Apache Spark training

Table of Contents

[Installation 2](#_Toc56695255)

[Pre-Req: 2](#_Toc56695256)

[Release Notes: 3](#_Toc56695257)

[Guide: 3](#_Toc56695258)

[Steps: 3](#_Toc56695259)

[Start and Stop 5](#_Toc56695260)

[Components: 6](#_Toc56695261)

[List of All Components: 6](#_Toc56695262)

[Status of All Components: 6](#_Toc56695263)

[VM Access 7](#_Toc56695264)

[Using Browser: 7](#_Toc56695265)

[Using Terminal: 7](#_Toc56695266)

[Sandbox 7](#_Toc56695267)

[Components: 8](#_Toc56695268)

[HDFS: 8](#_Toc56695269)

[Overview: 8](#_Toc56695270)

[Check 8](#_Toc56695271)

[Hive: 8](#_Toc56695272)

[Overview: 8](#_Toc56695273)

[Details: 9](#_Toc56695274)

[Commands: 9](#_Toc56695275)

[Config Properties 9](#_Toc56695276)

[Kafka 9](#_Toc56695277)

[Overview: 9](#_Toc56695278)

[Topic: 9](#_Toc56695279)

[Partition: 9](#_Toc56695280)

[Retention Period: 9](#_Toc56695281)

[Topic and Partition: 10](#_Toc56695282)

[Offset Positions Per Consumer: 10](#_Toc56695283)

[Kafka Cluster: Multi-Node Kafka Servers. 11](#_Toc56695284)

[Kafka Services in HDP: 11](#_Toc56695285)

[Check: 12](#_Toc56695286)

[Create Kafka Topic: 12](#_Toc56695287)

[Produce Data into Topic: 13](#_Toc56695288)

[Consume Data from Topic: 13](#_Toc56695289)

[Topic Offsets: 13](#_Toc56695290)

[Consume Group Offsets: 14](#_Toc56695291)

[Spark 15](#_Toc56695292)

[Overview: 15](#_Toc56695293)

[Stack: 15](#_Toc56695294)

[Input: 16](#_Toc56695295)

[spark.sql.Row object 16](#_Toc56695296)

[Structured Streaming 17](#_Toc56695297)

[Concepts: 19](#_Toc56695298)

[Cluster Mode: 21](#_Toc56695299)

[YARN: 21](#_Toc56695300)

[Driver: 22](#_Toc56695301)

[Executors: 22](#_Toc56695302)

[Sources: 22](#_Toc56695303)

[Sink: 22](#_Toc56695304)

[Streaming: 22](#_Toc56695305)

[Spark + Kafka 24](#_Toc56695306)

[Kerberos: 26](#_Toc56695307)

[Broadcast: 27](#_Toc56695308)

[HBase: 27](#_Toc56695309)

[Features: 27](#_Toc56695310)

[Hive: 28](#_Toc56695311)

[Data Formats: 29](#_Toc56695312)

[AVRO 29](#_Toc56695313)

[ORC: 29](#_Toc56695314)

[PARQUET: 29](#_Toc56695315)

[Scala Program: 29](#_Toc56695316)

[Build and Deploy: 29](#_Toc56695317)

[Project: 29](#_Toc56695318)

[IDE: 29](#_Toc56695319)

[Project: 29](#_Toc56695320)

[Build: 31](#_Toc56695321)

[Compile & Package: 31](#_Toc56695322)

[Payload: 31](#_Toc56695323)

[Upload: 31](#_Toc56695324)

[Run Program: 32](#_Toc56695325)

[Sandbox: 32](#_Toc56695326)

[GCP: 32](#_Toc56695327)

[Applications 32](#_Toc56695328)

[Produce Data to Kafka 32](#_Toc56695329)

[Consume Data from Kafka 32](#_Toc56695330)

[Kafka Streams Application 33](#_Toc56695331)

[Spark Application: 33](#_Toc56695332)

[Read from Kafka: 33](#_Toc56695333)

[Apply Transformation 33](#_Toc56695334)

[Write to HDFS 33](#_Toc56695335)

[Write to HIVE 33](#_Toc56695336)

[Write to HBase 33](#_Toc56695337)

[Troubleshoot: 33](#_Toc56695338)

[Interview Preparation: 34](#_Toc56695339)

# Installation

## Pre-Req:

Make sure your computer is having minimum 16GB RAM and Quad Core CPU. HDP platform would need 10+GB to start the sandbox.

## Release Notes:

<https://www.cloudera.com/tutorials/hortonworks-sandbox-guide/1.html>

## Guide:

https://www.cloudera.com/tutorials/getting-started-with-hdp-sandbox.html

## Steps:

* Download Sandbox:
  + https://www.cloudera.com/downloads/hortonworks-sandbox.html
* Download Virtual Box:
  + <https://www.virtualbox.org/wiki/Downloads>
* Install Virtual Box
* Start Virtual Box
* Virtual Box: Import OVA File download for Sandbox
* Virtual Box: Settings -> Network -> Bridge

A screenshot of a social media post

Description automatically generated

* Virtual Box: Start Server (will take some time, only first time)

A screenshot of a social media post

Description automatically generated

* Verify:

A screenshot of a cell phone

Description automatically generated

## Start and Stop

* Start your virtual box
* Start the Imported Sandbox VM
* Wait for it to start the VM and print the welcome page

A screenshot of a cell phone

Description automatically generated

* Take a note of the IP address.
* Open Browser and access the web page with given IP and Port.
  + Example: <http://192.168.5.137:1080/>
* Access Quick Links
  + <http://192.168.5.137:1080/splash2.html>
* Access Ambari
  + <http://192.168.5.137:8080/#/login>, User and password is given in the same page.
  + username & password: raj\_ops
* Check to make sure all the required components are installed, started and healthy.

# Components:

## List of All Components:

<https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.0.1/release-notes/content/comp_versions.html>

## Status of All Components:

<http://192.168.5.137:8080/#/main/hosts/sandbox-hdp.hortonworks.com/summary>

Important components that we need for this training:

* HDFS
* YARN
* MapReduce2
* Tez
* Hive
* Hbase (may need more resources like RAM and CPU)
* Zookeeper
* Kafka
* Spark2

# VM Access

In case you want to access terminal and use command line tools.

username & password: **root / hadoop**

Note: system will ask you to change password on first login.

### Using Browser:

<http://192.168.5.137:4200/>

A screenshot of a cell phone

Description automatically generated

## Using Terminal:

### Sandbox

>ssh [root@192.168.5.137](mailto:root@192.168.5.137) (IP address refer to virtual box->popup screen)

>>ssh [root@172.18.0.2](mailto:root@172.18.0.2)

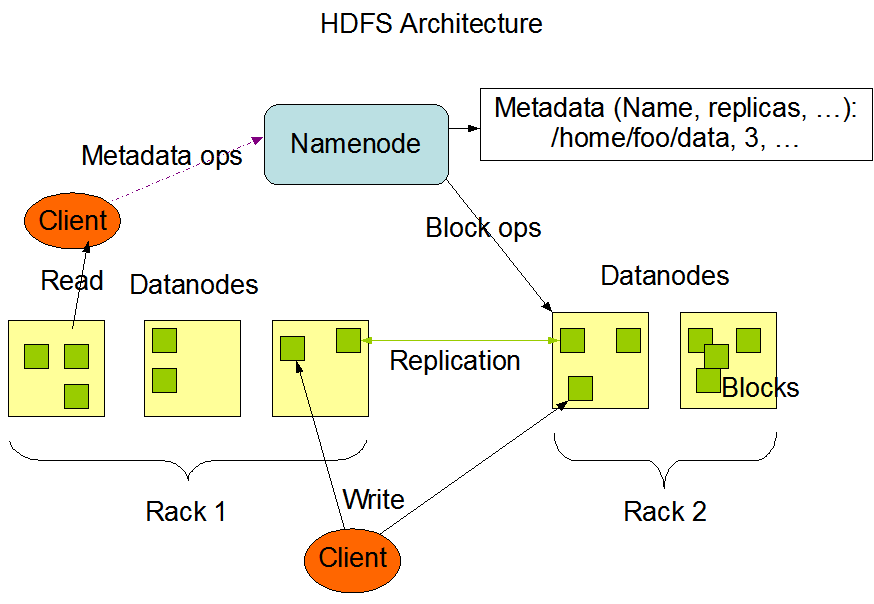
# Components:

## HDFS:

### Overview:

Hadoop Distributed File System (HDFS), a distributed storage used by Hadoop applications.

A HDFS cluster primarily consists of a NameNode that manages the file system metadata and DataNodes that store the actual data



### Check

Make sure HDFS is running and healthy from Ambari.

[root@sandbox-hdp ~]# hadoop fs -ls /

## Hive:

### Overview:

A data warehouse like system with reading, writing, and managing large datasets residing in distributed storage using SQL.

### Details:

<https://cwiki.apache.org/confluence/display/Hive/Home>

### Commands:

>hive

>> show databases;

>> use foodmart;

>> show tables;

>> desc days;

>> show create table days;

### Config Properties

<http://192.168.5.137:8080/#/main/services/HIVE/configs>

## Kafka

### Overview:

A streaming platform has three key capabilities:

* Publish and subscribe to streams of records, like message queues.
* Store streams of records in a fault-tolerant durable way.
* Process streams of records as they occur

Kafka stores streams of records in categories called topics. Each record consists of a key, a value, and a timestamp.

### Topic:

A topic is a category or feed name to which records are published. Topics in Kafka are always multi-subscriber; that is, a topic can have zero, one, or many consumers that subscribe to the data written to it.

### Partition:

Each partition is an ordered, immutable sequence of records that is continually appended. Each record in the partitions are each assigned a sequential id number called the offset that uniquely identifies each record within the partition.

### Retention Period:

If the retention policy is set to two days, then for the two days after a record is published, it is available for consumption, after which it will be discarded to free up space.

### Topic and Partition:

A screenshot of a cell phone

Description automatically generated

### Offset Positions Per Consumer:

A picture containing bird

Description automatically generated

### Kafka Cluster: Multi-Node Kafka Servers.

A picture containing drawing, table

Description automatically generated

A picture containing bird

Description automatically generated

### Kafka Services in HDP:

Wait for some time for all the services to start. Check the status here:

A screenshot of a social media post

Description automatically generated

Started Fine:

A screenshot of a cell phone

Description automatically generated

Ambari: <http://192.168.5.137:8080/#/main/dashboard/metrics>

### Check:

* Cluster is up
* Zookeeper is up
* Bootstrap server is up
* Kafka service is up

### Create Kafka Topic:

#### Sandbox:

> /usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 2 --topic tp\_in\_profileupdat

Es

#### GCP:

>sh /usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --create --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --replication-factor 1 --partitions 2 --topic tp\_in\_profileupdates

Created topic "tp\_in\_profileupdates".

>sh /usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --create --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --replication-factor 1 --partitions 2 --topic SourceTopic

Created topic "SourceTopic".

>sh /usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --create --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --replication-factor 1 --partitions 2 --topic SinkTopic

Created topic "SinkTopic".

Partitions: 2

Replication Factor: 1

#### List Topics:

Sandbox

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181

#### Describe Topic:

Sandbox

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic tp\_in\_profileupdates

GCP Cluster:

>/usr/hdp/3.1.4.0-315/kafka/bin/kafka-topics.sh --zookeeper instance-1.us-east1-b.c.wired-effort-272101.internal:2181 --describe --topic tp\_in\_profileupdates

Topic:tp\_in\_profileupdates PartitionCount:2 ReplicationFactor:1 Configs:

Topic: tp\_in\_profileupdates Partition: 0 Leader: 1001 Replicas: 1001 Isr: 1001

Topic: tp\_in\_profileupdates Partition: 1 Leader: 1001 Replicas: 1001 Isr: 1001

### Produce Data into Topic:

Sandbox:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-console-producer.sh --broker-list sandbox-hdp.hortonworks.com:6667 --topic tp\_in\_profileupdates

>{"user\_fname":"sunil", "user\_lname":"miriyala", "user\_id":"sunil.miriyala", "updated\_at":"2020-03-22 14:15:16.000", "modified\_fields":[{"addr

ess":"new value"}]}

>

GCP Cluster:

>cd /usr/hdp/3.1.4.0-315/kafka/bin/

>/usr/hdp/3.1.4.0-315/kafka/bin//kafka-console-producer.sh --broker-list localhost:6667 --topic tp\_in\_profileupdates

### Consume Data from Topic:

Using Bootstrap:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-console-consumer.sh --bootstrap-server sandbox-hdp.hortonworks.com:6667 --topic tp\_in\_profileupdates --group tp\_in\_userprof\_grp

{"user\_fname":"sunil", "user\_lname":"miriyala", "user\_id":"sunil.miriyala", "updated\_at":"2020-03-22 14:15:16.000", "modified\_fields":[{"addre

ss":"new value"}]}

### Topic Offsets:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic tp\_in\_profileupdates

Topic:tp\_in\_profileupdates PartitionCount:2 ReplicationFactor:1 Configs:

Topic: tp\_in\_profileupdates Partition: 0 Leader: 1001 Replicas: 1001 Isr: 1001

Topic: tp\_in\_profileupdates Partition: 1 Leader: 1001 Replicas: 1001 Isr: 1001

### Consume Group Offsets:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-consumer-groups.sh --bootstrap-server sandbox-hdp.hortonworks.com:6667 --describe --group tp\_in\_userprof\_grp

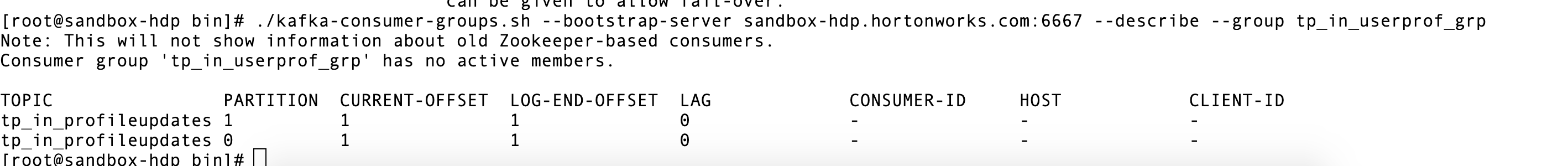
GCP Cluster:

>/usr/hdp/3.0.1.0-187/kafka/bin/kafka-consumer-groups.sh --bootstrap-server instance-1.us-east1-b.c.wired-effort-272101.internal:6667 --describe –group nptest1\_

TOPIC PARTITION CURRENT-OFFSET LOG-END-OFFSET LAG CONSUMER-ID HOST CLIENT-ID

tp\_in\_profileupdates 1 1 1 0 - - -

tp\_in\_profileupdates 0 1 1 0 - - -

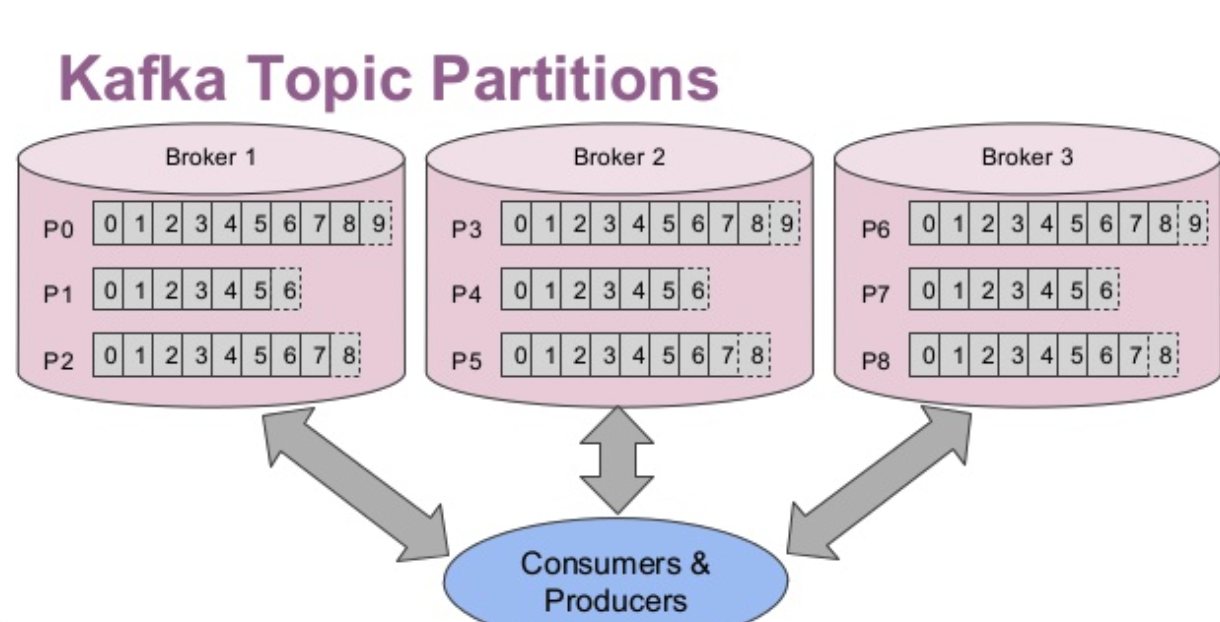


A screenshot of a cell phone

Description automatically generated

A close up of a device

Description automatically generated



## Spark

### Overview:

A high-performance large-scale data processing analytical engine that supports both batch and streaming data. It uses DAG schedulers, query optimizers and physical execution engines to achieve this performance.

It supports running on Hadoop, Kubernetes, Standalone and Cloud.

### Stack:

A close up of a sign

Description automatically generated

### Input:

spark.sql.Row object

Row[0]

Columns: item\_name, item\_desc, is\_active, created\_date

Row[1]

Columns: item\_name, item\_desc, is\_active, created\_date

#### Map Function:

Row\_In 🡪 Map Function 🡪 Row\_Out

FlatMap Function

Row\_In 🡪 Flat Map Function 🡪 [Row\_Out, …]

Row\_In (item\_name, item\_desc, is\_active, created\_date) 🡪 Flat Map Function 🡪 [item\_name, item\_desc, is\_active, created\_date]

This converted Single Row with 4 Columns as 4 Rows with Single Column. This is not correct.

Row\_In (item\_name, item\_desc, is\_active, created\_date) 🡪 Flat Map Function 🡪 [Row\_Out (Item1, MyItemDesc, Active/InActive, 2020-11-01), …]

Since we cannot create spark.sql.Row, we should create a POJO and set all required values in it and add to Iterator.

#### FlatMap Function

|  |  |
| --- | --- |
| **Input** | **Output** |
| org.apache.spark.sql.Row (Item1, MyDesc, 1, 2020-11-01T01:01:01 | com.css.java.bean.Items(Item1, MyDesc, Active, 2020-11-01)  com.css.java.bean.Items(Item1, MyDesc, Active, 2020-11-01)  com.css.java.bean.Items(Item1, MyDesc, Active, 2020-11-01)  … |

Map

|  |  |
| --- | --- |
| **Input** | **Output** |
| org.apache.spark.sql.Row (Item1, MyDesc, 1, 2020-11-01T01:01:01 | com.css.java.bean.Items(Item1, MyDesc, Active, 2020-11-01) |

### Structured Streaming

The key idea in Structured Streaming is to treat a live data stream as a table that is being continuously appended.

Note that Structured Streaming does not materialize the entire table. It reads the latest available data from the streaming data source, processes it incrementally to update the result, and then discards the source data. It only keeps around the minimal intermediate state data as required to update the result.

#### Windowing:

* window-based aggregations (e.g. number of events every minute) to be just a special type of grouping and aggregation on the event-time column.
* each time window is a group
* each row can belong to multiple windows/groups
* “watermarking allows the user to specify the threshold of late data, and allows the engine to accordingly clean up old state”.
* Aggregations over a sliding event-time window are straightforward with Structured Streaming and are very similar to grouped aggregations.
* windowing is similar to grouping, in code, you can use groupBy() and window() operations to express windowed aggregations.
* **Output mode must be Append or Update**
* The aggregation must have either the event-time column, or a window on the event-time column
* withWatermark must be called on the same column as the timestamp column used in the aggregate
* withWatermark must be called before the aggregation for the watermark details to be used

#### Example: “we want to count words within 10 minute windows, updating every 5 minutes.”

word counts in words received between 10 minute windows 12:00 - 12:10, 12:05 - 12:15, 12:10 - 12:20, etc. Note that 12:00 - 12:10 means data that arrived after 12:00 but before 12:10

Now, consider a word that was received at 12:07. This word should increment the counts corresponding to two windows 12:00 - 12:10 and 12:05 - 12:15. So the counts will be indexed by both, the grouping key and the window.

A picture containing diagram

Description automatically generated

#### Example “what happens if one of the events arrives late to the application”

A word generated at 12:04 (i.e. event time) could be received by the application at 12:11. The application should use the time 12:04 instead of 12:11 to update the older counts for the window 12:00 - 12:10.

Structured Streaming can maintain the intermediate state for partial aggregates for a long period of time such that late data can update aggregates of old windows correctly,

A picture containing diagram

Description automatically generated

* System should bound the amount of intermediate in-memory state it accumulates.
* System needs to know when an old aggregate can be dropped from the in-memory state
* Watermarking: lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly

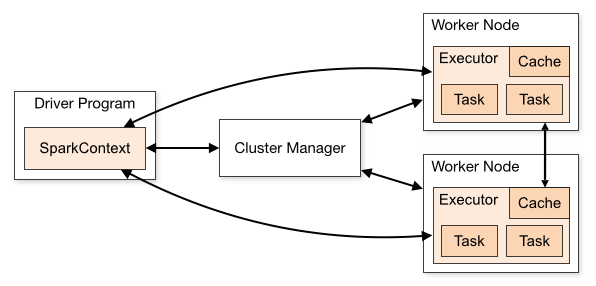
#### Important Points:

* Delivering end-to-end exactly-once semantics was one of key goals behind the design of Structured Streaming.
* It can handle any kind of failure by restarting and/or reprocessing.
* The engine uses checkpointing and write-ahead logs to record the offset range of the data being processed in each trigger.
* Streaming sinks are designed to be idempotent for handling reprocessing
* Together, using replayable sources and idempotent sinks, Structured Streaming can ensure **end-to-end exactly-once semantics** under any failure.
* DataFrames and Datasets can represent static, bounded data, as well as streaming, unbounded data

### Concepts:

* Submit Job:
  + Deploy-Mode: cluster, client
  + Master: Spark, Local, Yarn, Mesos, Kubernetes(k8)
* Initialize
  + Build SparkConf, contains information about your application.
    - new SparkConf().setAppName(appName).setMaster(master)
    - master:
      * Spark/Mesos/Yarn
  + Create SparkContext, tells how to access the Spark Cluster
    - new SparkContext(conf)
* Driver
  + Acts as a client to the cluster.
  + Runs the user’s “main” function
  + Executes various parallel operations on a cluster.
* Executors
  + This is where actual tasks are executed. Code inside Action calls, Map, FlatMap, Filter executes.
* Spark Context:
  + One per JVM. Represents connection to Spark Cluster and should be used to create RDD/Datasets and cache variables.
* Spark-Shell
* Sources: Files, Streaming Systems, Storage Systems
  + File: textFile, wholeTextFiles, kafka, jdbc(rdbma), ..
* Datasets
  + Distributed collection of data.
  + Datasets can be created from Hadoop InputFormats or by transforming other datasets.
* DataFrame:
  + Dataset organized into named columns. Dataset of rows.
  + Equivalent to a table in RDBMS.
* RDD
  + Main abstraction that spark provides. A collection of elements partitioned across the nodes of the cluster and can be operated in parallel.
  + Resilient Distributed Dataset, a fault-tolerant collection of elements that can be operated in parallel.
  + Function: parallelize
  + Operations: Transformations (map), Actions (reduce)
  + Allows persisting RDDs.
  + Fault-Tolerant and can recovered from failures.
* RDD Operations
  + Transformations: create a new dataset from an existing one
  + Actions: return a value to the driver program after running a computation on the dataset.
* Shared Variables
  + Can be accessed in parallel operations.
  + Default spark ships the copy of data to the remote workers for parallel operation. Spark supports Broadcast and Accumulators to support shared variables.
* Functions/Transformations: Map, Flatmap, Filter
  + Map: one-to-one
  + Flatmap: One-to-Many
* Cache: Broadcast, Accumulators
  + By default, each transformed RDD may be recomputed each time you run an action on it.
  + you may also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it
  + Save RDD on disk or in-memory.
* Sinks: Files, Streaming Systems, Storages
* Lazy Execution Model:
  + Map executes its functions only when the result of it is written somewhere.
  + The transformations are only computed when an action requires a result to be returned to the driver program
* Closures
* Shuffle: redistributes the data so that it is grouped again differently. Expensive operation. Use at caution. Spark must read from all partitions to find all the values for all keys, and then bring together values across partitions to compute the final result for each key.
  + Repartition, coalesce, groupByKey, reduceByKey, join, cogroup.
  + @TODO difference between Repartition and Coalesce.
    - Repartition: number can be increased or decreased. It makes new partitions and spread data over them evenly. It’s a full shuffle operation.
      * Pros: Equal sized data on each partition.
    - Coalesce: can only be decreased and optimized to use this instead of repartition. This uses existing partitions and moves only chosen partitions.
* Actions: foreach, reduce, count, take, saveAs,
* Spark-SQL
  + SQL API or Dataframe API
  + Read from Hive tables.
* User Defined Functions (UDF)
  + Refer to example code.
* Spark Streaming
* Structured Streaming:
  + A scalable and fault-tolerant stream processing engine with support for fast, scalable, fault-tolerant, end-to-end exactly-once.
  + Atleast Once: refer to examples shared
  + Exactly-Once:
  + Micro-Batch (100 millis) and Continuous (1 milli)
* Window Operations
  + Allows transformations over a sliding window of data.
* Checkpoint
  + Metadata Checkpoint: Config, DStream Operations, Incomplete Batches (queue and not yet completed)
  + Data Checkpoint: Save generated RDDs to reliable storages (Stateful transformations: updateStateByKey, reduceByKeyAndWindow).

### Cluster Mode:



* Driver program create SparkContext.
* Sends application code to the executors
* SparkContext send tasks to the executors to run
* Isolates one application from another sharing the same set of executors. When application is created, context, it gets a set of executor processes. It stays for length of the execution and runs tasks in multiple threads.
* Driver program controls schedule of tasks on the cluster. It should coexist with the worker for better efficiency.

### YARN:

* Acts as resource allocator and Cluster Manager.
* Alternatives: Kubernetes, Apache Mesos.
* YARN-Cluster: Spark driver runs inside an application master process which is managed by yarn on the cluster. Client is released once app is initialized.
* YARN-Client: Drives runs in the client process and the application master is only used for requesting resources from yarn.
* YARN UI: For application monitoring and debugging performance issues:

<https://databricks.com/blog/2015/07/08/new-visualizations-for-understanding-apache-spark-streaming-applications.html>

### Driver:

* Runs main programs
* Creates various parallel operations to perform on executors
* All aggregations runs on driver nodes like: collect(), take()

### Executors:

* All Parallel executions happens on executors
* Execution unit inside each function runs here.

### Sources:

* spark.read.load("...")
  + spark.read.format("json").load("..") //json, parquet, jdbc, orc, libsvm, csv, text
* spark.readStream.format("..”).load()
  + spark.readStream.format("kafka").option("kafka.bootstrap.servers", "host1:port1").option("subscribe", "topic1").load()

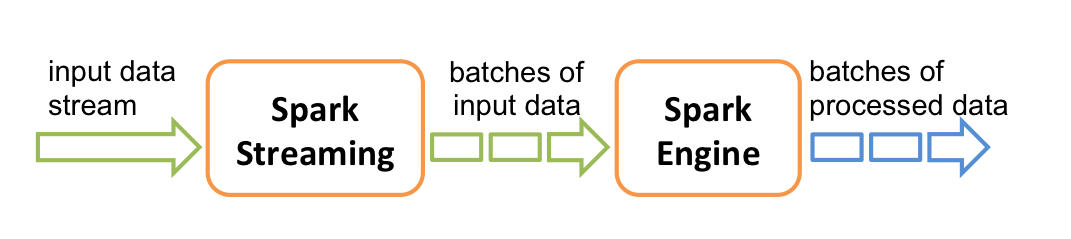
### Sink:

* DF.write.format("orc").option(..).save("..")
* DF.write.partitionBy("column\_name").format("parquet").save("..")
* DF.writeStream.format("..").start()
  + DF.writeStream.format("kafka").option("kafka.bootstrap.servers", "host1:port1").option("topic", "topic2").start()

### Streaming:

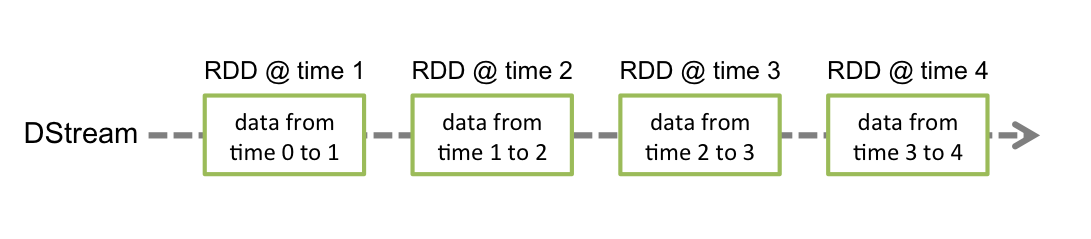
An extension that supports scalable, high-throughput, fault-tolerant stream processing engine.





#### DStream:

Discrete Streams, a continuous stream of data. Also referred as sequence of RDDs.



Input DStreams are DStreams representing the stream of input data received from streaming sources like Kafka.

Each RDD in a DStream contains data from a certain interval

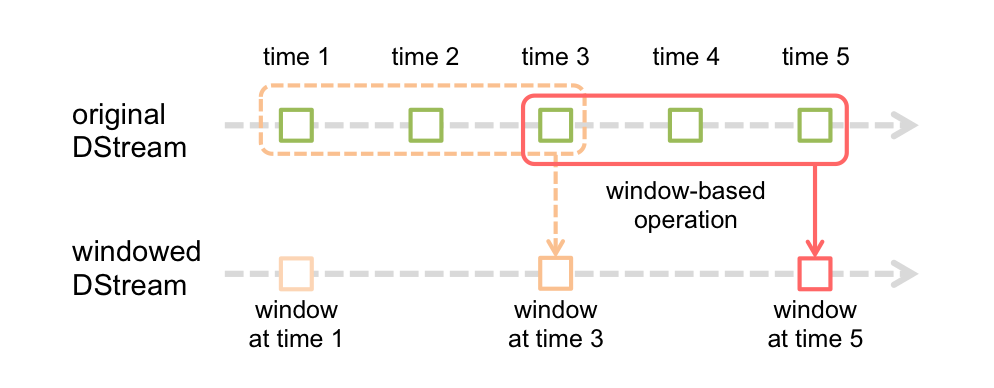
Any operation applied on a DStream translates to operations on the underlying RDDs, like split etc.

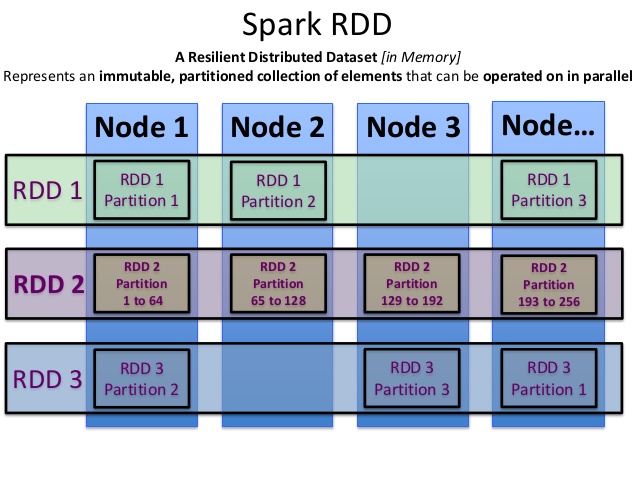
Each RDD pushed into the queue will be treated as a batch of data in the DStream, and processed like a stream.

Input Sources 🡪 DStream 🡪 Receiver 🡪 Spark In-Memory

#### Window Operations:

Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream





### Spark + Kafka

Provides simple parallelism, 1:1 correspondence between Kafka partitions and Spark partitions, and access to offsets and metadata.

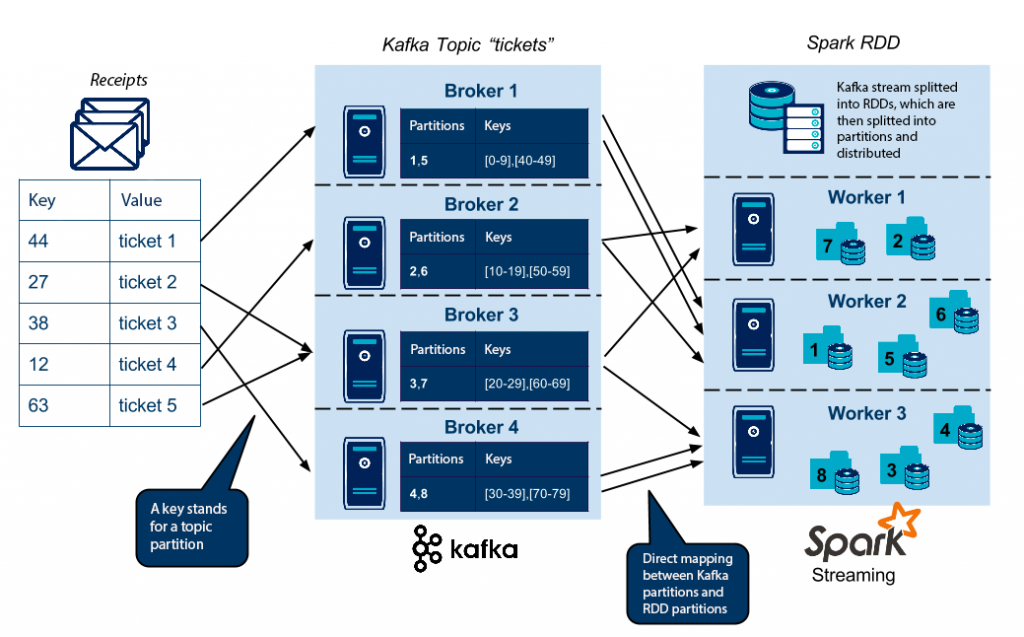
LocationStrategies.PreferConsistent: will distribute partitions evenly across available executors

PreferFixed: allows you to specify an explicit mapping of partitions to hosts

Direct Streaming:

A screenshot of a cell phone

Description automatically generated



#### RDD (Resilient Distributed Dataset)

An RDD is a fault-tolerant collection of elements that can be operated on in parallel. You can create them parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

#### Stateful:

Spark streaming supports stateful exactly-once out of box.

#### Exactly-Once:

* *At most once*: Each record will be either processed once or not processed at all.
* *At least once*: Each record will be processed one or more times. This is stronger than *at-most once* as it ensure that no data will be lost. But there may be duplicates.
* *Exactly once*: Each record will be processed exactly once - no data will be lost and no data will be processed multiple times. This is obviously the strongest guarantee of the three.

#### Spark-SQL:

Useful for querying structured data using DataFrame API or SQL syntax. Supports filtering, joins, grouping and more. Support HiveQL syntax, UDF’s and direct access to hive data stores.

Uses internally Cost Based Optimization for executing queries. It uses columnar storage and code generation that assist in running performant queries. It will launch spark jobs and use the power of cluster and scale to high volume data queries.

A picture containing drawing, woman

Description automatically generated

## Kerberos:

Part:1 Define JAAS File:

KafkaClient {

com.sun.security.auth.module.Krb5LoginModule required

doNotPrompt=true

useKeyTab=true

storeKey=true

keyTab="grsld.keytab"

renewTicket=true

serviceName="kafka"

principal="sa\_grsld@XYZ.GLOBAL";

};

Client {

com.sun.security.auth.module.Krb5LoginModule required

doNotPrompt=true

useKeyTab=true

storeKey=true

keyTab="grsld.keytab"

renewTicket=true

useTicketCache=true

serviceName="zookeeper"

client=true

principal="sa\_grsld@XYZ.GLOBAL";

};

Part:2

Unset and Re-Init before spark-submit:

unset KRB5CCNAME

kinit sa\_grsld@IHGINT.GLOBAL -kt /etc/grsld.keytab

klist

Include jaas in the spark-submit

--driver-java-options "-Djava.security.auth.login.config=kafka\_client\_jaas.conf" \

--files "kafka\_client\_jaas.conf,/etc/grsld.keytab,./subChannelDescription.csv,/etc/spark2/conf/hbase-site.xml" \

--conf "spark.executor.extraJavaOptions=-Djava.security.auth.login.config=kafka\_client\_jaas.conf" \

## Broadcast:

Setting Broadcast:

//query an existing hive table and save data in to broadcast

Dataset<Row> dtoLookUpDF = \_sparkSession.sql(kafkaConfig.getDtoLookUpQuery());

List<Row> rows = dtoLookUpDF.collectAsList(); //fetch all data in to a list

//create a broadcast variable and save in to it.

Broadcast<List> lookUpBroadcastVar = \_sparkSession.sparkContext().broadcast(rows,

scala.reflect.ClassManifestFactory.fromClass(List.class));

brdCstTblRefs.put("LOOKUP-QUERY", lookUpBroadcastVar);

//Accessing the save broadcast var:

Pass this: Broadcast<List> lookUpBroadcastVar to the methods

if (broadcastMap != null && broadcastMap.get("LOOKUP-QUERY") != null)

hotelBrandDFRows = broadcastMap.get("LOOKUP-QUERY").value();

## HBase:

A highly scalable, distributed big data storage system. It is a column-oriented non-relational database modeled after Google’s Bigtable. It runs on HDFS. Fault-tolerant ways to store sparse datasets.

Supports random realtime read/write access to Big Data.

Cons:

* Lack of supports for sql syntax.
* Data cannot be written in AVRO etc formats.
* Every record access must happen using primary-key.

### Features:

* Linear and modular scalability.
* Strictly consistent reads and writes.
* Automatic and configurable sharding of tables
* Automatic failover support between RegionServers.
* Block cache and Bloom Filters for real-time queries.
* Query predicate push down via server side Filters

HBase allows for many attributes to be grouped together into column families, such that the elements of a column family are all stored together. This is different from a row-oriented relational database, where all the columns of a given row are stored together. With HBase you must predefine the table schema and specify the column families. However, new columns can be added to families at any time, making the schema flexible and able to adapt to changing application requirements.

Command Line:

@instance-1 spark]$ hbase shell

hbase(main):001:0> list

TABLE

log\_agg

1 row(s)

Took 0.5507 seconds

=> ["log\_agg"]

hbase(main):002:0> scan 'log\_agg'

ROW COLUMN+CELL

\x00\x00\x00\x00\x00\x00\x00\x00 column=cf:application, timestamp=1585964037762, value=XYZ

\x00\x00\x00\x00\x00\x00\x00\x00 column=cf:message, timestamp=1585964037762, value=Message-1

\x00\x00\x00\x00\x00\x00\x00\x00 column=cf:module, timestamp=1585964037762, value=ModuleX

\x00\x00\x00\x00\x00\x00\x00\x01 column=cf:application, timestamp=1585964037762,

## Hive:

Apache Hive is an open source data warehouse software for reading, writing and managing large data set files that are stored directly in either the [Apache Hadoop Distributed File System (HDFS)](https://www.ibm.com/analytics/hadoop/hdfs) or other data storage systems such as [Apache HBase](https://www.ibm.com/analytics/hadoop/hbase). Hive enables SQL developers to write Hive Query Language (HQL) statements that are similar to standard SQL statements for data query and analysis.

Hive metastore, which enables you to apply a table structure onto large amounts of unstructured data.

Once you create a Hive table, defining the columns, rows, data types, etc., all of this information is stored in the metastore

Hive looks like traditional database code with SQL access. However, Hive is based on Apache Hadoop and Hive operations, resulting in key differences

First, Hadoop is intended for long sequential scans and, because Hive is based on Hadoop, queries have a very high latency (many minutes). This means Hive is less appropriate for applications that need very fast response times.

Second, Hive is read-based and therefore not appropriate for transaction processing that typically involves a high percentage of write operations. It is better suited for data warehousing tasks such as extract/transform/load (ETL), reporting and data analysis and includes tools that enable easy access to data via SQL.

## Data Formats:

### AVRO

Avro, as a component, supports a rich set of primitive data types including: numeric, binary data and strings; and a number of complex types including arrays, maps, enumerations and records. A sort order can also be defined for the data.

### ORC:

### PARQUET:

# Scala Program:

# Build and Deploy:

## Project:

### IDE:

IntelliJ

### Project:

New -> Project from Existing Source

Select the folder where you have unzipped the Spark.zip file

Or

New -> Project -> Scala -> sbt

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

## Build:

This is based on “sbt”. Please refer to “sbt” documentation for more information.

<https://www.scala-sbt.org/1.x/docs/sbt-by-example.html>

### Compile & Package:

>sbt clean compile package

## Payload:

>cd <source-dir>

>ls -lh target/scala-2.11/

Check for “spark-streaming\_2.11-0.1.jar”

## Upload:

#### To Sandbox:

Local to Sandbox:

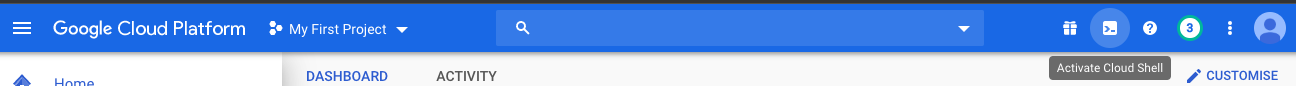
> scp target/scala-2.11/spark-streaming\_2.11-0.1.jar [root@192.168.5.137:/tmp](mailto:root@192.168.5.137:/tmp)

Sandbox to Final-Cluster:

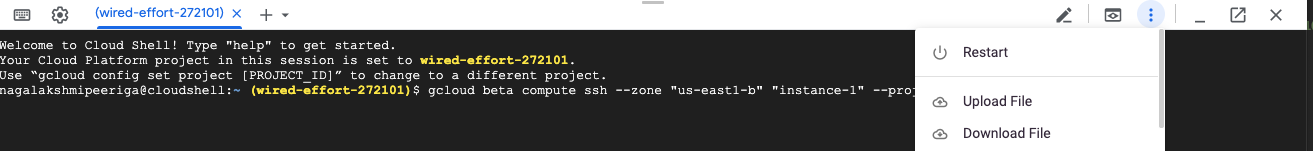
>scp root@192.168.5.137:/tmp/spark-streaming\_2.11-0.1.jar /opt/training/spark/

#### To GCP:

1: Gcp-cloud-console: <https://console.cloud.google.com/>

2: 

3: Upload File:



4: Select the payload jar file “spark-streaming\_2.11-0.1.jar” and upload it.

5: Upload to final Hadoop-cluster-node:

>gcloud beta compute scp --project="wired-effort-272101" --zone="us-east1-b" --recurse ./spark-streaming\_2.11-0.1.jar "instance-1":~/

>gcloud beta compute scp --project="wired-effort-272101" --zone="us-east1-b" --recurse ./runcmd.sh.\* "instance-1":~/

>gcloud beta compute scp --project="wired-effort-272101" --zone="us-east1-b" --recurse ./config.properties "instance-1":~/

A screenshot of a cell phone

Description automatically generated

6: Login to the hadoop-cluster-node:

> gcloud beta compute ssh --zone "us-east1-b" "instance-1" --project "wired-effort-272101"

## Run Program:

### Sandbox:

> cd /opt/training/spark

> sh runcmd.sh.produce

> sh runcmd.sh.consume (new shell window)

### GCP:

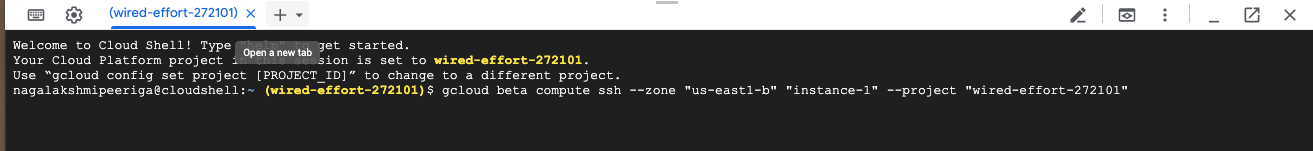
> gcloud beta compute ssh --zone "us-east1-b" "instance-1" --project "wired-effort-272101"

> cd /opt/training/spark

> vi config.properties (edit this file and change the properties like bootstrap.servers, topic.name etc)

> sh runcmd.sh.produce

> sh runcmd.sh.consume (new shell window)



# Applications

## Produce Data to Kafka

Program: DataProducer.scala

Partition Param: Code that support partition key hash.

References: <http://kafka.apache.org/documentation.html#producerapi>

## Consume Data from Kafka

Program: DataConsumer.scala

References: <http://kafka.apache.org/documentation.html#consumerapi>

## Kafka Streams Application

Program: KafkaStreaming.scala

References: <http://kafka.apache.org/documentation.html#streamsapi>

Configuration:

<http://kafka.apache.org/documentation.html#configuration>

## Spark Application:

### Read from Kafka:

com.cirrus.training.SparkStreaming

Submit File: runcmd.sh.spark (change config.properties and runcmd.sh.spark as needed)

### Apply Transformation

### Write to HDFS

### Write to HIVE

### Write to HBase

# Troubleshoot:

1:

Brokers:

>cd /usr/hdp/3.0.1.0-187/zookeeper/bin

>./zkCli.sh -server localhost:2181

[zk: localhost:2181(CONNECTED) 8] get /brokers/ids/1001

{"listener\_security\_protocol\_map":{"PLAINTEXT":"PLAINTEXT"},"endpoints":["PLAINTEXT://sandbox-hdp.hortonworks.com:6667"],"jmx\_port":-1,"host":

"sandbox-hdp.hortonworks.com","timestamp":"1584900008505","port":6667,"version":4}

2:

Caused by: java.lang.ClassNotFoundException: scala.Predef$

at java.net.URLClassLoader.findClass(URLClassLoader.java:382)

Solution: Make sure scala-lang library is picked properly in the classpath.

Ex: /usr/hdp/3.0.1.0-187/kafka/libs/scala-library-2.11.12.jar

3: Ambari Related Actions:

[root@sandbox-hdp ~]# ambari

ambari-admin-password-reset ambari-python-wrap ambari\_server\_main.py ambari-thp-disable.sh

ambari-agent ambari-server ambari-server.py

4: Zookeeper Related Actions:

[root@sandbox-hdp ~]# zookeeper-

zookeeper-client zookeeper-server zookeeper-server-cleanup

# Interview Preparation:

|  |  |
| --- | --- |
| Application | User program built on Spark. Consists of a driver program and executors on the cluster. |
| Application jar | A jar containing the user's Spark application. In some cases users will want to create an  "uber jar" containing their application along with its dependencies. The user's jar should never  include Hadoop or Spark libraries, however, these will be added at runtime. |
| Driver program | The process running the main() function of the application and creating the SparkContext |
| Cluster manager | An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos,  YARN) |
| Deploy mode | Distinguishes where the driver process runs. In "cluster" mode, the framework launches the  driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the  cluster. |
| Worker node | Any node that can run application code in the cluster |
| Executor | A process launched for an application on a worker node, that runs tasks and keeps data in  memory or disk storage across them. Each application has its own executors. |
| Task | A unit of work that will be sent to one executor |
| Job | A parallel computation consisting of multiple tasks that gets spawned in response to a Spark  action (e.g. save, collect); you'll see this term used in the driver's logs. |
| Stage | Each job gets divided into smaller sets of tasks called stages that depend on each other  (similar to the map and reduce stages in MapReduce); you'll see this term used in the  driver's logs. |