Operations & Technology

Enterprise Analytics Platform (EAP)
ETS : Introduction to Spark using PySpark

Citi Internal

February 25th, 2017

CATE: BIG DATA & ANALYTICS ENGINEERING

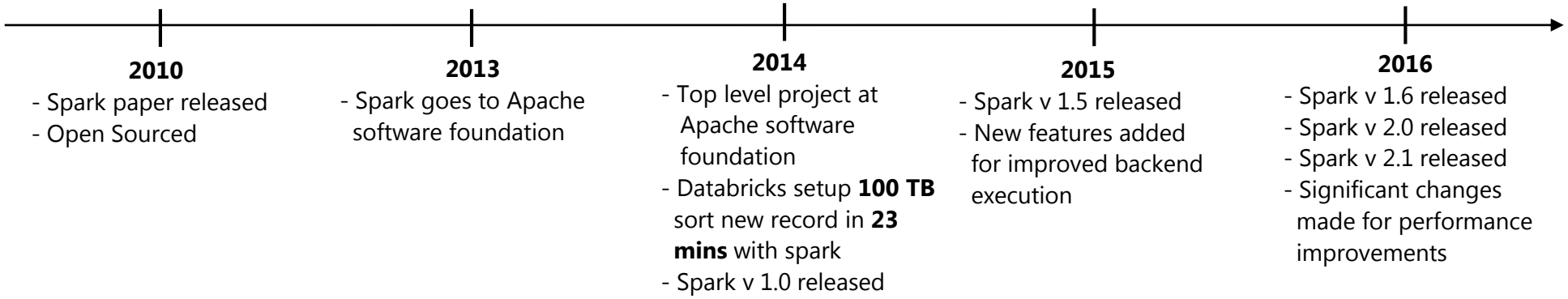
Data Science

Agenda

- **Introduction to Spark & its features**
  - Overview – What & Why
  - Architecture & Performance Benchmarking
  - Spark Ecosystem
    - Core Components – Spark Streaming, SQL, MLlib, GraphX & Spark R
    - APIs – Java, Scala, R & Python
      - Comparison of Spark API's
- **Introduction to RDDs & Dataframes**
  - Overview, Comparison & Operations
- **Programming**
  - Basic Operations – Importing files, data types, count, describe, output etc.
  - Transformation Functions – Filter, union, joins, Groupby etc.
  - Application Deployment in Spark

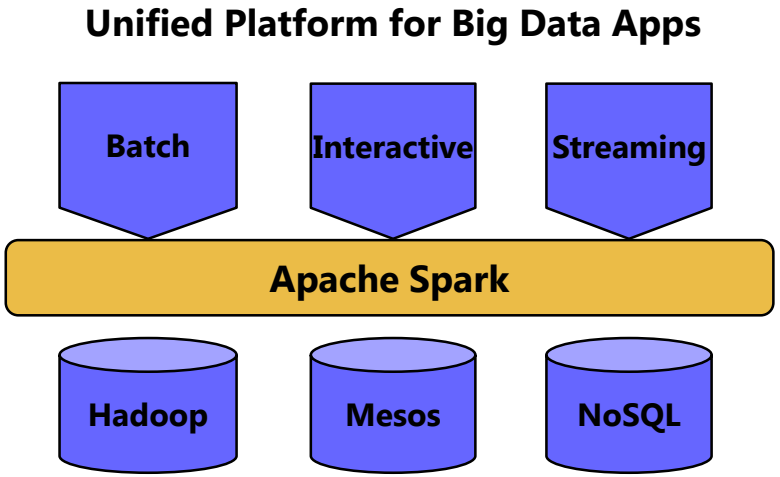
Introduction to Spark : Overview

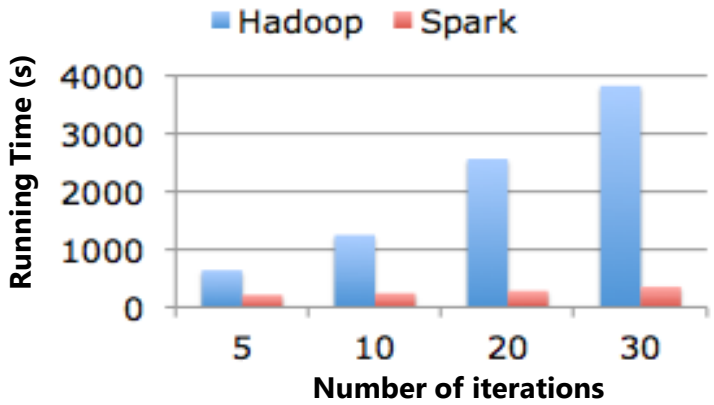
- Big data is about volume, velocity and variety. Nowadays, velocity has taken significance for real time application along with volume
- Hadoop –
  - Open source framework that allows for distributed processing of large unstructured datasets across clusters of computers
  - Based on a simple distributed programming model (MapReduce)
  - Computing is scalable, flexible, cost-effective & fault tolerant
  - **Concern** – Emphasis is on high throughput of data access rather than low latency of data access
- Spark, a fast computing engine designed for large scale data processing, was introduced to speed up the Hadoop computational process
- Developed in the AMP Lab at University of California, Berkeley but was later donated to the Apache Software Foundation
- Brief history of Spark –



Introduction to Spark : What & Why

- Spark is focused on –
  - Iterative programs (Machine Learning, Graph Analysis etc.)
  - Interactive Querying
  - Unifying real time and batch processing
- Spark is integrated with Hadoop and is used in tandem
  - Spark is not dependent on Hadoop as it has it's own cluster manager
  - Uses Hadoop mainly for storage
- **Features** -
  - ***Speed*** – x100 faster in memory processing (As it stores intermediate processing data **in memory**)
    – x10 faster for batch processing **on disk**
  - ***Supports multiple languages*** – Has built-in APIs in Java, Scala, Python & R
  - ***Advanced analytics*** – Supports map-reduce, SQL queries, streaming data, machine learning & graph algorithms
  - ***Runs on –*** Hadoop, Mesos, standalone or in cloud and can access diverse data sources including HDFS, Hbase, Cassandra, etc.





Introduction to Spark : Architecture

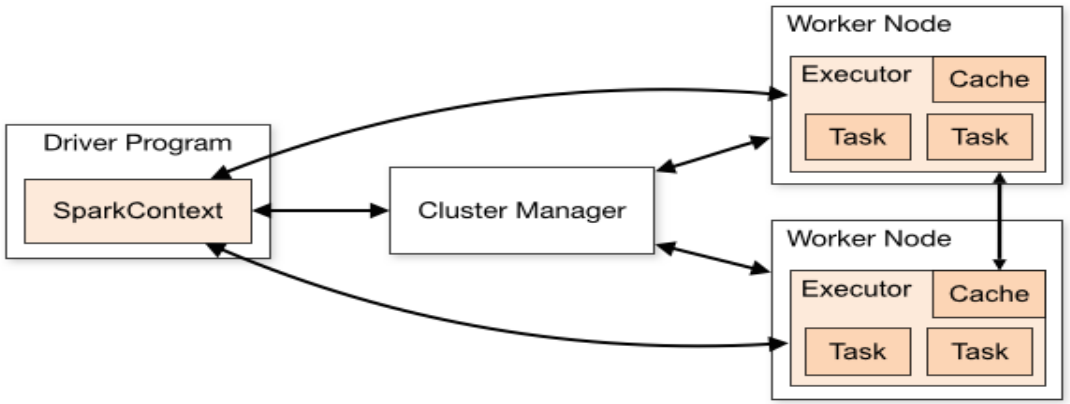
Spark Cluster is a **Master Slave Architecture** with two main processes –

1. ***Master Daemon*** (Driver/Master process) which executes on the Spark Master node
   - Driver processes manage executor processes via external resource manager (like YARN or Mesos)
2. ***Worker Daemon*** (Slave process) which executes on the Spark Worker node
   - Normally to leverage data locality, this process would run on Hadoop Data Node where data is physically available

Major components of Spark Architecture are –

- ***Spark context*** – Independent process through which spark application runs over a cluster
- ***Driver program*** – The application program which uses Spark context object to schedule jobs execution and negotiate with cluster manager
- ***Executors*** – Processes in the distributed nodes that run computations and store data for the applications

To run on a cluster, Spark context connects to cluster manager which allocates resources across applications → Spark takes over executors on distributed nodes in the cluster → Application codes are sent to the executors through spark context → Tasks are sent to the executors to run and complete it



Introduction to Spark : Comparison

- Comparison between Spark and Map Reduce –

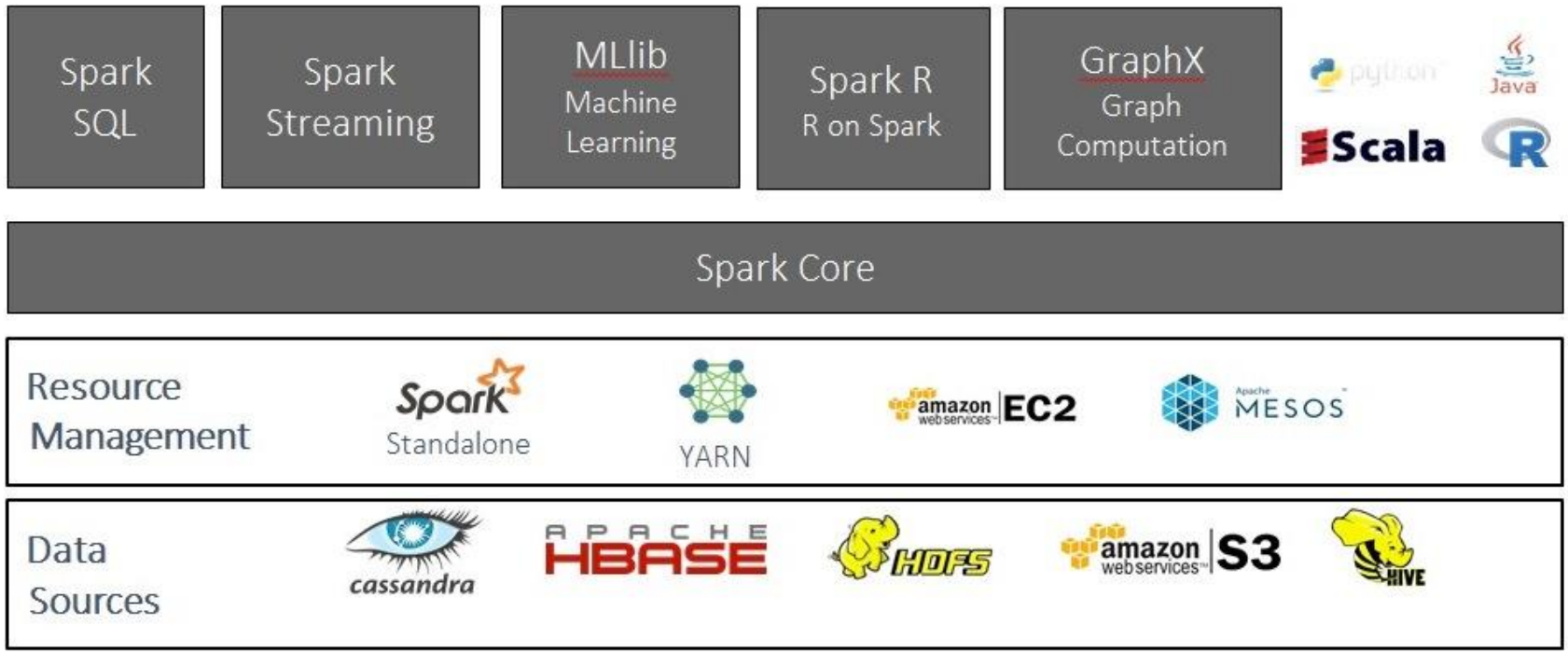
Metrics	Map Reduce	Spark
Speed	Slow due to replication and disk I/O, spends 90% of the time in HDFS read-write operations	10 to 100 times faster as it supports in-memory processing computation
Intermediate Storage	Stores intermediate results on disk (stable storage)	Stores intermediate results in a distributed memory
Pattern	Every use case has to be converted to map-reduce pattern	Uses Directed Acyclic Graph (DAG) pattern
Optimization	Not optimized, has high latency	Optimized execution as it does lazy evaluation (DAG)
Processing	Mainly used for batch processing	Used for batch, real time, streaming & interactive querying
Framework	Runs on Hadoop	Can be run on Hadoop (Yarn), Mesos or in standalone mode

Spark Ecosystem

Spark core is the general purpose system providing scheduling, parallelism and basic I/O functionalities. It uses a fundamental data structure called RDDs. It includes additional libraries which add powerful capabilities in Big Data Analytics like –

- ***Spark SQL***
  - Provides SQL like interface over Dataframes to query 'structured' data
  - Allows users to explore, transport & load data from different formats and query it
  - Supports JDBC & ODBC connections for integration with existing database
- ***Spark Streaming***
  - Used for processing real time streaming data
  - Uses a series of RDDs in form of micro batches for processing
- ***MLlib***
  - Spark's scalable machine learning library consists of common learning utilities
  - Includes algorithms like classification, regression, clustering etc.
- ***GraphX***
  - API for graphs and graph-parallel computations
  - Includes a collection of graph algorithms (like PageRank, Triangle Counting) to simplify graph analytics tasks
- ***Spark R***
  - R package providing an R frontend to Apache Spark
  - Allows data scientists to analyze large datasets and interactively run jobs on them from R shell

Spark Ecosystem

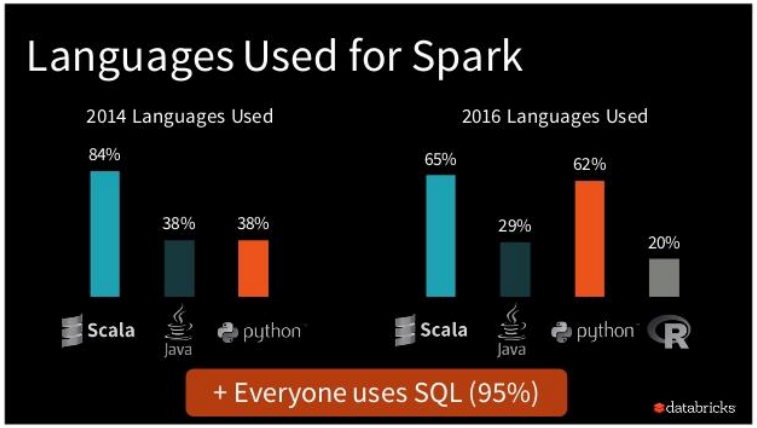


Comparison of Scala, Java & Python on Spark

PySpark API is recommended for **data scientists** on Citi EAP due to the following reasons –

- ***Presence of libraries*** – Python offers out of the box machine learning, statistical & numerical packages, and deep learning frameworks like keras, theano etc. specializing in rapid computation & experimentations which can be found only on Python. These help to clearly communicate output and predictions at any level of a business
- ***Speed*** – The advantage of speed offered by Scala over Python is insignificant compared to the overall speed improvement offered by Spark in general. Any slight loss of performance introduced by Python (in comparison to Scala) can be compensated in the design and operations of the cluster in order to fine tune the jobs
- ***Ease of Use*** – Python has a great interactive shell and a very wide community base. It's less verbose, has dynamic typing along with an easier syntax when compared to Scala and Java

Metrics	Scala	Python	Java
Verbose	Verbose	Less Verbose	Too Verbose
Syntax	Difficult	Easy	Moderate
Execution	Faster	Slower	Faster
Typing	Static	Dynamic	Static
ML & NLP Libraries Availability	Moderate	High	Moderate



* Word count example codes for Java, Scala & Python on Spark are attached below –

Wordcount Java.txt Wordcount Scala.txt Wordcount Python.txt

What are RDDs?

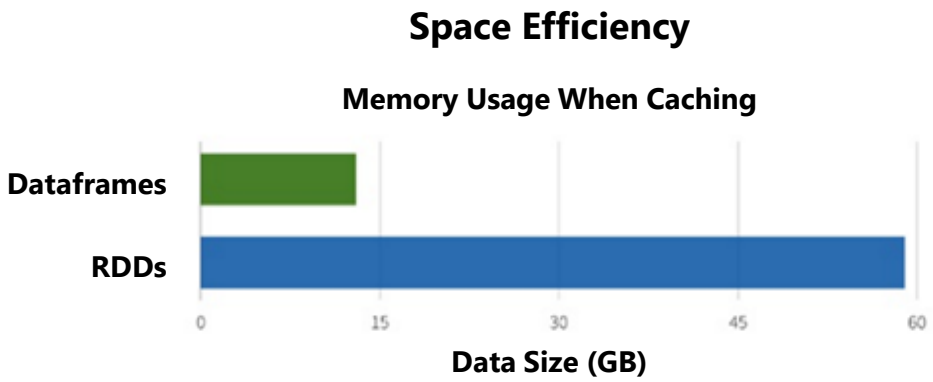
- It's a fundamental data structure of Spark
- Going by its name –
  - ***Resilient*** – Fault-tolerant with the help of RDD lineage graph, so able to recompute missing or damaged partitions due to node failure
  - ***Distributed*** – Data resides on multiple nodes in a cluster
  - ***Dataset*** – Collection of partitioned data
- **Properties** –
  - ***In-Memory storage*** – Data in stored in memory
  - ***Read-only*** – It does not change once created and can only be transformed using transformations to new RDDs
  - ***Lazy evaluation*** – Data is not transformed until an action is executed
  - ***Parallel processing*** – Processes data in parallel
  - ***Partitioned*** – The data inside a RDD is partitioned (split into partitions and then distributed)
- **Types of Operations** –
  - ***Transformations*** – Lazy operations that return another RDD
  - ***Actions*** – Operations that trigger computation and return values
- **Limitations** –
  - No inbuilt optimization engine
  - Doesn't infer schema of ingested data

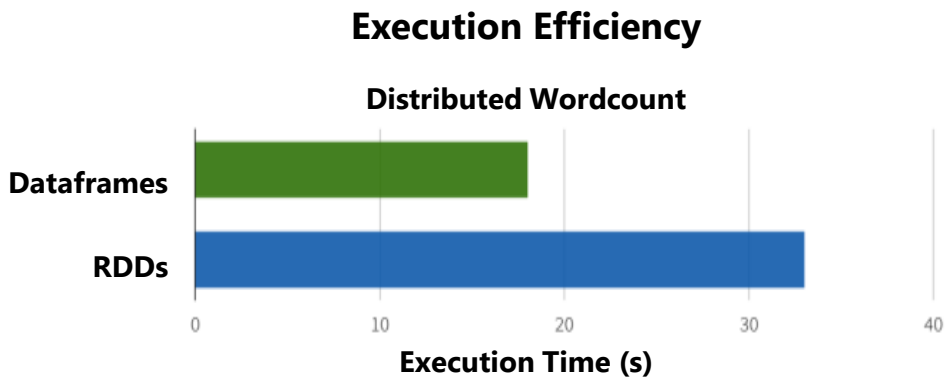
What are Dataframes?

- An abstraction which gives a schema view of the data
- **Properties** –
  - Supports different data formats and storage systems
  - Can be easily integrated with all big data tools
  - Ability to process the data in the size of kilobytes to petabytes on a single node cluster
  - It's API is available in Java, Scala, Python & R
- Like RDD, execution in Dataframes is lazy triggered
- Dataframe offers huge performance improvement over RDDs because of 2 powerful features –
  - ***Custom Memory management (aka Project Tungsten)***
    - Data is stored in off-heap memory in binary format
    - Also, there is no Garbage Collection overhead involved
  - ***Optimized Execution Plans (aka Catalyst Optimizer)***
    - Query plans are created for execution using Spark catalyst optimizer
- Other features –
  - ***Safety*** – Provides higher degree of type-safety at compile time. Dataframes can catch errors at compile time, which saves developer-time and costs
  - ***Ease-of-use of APIs with structure*** – Processing of high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access & use of lambda functions on semi-structured data is relatively more efficient

RDDs vs. Dataframes

Metrics	RDDs	Dataframes
Optimization	Execution plan needs to be optimized	Optimized execution plan
UDFs	Can use anonymous functions without registering them	UDFs must be registered
Transformation	No schema needs to be specified	Schema specification is mandatory
Optimizer	No parser or optimizer	SQL parser & optimizer
Performance	Different in different language APIs	Same across all different languages





Creating Dataframes

Using Spark SQL's Data Source API, we can read and write Dataframes using a variety of formats

Built in

External

Creating Dataframes

- Creating Dataframe from Hive tables

```
from pyspark.sql import HiveContext
sqlContext = HiveContext(sc)

dataframe = sqlContext.table("eaptraining_work.eap_loans")
```

- Creating Dataframe from existing RDD

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)

dataframe = loans_rdd.toDF()
```

- Creating Dataframe from csv files

```
from pyspark import SQLContext
sqlContext = SQLContext(sc)

dataframe_rdd = sc.textFile("/data/eaptraining/work/ETS training/loanStats.csv").map(lambda line: line.split(","))
dataframe  = dataframe_rdd.toDF(["col1","col2","col3",...])
```

Data prep and load checks

- Check data presence

```
dataframe.show(10)
```

- Check schema & data format

```
dataframe.printSchema()
```

- Count checks

```
dataframe.count()
dataframe.select(dataframe.id).count()
```

- Check using summary statistics of columns

```
dataframe.describe()
```

- Select

```
dataframe_sub = dataframe.select("id","loan_amnt")
```

- Drop

```
dataframe_sub = dataframe.drop("sub_grade","url")
```

Programming – Transforming Functions

- Filter

```
dataframe_refined = dataframe.filter((dataframe.loan_amnt > 3000) & (dataframe.int_rate > 9))
```

- GroupBy

```
dataframe_groupby = dataframe.groupBy(dataframe.grade)
```

- Aggregate

```
dataframe_agg = dataframe.groupBy(dataframe.term).agg({ “int_rate” : “max”}).show()
```

- Joins
  - Left Join

```
dataframe_joined = dataframe.join(state_info, dataframe.id = state_info.id, “leftouter”).drop(state_info.id)
```

  - Right join

```
dataframe_joined = dataframe.join(state_info, dataframe.id = state_info.id, “right”).drop(state_info.id)
```

- Union

```
dataframe_union = dataframe_1.unionAll(dataframe_2)
```

PySpark – Deployment

- Spark codes need to be executed & application deployed via **SSH Tectia (CLI)**
- **Apache Zeppelin and Jupyter (formerly IPython)** notebooks are in pipeline for Integrated Development Environment (IDE)
- These are web applications that allow users to create, share & document code, text, plots, etc.

- Steps to deploy on EAP –
  - ***spark-submit*** – Shell command used to deploy the Spark application on a cluster across all managers uniformly

```
spark-submit [options]* <python file> [app arguments]
```

  - The below example demonstrates submitting of **pyspark application**# by changing timeout, executor/driver memory from CLI

```
[nb50720@bdhkproxy01i1u ~]$ export PYSPARK_DRIVER_PYTHON=/opt/cloudera/parcels/Anaconda3/bin/python
[nb50720@bdhkproxy01i1u ~]$ export PYSPARK_PYTHON=/opt/cloudera/parcels/Anaconda3/bin/python
[nb50720@bdhkproxy01i1u ~]$ spark-submit --conf "spark.network.timeout=500s" --driver-memory 20g --executor-memory 14g --conf spark.executorEnv.PYTHONHASHSEED=0 /home/nb50720/py_dummy_v13
_poslogic_v2.py > /home/nb50720/dummy_filedescriptor_test2.out 2>&1
```

- Tracking URL from the job logs can be used to obtain the below information –
  - Application ID
  - Total uptime & event timeline (used to identify failed executors)
  - Scheduling mode & environment variable parameters like driver host, memory, timeout, etc.
  - Job details like shuffle Read/Write, Description, Task succeeded & DAG visualization

- The **Environment** # tab lists Spark properties like Application properties, Runtime environment, Shuffle behavior, Executor behavior etc.

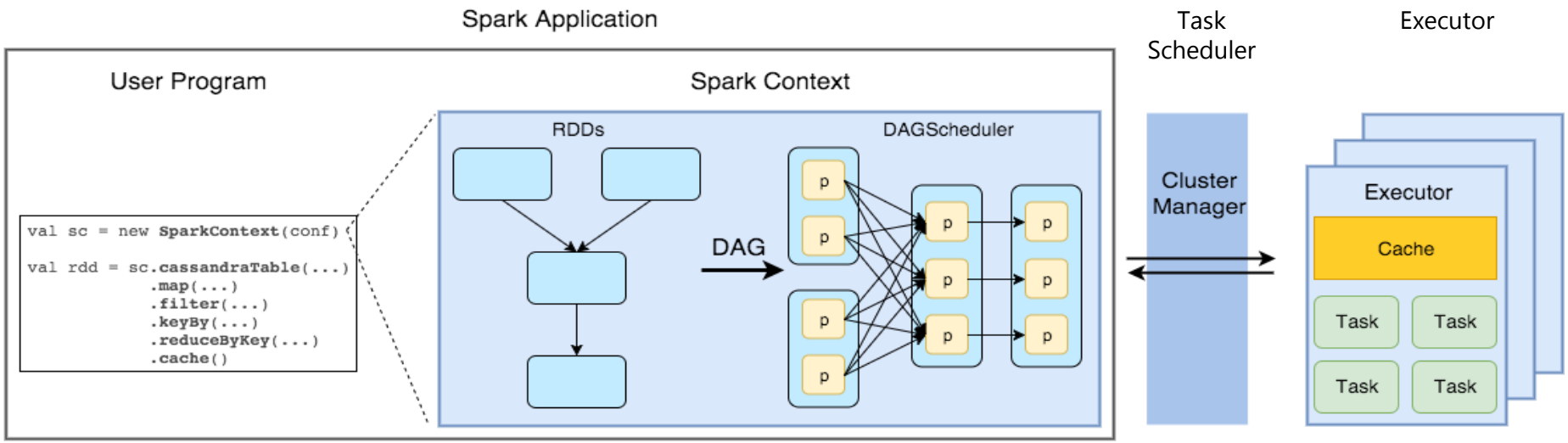
For list of important options refer to the **appendix*
#*For the application parameters refer to the **appendix***

Appendix

- Spark Architecture
- Spark Deployment
- Spark Job & Executor Details
- PySpark Job Parameters
- Memory Management in Spark

Spark Architecture

- **Spark application** (Driver program) consists of **Spark context** and **User Program** which interacts with it creating RDDs & performing a series of transformations
- These RDD transformations are then translated into **DAG** and submitted to **DAG scheduler** to be executed on a set of worker nodes
- **DAG Scheduler** computes a DAG of stages for each task & submits them to **Task Scheduler**. It also determines preferred locations for tasks (based on cache status or shuffle file locations) and finds minimum schedule to run the jobs
- **Task Scheduler** sends the jobs received from DAG Scheduler to the cluster, retrying in case of failures & mitigates stragglers
- Cluster basically has 2 components – Resource Manager & Node Manager
- When job comes to the cluster, **Resource Manager** determines the Node Manager to be contacted which would provide the execution containers (JVMs with requested heap size). Also, it decides the location of the execution container
- The **Node Manager** controls the node resource utilization and looks into the execution of the tasks in the nodes. The tasks spawn JVMs for computations



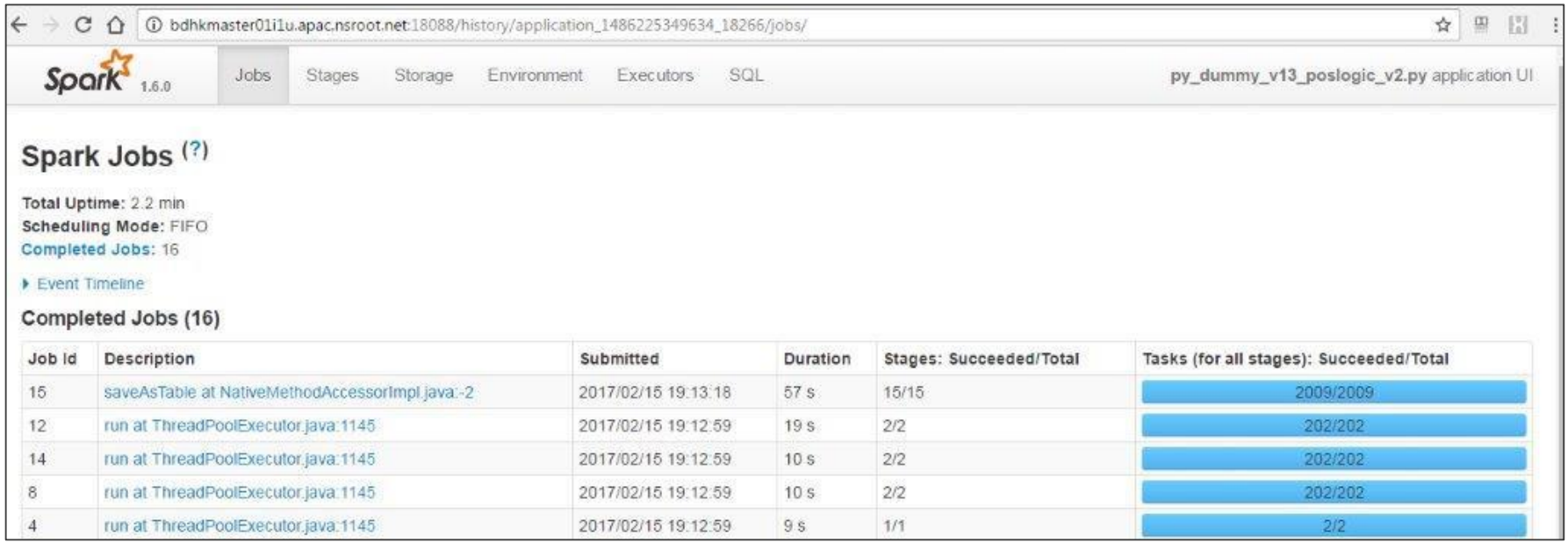
Spark – Deployment (1/2)

SL No.	Option	Description
1	--deploy-mode	Launches the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster")
2	--name	Name of your application
3	--py-files	Comma-separated list of .zip, .egg, or .py files to place on the PYTHON PATH for Python apps
4	--files	Comma-separated list of files to be placed in the working directory of each executor
5	--properties-file	Path to a file from which to load extra properties. If not specified, this will look for conf/spark-defaults
6	--driver-memory	Memory for driver

Spark – Deployment (2/2)

SL No.	Option	Description
7	--executor-memory	Memory per executor
8	--verbose, -v	Print additional debug output
9	--executor-cores	Number of cores per executor
10	--total-executor-cores	Total cores for all executors
11	--status	Requests the status of the driver specified
12	--driver-memory	Memory for driver

Spark Job and Executor details



bdhkmaster01i1u.apac.nsroot.net:18088/history/application_1486225349634_18266/jobs/

Spark 1.6.0 | Jobs | Stages | Storage | Environment | Executors | SQL | py_dummy_v13_poslogic_v2.py application UI

Spark Jobs (?)

Total Uptime: 2.2 min
Scheduling Mode: FIFO
Completed Jobs: 16

▸ Event Timeline

Completed Jobs (16)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
15	saveAsTable at NativeMethodAccessorImpl.java:-2	2017/02/15 19:13:18	57 s	15/15	2009/2009
12	run at ThreadPoolExecutor.java:1145	2017/02/15 19:12:59	19 s	2/2	202/202
14	run at ThreadPoolExecutor.java:1145	2017/02/15 19:12:59	10 s	2/2	202/202
8	run at ThreadPoolExecutor.java:1145	2017/02/15 19:12:59	10 s	2/2	202/202
4	run at ThreadPoolExecutor.java:1145	2017/02/15 19:12:59	9 s	1/1	2/2

Executors (173)

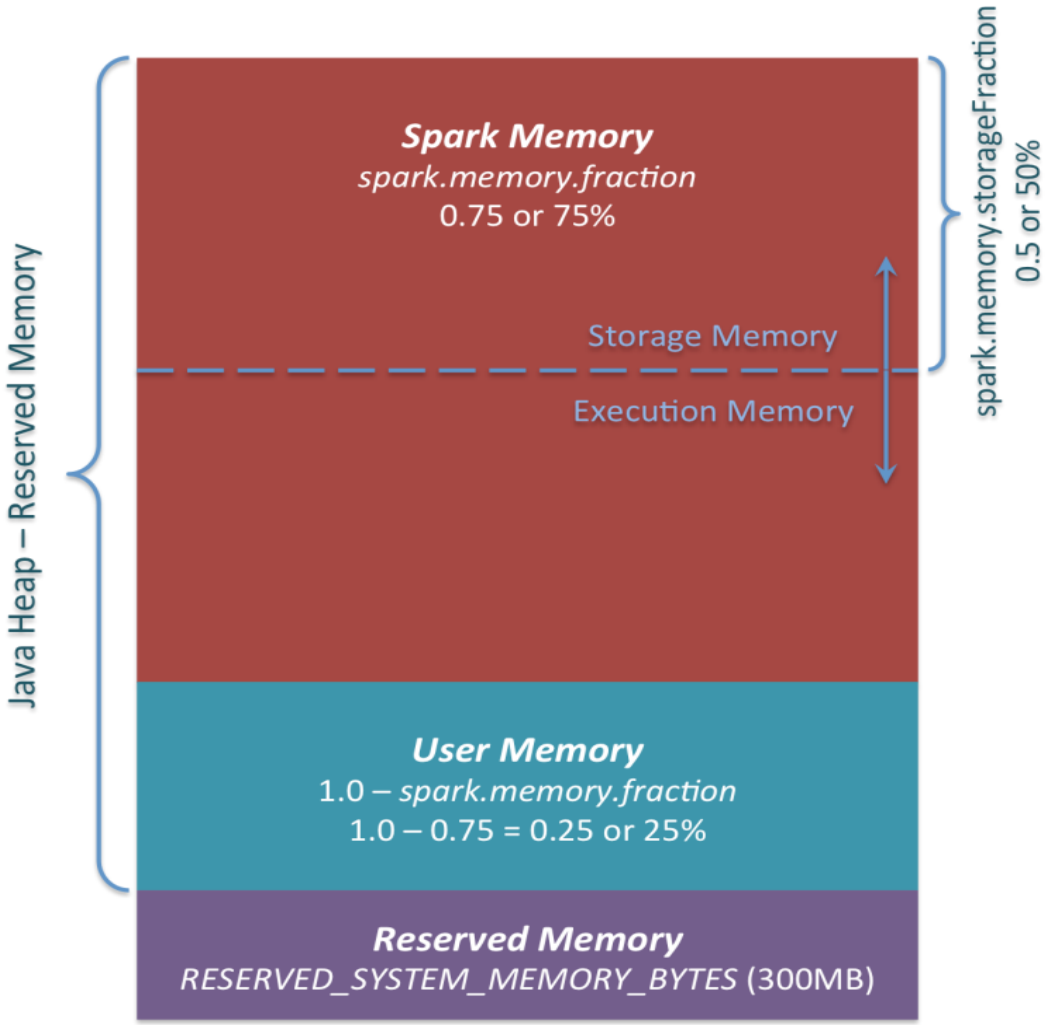
Memory: 0.0 B Used (1256.6 GB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
1	bdhkr003d07i2u.apac.nsroot.net:39187	0	0.0 B / 7.2 GB	0.0 B	0	0	10	10	11.3 s	384.0 KB	7.6 KB	9.8 KB	stdout stderr
10	bdhkr002d03i1u.apac.nsroot.net:39745	0	0.0 B / 7.2 GB	0.0 B	0	0	83	83	28.1 s	8.4 KB	26.3 KB	78.1 KB	stdout stderr

PySpark application parameters

spark.yarn.appMasterEnv.PYSPARK_PYTHON	/opt/cloudera/parcels/Anaconda3-4.1.2/bin/python
spark.yarn.historyServer.address	http://bdhkmaster01i1u.apac.nsroot.net:18088
spark.app.name	pyspark_step3_v11.py
spark.network.timeout	600s
spark.dynamicAllocation.schedulerBacklogTimeout	1
spark.yarn.am.extraLibraryPath	/opt/cloudera/parcels/CDH-5.7.1-1.cdh5.7.1.p1876.1944/lib/hadoop/lib/native
spark.scheduler.mode	FIFO
spark.driver.memory	20g
spark.yarn.config.gatewayPath	/opt/cloudera/parcels
spark.executor.id	driver
spark.yarn.config.replacementPath	{{HADOOP_COMMON_HOME}}/../../..
spark.yarn.appMasterEnv.PYSPARK_DRIVER_PYTHON	/opt/cloudera/parcels/Anaconda3-4.1.2/bin/python
spark.submit.deployMode	client
spark.shuffle.service.port	7337
spark.master	yarn-client
spark.authenticate	false
spark.ui.filters	org.apache.hadoop.yarn.server.webproxy.amfilter.AmIpFilter
spark.executor.extraLibraryPath	/opt/cloudera/parcels/CDH-5.7.1-1.cdh5.7.1.p1876.1944/lib/hadoop/lib/native
spark.executor.memory	14g
spark.eventLog.dir	hdfs://apacuat/user/spark/applicationHistory
spark.dynamicAllocation.enabled	true

Memory Management In Spark



- **Spark Memory –** Memory pool managed by Spark is split into 2 regions –
  - ***Storage Memory*** – Stores cached data along with all the 'broadcast variables' which get stored as cached blocks
  - ***Execution Memory*** – Stores objects required during the execution (sorting, shuffling etc.) of Spark tasks

The memory management is automized. In case of memory pressure, one region grows by borrowing space from the other

- **User Memory –** Memory that can be utilized by the user itself to store anything like maintaining hash tables, storing the data structures needed for RDD transformations etc.

- **Reserved Memory –** Memory reserved by the system whose size is hardcoded (300MB). It stores Spark's internal objects and doesn't participate in Spark memory region size calculations

Memory Management In Spark

Property Name	Default	Meaning
spark.memory.fraction	0.75	It sets aside memory for internal metadata. Leaving it at default value is recommended. The lower this is, the more frequently spills and cache data eviction occurs
spark.memory.storageFraction	0.5	The higher this is, the lesser working memory will be available to execution. Leaving this at default is recommended
spark.memory.useLegacyMode	False	It rigidly partitions the heap space into fixed-size regions, potentially leading to excessive spilling if the application was not tuned. It is disabled by default.
spark.shuffle.memoryFraction	0.2	This is read only if *spark.memory.useLegacyMode* is enabled. The collective size of all in-memory maps used for shuffles is bounded by this limit, beyond which the contents will begin to spill to disk
spark.storage.memoryFraction	0.6	This is read only if *spark.memory.useLegacyMode* is enabled. If spills to disk are often, using this parameter can increase the value of *spark.shuffle.memoryFraction*
spark.storage.unrollFraction	0.2	This is read only if *spark.memory.useLegacyMode* is enabled. It represents the fraction of *spark.shuffle.memoryFraction* to be used for unrolling blocks in memory