Data Modeler,20,125 S West St

Shiva,Project Manager,21,120 S West St

We will make key-value pairs of above file using user id which is 3rd attribute of the string

scala> val keybyRDD = sc.textFile("file:/home/cloudera/keyby\_rdd").keyBy(line => line.split(',')(2))

keybyRDD: org.apache.spark.rdd.RDD[(String, String)] = MapPartitionsRDD[9] at keyBy at <console>:27

scala> keybyRDD.collect.foreach(println)

(18,Ankit,Data Engineer,18,75 Liberty Ave)

(20,Anand,Data Modeler,20,125 S West St )

(21,Shiva,Project Manager,21,120 S West St)

Another example

[cloudera@quickstart ~]$ cat zipcode\_example

00210 43.005895 -71.013202

01014 42.170731 -72.604842

01062 42.324232 -72.67915

01263 42.392933 -73.228483

scala> val zipRDD = sc.textFile("file:/home/cloudera/zipcode\_example").map(line => line.split('\t')).map(fields => (fields(0),(fields(1),fields(2))))

zipRDD: org.apache.spark.rdd.RDD[(String, (String, String))] = MapPartitionsRDD[17] at map at <console>:27

scala> zipRDD.collect.foreach(println)

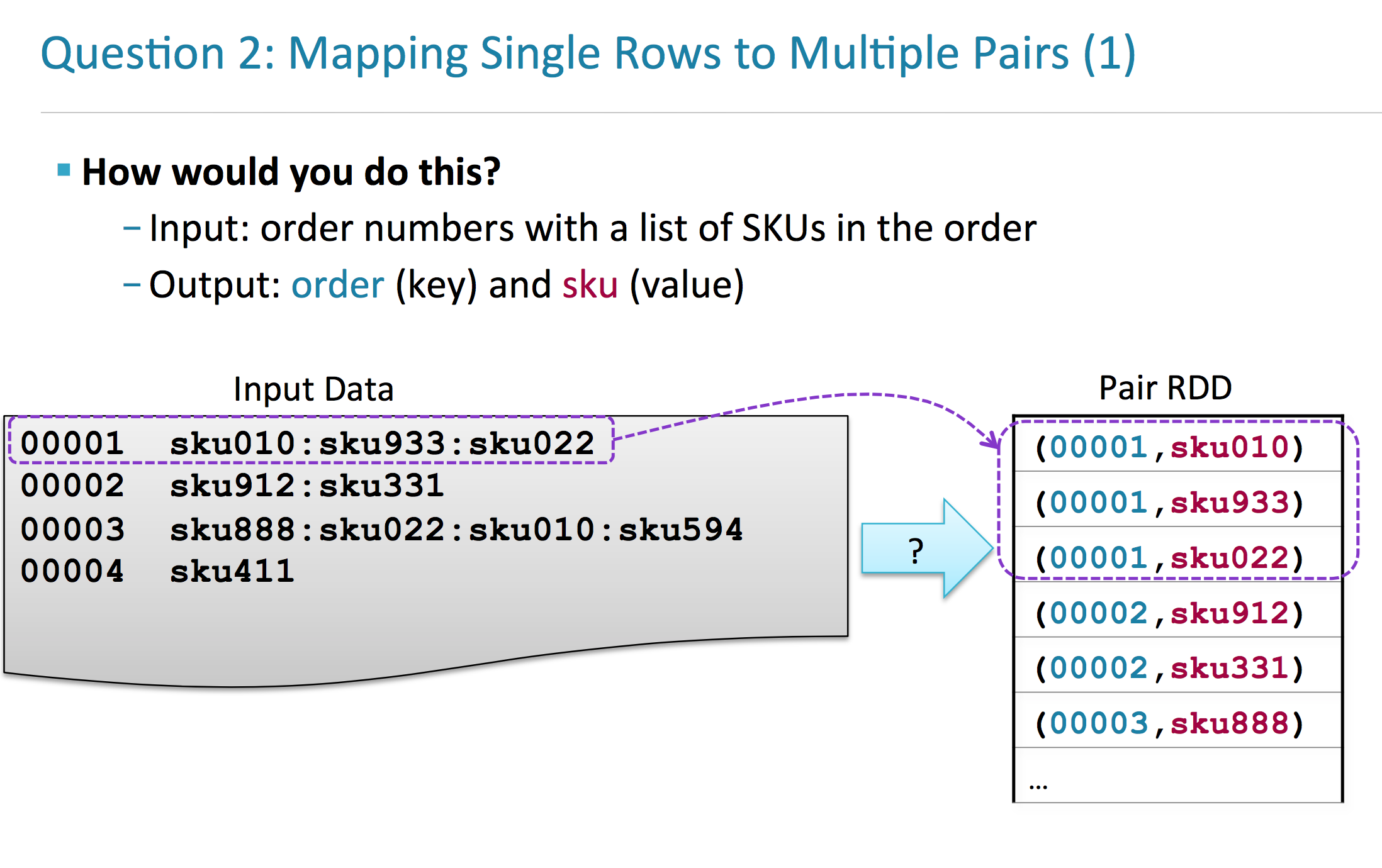
(00210,(43.005895,-71.013202 ))

(01014,(42.170731,-72.604842 ))

(01062,(42.324232,-72.67915 ))

(01263,(42.392933,-73.228483))

**Mapping a single row to multiple pairs (using flatMapValues)**



[cloudera@quickstart ~]$ cat flatmap\_example

00001 sku010:sku933:sku022

00002 sku912:sku331

00003 sku888:sku022:sku010:sku594

00004 sku411

scala> val flatmapRDD = sc.textFile("file:/home/cloudera/flatmap\_example").map(line => line.split('\t')).map(fields => (fields(0),fields(1))).flatMapValues(line => line.split(':'))

scala> flatmapRDD.collect.foreach(println)

(00001,sku010)

(00001,sku933)

(00001,sku022)

(00002,sku912)

(00002,sku331)

(00003,sku888)

(00003,sku022)

(00003,sku010)

(00003,sku594)

(00004,sku411)



**AVRO format**

Step 1 - Sqoop a table from mysql in avro format

[cloudera@quickstart avro]$ sqoop import --connect jdbc:mysql://localhost/training --username root --password cloudera --table emp --as-avrodatafile --num-mappers 1 --target-dir /user/cloudera/emp

.avsc file i.e. schema file will automatically generate in the folder where you are running scoop command

Step 2 -

create a hive table in avro format using schema file and location of the scooped data

hive> create table emp\_avro3 stored as avro location '/user/cloudera/emp' tblproperties('avro.schema.url'='file:///home/cloudera/emp.avsc');

**AVRO tool (to convert json to avro and vice-versa)**

Create a sample json file

[cloudera@quickstart ~]$ cat emp.json

{ "eno": {"int": 999},

"ename":{"string":"test\_99"},

"deptno": {"int": 99},

"insrt\_ts":{"long":1468965888000}

}

we will convert this json data file to avro format using avro tools jar

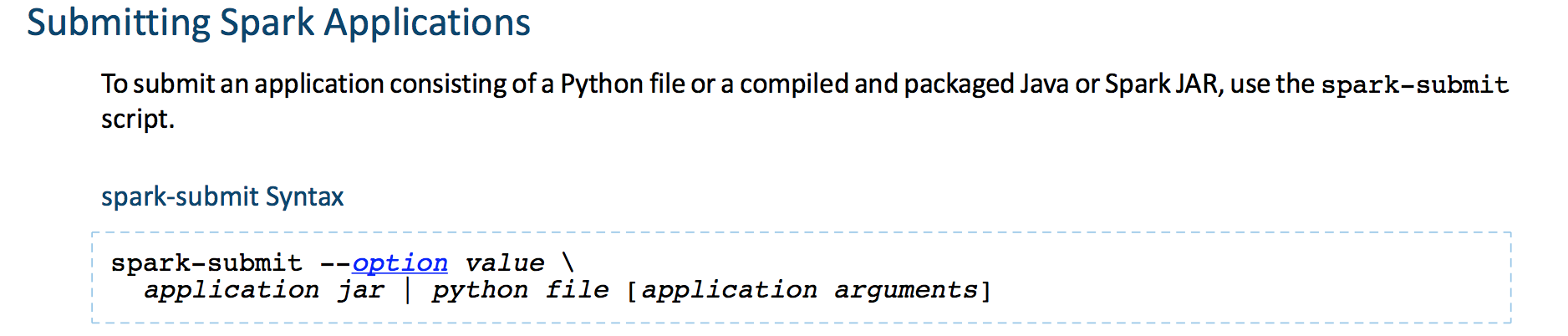
avtro tools jar is present it /usr/lib/avro folder

[cloudera@quickstart ~]$ java -jar /usr/lib/avro/avro-tools.jar fromjson --schema-file /home/cloudera/emp.avsc emp.json > emp99.avro

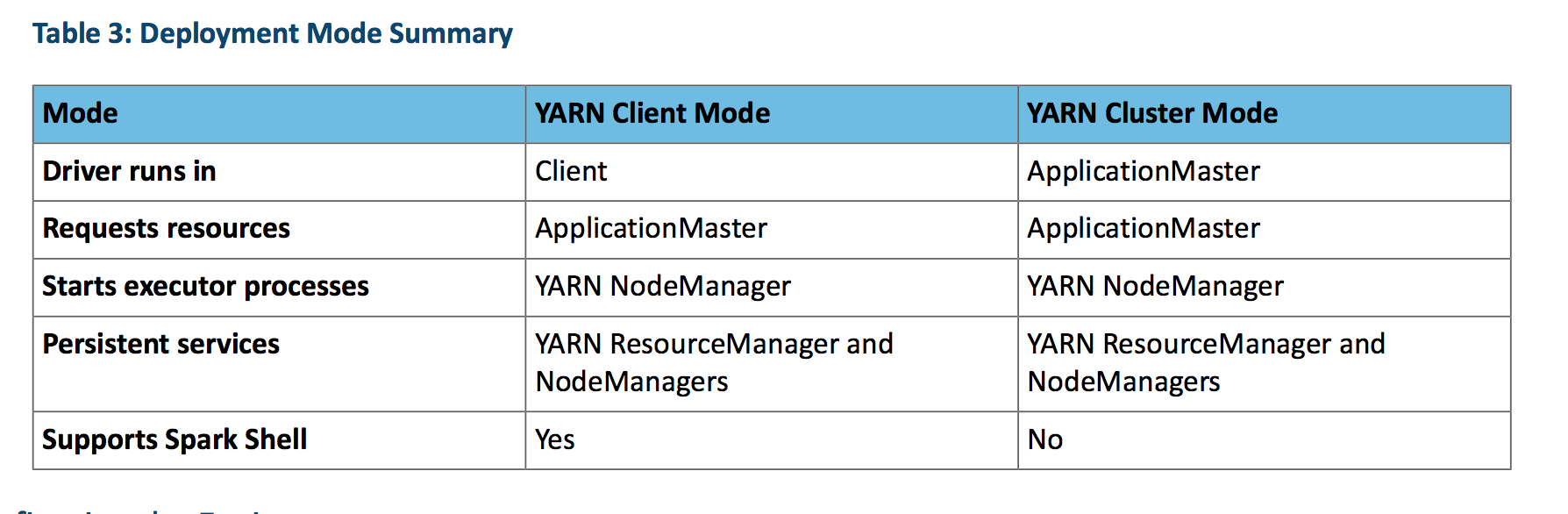
Configure Spark Applications

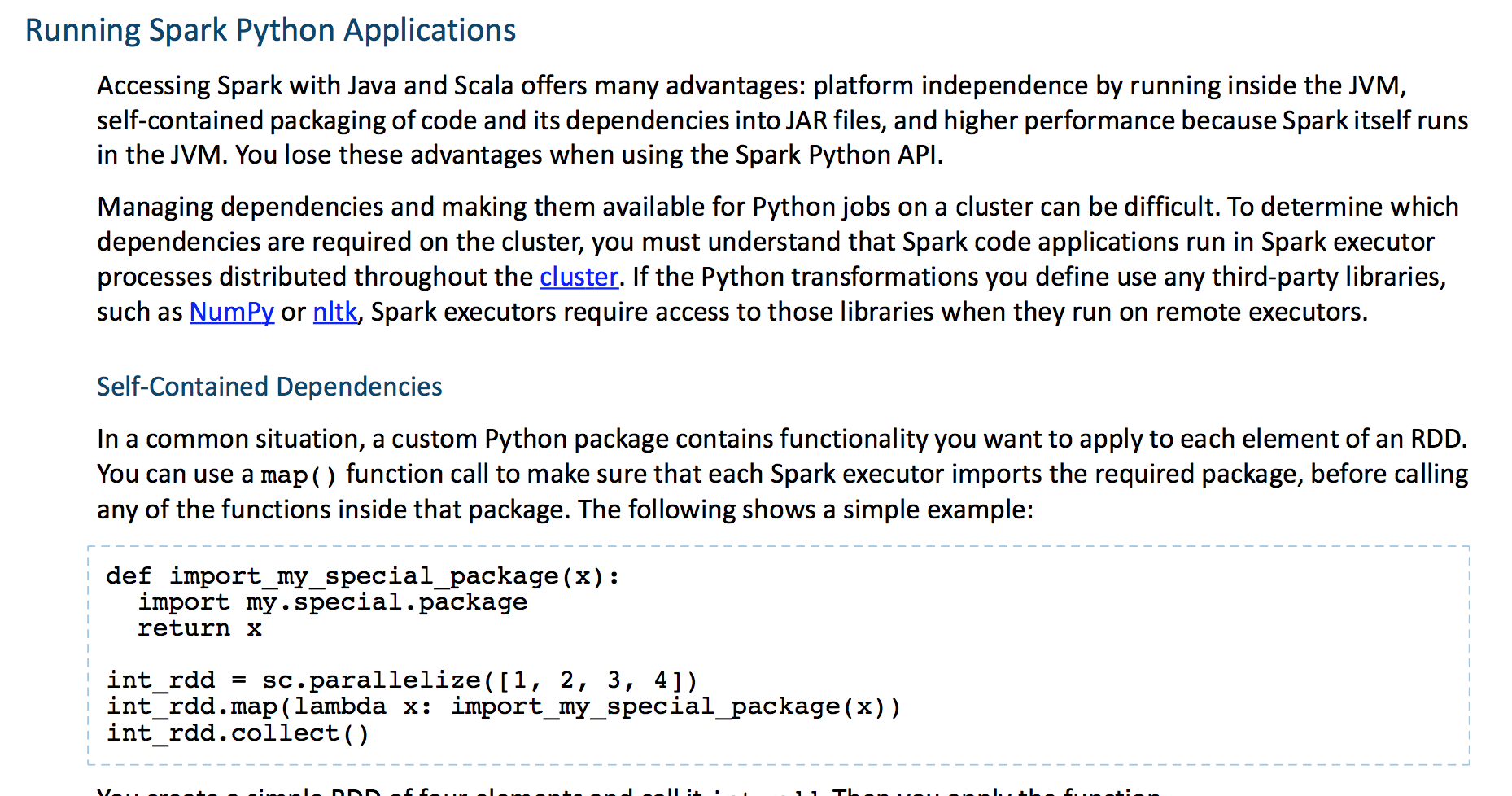
spark-submit \ --class com.cloudera.example.YarnExample \ --master yarn \ --deploy-mode cluster \ **--conf "spark.eventLog.dir=hdfs:///user/spark/eventlog"** \ lib/yarn-example.jar \

* The order of precedence in configuration properties is:
* **1.** Properties passed to SparkConf.
* **2.** Argumentspassedtospark-submit,spark-shell,orpyspark.
* **3.** Properties set in spark-defaults.conf.

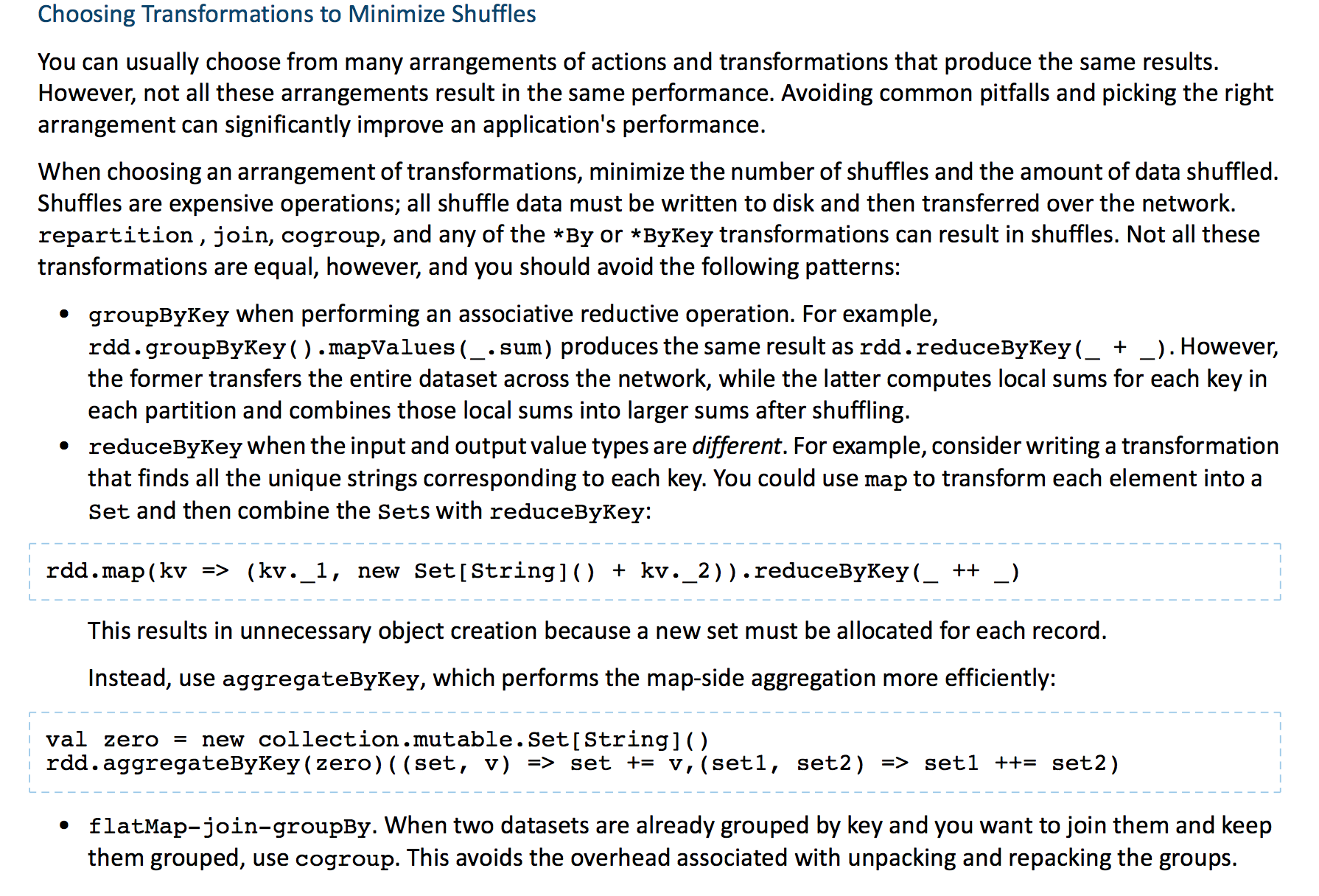


YARN Deployment Nodes





Tuning Spark Applications



**Configuration files**

In Hadoop 2.7.3, the configuration files are located under **$HADOOP\_HOME/etc/Hadoop/**   
  
These files are used to configure a hadoop cluster.  
  
**core-site.xml:**  
  
All Hadoop services and clients use this file to **locate the NameNode**, so this file must be copied to each node that is either running a Hadoop service or is a client node. The Secondary NameNode uses this file to determine the location for storing fsimage and edits log namefs.checkpoint.dir/name locally, and the location of the NameNode namefs.namedefault.name/name. Use the core-site.xml file to isolate communication issues with the NameNode host machine  
  
**hdfs-site.xml:**  
  
HDFS services use this file, which contains a number of important properties. These include:

HTTP addresses for the two services  
Replication for DataNodes namedfs.replication/name>  
DataNode block storage location namedfs.data.dir/name  
NameNode metadata storage namedfs.name.dir/name

Use the hdfs-site.xml file to isolate NameNode start-up issues. Typically, NameNode start-up issues are caused when NameNode fails to load the fsimage and edits log to merge. Ensure that the values for the location properties in hdfs-site.xml are valid locations.

**log4j.properties:**  
Use the log4j.properties file to modify the log purging intervals of the HDFS logs. This file defines logging for all the Hadoop services. It includes, information related to appenders used for logging and layout. For more details, see the[log4j documentation](http://logging.apache.org/log4j/2.x/).

**Hadoop-policy.xml**

hadoop-policy.xml configuration file is used to define the access control lists for various Hadoop services. Service Level Authorization is performed much before to other access control checks such as file-permission checks, access control on job queues etc

**yarn-site.xml**

The fundamental idea of YARN is to split up the functionalities of resource management and job scheduling/monitoring into separate daemons. The idea is to have a global ResourceManager (*RM*) and per-application ApplicationMaster (*AM*). An application is either a single job or a DAG of jobs.

**mapred-site.xml**

A MapReduce *job* usually splits the input data-set into independent chunks which are processed by the *map tasks* in a completely parallel manner. The framework sorts the outputs of the maps, which are then input to the *reduce tasks*. Typically both the input and the output of the job are stored in a file-system. The framework takes care of scheduling tasks, monitoring them and re-executes the failed tasks.

**Performance Tuning Tips**

* **repartition** does full shuffle of data. If you want to reduce partitions use **coalesce** instead which avoid full shuffle.
* **Shuffles are expensive** operations and should be avoided.
* Avoid groupByKey() when performing ***associative reductive operations***, use reduceByKey() instead.

e.g. rdd.groupByKey().mapValues(\_.sum)

* **groupByKey transfers entire dataset** across the network.
* **reduceByKey comutes local sums** in each partition and then combines them.
* **Avoid reduceByKey() when input and output values are different**, instead use aggregateByKey() which performs map-side aggregations more efficiently.
* Avoid **flatMap-join-groupBy** pattern when 2 datasets are already grouped by key and you want to join them, **use cogroup instead**. It avoids overhead of unpacking and repacking of groups.
* To avoid shuffles when joining two datasets, you **can use broadcast variables** (smaller one), broadcast variable will put it in hash table in memory and broadcast to every executor where it can be used.
* Sometimes shuffles are beneficial.

When files are not split efficiently; this reduces parallelism. Use **repartition** here to increase the number of partitions.

* If there are too many partitions, aggregate or reduce operations can be expensive. So use reduceByKey or aggregateByKey first to reduce the number of partitions (it happens in parallel).

**Specify**

***==executor-cores***

***==executor-memory***

***==num-executors***

* Number of tasks per stage is important in determining performance. Main goal is to have enough number of tasks that the data destined for each task fits in memory.
* **Reduce the size of the data structures**

To avoid **spill on disk**

Large objects have greater disk space and network I/O

* **Avoid** storing lots of **JSON** files, instead **use Avro, Parquet and Thrift** etc which are stored in a sequence file.

**DataFrames offer performance because**

* **Custom Memory Management** (Project Tungsten)
* Data is stored in **off-heap memory in binary form**. Serialization is avoided.
* Optimized Execution Plans (**Catalyst Optimizer**: predicate pushdown, JVM bytecode for physical plans intelligently between broadcast joins and shuffle joins)

**Datasets = RDD + DataFrame fatures + Encoders** (helps to interact with off-heap data and provide access to individual attributes, used to serialize and deserialize the records of dataset by knowing the schema)

***Scala WordCount***

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext.\_

import org.apache.spark.SparkConf

object SparkWordCount {

def main(args: Array[String]) {

// create Spark context with Spark configuration

val sc = new SparkContext(new SparkConf().setAppName("Spark Count"))

// get threshold

val threshold = args(1).toInt

// read in text file and split each document into words

val tokenized = sc.textFile(args(0)).flatMap(\_.split(" "))

// count the occurrence of each word

val wordCounts = tokenized.map((\_, 1)).reduceByKey(\_ + \_)

// filter out words with fewer than threshold occurrences

val filtered = wordCounts.filter(\_.\_2 >= threshold)

// count characters

val charCounts = filtered.flatMap(\_.\_1.toCharArray).map((\_, 1)).reduceByKey(\_ + \_)

System.out.println(charCounts.collect().mkString(", "))

}

}

***Python WordCount***

import sys

from pyspark import SparkContext, SparkConf

if \_\_name\_\_ == "\_\_main\_\_":

# create Spark context with Spark configuration

conf = SparkConf().setAppName("Spark Count")

sc = SparkContext(conf=conf)

# get threshold

threshold = int(sys.argv[2])

# read in text file and split each document into words

tokenized = sc.textFile(sys.argv[1]).flatMap(lambda line: line.split(" "))

# count the occurrence of each word

wordCounts = tokenized.map(lambda word: (word, 1)).reduceByKey(lambda v1,v2:v1 +v2)

# filter out words with fewer than threshold occurrences

filtered = wordCounts.filter(lambda pair:pair[1] >= threshold)

# count characters

charCounts = filtered.flatMap(lambda pair:pair[0]).map(lambda c: c).map(lambda c: (c, 1)).reduceByKey(lambda v1,v2:v1 +v2)

list = charCounts.collect()

print repr(list)[1:-1]



**Creating DataFrames (abstraction which gives schema view of data, column names/types)**

*# spark is an existing SparkSession*

df = spark.read.json("examples/src/main/resources/people.json")

df.show()

*# Print the schema in a tree format*

df.**printSchema**()

*# Select only the "name" column*

df.**select**(df.name).show()

*# Select everybody, but increment the age by 1*

df.select(df.name, df.age + 1).show()

*# Select people older than 21*

df.**filter**(df.age > 21).show()

df.**where**($”sal” > 140000).show()

*# Count people by age*

df.**groupBy**(df.age).count().show()

df.**join**(logs, logs.userId == df.userId, “left\_outer”)

*# Register the DataFrame as a SQL temporary view*

df.**createOrReplaceTempView**("people")

sqlDF = spark.**sql**("SELECT \* FROM people")

sqlDF.show()

df.**unionAll**(otherDF)

df.**intersect**(otherDF)

df.**except**(otherDF)

df.**dropDuplicates**()

## Creating Datasets

**case class Person(name: String, age: Long)**

// Encoders are created for case classes

**val** myDS **=** **Seq**(**Person**("Andy", **32**)).**toDS()**

myDS.show()

// Encoders for most common types are automatically provided by importing spark.implicits.\_

**val** primitiveDS **=** **Seq**(**1**, **2**, **3**).toDS()

primitiveDS.map(**\_** + **1**).collect() // Returns: Array(2, 3, 4)

// DataFrames can be converted to a Dataset by providing a class. Mapping will be done by name

**val** path **=** "examples/src/main/resources/people.json"

**val** peopleDS **=** spark.read.json(path).as[**Person**]

peopleDS.show()

Inferring the Schema Using Reflection



**Accessing Hive from Spark**

**from** **os.path** **import** expanduser, join, abspath

**from** **pyspark.sql** **import** SparkSession

**from** **pyspark.sql** **import** Row

# warehouse\_location points to the default location for managed databases and tables

warehouse\_location = abspath('spark-warehouse')

spark = SparkSession \

.builder \

.appName("Python Spark SQL Hive integration example") \

**.config("spark.sql.warehouse.dir", warehouse\_location)** \

**.enableHiveSupport()** \

.getOrCreate()

# spark is an existing SparkSession

spark.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING) USING hive")

spark.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")

# Queries are expressed in HiveQL

spark.sql("SELECT \* FROM src").show()

# Aggregation queries are also supported.

spark.sql("SELECT COUNT(\*) FROM src").show()

# The results of SQL queries are themselves DataFrames and support all normal functions.

sqlDF = spark.sql("SELECT key, value FROM src WHERE key < 10 ORDER BY key")

# The items in DataFrames are of type Row, which allows you to access each column by ordinal.

stringsDS = sqlDF.rdd.map(**lambda** row: "Key: %d, Value: %s" % (row.key, row.value))

**for** record **in** stringsDS.collect():

**print**(record)

# You can also use DataFrames to create temporary views within a SparkSession.

Record = Row("key", "value")

recordsDF = spark.createDataFrame([Record(i, "val\_" + str(i)) **for** i **in** range(**1**, **101**)])

recordsDF.createOrReplaceTempView("records")

# Queries can then join DataFrame data with data stored in Hive.

spark.sql("SELECT \* FROM records r JOIN src s ON r.key = s.key").show()

### 2.1. Create a new RDD

**a)** **Read File from local filesystem and create an RDD**.

|  |  |
| --- | --- |
| 1 | scala> val data = sc.textFile("data.txt") |



**spark-submit**

*# Run on a YARN cluster*

export HADOOP\_CONF\_DIR=XXX

**./bin/spark-submit** **\**

**--class** org.apache.spark.examples.SparkPi **\**

**--master** yarn **\**

**--deploy-mode** cluster **\**  *# can be client for client mode*

**--executor-memory** 20G **\**

**--num-executors** 50 **\**

/path/to/examples.jar **\**

1000

**Simple CSV extract program using DataBricks**

**%python**

**from pyspark import SQLContext**

**from pyspark.sql import Row**

**csv\_file = "/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv"**

**myRDD = sc.textFile(csv\_file)**

**myRDD.take(5)**

**myRDD2 = myRDD.map(lambda line: line.upper()).map(lambda line: line.split(",")).map(lambda line: Row(col1=line[0],col2=line[1],col3=line[2])).toDF()**

**myRDD3 = myRDD2.filter(myRDD2.col2 > 0.3).show()**

**myRDD2.registerTempTable("csv\_data\_table")**

**myRDD4 = spark.sql("select \* from csv\_data\_table where col2 > 0.3 ")**

**myRDD4.show()**

**Simple JSON extract program using Jupyter Notebooks**

import findspark

findspark.init()

import pyspark

from pyspark import SQLContext

from pyspark.sql import Row

import json

sc = pyspark.SparkContext(appName="jsonDF")

sqlContext = SQLContext(sc)

json\_text = """ {"id": 1,"name": "Mummy","age": 62,"city": "Bhopal"}

{"id": 2,"name": "Charu","age": 23,"city": "Bhopal"}

{"id": 3,"name": "Ashutosh","age": 39,"city": "Pune"}

{"id": 1,"name": "Ankit","age": 34,"city": "San Francisco"}"""

jsonDF = sqlContext.read.json("sample\_json.json")

jsonDF.printSchema()

jsonDF.createOrReplaceTempView("family")

sqlContext.sql("select name,age,case when city = 'Bhopal' then 'My City' else city end from family").show()

# to write in Hive table

jsonDF.write("append").partitionBy("city").insertInto("family\_table")

**Simple CSV extract program using Jupyter Notebooks**

import findspark

findspark.init()

import pyspark

import random

from pyspark.sql import Row

sc = pyspark.SparkContext(appName="Pi")

from pyspark import SQLContext

sqlContext = SQLContext(sc)

myRDD = sc.textFile("/Users/ankitupadhyay/Downloads/bikeshare.csv")

myRDD.take(5)

myRDD2 = myRDD.map(lambda line: line.split(",")).map(lambda line: Row(date\_col=line[0],season\_col=line[1],holiday\_col=line[2])).toDF()

#filter the dataframe

myRDD3 = myRDD2.filter(myRDD2.season\_col > 0.3).show()

#select only few columns

myRDD4.select('col2','col3').show(5)

myDF = myRDD2.createOrReplaceTempView("bike")

sqlContext.sql("select \* from bike").show()

sc.stop()