HDP Developer: Enterprise Apache Spark

Connection before Content

Lester Martin – Hortonworks Training & Consulting

lmartin@hortonworks.com

http://lester.website

(links to blog, twitter, github, LI, FB, etc)

Course Agenda

The following topics will be covered during this course on developing with Apache Spark:

- Monday
  - HDP and Spark Overviews
  - RDD and PairRDD Programming
- Tuesday
  - Spark Streaming
  - Spark SQL (DataFrames)
  - Visualization with Zeppelin
- Wednesday
  - Monitoring and Performance Considerations
  - Stand-alone Applications
  - Introduction to MLlib

Introductions

- Your name and physical location
- Your job role and responsibilities
- Your Big Data / Hadoop and Apache Spark experience, if any
- Your expectations for this course
- A favorite hobby

Logistics

- Mon-Fri from 10:30am-5pmEST – *Instructor will stay on WebEx for a while and join early*
- Breaks and lunch
  - Generally, let's take breaks with labs
    - BUT... **chime-up if we could benefit from one at any time!**
  - Lunch: 1:30pmEST
  - Any team meetings to work around?

Courseware & Lab Environments

- Downloaded PDFs
  – Student Guide
  – Lab Guide
    • PLUS one called **AdditionalLabs.pdf** I will distribute
- AWS hosted VMs
  – IP's handed out at first lab
  – Destroyed after class is over

7

Objectives

After completing this lesson, students should be able to:

- Describe the characteristics and types of Big Data
- Define HDP and how it fits into overall data lifecycle management strategies
- Describe and use HDFS
- Explain the purpose and function of YARN

- Defining Big Data

Objectives

What Makes Data "Big" Data?

- The term Big Data comes from the computational sciences
- It is used to describe scenarios where the volume, rate of creation, and types of data threaten to overwhelm the tools used to store and process it

Three V's	Description
VOLUME	Petabytes and more, spurred by exponential growth in computers, sensors, social media, and regulatory requirements.
Velocity	Gigabytes per *second,* and faster, plus new data and new ways to create data are generated an an increasing rate.
Variety	Structured, semi-structured, unstructured. Databases, XML, JSON, text, photo, video, audio, etc.

Common Types of Data in Hadoop

- There are six types of data commonly found in Hadoop.
  - Sentiment data: how customers react
  - Clickstream data: website visitor behavior
  - Sensor or machine data: data from remote devices
  - Geographic data: location-based data
  - Server log data: failure and security logs
  - Text: email, web pages, documents, etc.

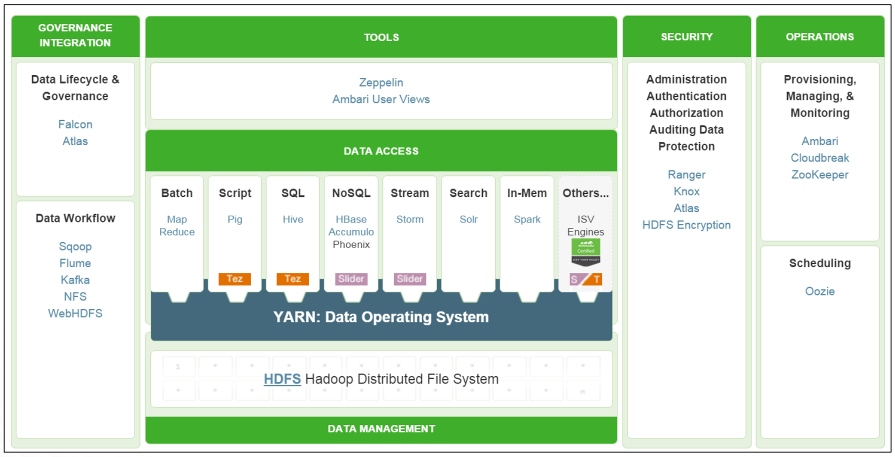
Objectives

- Defining Big Data
- HDP Introduction

What is Hadoop?

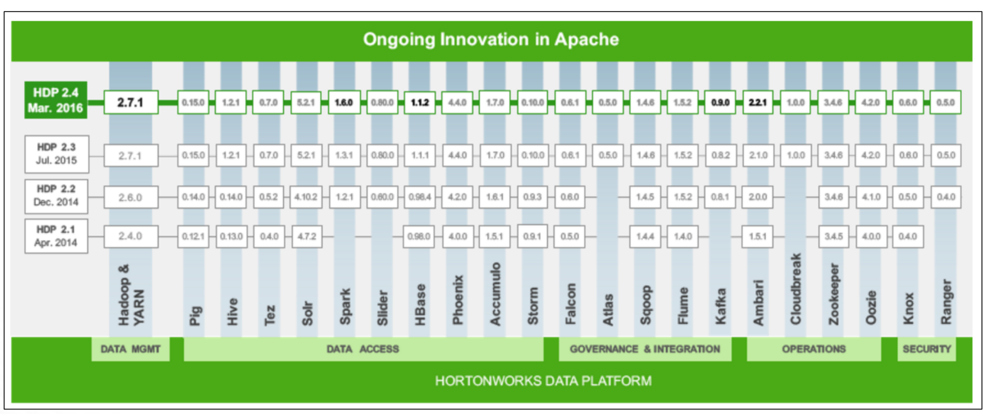
- Hadoop:
  - Is a collection of open source software frameworks for the distributed storing and processing of large sets of data
  - Is scalable and fault tolerant
  - Works with commodity hardware
  - Processes all types of Big Data
- Hadoop design goals:
  - Use inexpensive, enterprise-grade hardware to create very large clusters
  - Achieve massive scalability through distributed storage and processing
- HDP is an enterprise-ready collection of these frameworks
  - Supported by Hortonworks for business clients

Hortonworks Data Platform



15

HDP Introduction



16

Data Management and Operations Frameworks

Framework	Description
Hadoop Distributed File System (HDFS)	A Java-based, distributed file system that provides scalable, reliable, high-throughput access to application data stored across commodity servers
Yet Another Resource Negotiator (YARN)	A framework for cluster resource management and job scheduling

Framework	Description
Ambari	A Web-based framework for provisioning, managing, and monitoring Hadoop clusters
ZooKeeper	A high-performance coordination service for distributed applications
Cloudbreak	A tool for provisioning and managing Hadoop clusters in the cloud
Oozie	A server-based workflow engine used to execute Hadoop jobs

These brief descriptions are provided for quick convenience. More detailed descriptions are available online or in other lessons and courses.

17

Data Access Frameworks

Framework	Description
Pig	A high-level platform for extracting, transforming, or analyzing large datasets
Hive	A data warehouse infrastructure that supports ad hoc SQL queries
HCatalog	A table information, schema, and metadata management layer supporting Hive, Pig, MapReduce, and Tez processing
Cascading	An application development framework for building data applications, abstracting the details of complex MapReduce programming
HBase	A scalable, distributed NoSQL database that supports structured data storage for large tables
Phoenix	A client-side SQL layer over HBase that provides low-latency access to HBase data
Accumulo	A low-latency, large table data storage and retrieval system with cell-level security
Storm	A distributed computation system for processing continuous streams of real-time data
Solr	A distributed search platform capable of indexing petabytes of data
Spark	A fast, general purpose processing engine use to build and run sophisticated SQL, streaming, machine learning, or graphics applications.

18

Governance and Integration Frameworks

Framework	Description
Falcon	A data governance tool providing workflow orchestration, data lifecycle management, and data replication services.
WebHDFS	A REST API that uses the standard HTTP verbs to access, operate, and manage HDFS
HDFS NFS Gateway	A gateway that enables access to HDFS as an NFS mounted file system
Flume	A distributed, reliable, and highly-available service that efficiently collects, aggregates, and moves streaming data
Sqoop	A set of tools for importing and exporting data between Hadoop and RDBM systems
Kafka	A fast, scalable, durable, and fault-tolerant publish-subscribe messaging system
Atlas	A scalable and extensible set of core governance services enabling enterprises to meet compliance and data integration requirements

Security Frameworks

Framework	Description
HDFS	A storage management service providing file and directory permissions, even more granular file and directory access control lists, and transparent data encryption
YARN	A resource management service with access control lists controlling access to compute resources and YARN administrative functions
Hive	A data warehouse infrastructure service providing granular access controls to table columns and rows
Falcon	A data governance tool providing access control lists that limit who may submit Hadoop jobs
Knox	A gateway providing perimeter security to a Hadoop cluster
Ranger	A centralized security framework offering fine-grained policy controls for HDFS, Hive, HBase, Knox, Storm, Kafka, and Solr

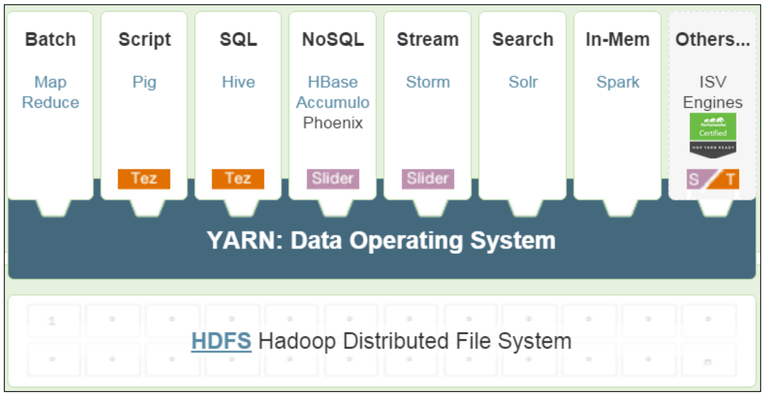
Pre-Lab: Setting Up the Lab Environment

Objectives

- Defining Big Data
- HDP Introduction
- HDFS Overview

22

HDFS and YARN are the Core of HDP



HDFS – The HDP File System

- Hadoop stores files using the Hadoop distributed file system (HDFS).
- HDFS is the basis for Hadoop's storage scalability and availability. HDFS:
  - Splits large data files into smaller chunks called blocks
  - Spreads those blocks across different slave/worker nodes
  - Tracks data block location
  - Automatically replicates data for high availability
- Scaling storage is easy – simply add more nodes!

HDFS Command Line Interaction

`hdfs dfs –command [args]`

- `-cat`: display file content (uncompressed)
- `-text`: just like `-cat` but works on compressed files
- `-mkdir`: create a directory in HDFS
- `-put`, `-get`, `-mv`: copies files between local file system and HDFS, as well as move within HDFS.
- `-ls`, `-rm`: list and remove files/directories (add `-R` to make commands recursive)
- `-chgrp`, `-chmod`, `-chown`: changes file permissions
- `-stat`: statistical info for a given file

HDFS Commands and Permissions

- `hdfs dfs -mkdir mydata`
- `hdfs dfs -put numbers.txt mydata/`
- `hdfs dfs -ls mydata`

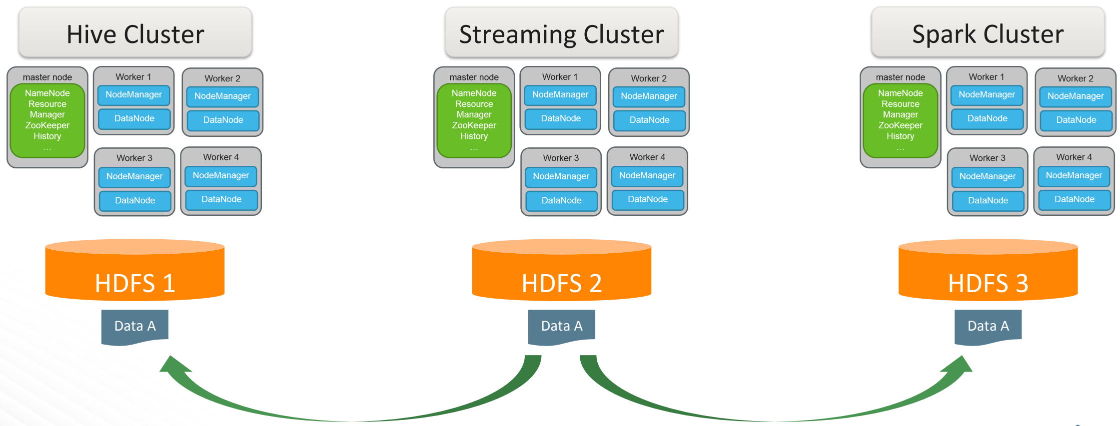
- HDFS implements a POSIX-style permissions model
  - User, group, and other rwx permissions for files and directories
  - Files: r = read, w = write or append
  - Directories: r = list contents, w = create or delete files or subdirectories, x = access a child object

Lab: Using HDFS Commands

Objectives

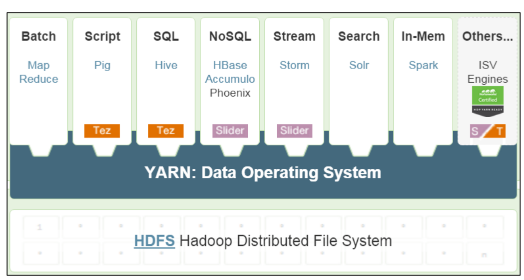
- Defining Big Data
- HDP Introduction
- HDFS Overview
- YARN Overview

Hadoop Applications Without YARN



YARN – the HDP Operating System

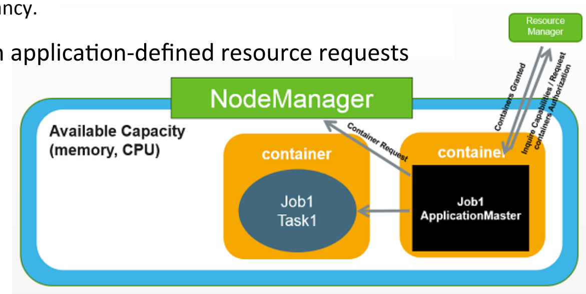
- Apache Hadoop YARN is the data operating system for Hadoop 2.
- YARN is:
  - Responsible for scheduling tasks and managing CPU and memory resources
  - Designed to enable multiple distributed applications to utilize cluster resources in a shared, secure, and multi-tenant manner



YARN Resource Containers

A resource container:

- Is the abstraction used to represent a discreet amount of CPU and memory resources on a machine
  - Hadoop applications run inside containers.
- Is managed and scheduled by YARN
- Is logically isolated from other containers running on the same machine
  - Isolation supports application multi-tenancy.
- Is allocated in different sizes based on application-defined resource requests



31

Knowledge Check

Questions

1. Name the three V's of big data.
2. Name four of the six types of data commonly found in Hadoop.
3. Why is HDP comprised of so many different frameworks?
4. What two frameworks make up the core of HDP?
5. What is the base command-line interface command for manipulating files and directories in HDFS?
6. YARN allocates resources to applications via ______________________.

Summary

Summary

- Data is made "Big" Data by ever-increasing Volume, Velocity, and Variety
- Hadoop is often used to handle sentiment, clickstream, sensor/machine, server, geographic, and text data
- HDP is comprised of an enterprise-ready and supported collection of open source Hadoop frameworks designed to allow for end-to-end data lifecycle management
- The core frameworks in HDP are HDFS and YARN
- HDFS serves as the distributed file system for HDP
- The `hdfs dfs` command can be used to create and manipulate files and directories
- YARN serves as the operating system and architectural center of HDP, allocating resources to a wide variety of applications via containers

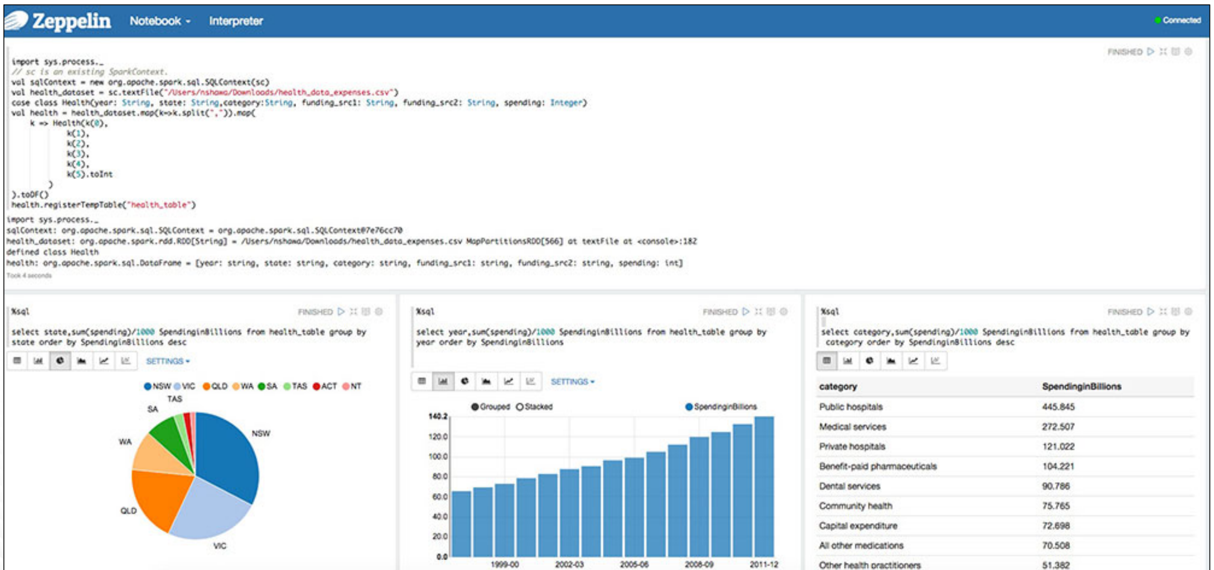
35

Objectives

- Use Apache Zeppelin to work with Spark
- Describe the purpose and benefits of Spark
- Define Spark REPLs and application architecture

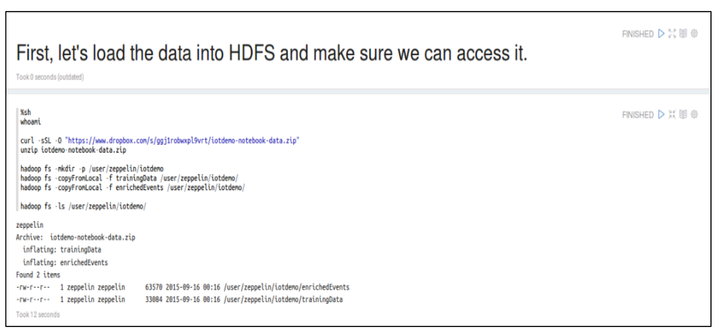
- Zeppelin Overview

Objectives



Zeppelin Major Functions

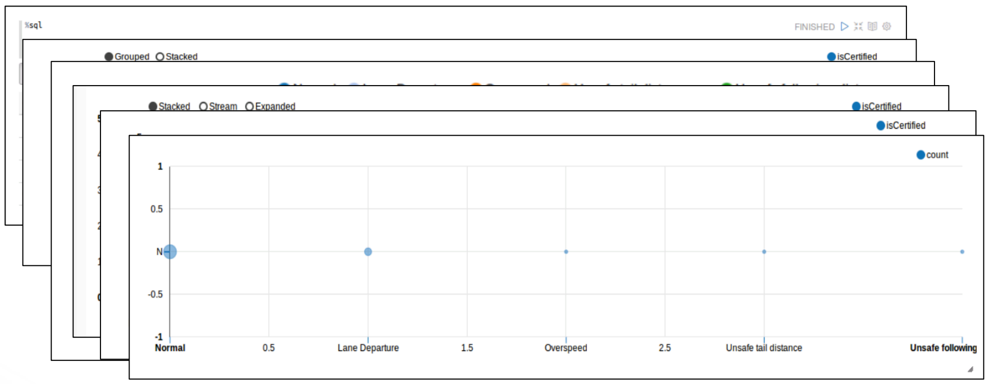
- Data Ingestion
- Data Discovery
- Data Analytics



- Data Visualization and Collaboration

Data Visualization

- Several built-in ways to interactively view / visualize data
  - Table
  - Column
  - Pie
  - Area
  - Line
  - Scatter



What Does "Tech Preview" Mean?

- Tech Preview status means a technology is not yet fully recommended for use in production environments

- Why Zeppelin is used in this course:
  - The issues that keep it in tech preview do not affect learning - *nearly* all labs could be run in either command line or Zeppelin with identical steps
  - Zeppelin for data visualization and collaboration is currently the best solution in HDP - even in tech preview - for multi-language support (Python, Scala, etc.)
  - When Zeppelin comes out of tech preview (may already be true by the time you take this course), you will already have significant hands-on experience

- Zeppelin Overview
- Spark Overview

Objectives

43

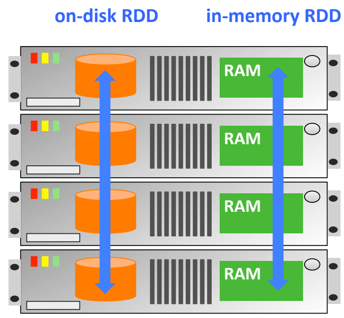
Spark Introduction

- Large-scale, cluster-based, in-memory data processing platform
- Development APIs for Scala, Java, Python, and R
- Supports SQL-like operations, streaming, and machine learning
- Runs on YARN, providing access to shared datasets across various HDP applications

44

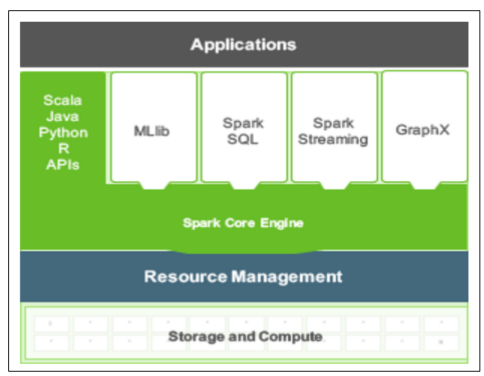
Spark RDDs – Scalability and Performance

- Leverages HDP's horizontal scalability
- Fault-tolerant collection of data elements.
- Enables parallel processing across the cluster



Spark High-Level Tools

- The Spark Core Engine supports four high-level tools to build applications that are part of the Spark project:
  - Spark SQL
  - Spark Streaming
  - MLlib
  - GraphX
- Spark also integrates with other HDP platforms to extend and enhance its capabilities - for example:
  - Hive
  - Zeppelin



Spark and HDP

- HDP 2.4.0 – Spark 1.6.0
- HDP 2.3.4 – Spark 1.5.2
- HDP 2.3.2 – Spark 1.4.1
- HDP 2.2.8 – Spark 1.3.1
- HDP 2.2.4 – Spark 1.2.1

- For this class we will use Spark 1.6 on HDP 2.4.

Objectives

- Zeppelin Overview
- Spark Overview
- Spark REPLs and Application Architecture

REPL Spark Shells

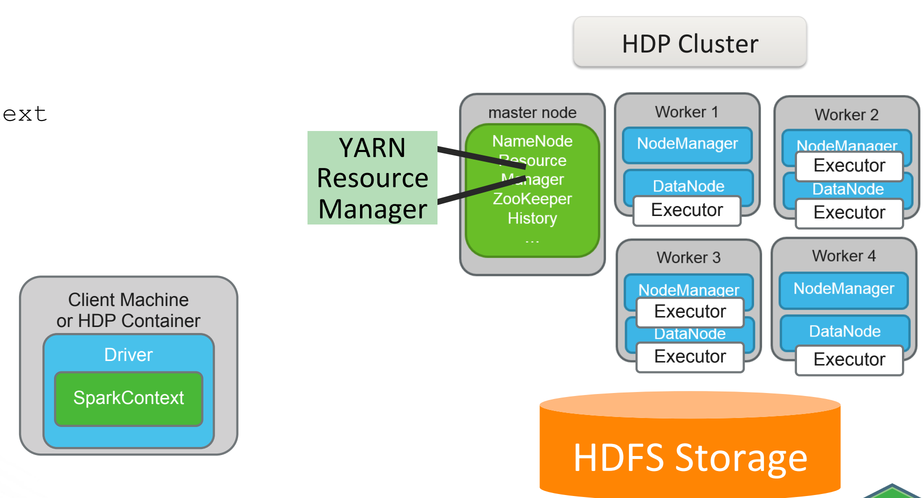
- The Spark Shell provides an interactive way to learn Spark, explore data, and debug applications
- Available for Python and Scala

```
$ pyspark
$ spark-shell
```

- REPL
  - Read Evaluate Print Loop

Enterprise Spark Application Components in HDP

- Driver
- `SparkContext`
- YARN
- HDFS
- Executors

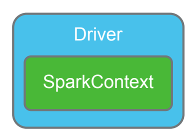


Spark Driver

- Contains the `main()` function
  - Spark REPLs are Spark driver programs
- Creates `SparkContext` and uses it to access Spark
- Manages writing and displaying log files
- Single point of failure when running YARN client (as opposed to cluster) applications

51

`SparkContext`



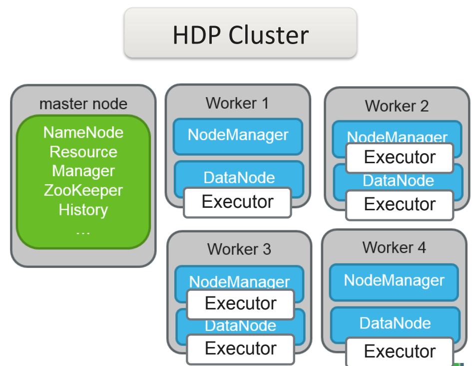
- Manages the connection to Spark
- Contacts YARN ResourceManager to launch Spark executors
- Schedules tasks for Spark executors
- Automatically created as `sc` by a REPL at startup

```
from pyspark import SparkContext, SparkConf
conf = SparkConf()
sc = SparkContext(conf=conf)
```

52

Spark Executors

- Responsible for all application workload processing
  - The "workers" of a Spark application, with `SparkContext` serving as the "master"
- Exist for the life of the application
- Interchangeable workspaces
  - Tasks assigned to a lost executor will be reassigned
  - Data lost will be recomputed on another executor
- Behavior and performance can be controlled programmatically



Lab: Introduction to Spark REPLs and Zeppelin

Knowledge Check

Questions

1. Name the tool in HDP that allows for interactive data analytics, data visualization, and collaboration with Spark.
2. What programming languages does Spark currently support?
3. What is the primary benefit of running Spark on YARN?
4. Name the five components of an enterprise Spark application running in HDP.
5. Which component of a Spark application is responsible for application workload processing?

Summary

Summary

- Zeppelin is a web-based notebook that supports multiple programming languages and allows for data engineering, analytics, visualization, and collaboration using Spark
- Spark is a large-scale, cluster-based, in-memory data processing platform that supports parallelized operations on enterprise-scale datasets
- Spark provides REPLs for rapid, interactive application development and testing
- The five components of an enterprise Spark application running on HDP are:
  - Driver
  - SparkContext
  - YARN
  - HDFS
  - Executors

Working with RDDs

Objectives

- Explain the purpose and function of RDDs
- Explain Spark programming basics
- Define and use basic Spark transformations
- Define and use basic Spark actions
- Invoke functions for multiple RDDs, create named functions, and use numeric operations

Objectives

- Introduction to RDDs

Resilient Distributed Datasets (RDDs)

- Distributed collection of immutable elements (typically stored in-memory)
- Dataset divided into partitions, which allows for parallel operation
  - Node selection for RDD partitions is aligned with HDFS blocks to maximize parallelism and HDP infrastructure benefits
- If individual partition is lost, will be recreated on another node

Create RDDs Programmatically - Simple Lists

- Use `sc.parallelize()` to create an RDD, assigned to a local variable name, composed of lists of numbers and verify with `collect()`

```
rddNumList = sc.parallelize([5, 7, 11, 14])
rddNumList.collect()
[5, 7, 11, 14]

rddTextList = sc.parallelize(["car", "house", "garage"])
rddTextList.collect()
['car', 'house', 'garage']
```


Create RDDs Programmatically - Variables as Input

- RDDs can also take variable values as input:

```
maryFile = ("Mary had a little lamb")
rddMary = sc.parallelize([maryFile])
rddMary.collect()
['Mary had a little lamb']

textList = ["car", "house", "garage"]
rddTextList = sc.parallelize(textList)
rddTextList.collect()
['car', 'house', 'garage']
```

Create Simple RDDs From Text Files

- An RDD can also be created from a text file on local, HDFS, or other locations (such as network or cloud storage) using `sc.textFile()`

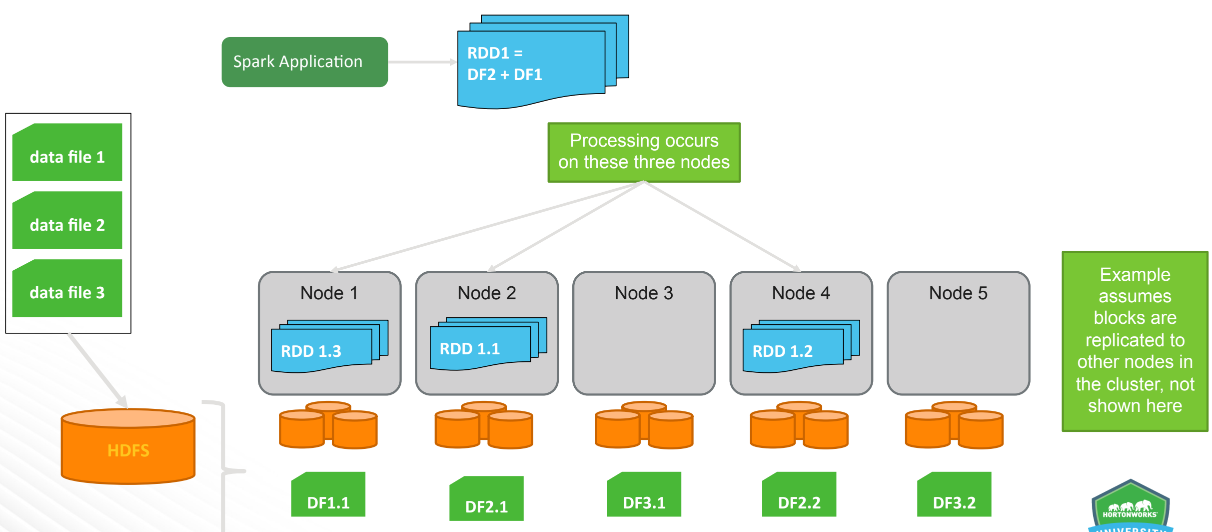
```
rddLocal = sc.textFile("file:/localPathToFile/filename.txt")
rddHDFS = sc.textFile("/HDFSpath/filename.txt")
```

- Multiple files can be combined as part of a single RDD using a comma-separated list or a wildcard character

```
rddComma = sc.textFile("fileLocation/file1.txt,fileLocation/file2.txt")
rddWild = sc.textFile("fileLocation/*.txt")
```

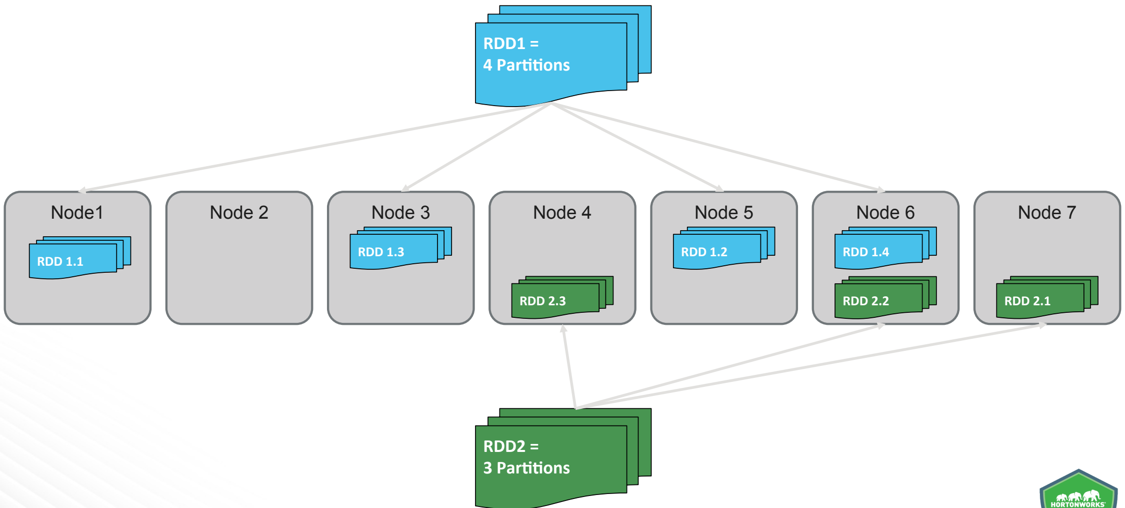
65

From Data Files to HDFS to RDD



66

Multiple RDDs in a Cluster



67

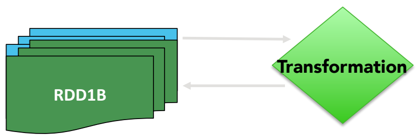
RDD Characteristics

- Can contain any type of ***serializable*** element, meaning those that can be converted to and from a byte stream
  - Examples: int, float, bool, and sequences/iteratives like arrays, lists, tuples, and strings
- Element types can be mixed - for example, an array of strings and int values.
- Non-serializable elements (for example: objects created with certain third-party JAR files or other external resource) cannot be made into RDDs

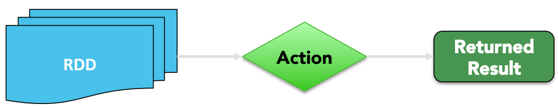
68

RDD Operations

- Two operations can be performed on an RDD
  - Transformations: apply a function and create new RDD partitions based on the output



  - Actions: return a result of a function as output to a screen, file, etc.



69

Objectives

- Introduction to RDDs
- Spark Programming Basics

70

Functional Programming Implications in Spark

- **Immutable data:** RDD1A can be transformed into RDD1B, but an individual element within RDD1A cannot be independently modified
- **No state or side effects:** No interaction with or modification of any values or properties outside of the function
- **Behavioral consistency:** If you pass the same value into a function multiple times, you will always get the same result - changing order of evaluation does not change results
- **Functions as arguments:** function results (including anonymous functions) can be passed as input/arguments to other functions
- **Lazy evaluation:** function arguments are not evaluated / executed until required

Anonymous (a.k.a. Lambda) Functions

- Passed as an argument to another function, called using the `lambda` keyword
- Element variable is defined to the left of a colon, function body defined to the right
  - Example using z as the anonymous function variable and z + 1 as the function body:

```
rddNumList = sc.parallelize([5, 7, 11, 14])

rddAnon = rddNumList.map(lambda z: z + 1)
```

Spark Function (map) — Anonymous Function Call (lambda) — Anonymous Function Body / Definition (z + 1)

```
rddAnon.collect()
[6, 8, 12, 15]
```

Objectives

- Introduction to RDDs
- Spark Programming Basics
- Basic Spark Transformations

map()

- Applies a function supplied as its argument to each element of the RDD

```
rddNumList = sc.parallelize([5, 7, 11, 14])
rddNumList.map(lambda z: z + 1).collect()
[6, 8, 12, 15]
```

Mary had a little lamb Its fleece was white as snow And everywhere that Mary went The lamb was sure to go

Array(Mary, had, a, little, lamb)
Array(Its, fleece, was, white, as, snow)
Array(And, everywhere, that, Mary, went)
Array(The, lamb, was, sure, to, go)

`flatMap()`

- Similar to map(), but after a map function has been performed, takes an additional step and flattens the file

```
rddLineSplit = rddMary.map(lambda line: line.split(" "))
```

Array(Mary, had, a, little, lamb)
Array(Its, fleece, was, white, as, snow)
Array(And, everywhere, that, Mary, went)
Array(The, lamb, was, sure, to, go)

```
rddFlat = rddMary.flatmap(lambda line: line.split(" "))
```

Mary
had
a
little
***
sure
to
go

75

`filter()`

- Keeps elements that meet a defined criteria
  - If the element meets that criteria, it is passed on to the new RDD
  - If not, the element is discarded

```
rddNumList = sc.parallelize([5, 7, 11, 14])
rddNumList.filter(lambda number:  number <= 10).collect()
[5, 7]

months = ["January", "March", "May", "July", "September"]
rddMonths = sc.parallelize(months)
rddMonths.filter(lambda name: len(name) > 5).collect()
['January', 'September']
```

76

distinct()

```
rddBigList = sc.parallelize([5, 7, 11, 14, 2, 4, 5, 14, 21])
rddBigList.collect()
[5, 7, 11, 14, 2, 4, 5, 14, 21]

rddDistinct = rddBigList.distinct()
rddDistinct.collect()
[4, 5, 21, 2, 14, 11, 7]
```


Objectives

- Introduction to RDDs
- Spark Programming Basics
- Basic Spark Transformations
- Basic Spark Actions

`collect()`, `first()`, and `take()`

- `collect()` returns an entire RDD
- `first()` returns only the first element in an RDD
- `take()` returns a specified number of elements in an RDD

```
rddNumList = sc.parallelize([5, 7, 11, 14])

rddNumList.collect()
[5, 7, 11, 14]

rddNumList.first()
5

rddNumList.take(2)
[5, 7]
```


`count()`

- Returns the number of elements in an RDD

```
rddNumList = sc.parallelize([5, 7, 11, 14])

rddNumList.count()
4


rddMary = sc.textFile("mary.txt")

rddMary.count()
4
```

Mary had a little lamb
Its fleece was white as snow
And everywhere that Mary went
The lamb was sure to go

saveAsTextFile()

- Writes the contents of RDD partitions as a set of text files to a specified location (`hdfs://`, `file:/`, etc.) and directory

```
rddBigList.saveAsTextFile("/desiredLocation/foldername")
```

- In this example, can verify the file was successfully written from the command line

```
$ hdfs dfs -ls desiredLocation/foldername
```

```
Found 5 items
-rw-r--r--   3 root hdfs          0 2016-04-25 14:57 numList.txt/_SUCCESS
-rw-r--r--   3 root hdfs          2 2016-04-25 14:57 numList.txt/part-00000
-rw-r--r--   3 root hdfs          2 2016-04-25 14:57 numList.txt/part-00001
-rw-r--r--   3 root hdfs          3 2016-04-25 14:57 numList.txt/part-00002
-rw-r--r--   3 root hdfs          3 2016-04-25 14:57 numList.txt/part-00003
```


Transformations vs. Actions: Lazy Evaluation

- Transformations are lazy - they do not compute until an action is performed

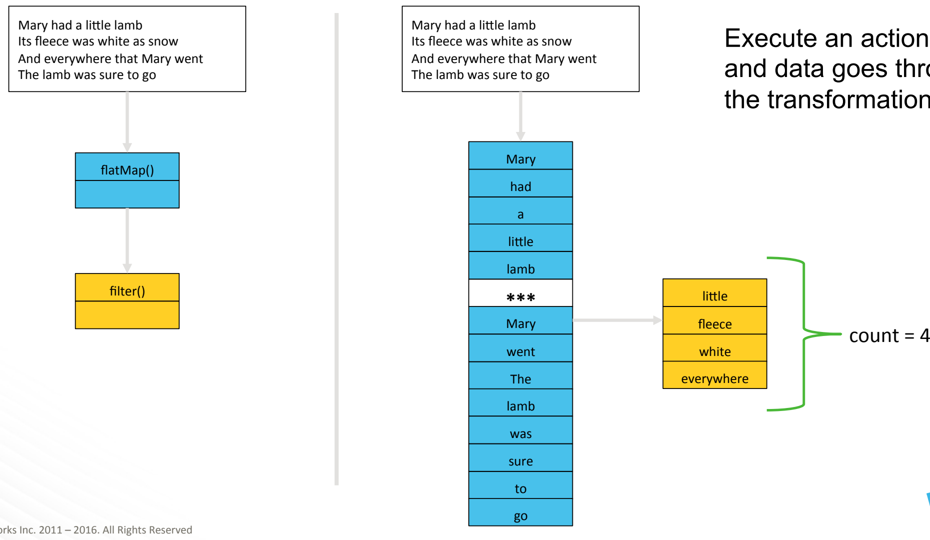
```
rddMary = sc.textFile("mary.txt")
rddFlat = rddMary.flatmap()
rddFilter = rddFlat.filter(lambda words: len(words) > 4)
```

Series of transformations is built and tracked by the Spark driver

```
rddFilter.count()
```

Action triggers execution of the series of transformations

Lazy Evaluation Visualized



Execute an action and data goes through the transformations

83

Objectives

- Introduction to RDDs
- Spark Programming Basics
- Basic Spark Transformations
- Basic Spark Actions
- RDD Special Topics
  - Multiple RDDs
  - Named Functions
  - Numeric Operations

84

Multiple RDDs: `union()` and `intersection()`

```
rddNumList = sc.parallelize([5, 7, 11, 14])
rddNumList2 = sc.parallelize([2, 4, 5, 14, 21])

rddCombined = rddNumList.union(rddNumList2)
rddCombined.collect()
[5, 7, 11, 14, 2, 4, 5, 14, 21]

rddInter = rddNumList.intersection(rddNumList2)
rddInter.collect()
[5, 14]
```


Named Functions

- Functions used multiple times or those that require more than a single line of code should be explicitly defined and named

```
def gradeAorNot(percentage):
    if percentage >  89:
        return "A"
    else:
        return "Not an A"

rddGrades = sc.parallelize([87, 94, 41, 90])
rddGrades.map(gradeAorNot).collect()
['Not an A', 'A', 'Not an A', 'A']
```

Numeric Operations

- Math and statistical functions can be performed on RDDs
  - mean, count, stDev, sum, stats, max, min, etc.

```
rddNumList = sc.parallelize([5, 7, 11, 14])

rddNumList.stats()
(count: 4, mean: 9.25, stdev: 3.49…, max: 14, min: 5)

rddNumList.min()
5
```

87

More Functions: Spark Documentation

- http://spark.apache.org/docs/<version>/api/
- Version options
  - Official version number, such as "1.4.0" or "1.6.1"
  - "latest" for the newest release

Index of /docs/latest/api

Name	Last modified	Size	Description
Parent Directory		-	
R/	2016-03-10 19:28	-	
java/	2016-03-10 19:28	-	
python/	2016-03-10 19:28	-	
scala/	2016-03-10 19:28	-	

88

Lab: Create and Manipulate RDDs

Knowledge Check

Questions

1. What does RDD stand for?
2. What two functions were covered in this lesson that create RDDs?
3. True or False: Transformations apply a function to an RDD, modifying its values
4. What operation does the lambda function perform?
5. Which transformation will take take all of the words in a text object and break each of them down into a separate element in an RDD?
6. True or False: The count action returns the number of lines in a text document, not the number of words it contains.
7. What is it called when transformations are not actually executed until an action is performed?
8. True or False: The distinct function allows you to compare two RDDs and return only those values that exist in both of them
9. True or False: Lazy evaluation makes it possible to run code that "performs" hundreds of transformations without actually executing any of them

91

Summary

Summary

- Resilient Distributed Datasets (RDDs) are *immutable* collection of elements that can be operated on in parallel
- Once an RDD is created, there are two things that can be done to it: transformations and actions
- Spark makes heavy use of functional programming practices, including the use of anonymous functions
- Common transformations include `map()`, `flatmap()`, `filter()`, `distinct()`, `union()`, and `intersection()`
- Common actions include `collect()`, `first()`, `take()`, `count()`, `saveAsTextFile()`, and certain mathematic and statistical functions

Learning Objectives

- Define and create Pair RDDs
- Perform common operations on Pair RDDs

- Pair RDD Introduction

Objectives

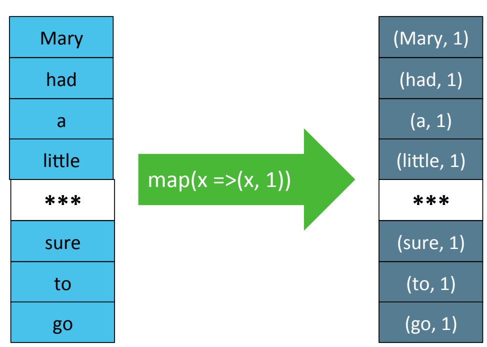
Pair RDD Introduction

- A Pair RDD has elements comprised of a key-value pairs
- Allows for additional key-value based functions and operations
  - Direct RDD interactions that can be used as an alternative to SQL-like APIs

Create Pair RDDs: `map()`

- Pair RDDs can be created from regular RDDs by using the map() transformation:

```
rddMary = sc.textFile("filelocation/mary.txt")
rddFlat = rddMary.flatMap(lambda line: line.split(' '))
kvRdd = rddFlat.map(lambda word: (word,1))
kvRdd.collect()
```



Create Pair RDDs: `keyBy()`

- Creates key-value pairs by applying a function on each data element
  - Function result becomes the key, data element becomes the value in the pair

```
rddTwoNumList = sc.parallelize([(1,2,3),(7,8)])
keyByRdd = rddTwoNumList.keyBy(len)

keyByRdd.collect()
[(3, (1, 2, 3)), (2, (7, 8))]
```


Create Pair RDDs: `zipWithIndex()`

- Creates key-value pairs by making element index (position) the value
  - Element becomes the key

```
rddThreeWords = sc.parallelize(["cat","A","spoon"])
zipWIRdd = rddThreeWords.zipWithIndex()

zipWIRdd.collect()
[('cat', 0), ('A', 1), ('spoon', 2)]
```

Create Pair RDDs: `zip()`

- Creates key-value pairs by taking elements from one RDD as the key and elements of another RDD as the value
  - Syntax: keyRDD.zip(valueRDD)
  - Assumes the two RDDs have the same number of partitions and elements

```
rddThreeWords = sc.parallelize(["cat", "A", "spoon"])
rddThreeNums = sc.parallelize([11, 241, 37])
zipRdd = rddThreeWords.zip(rddThreeNums)


zipRdd.collect()
[('cat', 11), ('A', 241), ('spoon', 37)]
```


- Pair RDD Introduction
- Pair RDD Operations

Objectives

`mapValues()`

- Performs a function on Pair RDD values, leaving keys unchanged

```
zipWIRdd = sc.parallelize([("cat", 0), ("A", 1), ("spoon", 2)])
rddMapVals = zipWIRdd.mapValues(lambda val: val + 1)

rddMapVals.collect()
[('cat', 1), ('A', 2), ('spoon', 3)]
```


`keys(), values(), and sortByKey()`

- `keys()` - returns a list of just the keys

```
rddMapVals.keys().collect()
['cat', 'A', 'spoon']
```

- `values()` - returns a list of just the values

```
rddMapVals.values().collect()
[1, 2, 3]
```

- `sortByKey(ascending=True/False)`
  - "`ascending=False`" sorts from largest to smallest; default is "`ascending=True`"

```
rddMapVals.sortByKey().collect()
[('A', 2), ('cat', 1), ('spoon', 3)]
```


Reorder Key-Value Pairs using map()

- Use pattern matching to reorder placement of key-value pair elements in an RDD

```
zipWIRdd = sc.parallelize([("cat", 0), ("A", 1), ("spoon", 2)])
rddReorder = zipWIRdd.map(lambda (key, value): (value, key))

rddReorder.collect()
[(0, 'cat'), (1, 'A'), (2, 'spoon')]
```

105

lookup(), countByKey(), and collectAsMap()

- `lookup(key)` - returns a list containing all values for a given key

```
keyByRdd.lookup(2)
[(7, 8)]
```

- `countByKey()` - counts the number times a key appears

```
keyByRdd.countByKey()
defaultdict(<type 'int'>,{2: 1, 3: 1})
```

- `collectAsMap()` - collects the result as a map
  - If multiple values exist for the same key only one will be returned

```
keyByRdd.collectAsMap()
{2: (7, 8), 3: (1, 2, 3)}
```

106

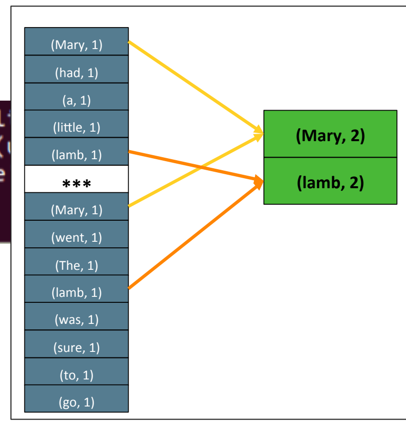
reduceByKey()

- Performs a reduce operation on elements of a Pair RDD and runs a function on any elements that share a key

```
kvReduced = kvRdd.reduceByKey(lambda a,b: a+b)

kvReduced.collect()
```

```
[(u'a', 1), (u'lamb', 2), (u'l
1), (u'the', 1), (u'as', 1), (u
s', 2), (u'white', 1), (u'sure
'to', 1), (u'Mary', 2)]
>>>
```

```
d', 1), (u'fleece',
(u'went', 1), (u'wa
1), (u'its', 1), (u
```



groupByKey()

- Returns an RDD with a grouping of values by key
  - Grouped values are returned as a single iterable object
  - Can be viewed by mapping the elements of the iterable object into a defined list

```
kvGroupByKey = kvRdd.groupByKey().map(lambda x : (x[0], list(x[1])))

kvGroupByKey.collect()
[(u'a', [1]), (u'lamb', [1, 1]), (u'little', [1]),…(u'Mary',[1, 1])]
```

When desired output can be obtained by reduceByKey(), use that instead

subtractByKey()

- Returns key-value pairs containing keys in the source RDD not found in another RDD

```
zipWIRdd = sc.parallelize([("cat", 0), ("A", 1), ("spoon", 2)])
rddSong = sc.parallelize([("cat", 7), ("cradle", 9), ("spoon", 4)])

rddSong.subtractByKey(zipWIRdd).collect()
[('cradle', 9)]
```

('A', 1) is not returned because it does not exist in the source RDD

Pair RDD Joins

- All joins types are supported: inner ("`join`"), full outer, left outer, right outer

```
zipWIRdd = sc.parallelize([("cat", 0), ("A", 1), ("spoon", 2)])
rddSong = sc.parallelize([("cat", 7), ("cradle", 9), ("spoon", 4)]

 rddSong.leftOuterJoin(ZipWIRdd).collect()
 [('spoon', (4, 2)), ('cradle', (9, none)), ('cat', (7, 0))]
```

More Functions: Spark Documentation

- `http://spark.apache.org/docs/<version>/api/`

- Version options
  - Official version number, such as "1.4.0" or "1.6.1"
  - "latest" for the newest release

Index of /docs/latest/api

Name	Last modified	Size	Description
Parent Directory		-	
R/	2016-03-10 19:28	-	
java/	2016-03-10 19:28	-	
python/	2016-03-10 19:28	-	
scala/	2016-03-10 19:28	-	

Lab: Create and Manipulate Pair RDDs, Advanced RDD Programming

Knowledge Check

Questions

1. An RDD that contains elements made up of key-value pairs is sometimes referred to as a _________________.
2. Name two functions that can be used to create a Pair RDD.
3. True or False: A key can have a value that is actually a list of many values.
4. Since `sortByKey()` only sorts by key, and there is no equivalent function to sort by values, how could you go about getting your Pair RDD sorted alphanumerically by value?
5. You determine either `reduceByKey()` or `groupByKey()` could be used in your program to get the same results. Which one should you choose?
6. How can you use `subtractByKey()` to determine *all* of the unique keys across two RDDs?

114

Summary

Summary

- Pair RDDs contain elements made up of key-value pairs
- Common functions used to create Pair RDDs include `map()`, `keyBy()`, `zipWithIndex()`, and `zip()`
- Common functions used with Pair RDDs include `mapValues()`, `keys()`, `values()`, `sortByKey()`, `lookup()`, `countByKey()`, `collectAsMap()`, `reduceByKey()`, `groupByKey()`, `flatMapValues()`, `subtractByKey()`, and various join types.

Objectives

After completing this lesson, students should be able to:

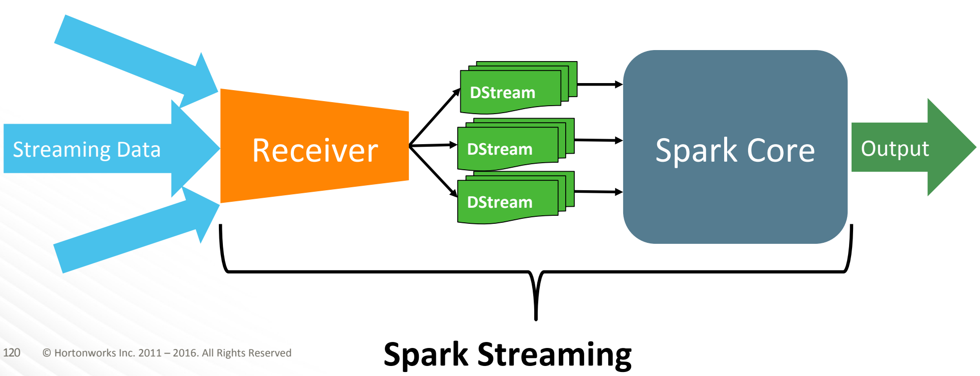
- Describe Spark Streaming
- Create and view basic data streams
- Perform basic transformations on streaming data
- Utilize window transformations on streaming data

Objectives

- Spark Streaming Overview

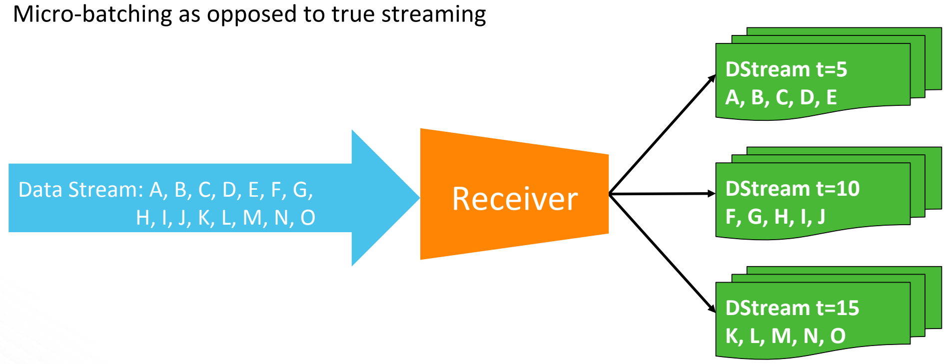
What is Spark Streaming?

- Implements a receiver and specialized RDDs called DStreams on top of Spark
- Enables micro-batch processing of live streaming data
- Allows for additional ROI on Spark platform investment



DStreams

- Batches of input data created at regular time intervals
  - Micro-batching as opposed to true streaming

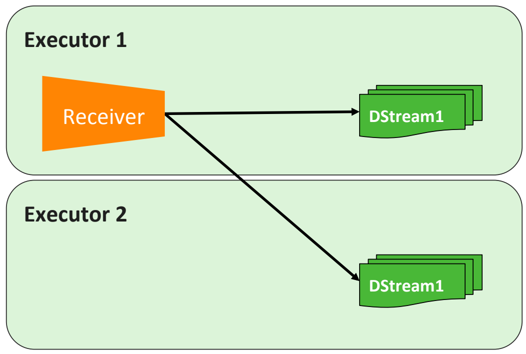


DStream vs. RDD

- DStreams contain data and physically exist in memory from moment of creation
  - Normal RDDs are just sets of instructions until an action is performed
- By default, DStreams are deleted after processing
- Outputs vs. Actions

DStream Replication

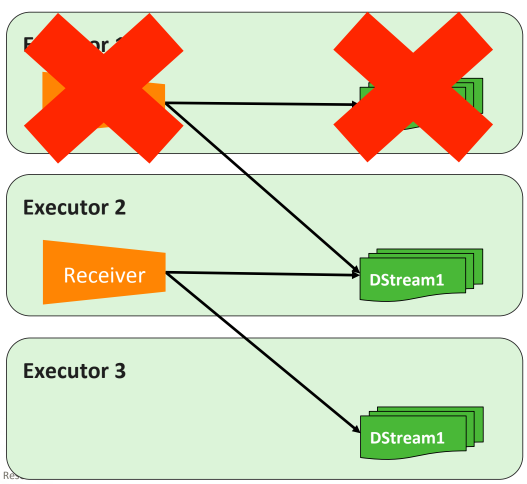
- Receiver duplicates data to two executors by default



123

Receiver Availability

- If an executor with a receiver goes down, it will be restarted in another executor



124

Receiver Reliability

- By default, receivers are "unreliable"
  - No acknowledgment between receiver and source
  - No record of whether data has been successfully written
  - No ability to ask for retransmission for missed data
  - Possibility for data loss if receiver is lost
- To implement a reliable receiver, a custom receiver must be created
  - Scala / Java only as of Spark 1.6.0
  - Not supported by Python APIs

125

Streaming Data Source Examples

- Basic Sources
  - Text files from an HDFS directory
  - Text via TCP socket connection
  - Queue of RDDs (for testing purposes)
- Advanced Sources
  - Kafka
  - Kinesis
  - Flume
  - MQTT

**Additional basic and advanced sources are available in Scala / Java*

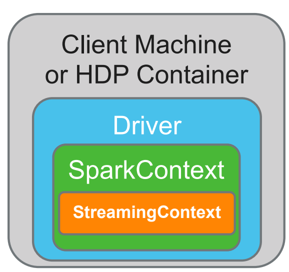
126

- Spark Streaming Overview
- Basic Streaming

Objectives

`StreamingContext`

- An extension of the `SparkContext`
- Entry point for streaming applications
- Sets up receiver and enables real-time transformations on Dstreams, as well as various output types



Modify REPL CPU Cores

- Streaming requires having two or more CPU cores available
  - One core for the receiver plus one core for each DStream being ingested
- This can be changed by modifying the MASTER environment variable when launching the REPL
  - To utilize two cores: `pyspark --master local[2]`

Launch `StreamingContext`

- Import the `StreamingContext` API
  - Example: `from pyspark.streaming import StreamingContext`
- Create an instance of the `StreamingContext` and supply the name of the `SparkContext` (when using the REPL, `sc`) and an interval time for micro-batching
  - Example setting a one-second interval: `ssc = StreamingContext(sc, 1)`
- Spark `StreamingContext` instances can be defined with varying time intervals based on needs
  - Only one `StreamingContext` is allowed per JVM

```
sscTen = StreamingContext(sc, 10)
```


Stream from HDFS Directories and TCP Sockets

- To create a stream by monitoring an HDFS directory and ingesting any new files:

```
hdfsInputDS = ssc.textFileStream("someHDFSdirectory")
```

- To create a stream by monitoring TCP socket source (hostname and port):

```
tcpInputDS = ssc.socketTextStream("someHostname", portNumber)
```


Output to Console and to HDFS

- Print output to the console:
  – Python: `DSvariableName.pprint()`
  – Scala/Java: `DSvariableName.print()`

- Suggestion: set sc log level to "ERROR" when printing to console to reduce screen clutter
  – Example: `sc.setLogLevel("ERROR")`

- Save output as a time-stamped text file on HDFS:
  – `DSVariable.saveAsTextFiles("HDFSlocation/prefix", "optionalSuffix")`
  – Directory permissions must be set accordingly

- Can use the same DStream to output to both console and HDFS text file

Start and Stop the Streaming Application

- All operations must be defined before the stream is started
- When ready: `ssc.start()`
- When finished: `ssc.stop()`

Simple Streaming Program Example Using a REPL

```
# pyspark --master local[2]
>>> sc.setLogLevel("ERROR")
>>> from pyspark.streaming import StreamingContext
>>> sscFive = StreamingContext(sc, 5)
>>> hdfsInputDS = sscFive.textFileStream("/user/root/test/")
>>> hdfsInputDS.saveAsTextFiles("/user/root/test/stream/name")
>>> hdfsInputDS.pprint()
>>> sscFive.start()
```


Lab: Basic Spark Streaming using HDFS Directories and TCP Sockets

Objectives

- Spark Streaming Overview
- Basic Streaming
- Basic Streaming Transformations

DStream Transformations

- Allow modification of DStream data similar to RDD transformations

- Familiar functions
  - `map()`
  - `flatMap()`
  - `filter()`
  - `repartition()`
  - `union()`
  - `count()`
  - `reduceByKey()`
  - `join()`
  - Etc.

Transformation using `flatMap()`

```
...

hdfsInputDS = ssc.textFileStream("someHDFSdirectory")

flatMapDS = hdfsInputDS.flatMap(lambda line: line.split(" "))

flatMapDS.pprint()

ssc.start()
```

Combine DStreams using `union()`

- A simple example that creates two DStreams from the same source and combines them

```
. . .
input1 = ssc.textFileStream("/user/root/test/")
input2 = ssc.textFileStream("/user/root/test/")
combined = input1.union(input2)
combined.pprint()
ssc.start()
```

139

Create Key-Value Pairs

```
. . .
hdfsInputDS = ssc.textFileStream("someHDFSdirectory")
kvPairDS = hdfsInputDS.flatMap(lambda line: line.split(" ").map(lambda word: (word, 1))
kvPairDS.pprint()
ssc.start()
```

140

reduceByKey()

```
. . .
hdfsInputDS = ssc.textFileStream("someHDFSdirectory")
kvPairDS = hdfsInputDS.flatMap(lambda line: line.split(" ").map(lambda word: (word, 1))
kvReduced = kvPairDS.reduceByKey(lambda a,b: a+b)
kvReduced.pprint()
ssc.start()
```


Lab: Basic Spark Streaming Transformations

Objectives

- Spark Streaming Overview
- Basic Streaming
- Streaming Transformations
- Window Transformations

Stateful vs. Stateless Operations

- By default, DStreams are discarded from memory when the next batch of data arrives
  - Assumes all operations performed are on single DStreams, not dependent on previous data
  - This is referred to as working with "stateless" operations / transformations
- However, it is sometimes beneficial to perform transformations and gather output using overlapping time slices, or across an entire collected dataset
  - Example: Every 15 seconds perform operations over the last 45 seconds worth of data
  - This is referred to as working with "stateful" operations / transformations

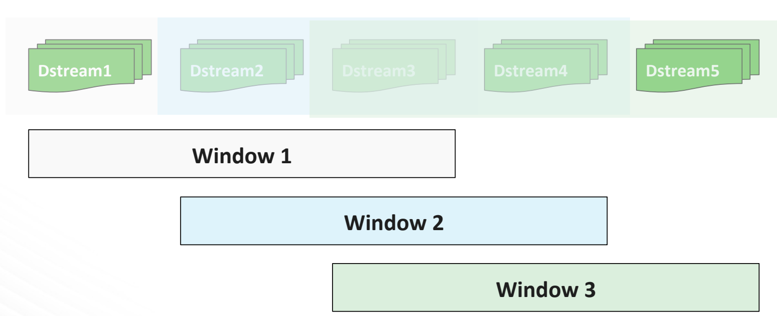
Checkpointing

- Used in stateful streaming operations to maintain state in the event of system failure
- To enable:

```
. . .

ssc.checkpoint("someHDFSdirectory")

. . .
```


Streaming Window Functions

- Window functions perform combined operations on a set of Dstreams
- The window length (size, in seconds) and interval (how often it is collected) are set during creation
  – These values must be a multiple of the `StreamingContext` interval value



Basic Window Transformations

- `window(windowLength, interval)` - returns a new DStream which is computed based on the length and interval provided
  - Functionally similar to a `union()` transformation
  - Example: `windowDS = streamingDS.window(30,10)`
- `countByWindow(windowLength, interval)` - returns a count of the number of elements in the stream
  - Example: `windowCountDS = streamingDS.countByWindow(30,10)`
  - Equivalent output to `streamingDS.window(30, 10).count()`, but more efficient if number of elements is large

147

Sample Window Application

```
# pyspark --master local[2]
>>> sc.setLogLevel("ERROR")
>>> from pyspark.streaming import StreamingContext
>>> ssc = StreamingContext(sc, 1)
>>> ssc.checkpoint("/user/root/test/checkpoint/")
>>> tcpInputDS = ssc.socketTextStream("sandbox",9999)
>>> windowDS = tcpInputDS.window(15, 5).
flatMap(lambda line: line.split(" ")).count()
>>> windowDS.pprint()
>>> ssc.start()
```

148

reduceByKeyAndWindow()

```
. . .
tcpInDS = ssc.socketTextStream("sandbox",9999)

redPrWinDS = tcpInDS.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).
reduceByKeyAndWindow(lambda a,b: a+b, lambda a,b: a-b, 10, 2)

redPrWinDS.pprint()

ssc.start()
```


Lab: Spark Streaming Window Transformations

Knowledge Check

Questions

1. Name the two new components added to Spark Core to create Spark Streaming.
2. If an application will ingest three streams of data, how many CPU cores should it be allocated?
3. Name the three basic streaming input types supported by both Python and Scala APIs.
4. What two arguments does an instance of `StreamingContext` require?
5. What is the additional prerequisite for any stateful operation?
6. What two parameters are required to create a window?

Summary

Summary

- Spark Streaming is an extension of Spark Core that adds the concept of a streaming data receiver and a specialized type of RDD called a DStream.
- DStreams are fault tolerant, whereas receivers are highly available.
- Spark Streaming utilizes a micro-batch architecture.
- Spark Streaming layers in a `StreamingContext` on top of the Spark Core `SparkContext`.
- Many DStream transformations are similar to traditional RDD transformations
- Window functions allow operations across multiple time slices of the same DStream, and are thus stateful and require checkpointing to be enabled.

Spark SQL

Objectives

After completing this lesson, students should be able to:

- Name the various components of Spark SQL and explain their purpose
- Describe the relationship between DataFrames, tables, and contexts
- Use various methods to create and save DataFrames and tables
- Manipulate DataFrames and tables

Objectives

- Spark SQL Components Overview

Spark SQL

- A Spark module for processing ***structured*** data
- Automated performance improvements compared to Spark Core API programs
- Allows leveraging of investments in Hive data and knowledge-building while taking advantage of Spark's in-memory processing capabilities

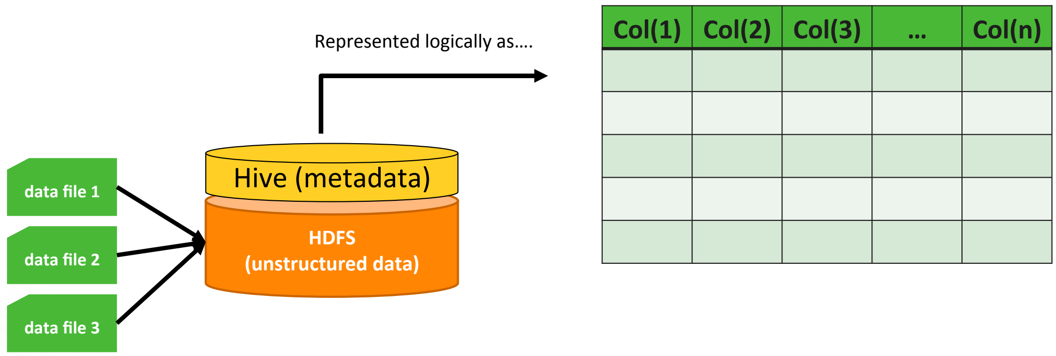
DataFrames

- Data organized into one or more columns, similar to a table
  – Underlying constructs = RDD
- Can be created from RDDs, Hive tables, and outside data sources
- Can be used to create SQL tables
- Three primary methods available to interact with DataFrames and tables
  – DataFrames API available for Java, Scala, Python, and R
  – Native Spark SQL (subset of SQL92)
  – HiveQL (with just a few exceptions)

Hive

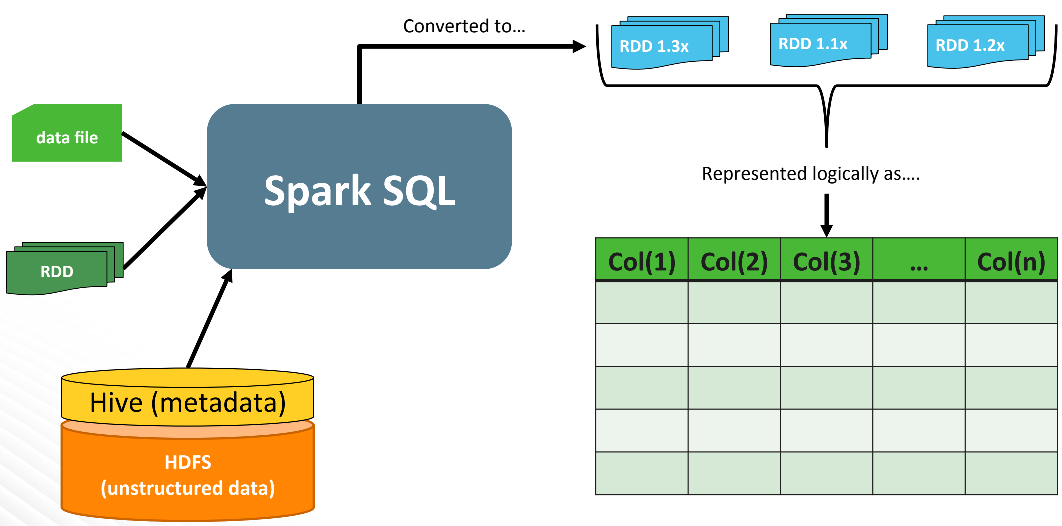
- Original data warehouse platform in Hadoop
  – Interacts with data using a SQL-like query language, HiveQL
- Represents unstructured data in HDFS as tables using a metadata overlay
- Ubiquitous
  – Every Hadoop distribution includes it
  – **Massive amounts of existing data managed by Hive**

Hive Data Visually



161

The DataFrame Visually



162

Spark SQL Contexts

- Two options:

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

or

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

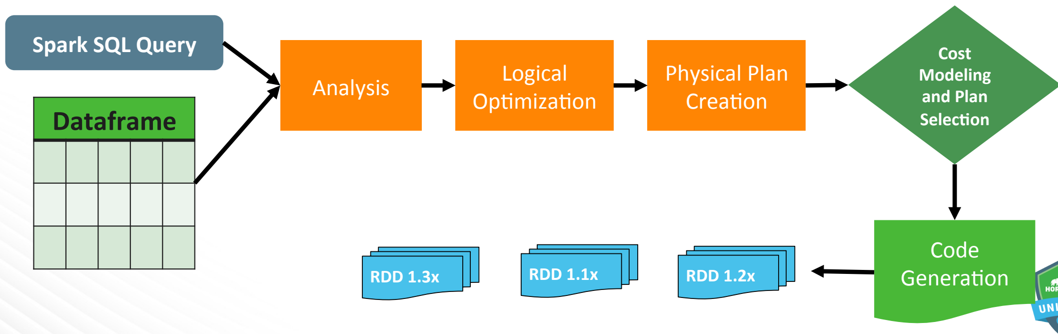
- Zeppelin uses `HiveContext` named `sqlContext` when running `%sql` code
- REPL also creates a `HiveContext` named `sqlContext` at launch

`SQLContext` vs. `HiveContext`

- `SQLContext`
  - Provides a generic SQL parser
- `HiveContext`
  - Superset of (extends) `SQLContext`
  - Enables numerous additional operations using the HiveQL parser
  - Allows ability to read data directly from and write back to Hive tables
  - Provides access to Hive User Defined Functions (UDFs)
- Which to use?
  - `SQLContext` has fewer dependencies and uses less resources if the limited API meets your needs
  - `HiveContext` allows greater flexibility and capabilities
  - When in doubt, use `HiveContext`

Catalyst, the Spark SQL Optimizer

- Accelerates query performance via:
  - Built-in catalog of optimizations
  - Intelligent, cost-based plan selection and execution
- Simpler to write a SQL statement than a series of `filter()`, `group()`, etc. calls
- Performance matches or outperforms equivalent core RDD programs



Objectives

- Spark SQL Components Overview
- Create and Save DataFrames and Tables

DataFrames and Tables - Row Objects

- Code looks different in Scala but the same basic flow applies - create formatted RDD, convert to DataFrame, register as a temporary table, and save to Hive

```
import org.apache.spark.sql.hive._
val sqlContext2 = new HiveContext(sc)
import sqlContext2.implicits._
case class DataSample(code: String, value: Long)
val df2 = sc.parallelize(Seq(DataSample("CC", 110000), DataSample("DD", 90000))).toDF()
df2.registerTempTable("TEST5")
sqlContext2.sql("CREATE TABLE permcd AS SELECT * FROM test5")
sqlContext2.sql("SHOW TABLES").show()

import org.apache.spark.sql.hive._
sqlContext2: org.apache.spark.sql.hive.HiveContext = org.apache.spark.sql.hive.HiveContext
import sqlContext2.implicits._
defined class DataSample
df2: org.apache.spark.sql.DataFrame = [code: string, value: bigint]
res74: org.apache.spark.sql.DataFrame = []
+------------+-----------+
|   tableName|isTemporary|
+------------+-----------+
|       test5|       true|
|      permab|      false|
|      permcd|      false|
|permenriched|      false|
```


Convert an RDD to a DataFrame

- An RDD with elements that adhere to a properly defined schema can be converted to a DataFrame using one of the following methods:

```
toDF():  dataframeX = rddName.toDF()

createDataFrame():
dataframeX = sqlContext.createDataFrame("rddName")
```

- In Python, if an RDD is properly formatted but lacks a schema, `createDataFrame()` can be used to infer the schema on DataFrame creation

```
rddName = sc.parallelize([('AA', 150000), ('BB', 80000)])
dataframeX = sqlContext.createDataFrame(rddName, ['code', 'value'])
```

Create DataFrames From Text Programmatically

```
from pyspark.sql import SQLContext, Row
sqlContext = SQLContext(sc)
##Want to create a DataFrame of People
##Attributes will be Name, Age

lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

##Create the DataFrame
peopleDF=sqlContext.createDataFrame(people)
```

169

Creating a DataFrame from a table in Hive

- Load the entire table

```
df = sqlContext.table("patients")
```

- Load using a SQL Query

```
sqlContext.sql("Use people")
df1 = sqlContext.sql("SELECT * from patients WHERE age>50")
df2 = sqlContext.sql("""
 SELECT col1 as timestamp, SUBSTR(date,1,4) as year, event
 FROM events
 WHERE year > 2014""")
```

170 170

Register DataFrames as Temporary Tables

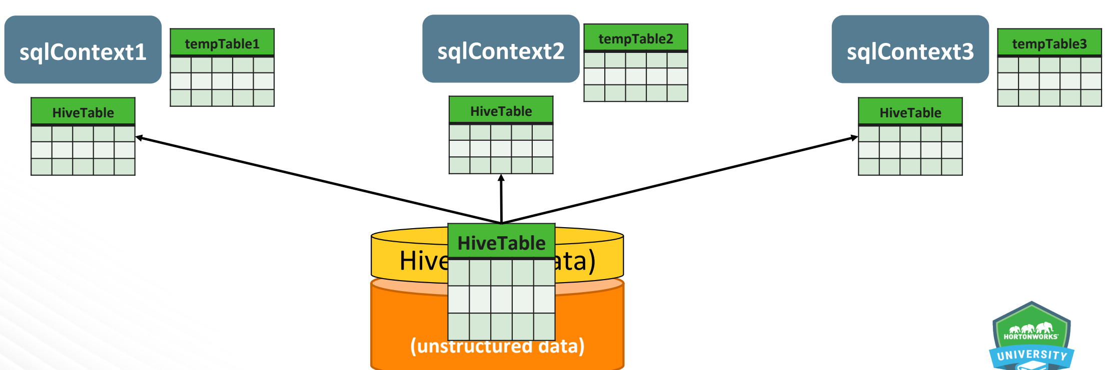
- Use `registerTempTable()` to make the DataFrame available to SQL within the current context

```
dataframe1.registerTempTable("table1")
```

```
%pyspark
dataframe1.registerTempTable("table1")
sqlContext.sql("SELECT * FROM table1").show()

+----+------+
|code| value|
+----+------+
|  AA|150000|
|  BB| 80000|
+----+------+
```


DataFrames and Tables - Summary

- Registering a temporary table makes that table available for either DataFrames API or SQL interactions while operating in that specific context, but storing tables in Hive (and using `HiveContext`) makes them available across contexts



sqlContext.sql() and show()

- The DataFrames API enables a user to run native SQL commands using the `sql()` function prepended by the name of the context (default is usually `sqlContext`)
- When displaying DataFrame contents or the output from a SQL command run from the DataFrames API, append `show()` to display the contents on-screen

```
sqlContext.sql("SELECT * FROM permcd").show()
```

```
%pyspark
sqlContext.sql("SELECT * FROM permcd").show()

+----+------+
|code| value|
+----+------+
|  CC|110000|
|  DD| 90000|
+----+------+
```


Saving Dataframe to Hive Table

- Use the HiveQL `CREATE TABLE` function to make a copy of a DataFrame as a permanent Hive table

```
sqlContext.sql("CREATE TABLE table1hive AS SELECT * FROM table1")
```

```
%pyspark
sqlContext.sql("CREATE TABLE table1hive AS SELECT * FROM table1")
sqlContext.sql("SHOW TABLES").show()

+------------+-----------+
|   tableName|isTemporary|
+------------+-----------+
|      table1|       true|
|       test4|       true|
|      permab|      false|
|      permcd|      false|
|permenriched|      false|
|  table1hive|      false|
```

DataFrameReader / DataFrameWriter API

- `DataFrameReader`
  - Interface used to load a DataFrame from external storage
  - `format(str)` – supports "`orc`", "`parquet`", "`json`", etc
  - load(path-to-file)
- `DataFrameWriter`
  - Interface used to store a DataFrame to external storage
  - `format(str)` – supports "`orc`", "`parquet`", "`json`", etc
  - `mode(str)` - what to do when file exists: "`append`", "`ignore`", "`overwrite`", "`error`"
  - `save(path-to-file)`

Create DataFrames from Files using `read()`

- DataFrames can be created easily from certain structured file types, including ORC, parquet, and if properly formatted, JSON (as well as others)

```
dataframeJSON = sqlContext.read.format("json").load("dfsamp.json")
```

Or, if reading from a folder of part-* files created using `write()`:

```
dataframeJSON = sqlContext.read.format("json").load("folderName/*")
```

```
%pyspark
dataframeJSON = sqlContext.read.format("json").load("dfsamp.json")
dataframeJSON.show()

+----+------+
|code| value|
+----+------+
|  AA|150000|
|  BB| 80000|
+----+------+
```

Save DataFrames as Files Using `write()`

- DataFrames can be saved to HDFS as files of many commonly used file formats, including ORC, JSON, and parquet.

```
dataframe1.write.format("json").save("dfjson")
dataframe1.write.format("orc").save("dforc")
dataframe1.write.format("parquet").save("dfparquet")
```

```
%pyspark
dataframe1.write.format("json").save("dfjson")
dataframe1.write.format("orc").save("dforc")
dataframe1.write.format("parquet").save("dfparquet")
```


Save Modes

- Save modes control behavior during save operations
  - `ErrorIfExists`: Default mode, returns an error if the data already exists
  - `Append`: Appends data to file or table if it already exists
  - `Overwrite`: Replaces existing data if it already exists
  - `Ignore`: Does nothing if the data already exists

```
dataframe1.write.format("orc").save("dforc", mode="overwrite")
```

```
%pyspark
dataframe1.write.format("orc").save("dforc", mode="overwrite")
```

Lab: Create and Save DataFrames

Objectives

- Spark SQL Components Overview
- Create and Save DataFrames and Tables
- Manipulate DataFrames and Tables

Working with Dataframes and sql()

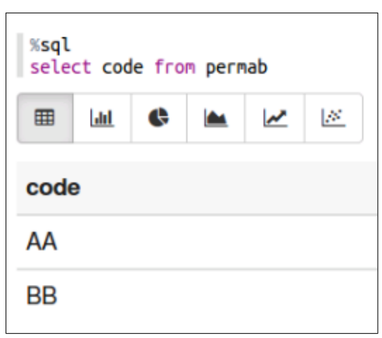
- SQL can be run against dataframes with just small modification

```
val df = sqlContext.table("myHiveTable")
df.registerTempTable("t1")

val df2 = sqlContext.sql("SELECT A, B, C from t1")
df2.registerTempTable("t2")

val df3 = sqlContext.sql("… from t2")
df3.registerTempTable("t3")
```


Zeppelin and the `%sql` binding

- In Zeppelin we have a shortcut for
  `sqlContext.sql()`

- In Zeppelin, we can use the %SQL on tables registered to the SQL Context (temp and hive tables).



Example DataFrames

For the next few slides, let's create two data frames:

```
df1 = sc.parallelize(
   [Row(cid='101', name='Alice', age=25, state='ca'), \
   Row(cid='102', name='Bob', age=15, state='ny'), \
   Row(cid='103', name='Bob', age=23, state='nc'), \
   Row(cid='104', name='Ram', age=45, state='fl')]).toDF()


df2 = sc.parallelize(
   [Row(cid='101', date='2015-03-12', product='toaster', price=200), \
   Row(cid='104', date='2015-04-12', product='iron', price=120), \
   Row(cid='102', date='2014-12-31', product='fridge', price=850), \
   Row(cid='102', date='2015-02-03', product='cup', price=5)]).toDF()
```


DataFrame Operations: Inspecting Content (1 of 2)

- `first()` – return the first row
- `take(n)` – return n rows

```
df1.first()
Row(age=23, cid=u'104', name=u'Bob', state=u'nc')

df1.take(2)
[Row(age=45, cid=u'104', name=u'Ram', state=u'fl')
Row(age=15, cid=u'102', name=u'Bob', state=u'ny')]
```


DataFrame Operations: Inspecting Content (2 of 2)

- `limit(n)`: reduce the DataFrame to n rows
  - Result is still a DataFrame, not a Python result list
- `show(n)`: prints the first n rows to the console

```
df1.show(3)

+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 25|101|Alice|   ca|
| 15|102|  Bob|   ny|
| 23|103|  Bob|   nc|
+---+---+-----+-----+
```

```
df1.limit(2).show()

+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 15|102|  Bob|   ny|
| 45|101|Alice|   ca|
+---+---+-----+-----+
```

185

DataFrame Operations: Inspecting Schema

```
df1.columns  #Display column names
[u'age', u'cid', u'name', u'state']

df1.dtypes  #Display column names and types
[('age', 'bigint'), ('cid', 'string'), ('name', 'string'), ('state',
'string')]

df1.schema  #Display detailed schema
StructType(List(StructField(age,LongType,true),
StructField(cid,StringType,true),
StructField(name,StringType,true),
StructField(state,StringType,true)))
```

186

DataFrame Operations: Counting Rows

- Count all the rows in a DataFrame

```
df1.count()
4
```


DataFrame Operations: Summary Statistics

```
df1.describe().show()

+-------+------------------+
|summary|               age|
+-------+------------------+
| count |                 4|
| mean  |              27.0|
| stddev|11.045361017187261|
| min   |                15|
| max   |                45|
+-------+------------------+
```

`Describe()` shows statistics for all numeric columns, ignoring others

DataFrame Operations: Removing Duplicates

- Remove duplicate rows

```
df1.distinct().show()

+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 23|103|  Bob|   nc|
| 15|102|  Bob|   ny|
| 45|104|  Ram|   fl|
| 25|101|Alice|   ca|
+---+---+-----+-----+
```


DataFrame Operations: Removing Rows by Key

- Removing duplicate rows by key, drops every row with the same key but the first occurrence

```
df1.drop_duplicates(["name"]).show()

+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 15|102|  Bob|   ny|
| 45|104|  Ram|   fl|
| 25|101|Alice|   ca|
+---+---+-----+-----+
```

DataFrame Operations: Sorting Rows

```
df1.sort(df1["age"].desc()).
show()
```

```
+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 45|104|  Ram|   fl|
| 25|101|Alice|   ca|
| 23|103|  Bob|   nc|
| 15|102|  Bob|   ny|
+---+---+-----+-----+
```

```
df1.sort("age",
ascending=True).show()
```

```
+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 15|102|  Bob|   ny|
| 23|103|  Bob|   nc|
| 25|101|Alice|   ca|
| 45|104|  Ram|   fl|
+---+---+-----+-----+
```


DataFrame Operations: Adding a Column

```
df1.withColumn("age-dog-years", df1["age"]*7).show()
```

```
+---+---+-----+-----+-------------+
|age|cid| name|state|age-dog-years|
+---+---+-----+-----+-------------+
| 25|101|Alice|   ca|          175|
| 15|102|  Bob|   ny|          105|
| 23|103|  Bob|   nc|          161|
| 45|104|  Ram|   fl|          315|
+---+---+-----+-----+-------------+
```


DataFrame Operations: Renaming a Column

```
df1.withColumnRenamed("age", "age2").show()
```

```
+----+---+-----+-----+
|age2|cid| name|state|
+----+---+-----+-----+
|  25|101|Alice|   ca|
|  15|102|  Bob|   ny|
|  23|103|  Bob|   nc|
|  45|104|  Ram|   fl|
+----+---+-----+-----+
```


DataFrame Operations: `select()` Operator

- `select(*cols)`
  - `Cols`: list of column names (strings) or list of "Column" expressions

```
df1.select("name", "age").show()

+-----+---+
| name|age|
+-----+---+
|Alice| 25|
|  Bob| 15|
|  Bob| 23|
|  Ram| 45|
+-----+---+
```

```
df1.select(df1["name"],
df1["age"]*7).show()

+-----+---------+
| name|(age * 7)|
+-----+---------+
|Alice|      175|
|  Bob|      105|
|  Bob|      161|
|  Ram|      315|
+-----+---------+
```


DataFrame Operations: `selectExpr()` Operator

- `selectExpr(*expr)` – Selects a set of SQL expressions.

```
df.selectExpr("colA","colB as newName","abs(colC)")
```

```
df1.selectExpr("substr(name,1,3)", "age*7").show()

+-----------------+---------+
|SUBSTR(name, 1, 3)|(age * 7)|
+-----------------+---------+
|              Ali|      175|
|              Bob|      105|
|              Bob|      161|
|              Ram|      315|
+-----------------+---------+
```


Column Expression

- Column objects can be created from a DataFrame

Select a column: `df1["age"]`
OR
Expression: `df1.age * 2 - 15`

- Operations on Column objects:

Cast to type: `df1["age"].cast("string")`

Rename a column: `df1["age"].alias("age2")`

Sort a column: `df1["age"].asc() or df["age"].desc()`

Substring: `df1["name"].substr(1,3)`

Between: `df1["age"].between(25, 34)`

DataFrame Operations: Dropping Columns

```
df1.drop("age").show()
```

```
+---+-----+-----+
|cid| name|state|
+---+-----+-----+
|101|Alice|   ca|
|102|  Bob|   ny|
|103|  Bob|   nc|
|104|  Ram|   fl|
+---+-----+-----+
```


Data Frame Operations: Filtering Rows

```
df1.filter(df1.age>21).show()
```

OR

```
df1.filter(df1["age"]>21).show()
```

```
+---+---+-----+-----+
|age|cid| name|state|
+---+---+-----+-----+
| 25|101|Alice|   ca|
| 23|103|  Bob|   nc|
| 45|104|  Ram|   fl|
+---+---+-----+-----+
```

Data Frame Operations: groupBy()

```
df1.groupBy("name").count().show()
```

```
+-----+-----+
| name|count|
+-----+-----+
|  Ram|    1|
|Alice|    1|
|  Bob|    2|
+-----+-----+
```


Data Frame Operations: groupBy() and sum()

```
df2.select(df2["date"].substr(1,4).alias("year"),
df2["price"]).groupBy("year").sum().show()
```

```
+----+----------+
|year|SUM(price)|
+----+----------+
|2014|       850|
|2015|       325|
+----+----------+
```


Data Frame Operations: `groupBy()` and `agg()`

- `agg(*exprs)` is a generic function for implementing aggregations after `groupBy`
- `Exprs`: a `dict` mapping column names to aggregate function (`min`, `max`, `count`, `avg`, `sum`)

```
df2.select(df2["date"].substr(1,4).alias("year"),
df2["price"])\
   .groupBy("year"). \
   agg({"price": "avg", "year": "count"}).show()

+----+------------------+-----------+
|year|        AVG(price)|COUNT(year)|
+----+------------------+-----------+
|2014|             850.0|          1|
|2015|108.33333333333333|          3|
+----+------------------+-----------+
```

201

Inner Join with Data Frames

```
df1.join(df2, df1["cid"]==df2["cid"], "inner").show()

+---+---+-----+-----+---+----------+-----+-------+
|age|cid| name|state|cid|      date|price|product|
+---+---+-----+-----+---+----------+-----+-------+
| 25|101|Alice|   ca|101|2015-03-12|  200|toaster|
| 15|102|  Bob|   ny|102|2014-12-31|  850| fridge|
| 15|102|  Bob|   ny|102|2015-02-03|    5|    cup|
| 45|104|  Ram|   fl|104|2015-04-12|  120|   iron|
+---+---+-----+-----+---+----------+-----+-------+
```

202

More About `join()`

- `df1.join(df2, joinExpr, joinType)`
- `joinType` is one of: `inner`, `outer`, `left_outer`, `right_outer` and `semijoin`
- `joinExpr` can be written in two ways
  - `df1.join(df2, "cid", "inner")`
  - `df1.join(df2, df1["cid"]==df2["cid"], "inner")`

Multiple Conditions in `joinExpr`

Notice the special way to join with multiple conditions:

```
df1.join(df2, (df1["cid"]==df2["cid"] & (df2["price"] > 200),
"inner").show()

+---+---+----+-----+---+----------+-----+-------+
|age|cid|name|state|cid|      date|price|product|
+---+---+----+-----+---+----------+-----+-------+
| 15|102| Bob|   ny|102|2014-12-31| 850 | fridge|
+---+---+----+-----+---+----------+-----+-------+
```


User Defined Functions (UDFs)

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

get_year = udf(lambda x: int(x[:4]), IntegerType())

df2.select(get_year(df2["date"]).alias("year"),
df2["product"]).collect()

+----+-------+
|year|product|
+----+-------+
|2015|toaster|
|2015|   iron|
|2014| fridge|
|2015|    cup|
+----+-------+
```


UDFS with Multiple Parameters

```
from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType

calc_mins = udf(lambda h,m: int(h*60+m), IntegerType())

df2.select(calc_mins(df2["hour"],
df2["mins"]).alias("my_mins"))
```


Using UDFs in SQL Statements

```
df2.registerTempTable("my_df")
hc.registerFunction("get_year", lambda x: int(x[:4]))
hc.sql("select get_year(date) as year FROM my_df").show()
```

```
+----+
|year|
+----+
|2015|
|2015|
|2014|
|2015|
+----+
```

207

`explain()`

- The `explain()` command describes Spark-SQL execution plan

```
df1.join(df2, (df1["cid"]==df2["cid"]) & (df2["price"] > 200),
"inner").show()
```

```
ShuffledHashJoin [cid#140], [cid#143], BuildRight
    Exchange (HashPartitioning 200)
        PhysicalRDD [age#139L,cid#140,name#141,state#142], MapPartitionsRDD[286]
at applySchemaToPythonRDD at NativeMethodAccessorImpl.java:-2
        Exchange (HashPartitioning 200)
            Filter (price#145L > 200)
                PhysicalRDD [cid#143,date#144,price#145L,product#146],
MapPartitionsRDD[295] at applySchemaToPythonRDD at NativeMethod
AccessorImpl.java:-2
```

208

DataFrame Operations: Underlying RDD

- A DataFrame is implemented with an RDD of Row objects; it is sometimes useful to access that underlying RDD:
  - `df.rdd`: returns the underlying rdd
- Several functions are shortcuts to the same function on that underlying RDD:
  - `df.map() = df.rdd.map()`
  - `df.flatMap() = df.rdd.flatMap()`
  - `df.foreach() = df.rdd.foreach()`
  - etc

More on DataFrames

- We covered a subset of data frames operations
- Other areas not covered here:
  - Data Frame windowing functions (`OVER` with rank, `first_value`, `last_value`, etc)
  - `cov`, `crosstab`, `corr`, `rollup`
  - `fillna()` to deal with missing values
- The best reference is the documentation: `https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame`

Lab: Work with Tables and DataFrames, Dataframes and UDFs, Hive + Spark SQL

Knowledge Check

Questions

1. While core RDD programming is used with [structured/unstructured/both] data Spark SQL is used with [structured/unstructured/both] data.
2. True or False: Spark SQL is an extra layer of translation over RDDs. Therefore while it may be easier to use, core RDD programs will generally see better performance.
3. True or False: A `HiveContext` can do everything that a `SQLContext` can do, but provides more functionality and flexibility.
4. True or False: Once a DataFrame is registered as a temporary table, it is available to any running sqlContext in the cluster.
5. Hive tables are stored [in memory/on disk].
6. Name two functions that can convert an RDD to a DataFrame.
7. Name two file formats that Spark SQL can use without modification to create DataFrames.

Summary

Summary

- Spark SQL gives developers the ability to utilize Spark's in-memory processing capabilities on structured data
- Spark SQL integrates with Hive via the HiveContext, which broadens SQL capabilities and allows Spark to use Hive HCatalog for table management
- DataFrames are RDDs that are represented as table objects which can used to create tables for SQL interactions
- DataFrames can be created from and saved as files such as ORC, JSON, and parquet
- Because of Catalyst optimizations of SQL queries, SQL programming operations will generally outperform core RDD programming operations

Objectives

After completing this lesson, students should be able to:

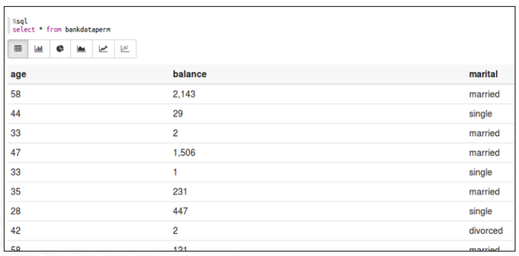
- Explain the purpose and benefits of data visualization
- Perform interactive data exploration using visualization in Zeppelin
- Collaborate with other developers and stakeholders using Zeppelin

- Data Visualization Overview

Objectives

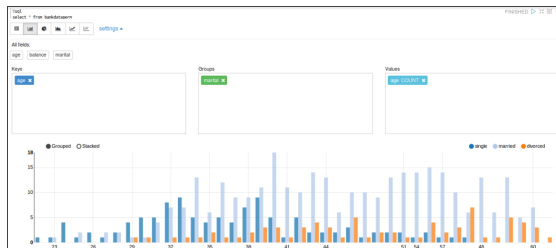
Data Visualization Introduction

- Table-based data is great for calculation and organization, but hard to use for decision making when working with large sets of data
- Data visualizations enable humans to make inferences and draw conclusions about large sets of data based on visual input alone



```
%sql
select * from bankdataperm
```

age	balance	marital
58	2,143	married
44	29	single
33	2	married
47	1,506	married
33	1	single
35	231	married
28	447	single
42	2	divorced



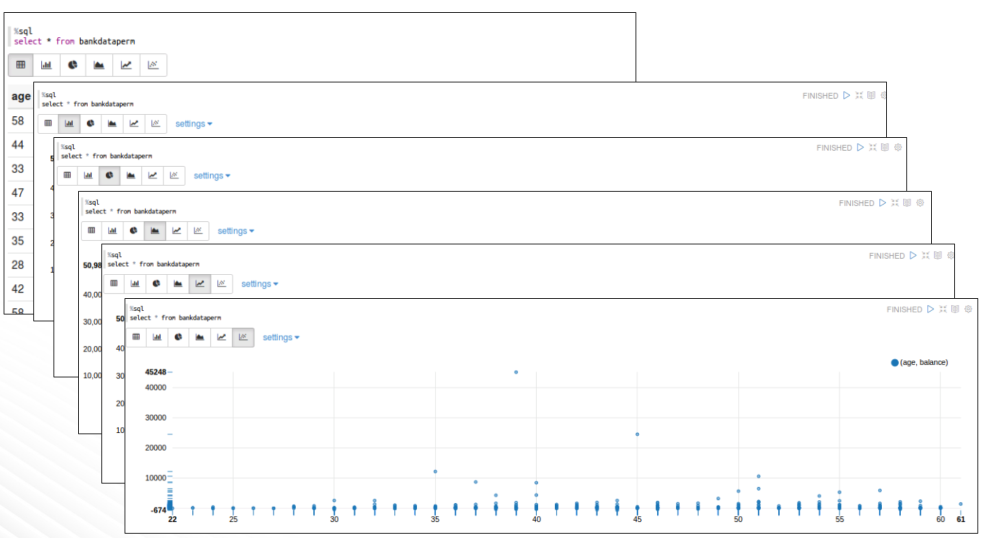
Data Visualization and Spark

- The Spark project contains a module called GraphX for visualizations
  - Scala only
  - Programmatic (difficult for non-coders to interact with)
- Zeppelin can be used for data visualization as well
  - Lots of built-in, easy to use visualizations
  - Virtually any visualization library from any supported language can be used
  - Easy collaboration with other developers and non-technical business owners

- Data Visualization Overview
- Data Exploration

Objectives

Visualizations on Tables (`%sql` default)



Visualizations on DataFrames

- `z.show(DataFrameName)`

```
%pyspark
bankDataFrame = sqlContext.table("bankdataperm")
z.show(bankDataFrame)
```

age	balance	marital
58	2,143	married
44	29	single
33	2	married
47	1,506	married
33	1	single
35	231	married
28	447	single
42	2	divorced
58	121	married

Visualizations on Other Formatted Data

- Use `%table` as part of the print instruction and, if formatted correctly, the data will be presented with visualizations enabled

```
println("%table code\tvalue\nAA\t150000\nBB\t80000\n")
```

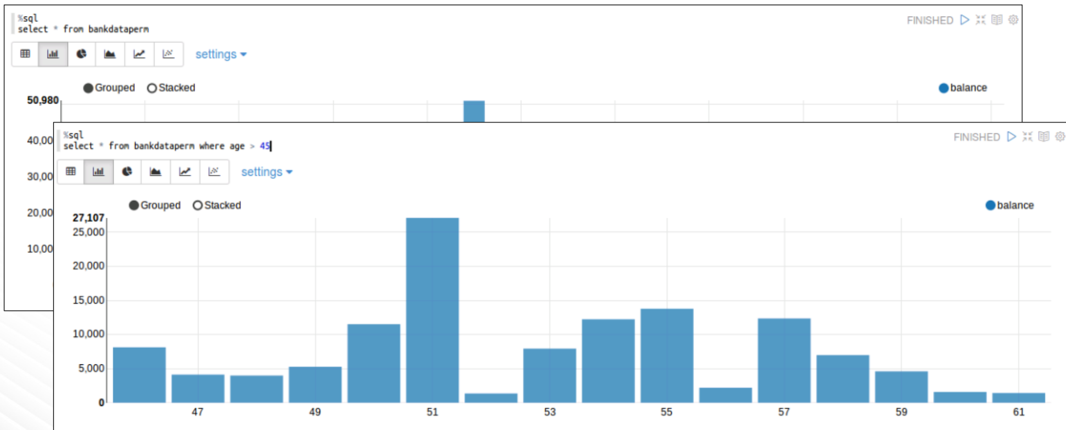
```
println("code\tvalue\nAA\t150000\nBB\t80000\n")

code    value
AA      150000
BB      80000
```

```
println("%table code\tvalue\nAA\t150000\nBB\t80000\n")
```

code	value
AA	150,000
BB	80,000

Interactive Visualization - Programmatic

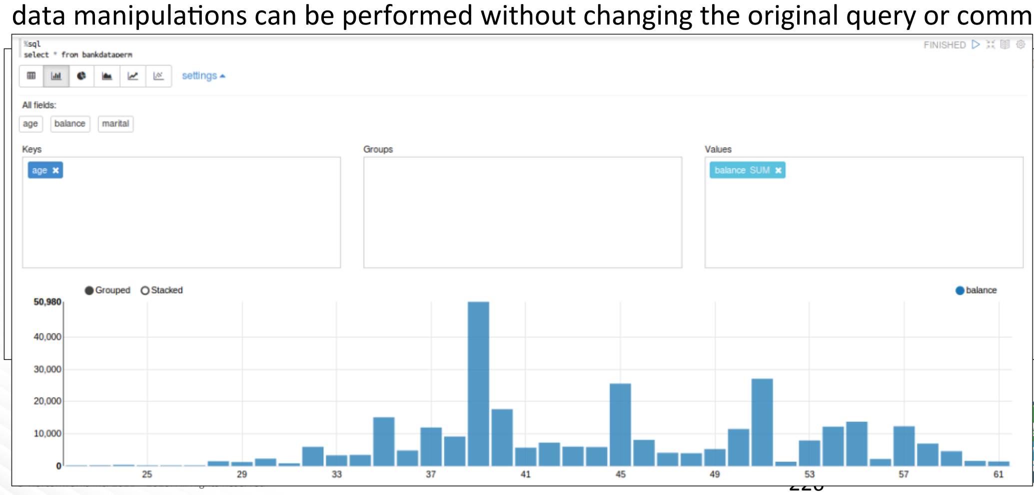
- Visualization displays change any time a new query (or other command) is executed



  225

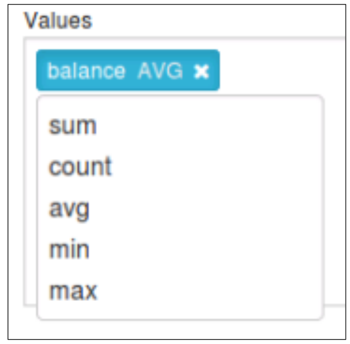
Interactive Visualization - Pivot Charts

- In addition, Zeppelin provides a Pivot Chart capability under Settings in which additional data manipulations can be performed without changing the original query or command



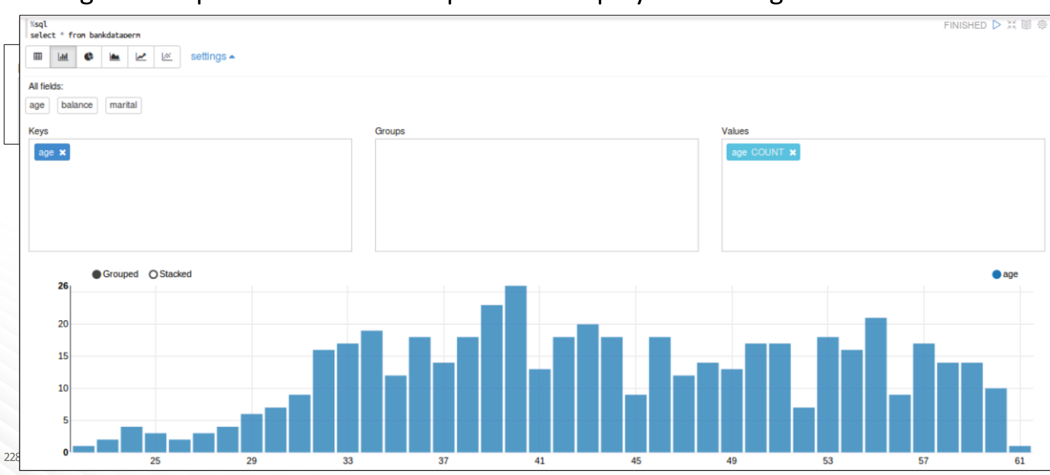
Pivot Chart - Value Options

- Click on the box under "Values" and a drop-down menu appears
- Use it to change the default value action
  - Switch between SUM, AVG, COUNT, MIN, and MAX



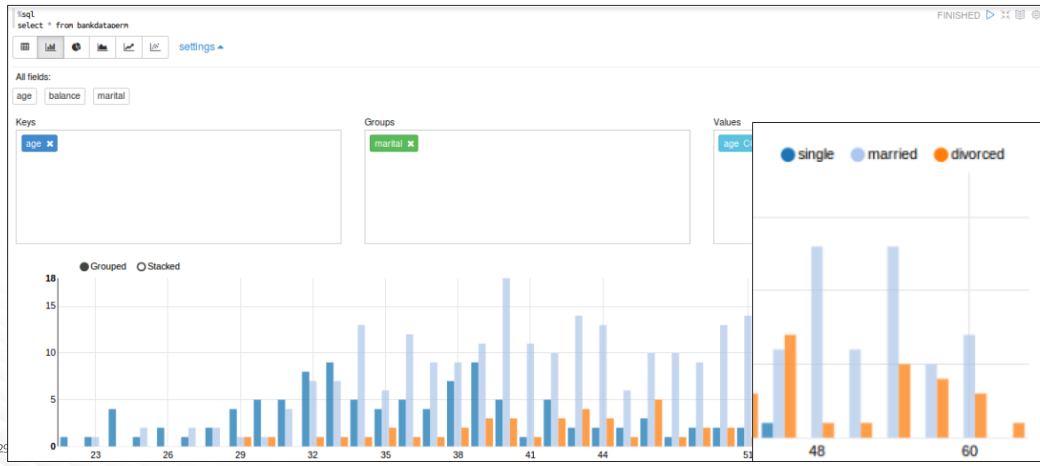
Pivot Chart - Change Values or Keys

- Click on the "x" to the right of a box to remove that from the appropriate column, then drag and drop from the column options to display something new

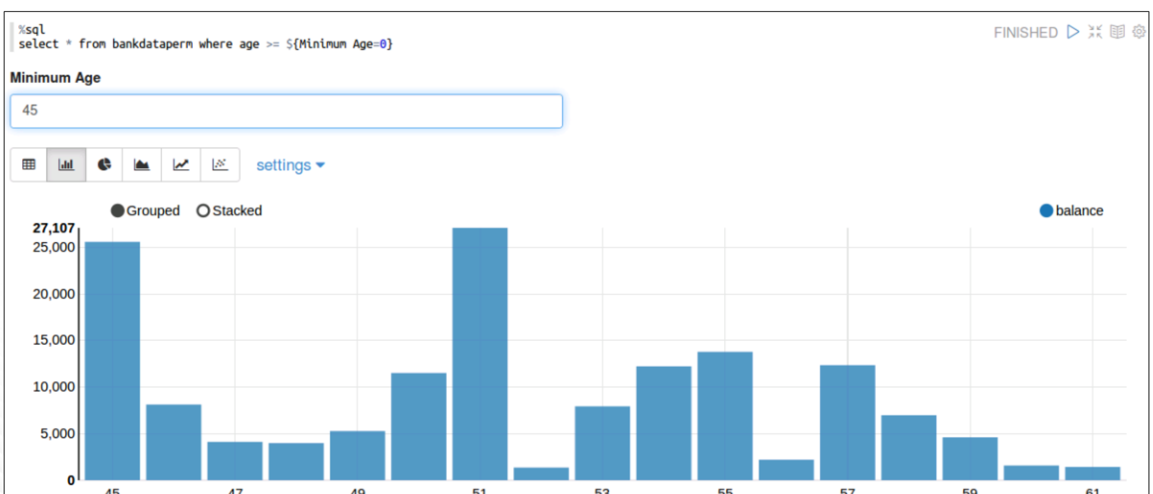


Pivot Chart - Add Groups

- Drag and drop the appropriate grouping category from the list of options to see the data further broken down

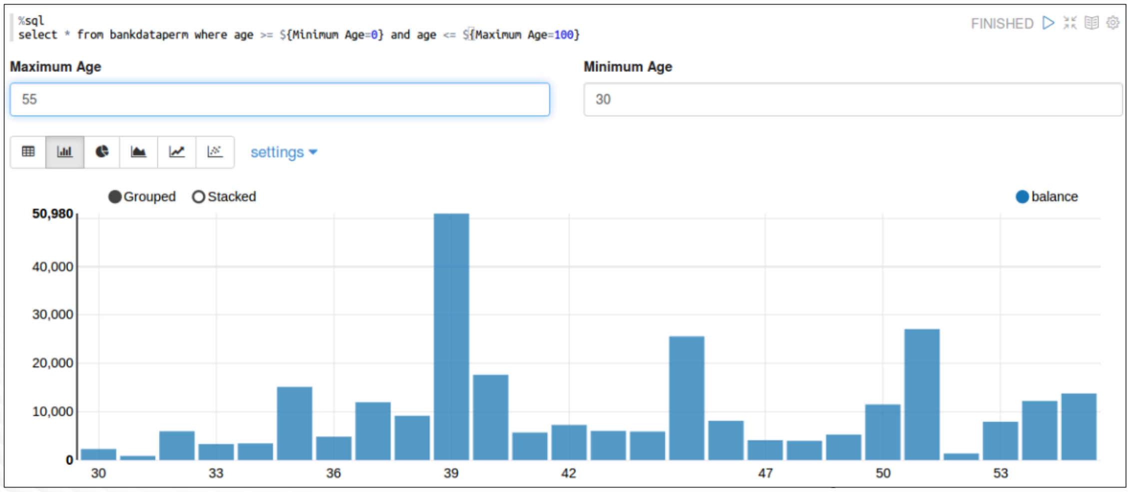


Dynamic Forms

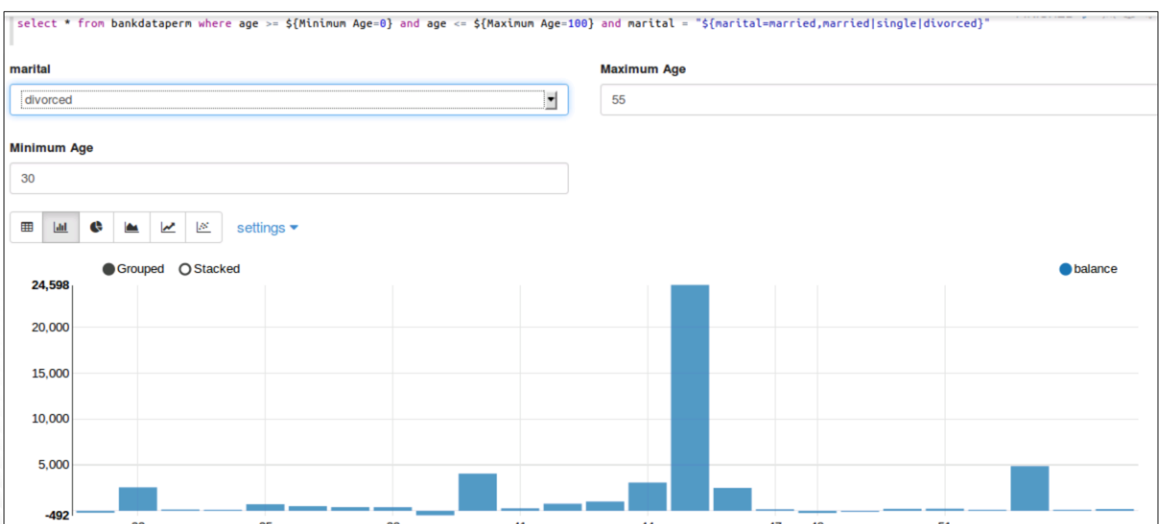


Dynamic Forms - Multiples

- Multiple variables can be included as Dynamic Forms



Dynamic Forms - Select



232

232

Objectives

- Data Visualization Overview
- Data Exploration
- Collaboration and Sharing

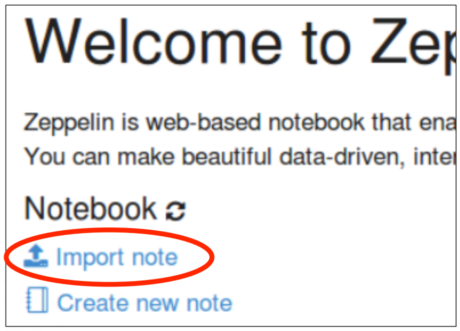
Clone and Export a Note

- Before sharing a note with others, it may be a good idea to make a copy of it
- Two ways to do this:
  – Clone: make a copy of the note in Zeppelin

  – Export: save a copy of the note in JSON format

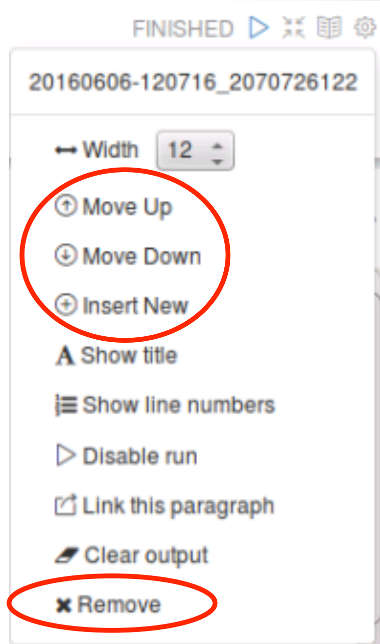
Import a Note

- Exported notes can be shared with and imported by another developer



Note Cleanup

- In-process notes can be messy and contain unnecessary duplicate code or alternatives
- Individual paragraphs that are no longer needed can be deleted from the note
- Paragraphs can also be reordered and new paragraphs can be inserted
  – For example, to add Markdown comments



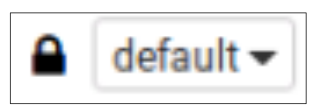
Interactive Note Sharing

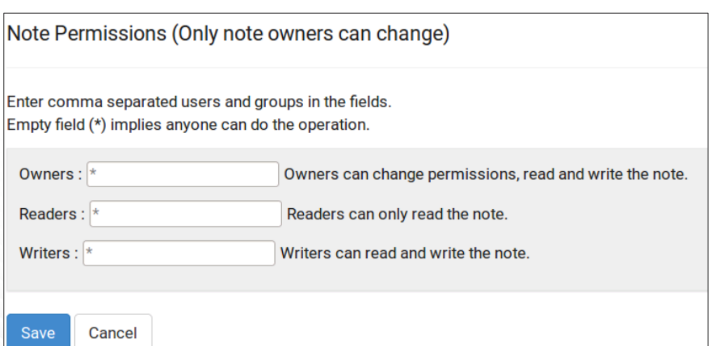
- Note URLs can be shared
  – All connections using this URL are live, real-time connections to the same note

Note Access Control

- By default, anyone with the note link can completely control the note

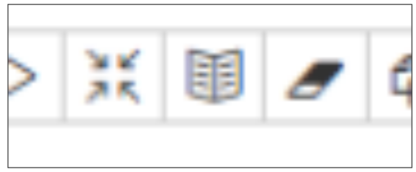
- To control access, click the Note Permissions (padlock) icon at the top-right corner of the note and set permissions accordingly



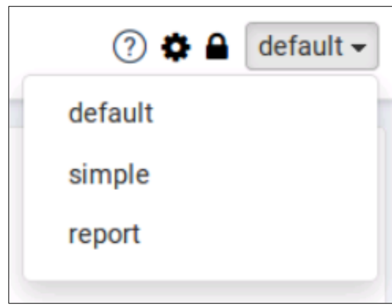


Note Formatting

- Note owners can control all paragraphs at the note level, including:
  - Hide/Show all code
  - Hide/Show all output
  - Clear all output

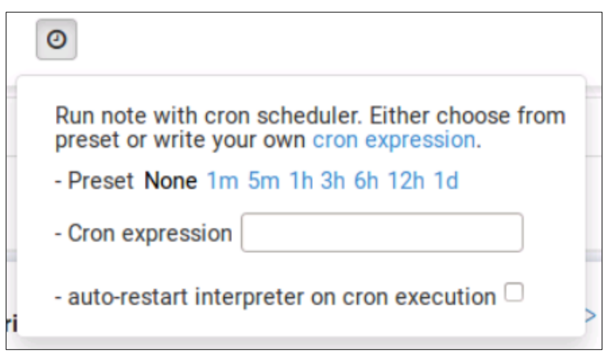


- There are also two additional note views
  - Simple: Removes note-level controls
  - Report: Removes note-level controls and all code



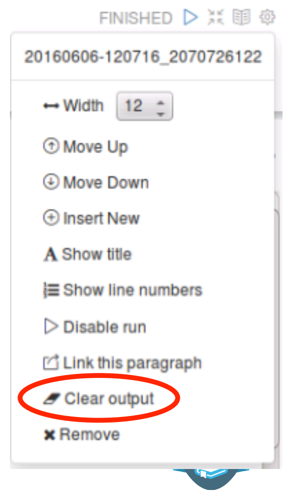
Automate Note Updates

- Entire notes can be played, paragraph by paragraph, at regular intervals



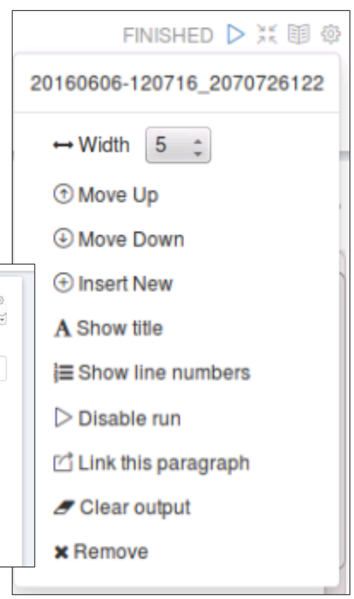
Paragraph Formatting

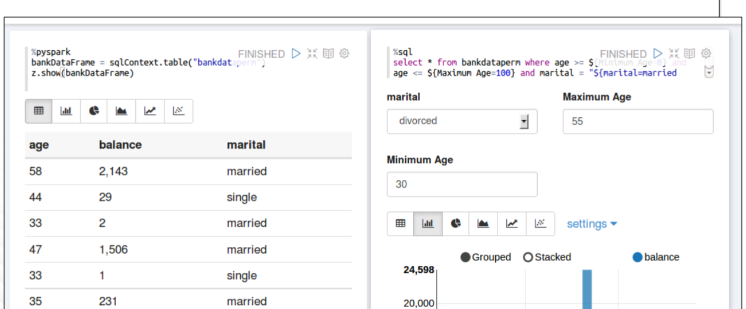
- Paragraphs also contain formatting settings, including:
  - Hide/Show paragraph code
  - Hide/Show paragraph output
  - Clear paragraph output is available in the settings menu (gear icon)



Paragraph Enhancement - Width

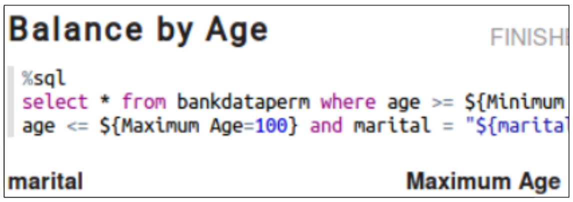
- Width: Controls width of the paragraph in the note, allowing multiple paragraphs to be displayed in a row

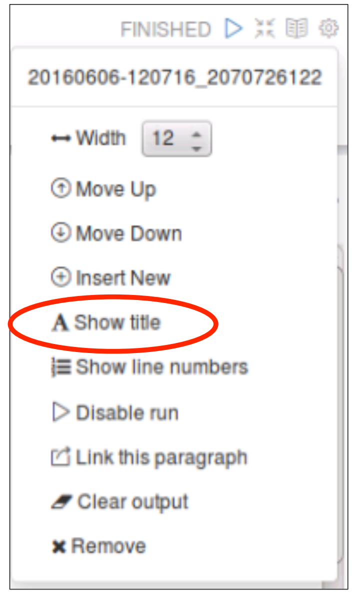




Paragraph Enhancement - Show Title

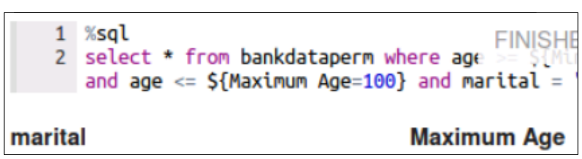
- Paragraph titles can be added for clarity

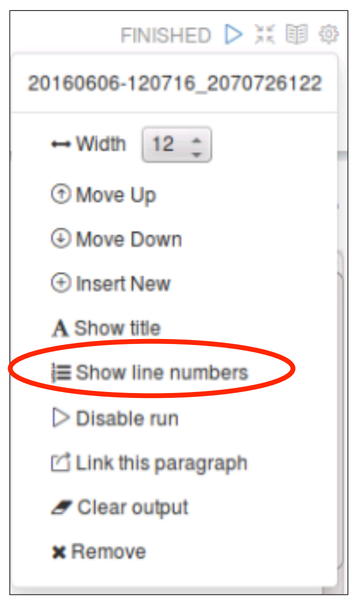




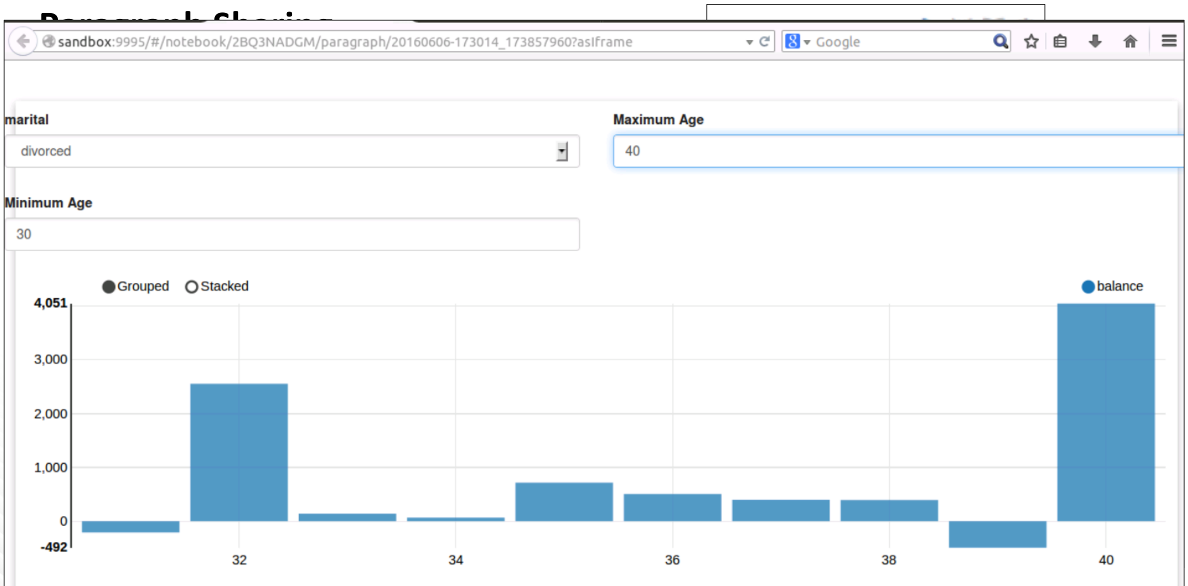
Paragraph Enhancement - Line Numbers

- Line numbers can be added to paragraph code



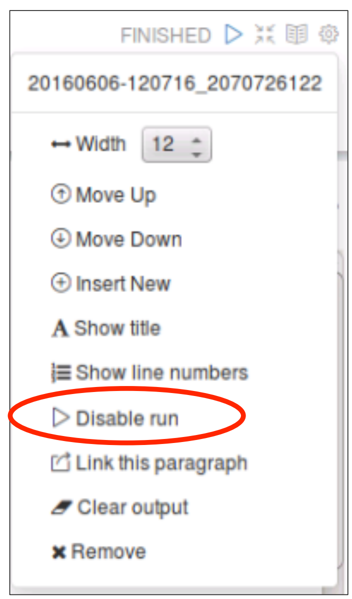


Paragraph Sharing



Disable Paragraph Output Changes

- Disable the paragraph run feature to lock the output of a paragraph
- Changes to Dynamic Forms or code will not be reflected in the paragraph



Lab: Data Visualization in Zeppelin

Knowledge Check

Questions

1. What is the value of data visualization?
2. How many chart views does Zeppelin provide by default?
3. How do you share a copy of your note (non-collaborative) with another developer?
4. How do you share your note collaboratively with another developer?
5. Which note view provides only paragraph outputs?
6. Which paragraph feature provides the ability for an outside person to see a paragraph's output without having access to the note?
7. What paragraph feature allows you to give outside users the ability to modify parameters and update the displayed output without using code?

Summary

Summary

- Data visualizations are important when humans need to draw conclusions about large sets of data
- Zeppelin provides support for a number of built-in data visualizations, and these can be extended via visualization libraries and other tools like HTML and JavaScript
- Zeppelin visualizations can be used for interactive data exploration by modifying queries, as well as the use of pivot charts and implementation of dynamic forms
- Zeppelin notes can be shared via export to a JSON file or by sharing the note URL
- Zeppelin provides numerous tools for controlling the appearance of notes and paragraphs which can assist in communicating important information
- Paragraphs can be shared via a URL link
- Paragraphs can be modified to control their appearance and assist in communicating important information

HORTONWORKS UNIVERSITY

251

Objectives

After completing this lesson, students should be able to:

- Describe the components of a Spark job
- Explain default parallel execution for stages, tasks, across CPU cores
- Monitor Spark jobs via the Spark Application UI

253

- Job Anatomy

Objectives

254

Spark Task/Stage/Job/ DAG Schedule

- Task is a unit of work (pipeline of operations that do not require a shuffle)
- Stage is a group of tasks separated by a operation that requires a shuffle
- A job is a grouping of stages

- The DAG scheduler tells Spark which stages to execute when
  - The next stage cannot start before all the tasks in the previous stage have finished

Wide and Narrow Operations

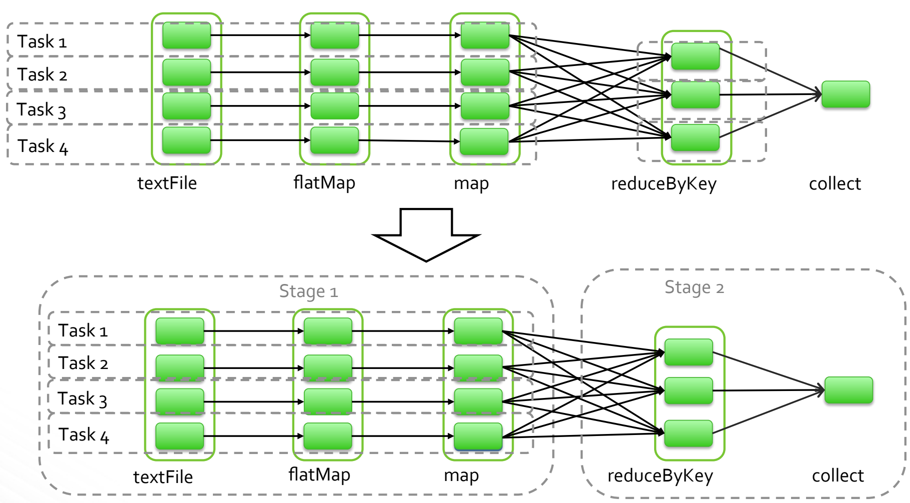
- Wide operations require a shuffling of data (many to 1 relationship)
  - reduceByKey
  - groupByKey
  - repartition
  - join

- Narrow operations can be executed locally (1 to many relationship)
  - map
  - filter
  - flatMap

RDD Graph

```
sc.textFile("/path/to/data")
    .flatMap(lambda line: line.split(" "))
    .map(lambda word: (word,1)))
    .reduceByKey(lambda a,b: a+b, numPartitions = 3)
    .collect()
```


DAG Scheduler

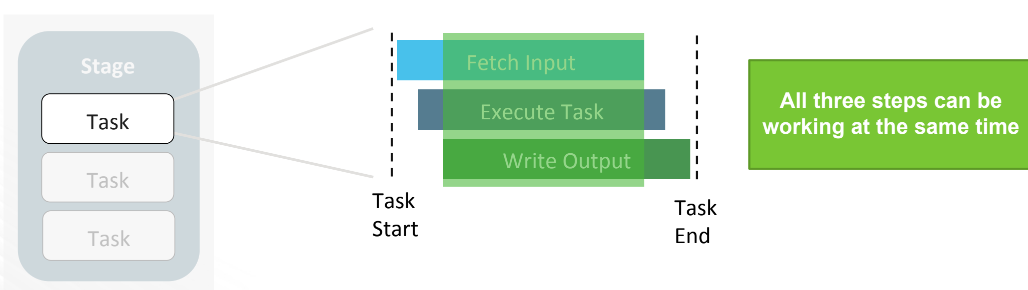


Objectives

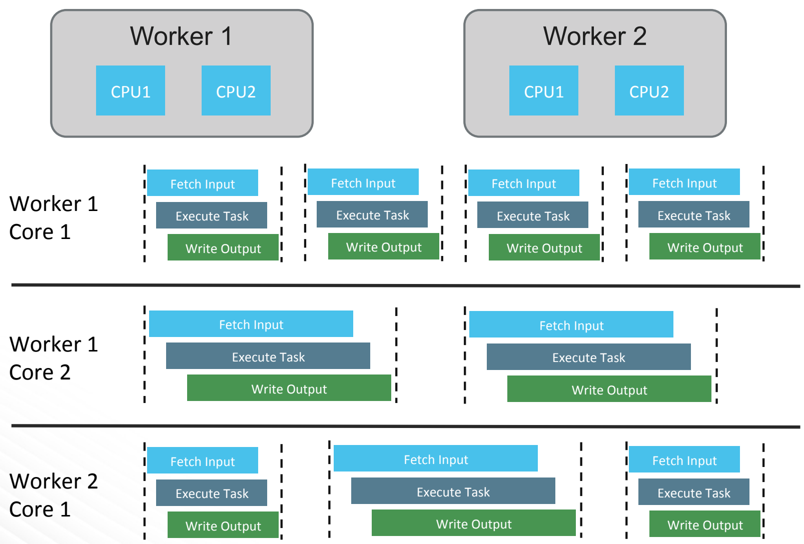
- Job Anatomy
- Parallel Execution

Task Steps

- A task consists of three steps
  - Fetch input data
  - Execute the operation
  - Produce output
- Parallel execution minimizes task completion times



Tasks and CPU Cores



Each CPU core on each node can process tasks independently

Inherent Parallelism – `parallelize()`

- `parallelize()` bases default partitioning on the number of cores across all executors it is assigned
  - Minimum of two partitions
  - Default behavior can be overridden
  - Intent is to maximize parallel operations
- Can be overridden at RDD creation time:

```
rdd1 = sc.parallelize([1,2,3,4,5,6],8)
```

Inherent Parallelism – `textFile()`

- `textFile()` partitions based on the number of HDFS blocks the file uses
  - A single-block file (default 128 MB or less) will get the minimum of two partitions
  - RDD partition number can be larger, but not smaller, than number of HDFS blocks
  - Goal is to avoid moving data between nodes

- Can be overridden at RDD creation time:

```
rdd2 = sc.textFile("statePopulations.csv",numPartitions=8)
```


Tune Data Parallelism

- Spark works with partitions as the mechanism for data processing parallelization
- Use `repartition()` or `coalesce()` to control parallelism when needed
  - Use coalesce when reducing partitions, repartition to increase

```
rdd.repartition(500)
rdd.coalesce(20)
```

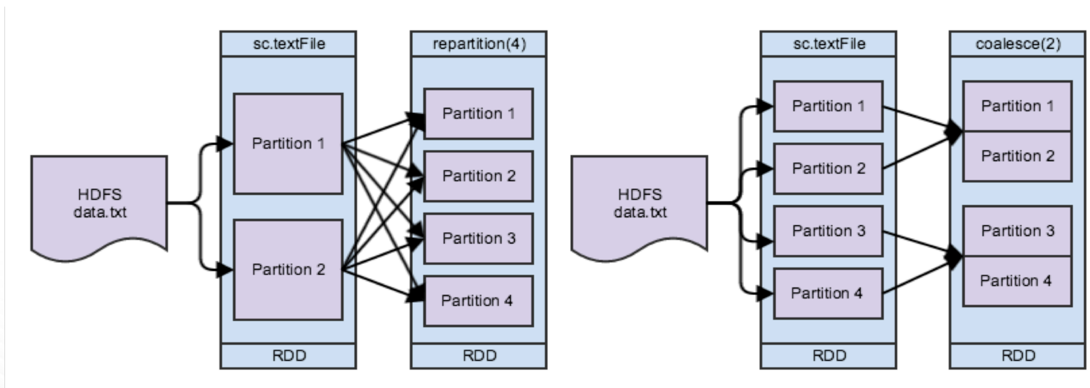
- Many operations include `numPartitions` as parameter that does this automatically

```
rdd.reduceByKey(lambda ab: a+b, numPartitions=10)
```

- In the REPL, users can check the number of partitions by executing the following:

```
rdd.getNumPartitions()
```

Changing the Level of Parallelism



Objectives

- Job Anatomy
- Parallel Execution
- Spark Application UI

Spark Application UI

- Generated by and available for the life of a SparkContext
  – When the SparkContext is exited, no longer available
- Accessed via <drivernode>:4040
  – In our environment: sandbox:4040
- If multiple SparkContext instances are launched, multiple Spark Application UIs will exist
  – Each new one incremented port number by one - for example: sandbox:4041, sandbox:4042
  – For example: running Zeppelin, open a PySpark REPL

Spark UI: Jobs View

Spark 1.4.1 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Spark Jobs (?)

Total Uptime: 1.8 h
Scheduling Mode: FIFO
Completed Jobs: 18

▸ Event Timeline

Completed Jobs (18)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
17	take at <console>:30	2015/11/11 17:52:08	2 s	3/3	9/9
16	take at <console>:26	2015/11/11 17:52:03	18 ms	1/1	1/1
15	take at <console>:26	2015/11/11 17:51:03	18 ms	1/1	1/1
14	take at <console>:26	2015/11/11 17:50:52	15 ms	1/1	1/1
13	take at <console>:30	2015/11/11 17:50:36	0.1 s	1/1 (2 skipped)	1/1 (8 skipped)
12	take at <console>:30	2015/11/11 17:50:36	0.5 s	1/1 (2 skipped)	4/4 (8 skipped)
11	take at <console>:30	2015/11/11 17:50:33	3 s	3/3	9/9
10	take at <console>:26	2015/11/11 17:50:19	26 ms	1/1	1/1
9	take at <console>:24	2015/11/11 17:49:43	21 ms	1/1	1/1

Spark UI: Single Job View

Spark 1.4.1 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Details for Job 17

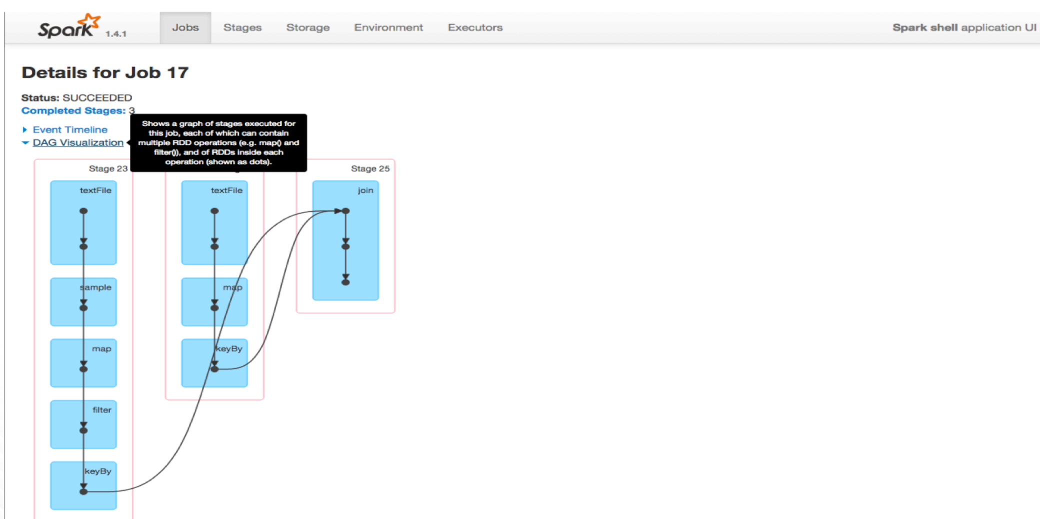
Status: SUCCEEDED
Completed Stages: 3

- Event Timeline
- DAG Visualization

Completed Stages (3)

Stage Id	Description		Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
25	take at <console>:30	+details	2015/11/11 17:52:09	0.4 s	1/1			1120.9 KB	
24	keyBy at <console>:23	+details	2015/11/11 17:52:08	1 s	2/2	64.1 KB			41.8 KB
23	keyBy at <console>:23	+details	2015/11/11 17:52:08	2 s	6/6	657.8 MB			4.1 MB

Spark UI: Single Job DAG Visualization



Spark UI: Inside a Stage

Spark 1.4.1 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Details for Stage 23 (Attempt 0)

Total Time Across All Tasks: 5 s
Input Size / Records: 657.8 MB / 7009728
Shuffle Write: 4.1 MB / 68613

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 6 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0.1 s	0.8 s	1 s	1 s	1 s
Scheduler Delay	2 ms	3 ms	5 ms	7 ms	17 ms
GC Time	8 ms	22 ms	0.1 s	0.1 s	0.1 s
Input Size / Records	17.5 MB / 184198	128.1 MB / 1351102	128.1 MB / 1368262	128.1 MB / 1369518	128.1 MB / 1372006
Shuffle Write Size / Records	110.3 KB / 1759	807.6 KB / 13134	825.0 KB / 13364	826.8 KB / 13413	841.8 KB / 13617

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input Size / Records	Shuffle Write Size / Records
driver	localhost:45917	5 s	6	0	6	657.8 MB / 7009728	4.1 MB / 68613

Spark UI: Inside a Stage, cont.

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	GC Time	Input Size / Records	Write Time	Shuffle Write Size / Records	Errors
0	33	0	SUCCESS	ANY	driver / localhost	2015/11/11 17:52:08	1 s	5 ms	95 ms	128.1 MB (hadoop) / 1372006	5 ms	823.3 KB / 13326	
1	34	0	SUCCESS	ANY	driver / localhost	2015/11/11 17:52:08	1 s	7 ms	0.1 s	128.1 MB (hadoop) / 1368262	7 ms	841.8 KB / 13617	
2	35	0	SUCCESS	ANY	driver / localhost	2015/11/11 17:52:08	1 s	3 ms	0.1 s	128.1 MB (hadoop) / 1369518	5 ms	825.0 KB / 13364	
3	36	0	SUCCESS	ANY	driver / localhost	2015/11/11 17:52:08	1 s	4 ms	0.1 s	128.1 MB (hadoop) / 1364642	6 ms	826.8 KB / 13413	
4	37	0	SUCCESS	ANY	driver / localhost	2015/11/11 17:52:09	0.8 s	17 ms	22 ms	128.1 MB (hadoop) / 1351102	4 ms	807.6 KB / 13134	
5	38	0	SUCCESS	ANY	driver / localhost	2015/11/11 17:52:09	0.1 s	2 ms	8 ms	17.5 MB (hadoop) / 184198	2 ms	110.3 KB / 1759	

Spark UI: Environment

Spark 1.4.1 | Jobs | Stages | Storage | Environment | Executors | **Spark shell** application UI

Environment

Runtime Information

Name	Value
Java Home	/usr/lib/jvm/java-1.7.0-openjdk-1.7.0.91.x86_64/jre
Java Version	1.7.0_91 (Oracle Corporation)
Scala Version	version 2.10.4

Spark Properties

Name	Value
spark.app.id	local-1447263545873
spark.app.name	Spark shell
spark.driver.extraJavaOptions	-Dhdp.version=2.3.2.0-2950
spark.driver.host	192.168.1.170
spark.driver.port	35557

273

Executor View

Spark 1.6.0 | Jobs | Stages | Storage | Environment | Executors | SQL | **Zeppelin** application UI

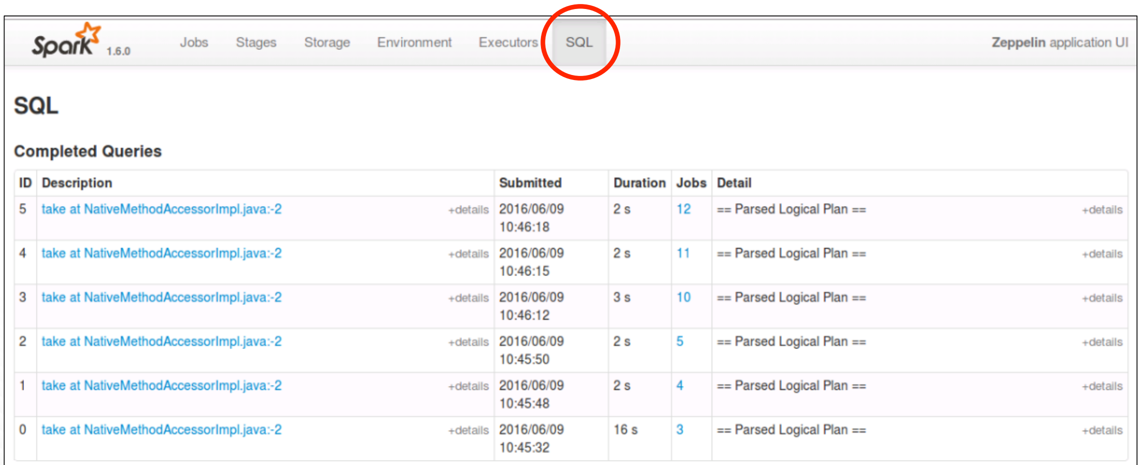
Executors (3)

Memory: 0.0 B Used (797.6 MB 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	Thread Dump
1	sandbox:52087	0	0.0 B / 143.3 MB	0.0 B	0	4	594	598	32.9 s	2.2 MB	10.8 KB	21.6 KB	stdout stderr	Thread Dump
2	sandbox:57010	0	0.0 B / 143.3 MB	0.0 B	0	4	628	632	32.8 s	2.0 MB	10.8 KB	21.7 KB	stdout stderr	Thread Dump
driver	172.17.0.1:52752	0	0.0 B / 511.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

274

SQL View

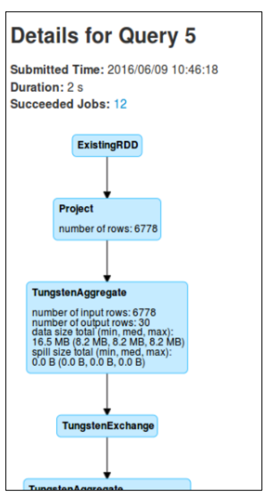


SQL

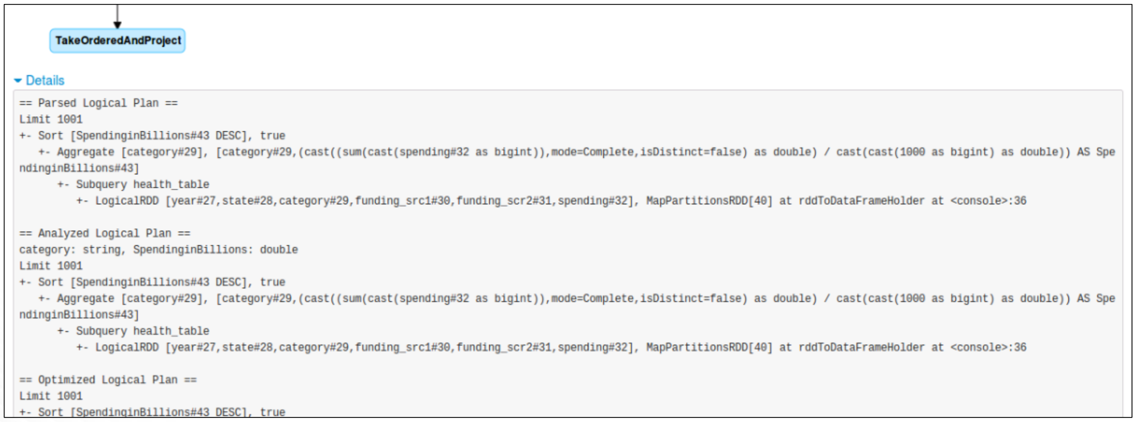
Completed Queries

ID	Description		Submitted	Duration	Jobs	Detail	
5	take at NativeMethodAccessorImpl.java:-2	+details	2016/06/09 10:46:18	2 s	12	== Parsed Logical Plan ==	+details
4	take at NativeMethodAccessorImpl.java:-2	+details	2016/06/09 10:46:15	2 s	11	== Parsed Logical Plan ==	+details
3	take at NativeMethodAccessorImpl.java:-2	+details	2016/06/09 10:46:12	3 s	10	== Parsed Logical Plan ==	+details
2	take at NativeMethodAccessorImpl.java:-2	+details	2016/06/09 10:45:50	2 s	5	== Parsed Logical Plan ==	+details
1	take at NativeMethodAccessorImpl.java:-2	+details	2016/06/09 10:45:48	2 s	4	== Parsed Logical Plan ==	+details
0	take at NativeMethodAccessorImpl.java:-2	+details	2016/06/09 10:45:32	16 s	3	== Parsed Logical Plan ==	+details

SQL Query Details - Visual



SQL Query Details - Text

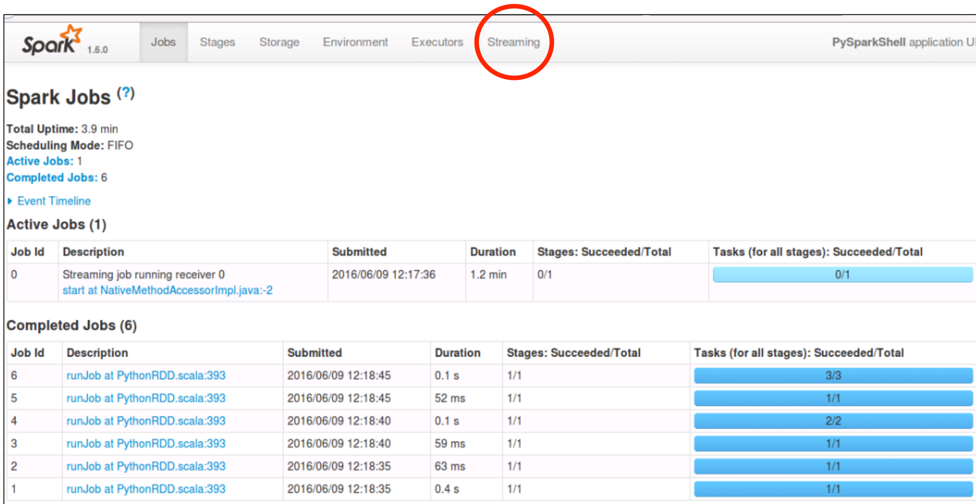


TakeOrderedAndProject

Details

```
== Parsed Logical Plan ==
Limit 1001
+- Sort [SpendinginBillions#43 DESC], true
   +- Aggregate [category#29], [category#29,(cast((sum(cast(spending#32 as bigint)),mode=Complete,isDistinct=false) as double) / cast(cast(1000 as bigint) as double)) AS Spe
ndinginBillions#43]
      +- Subquery health_table
         +- LogicalRDD [year#27,state#28,category#29,funding_src1#30,funding_scr2#31,spending#32], MapPartitionsRDD[40] at rddToDataFrameHolder at <console>:36

== Analyzed Logical Plan ==
category: string, SpendinginBillions: double
Limit 1001
+- Sort [SpendinginBillions#43 DESC], true
   +- Aggregate [category#29], [category#29,(cast((sum(cast(spending#32 as bigint)),mode=Complete,isDistinct=false) as double) / cast(cast(1000 as bigint) as double)) AS Spe
ndinginBillions#43]
      +- Subquery health_table
         +- LogicalRDD [year#27,state#28,category#29,funding_src1#30,funding_scr2#31,spending#32], MapPartitionsRDD[40] at rddToDataFrameHolder at <console>:36

== Optimized Logical Plan ==
Limit 1001
+- Sort [SpendinginBillions#43 DESC], true
```


Streaming Tab



Spark 1.6.0 | Jobs | Stages | Storage | Environment | Executors | Streaming | PySparkShell application UI

Spark Jobs (?)

Total Uptime: 3.9 min
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 6

Event Timeline

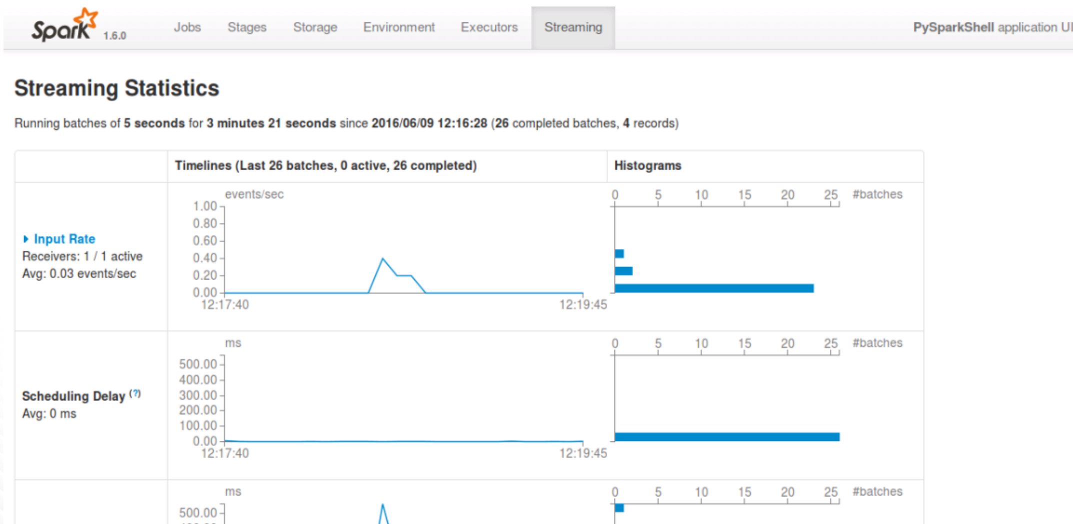
Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	Streaming job running receiver 0 start at NativeMethodAccessorImpl.java:-2	2016/06/09 12:17:36	1.2 min	0/1	0/1

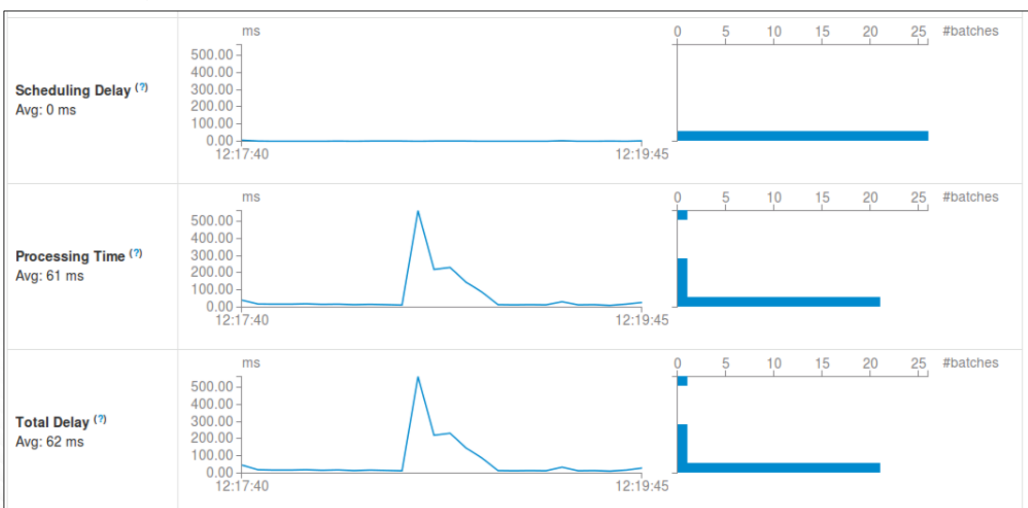
Completed Jobs (6)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
6	runJob at PythonRDD.scala:393	2016/06/09 12:18:45	0.1 s	1/1	3/3
5	runJob at PythonRDD.scala:393	2016/06/09 12:18:45	52 ms	1/1	1/1
4	runJob at PythonRDD.scala:393	2016/06/09 12:18:40	0.1 s	1/1	2/2
3	runJob at PythonRDD.scala:393	2016/06/09 12:18:40	59 ms	1/1	1/1
2	runJob at PythonRDD.scala:393	2016/06/09 12:18:35	63 ms	1/1	1/1
1	runJob at PythonRDD.scala:393	2016/06/09 12:18:35	0.4 s	1/1	1/1

Streaming View



Streaming Additional Charts



Streaming Batches

Active Batches (0)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Output Ops: Succeeded/Total	Status

Completed Batches (last 26 out of 26)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Total Delay (?)	Output Ops: Succeeded/Total
2016/06/09 12:19:45	0 events	2 ms	27 ms	29 ms	1/1
2016/06/09 12:19:40	0 events	0 ms	17 ms	17 ms	1/1
2016/06/09 12:19:35	0 events	1 ms	10 ms	11 ms	1/1
2016/06/09 12:19:30	0 events	0 ms	14 ms	14 ms	1/1
2016/06/09 12:19:25	0 events	0 ms	13 ms	13 ms	1/1
2016/06/09 12:19:20	0 events	3 ms	31 ms	34 ms	1/1
2016/06/09 12:19:15	0 events	0 ms	13 ms	13 ms	1/1
2016/06/09 12:19:10	0 events	0 ms	14 ms	14 ms	1/1
2016/06/09 12:19:05	0 events	0 ms	13 ms	13 ms	1/1
2016/06/09 12:19:00	0 events	0 ms	14 ms	14 ms	1/1
2016/06/09 12:18:55	0 events	0 ms	87 ms	87 ms	1/1

281

Batch Detail

Spark 1.6.0 Jobs Stages Storage Environment Executors Streaming PySparkShell application UI

Details of batch at 2016/06/09 12:19:45

Batch Duration: 5 s
Input data size: 0 records
Scheduling delay: 2 ms
Processing time: 27 ms
Total delay: 29 ms

Output Op Id	Description	Duration	Status	Job Id	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total	Error
0	callForeachRDD at NativeMethodAccessorImpl.java:-2 +details	28 ms	Succeeded	-	-	-	-	-

282

Lab: Job Monitoring

Knowledge Check

Questions

1. Spark jobs are divided into ____________, which are logical collections of ______________.
2. A job is defined as a set of tasks that culminates in a _______________.
3. What Spark component organizes stages into logical groupings that allow for parallel execution?
4. What is the default port used for the Spark Application UI?
5. If two SparkContext instances are running, what is the port used for the Spark Application UI of the second one?
6. As discussed in this lesson, what tabs in the Spark Application UI only appear if certain types of jobs are run?

Summary

Summary

- Spark applications consist of Spark jobs, which are collections of tasks that culminate in an action.
- Spark jobs are divided into stages, which separate lists of tasks based on shuffle boundaries and are organized for optimized parallel execution via the DAG Scheduler.
- The Spark Application UI provides a view into all jobs run or running for a given SparkContext instance, including detailed information and statistics appropriate for the application and tasks being performed.

Caching and Persisting Data, Checkpointing

Objectives

After completing this lesson, students should be able to:

- Understand the caching, persisting, and the different storage levels
- Describe and implement checkpointing

- Caching and Persisting Data

Objectives

Caching and Persisting Data

- Spark data is not maintained in memory by default
- Spark allows the developer to persist data in memory
  - Beneficial when an RDD is going to be used more than once - for example: an application where a "clean" file, reject file, and summary file are each created by processing the same original file
  - Very useful (and incredibly fast) for iterative applications

Memory Representations & Limitations

- Caching occurs at the partition level
- Cached datasets can be stored three ways
  - Serialized: objects are turned into compact byte streams - reduces memory usage, but require more processing resources to deserialize when needed
  - Raw: fastest to process, but can easily take up 2-10x more memory than serialized datasets
  - Off-heap: Utilize off heap memory to avoid GC's. Slower to access off heap memory (all data must be serialized, primative classes have encoders)
- Executor memory is a finite resource
  - Least Recently Used (LRU) algorithm determines which dataset(s) to evict when needed
  - If an operation tries to use cache that no longer exists, data will be recomputed and recached
    - Will discuss in more detail later

Caching Syntax

- `persist()` - developer can control caching storage level
  - `persist(StorageLevel.Selection)`
- `cache()` - simple operation
  - `cache() == persist(StorageLevel.MEMORY_ONLY)`
- `unpersist()` - remove data from cache
- Must import library to use it:

```
scala -> import org.apache.spark.storage.storageLevel._
python -> from pyspark import StorageLevel
```

- In `pyspark`: objects are always stored with the Pickle library
  - So `MEMORY_ONLY` and `MEMORY_ONLY_SER` are the same

Spark SQL:

sqlContext.cacheTable()
sqlContext.uncacheTable()

293

Physical Options for Caching

Storage Level	Memory	Disk	Serialized	Replicas
MEMORY_ONLY (default)	Yes	Never	No	No
MEMORY_AND_DISK	Yes	Spills	No	No
MEMORY_ONLY_SER	Yes	No	Yes	No
MEMORY_AND_DISK_SER	Yes	Spills	Yes	No
MEMORY_ONLY_2	Yes	No	No	Yes
MEMORY_AND_DISK_2	Yes	Spills	No	Yes
DISK_ONLY	No	Yes	No	No

294

Example

```
from pyspark import StorageLevel

rdd = sc.textFiles("/user/root/logs/*")
rdd.persist(StorageLevel.MEMORY_ONLY_SER)
rdd.map(…).saveAsTextFile("/user/root/cleanLogs.txt")
rdd.filter(…).saveAsTextFile("/user/root/filteredLogs.txt")
rdd.unpersist()
```


Which Storage Level to Choose?

- If the RDD fits in memory, use the default MEMORY_ONLY
- If RDDs are too big, try MEMORY_ONLY_SER with a fast serialization library (Scala only)
- If the RDDs are still too big:
  - Consider the time to compute this RDD from parent RDD vs the time to load it from disk
  - Re-computing an RDD may sometimes be faster than reading it from disk
- Replicated storage is good for fast fault recovery, but…
  - Usually this is overkill, and not a good idea if you're using a lot of data relative to total memory
- For DataFrames, use `cache()` instead of `persist(StorageLevel)`

Serialization Options

- For Scala, Spark provides two serialization libraries:
  – Java serialization (default)
  – Kryo serialization
- Kryo is much faster & more compact (often as much as 10x)
  – Used to require registration of custom classes, but this has since been addressed
- Python uses Pickle for RDD serialization
  – DataFrames generate Java byte code, so DataFrames should leverage Kryo

Kyro Serialization

- Using Kryo Serialization (always use it)

```
conf = SparkConf()
conf.set('spark.serializer',
   'org.apache.spark.serializer.KryoSerializer')
sc=SparkContext(conf=conf)
```

Lab: Caching and Persisting Data

Objectives

- Caching and Persisting Data
- Checkpointing

Recomputation Problem

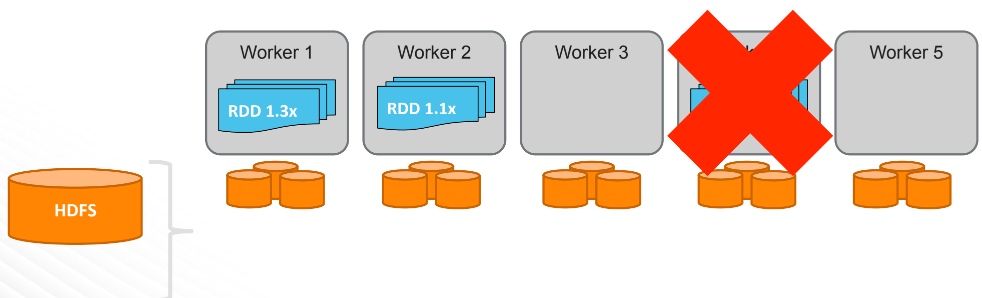
- As Spark transformations are processed they create a lineage
  – This lineage provides resilience, but can also cause problems as number of transformations grows
- If data is lost on an executor, re-computing that data can take a very long time
  – The data can potentially have to be reprocessed through hundreds/thousands of operations

Checkpointing

- Helps mitigate the recomputation problem
- Enabling checkpointing does the following
  – Data checkpointing that saves intermediate data to reliable storage (HDFS)
  – Metadata checkpointing, which stores file names and other configuration data
- Lineage is "reset" to the point of the last checkpoint
- Considerations:
  – Performed at the RDD, not the application, level
  – No current DataFrame support
  – There is an expense to persist to HDFS, but this is usually overshadowed by the benefits
  – No automatic cleanup of HDFS files

Node Loss Without Checkpointing

All processing must be repeated, potentially hundreds or thousands of transformations

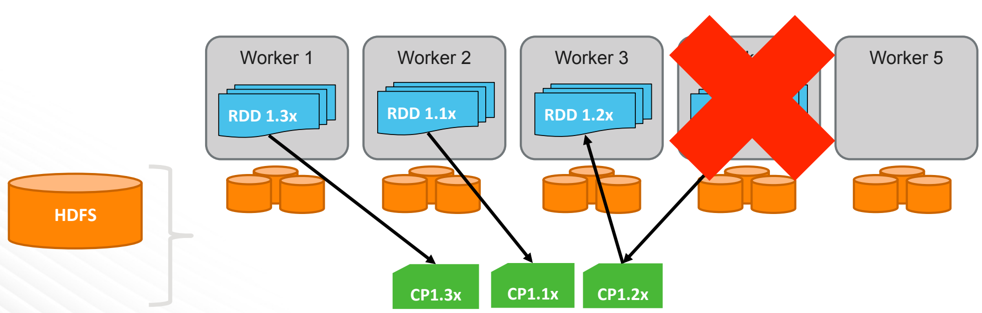


303

Node Loss With Checkpointing

Only processing since last checkpoint must be repeated

Trade performance while processing in exchange for faster recovery in case of node loss



304

Checkpointing vs Caching

- Checkpoint
  - Saves a permanent copy of the intermediate data
  - Lineage is then rebuilt from the intermediate data
  - If data is lost, recomputes the data from intermediate data
- Caching
  - Data is stored somewhere temporarily
  - Lineage is preserved
  - If data is lost, recomputes from base data

When to Use Checkpointing

- Window and other stateful streaming application transformations require it
- Iterative applications that may loop through data hundreds, or thousands of times
  - Machine learning algorithms typically do this

Implementing Checkpointing

- Set a checkpoint directory, and checkpoint the rdd:

```
sc.setCheckpointDir("somedir/")
rdd = sc.textFile("/path/to/file.txt")
while x in range(<large number>)
    rdd.map(…)
    if x % 5 == 0
        rdd.checkpoint()
rdd.saveAsTextFile("/path/to/output.txt")
```

307

Understand RDD lineage with toDebugString()

- Example:

```
rdd.toDebugString
```

(2) PythonRDD[29] at RDD at PythonRDD.scala:43 []

| MapPartitionsRDD[28] at mapPartitions at PythonRDD.scala:346 []

| ShuffledRDD[27] at partitionBy at NativeMethodAccessorImpl.java:-2 []
+-(2) PairwiseRDD[26] at reduceByKey at <ipython-input-8-1817f0de03c6>:2 []

| PythonRDD[25] at reduceByKey at <ipython-input-8-1817f0de03c6>:2 [] |
MapPartitionsRDD[24] at textFile at NativeMethodAccessorImpl.java:-2 []
| some-text-file HadoopRDD[23] at textFile at NativeMethodAccessorImpl.java:-2 []

308

Lab: Checkpointing and RDD Lineage

Spark Shared Variables

Objectives

After completing this lesson, students should be able to:

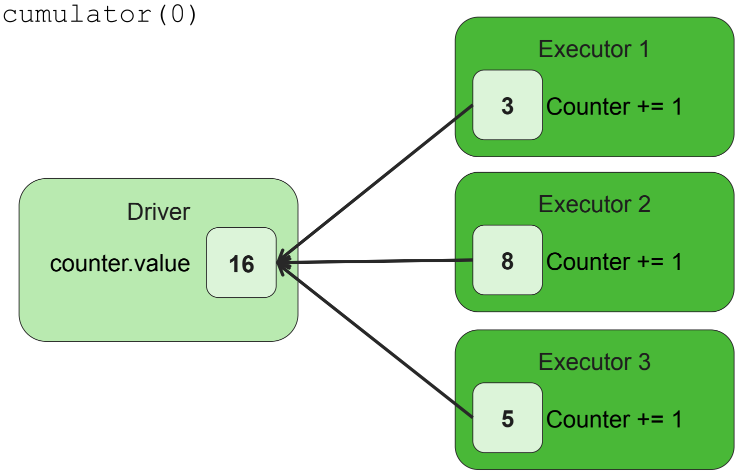
- Use accumulators
- Use broadcast variables

- Accumulators

Objectives

Accumulators

```
counter = sc.accumulator(0)
rdd.foreach(...
   counter += 1
)
counter.value()
```



Accumulators

- Accumulator = A variable that is only “added” to through an associated operation, and can therefore be efficiently supported in parallel.
- Accumulators can be used to implement counters (as in MapReduce) or sums.
- Only the driver can access the value.
  - Updates are sent to the driver, will get an exception if you use the `.value` on executors
- Spark natively supports accumulators of numeric types, and developers can add support for new types.
  - Doubles
  - Floats
  - Ints
- Most common uses
  - Count events that occur, like invalid records

Accumulators and Fault Tolerance

- Spark automatically deals with failed or slow machines by re-executing failed or slow tasks.
- Accumulators are returned at the end of successful tasks
- For accumulators used in actions, Spark applies each task's update to each accumulator only once
  - If a reliable counter is required, they must be used in an action, like foreach()
- For accumulators used in transformations, the guarantee does not exist
  - Transformations can happen more than once in an action, if there are slow or failed tasks
  - Accumulators in transformations should only be used for debugging

Accumulator in Transformation Example

```
rdd=sc.textFile(myfile.txt)
blanklines = sc.accumulator(0) ## Create an Accumulator[Int]
initialized to 0
rddNotBlank = rdd.map(lambda line: \
   if not line:
      blanklines += 1
   else:
      line).map(lambda line: line.split(',')

rddNotBlank.saveAsTextFile("myfile.txt")
```


Accumulator in Action Example

```
val rdd=sc.textFile(myfile.txt)
//Create Accumulator[Int] initialized to 0
val blanklines = sc.accumulator(0)
val rddNotBlank = rdd.filter(line => !line.isEmpty)
rdd.foreach(line =>
   if (line.isEmpty){
      blanklines +=1
})
rdd.join(otherrdd).saveAsTextFile()
blanklines.value
rddNotBlank.saveAsTextFile("myfile.txt")
```

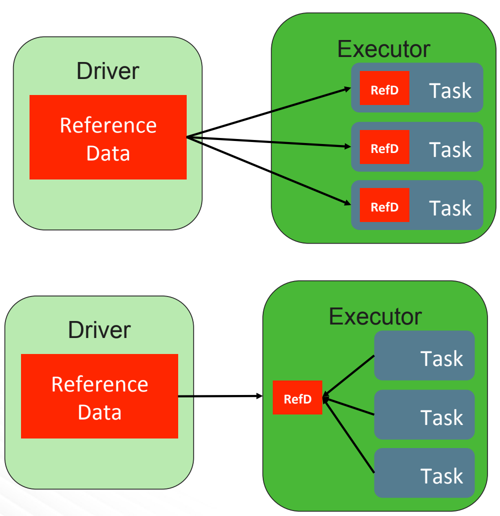
317

Lab: Using Accumulators to Check Data Quality

Objectives

- Using Accumulators
- Using Broadcast variables

How do Broadcast Variables Work?



- Without broadcast variables, reference data gets sent to every task on the executor, even though multiple tasks reuse the same variables.
- Using broadcast variables, Spark sends a copy to the node once, then the data is stored in memory. Each task will reference the local copy of the data.

Broadcast Variables

- Spark feature for sharing a variable throughout the application cluster
  - The broadcast variable must fit within an executor's memory
  - Not intended for RDDs or DataFrames
  - Immutable
- Give every node a copy of an input dataset in an efficient manner
  - Uses P2P torrenting concepts to efficiently distribute
  - Lazy - the first read of a broadcast variable will retrieve and store the data
  - Sent to each executor once

321

Why Use Broadcast Variables?

- Minimize network traffic by passing referenced variables to an executor only one time
  - Especially beneficial when local variables are 20kb or larger
- Complements Spark's task launching behavior for RDD programming, which is optimized for small tasks
  - Not used with Spark SQL

322

Implementing Broadcast Variables

```
rdd = sc.textFile(input.txt).map(….)…
toBroadcast = //some dictonary
lkp_bc = sc.broadcast(toBroadCast)
lookuprdd = rdd.map(lambda (key, value):
(key, lkp_bc.value[value])))
```


Lab: Using Broadcast Variables

Objectives

After completing this lesson, students should be able to:

- Control behavior and performance of Spark applications via:
  - `mapPartitions()` vs. `map()`
  - Modifying RDD parallelism / partitioning
  - Caching and persisting
  - Checkpointing
  - Using broadcast variables
  - Implementing joining strategies
  - Optimizing executors

Objectives

- `mapPartitions()` vs. `map()`

Improve Performance: `mapPartitions()` vs. `map()`

- `mapPartitions()` is a special kind of map transformation
  - Requires both input and output to be iterable
  - Operates at the RDD partition level, as opposed to the element level like `map()`
- Ex: Initialize a database with 2,000,000 elements spread across four RDD partitions
  - `map()` initializes each element individually, thus 2,000,000 initializations
  - `mapPartitions()` initializes each partition (four initializations total) and then can iterate through the elements in each partition
  - Can result in significant performance improvements

```
rdd1 = sc.parallelize((1,2,3,4,5,6,7,8),2)
rdd1.mapPartitions(lambda x: [sum(x)]).collect()
[(10, 26)]
```


map vs mapPartitions example cont.

- Lets fix this, and use a better parser -> Converting string to Array[String]

```
rdd = ##someRdd
rdd.mapPartitions(lambda lines: {
  myObject = simulateExpensiveOjectCreation()
  lines.map(lambda line: {
   myObject.map(lambda line: …
    })
  }).take(5).foreach(println)
```

- In this example we created a single instance of a an obect per partition, instead of per record.

```
def simulateExpensiveObjectCreation() {
  Thread sleep 10
}
```


Objectives

- `mapPartitions()` vs. `map()`
- Partition Optimization

PairRDD Parallelism – Hashed Partitions

- By default, files read into Spark are not necessarily organized so that matching keys are written to the same partition
  - Also, when `map()` is used to create a PairRDD, partitions are not reorganized
- PairRDDs **can be** partitioned so that matching keys are in the same partition
  - Can result in performance improvements, particularly when implementing joins
- Some operations create hashed partitions by design
  - `partitionBy()`, `cogroup()`, `join()`, `groupByKey()`, `reduceByKey()`, `sort()`
  - The default `HashPartitioner` guarantees identical keys go to same partition

Preserving Hashed Partitions

- Some Spark transformations maintain partitioning after hashing
  - No need to recreate hashed partitions after the first run
  - No new keys are created and partition placement is maintained
- Examples of operations that preserve hashed partitioning:
  - `mapValues()`, `flatMapValues()`, `filter()`, `reduceByKey()`, `groupByKey()`, and `join()`

Partitioning Optimization

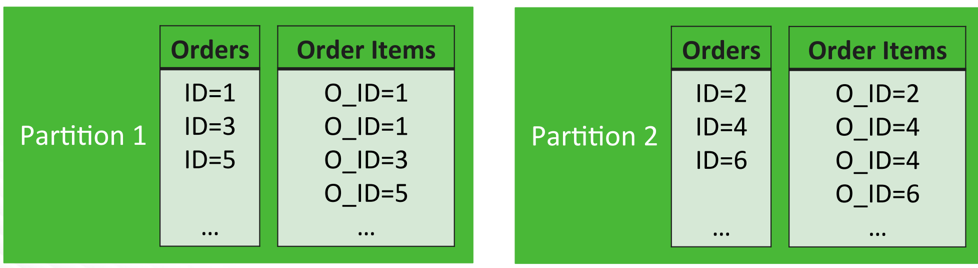
- Generally speaking, too many partitions is better than too few
  - Increase partitions by numbers by 50% until performance stops improving
  - Tasks will usually take at least 200ms
  - Scheduling tasks takes ~10ms-20ms regardless of the amount of data being processed
- Number of partitions should be a slightly less than a multiple of the number of executor cores
  - Ten executors with two cores each = RDDs with 39, 58, or 78 partitions
  - Reasons for a little less is to leave a couple cores open for speculative execution
- Spark SQL
  - uses "spark.sql.shufflePartitions" by default is 200
  - Best to have number partitions = output datasize / block size

Objectives

- `mapPartitions()` vs. `map()`
- Partition Optimization
- Joining Strategies

Spark Joins at the Partition Level

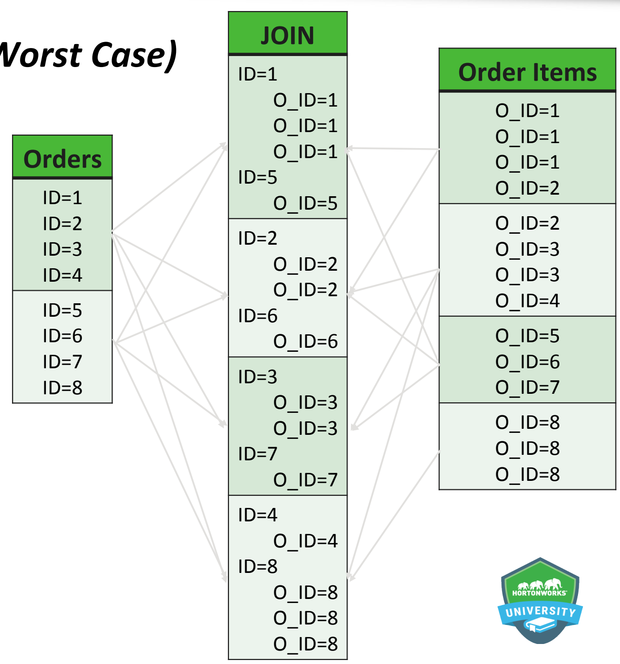
- To improve performance, joins technically occur at the partition, not the complete dataset, level
  - Framework ensures that join key placement in partitions aligns with all datasets
  - The collective results represent the comprehensive join request
- Leverages hash partitions and requires equal number of partitions for datasets to be joined



335

No Common Hash Partitioning *(Worst Case)*

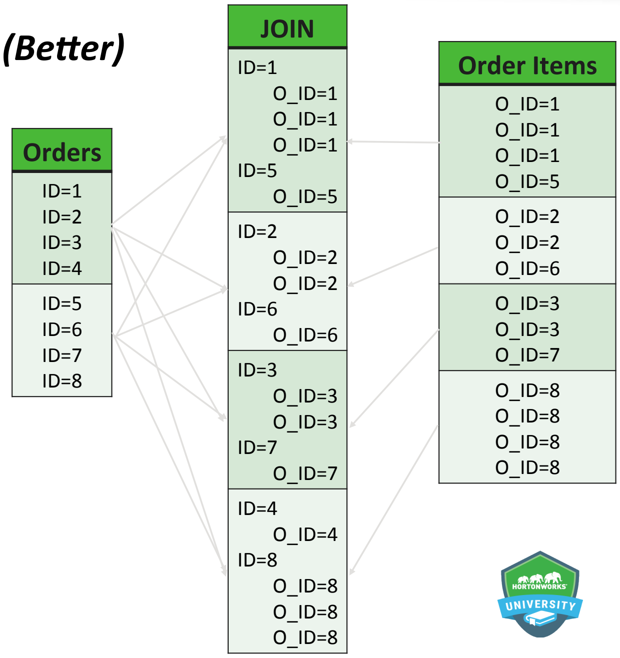
- Neither dataset is partitioned by the join key
- Both have to perform a `partitionBy()` transformation
  - Incurs a shuffle for each
- NOTE: The newly created hashed partitioned datasets use the number of partitions from the largest original



336

One Dataset Hashed Partitioned *(Better)*

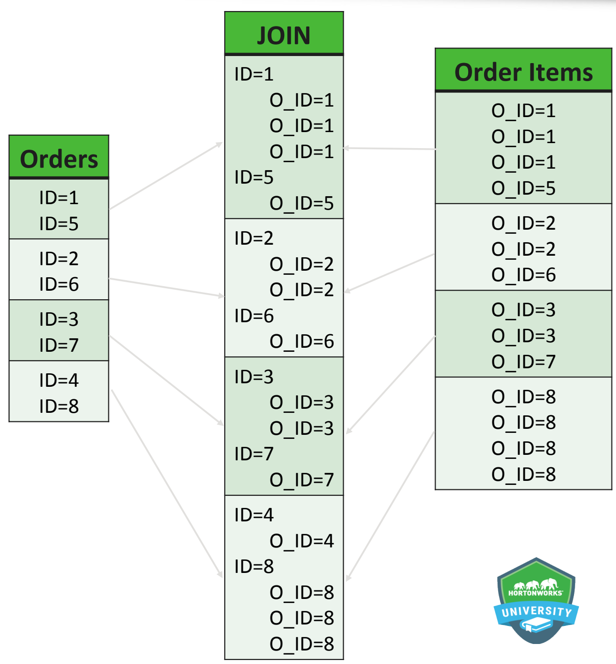
- The larger dataset is hashed partitioned by the join key
- The smaller one performs a `partitionBy()` transformation
  – Only one shuffle is required



337

Co-Partitioned *(Best Case)*

- Both datasets are hashed partitioned by the join key with the same number of partitions
  – Referred to as co-partitioned join
- **No shuffles are required**
  – This is a narrow operation!
  – Significant performance gains



338

Consideration and Guidelines

- RDD developers needs to consider order of operations
  – DataFrames benefit from Catalyst interventions, so not as critical here
- If possible, filter out irrelevant data from large datasets before joining to a smaller one
  – Will require less shuffling to occur during the join
- Cache any hash partitioned datasets that will be used in a subsequent join
  – Prevents the need to rebuild the intermediary dataset for each join

Objectives

- `mapPartitions()` vs. `map()`
- Partition Optimization
- Joining Strategies
- Executor Optimization and Memory Management and YARN

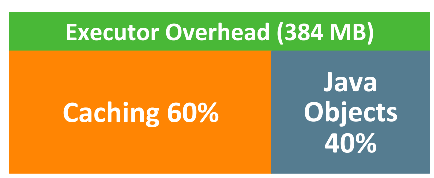
Memory Management Overview

- Memory usage in Spark falls under one of two categories
  - Execution
    - Memory used for computations in shuffles, joins, sorts and aggregations
  - Storage
    - Memory used for caching across the cluster

- New in Spark 1.6 is a unified memory region. Memory can be shared between the execution and storage. Storage takes a lower precedence, that is objects stored in storage can be evicted when memory is required by the execution side of things. We can set minimums (like YARN fair scheduler) on the amount of memory storage must have available.

Executor Optimization (Legacy)

- Executors are made up of three regions:
  - Overhead (384MB by default)
  - Space reserved for caching (60%)
  - Space reserved for java objects (40%)



- Data that will be cached is stored in the area for caching
  - The remaining will be used for creating objects

- Three main configurations to make up executor

```
--executor-memory
--executor-cores
--num-executors
```


Memory Management in Spark 1.6+

- Configs depreciated in Spark 1.6

```
spark.shuffle.memoryFraction
spark.storage.memoryFraction
spark.storage.unrollFraction
spark.memory.useLegacyMode (false, can be set to true to use old ways)
```

- Spark moved from a rigid memory structure to a more fluid, and (can leverage) off-heap managed memory
- Pros:
  - Less garbage collection
  - Less wasted resources
- Cons:
  - Data stored off-heap is slower to access than on-heap (still MUCH faster than disk)
  - Everything must be serialized/deserialized (encoders are available for primatives/java beans currently)
  - More knobs to tune

Using off-heap Memory

- Important new configs:
  - `spark.memory.offHeap.enabled (false by default)`
    - If true, Spark will attempt to use off-heap memory for certain operations. If off-heap memory use is enabled, then spark.memory.offHeap.size must be positive.
  - `spark.memory.offHeap.size (0 by default)`
  - `spark.memory.fraction (0.6 by default)`
  - `spark.memory.storageFraction`

Configuring Executors

- `executor-memory`
  - Should be between 8GB and 64GB
- `executor-cores`
  - At least 2, max 4
- `num-executors`
  - This is the most flexible
  - If caching data, desirable to have datasize * 2 as the total application memory

- EXAMPLE: YARN nodes with 128GB and 16 cores available would support a relatively common 16GB-memory / 2-core executor size
  - If caching a 100GB dataset, 13 executors could be ideal

Spark on YARN and Resource Request Implications

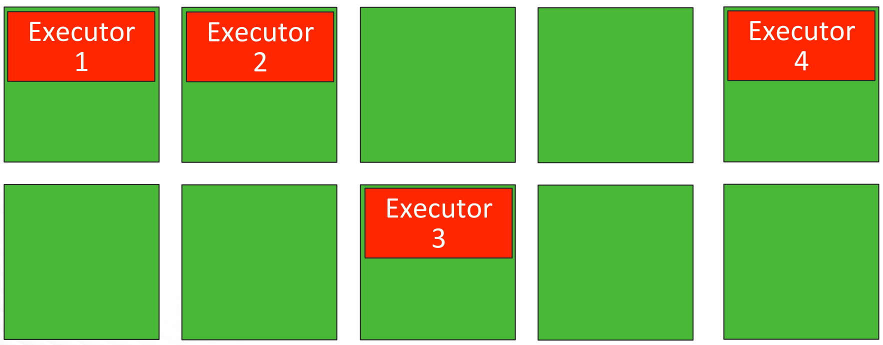
- Spark Applications generally have a higher resource usage footprint.
  - More RAM
  - More CPU's per worker
  - Larger JVM's
- General recommendation on resource requesting:
  - Fewer, larger executors are generally better than many smaller ones
    - Minimize shuffle
    - More data available locally to the worker
    - Less overhead

Spark on YARN and Resource Request Implications cont.

- Every machine has a finite amount of memory and CPU's available to it
- The larger the requested container, the harder it is to find a spot to allocate

Ex. Request 4 – 40gb RAM executors on a 10 node, 100gb RAM/machine cluster, no load

Easily to find resources for the application to run

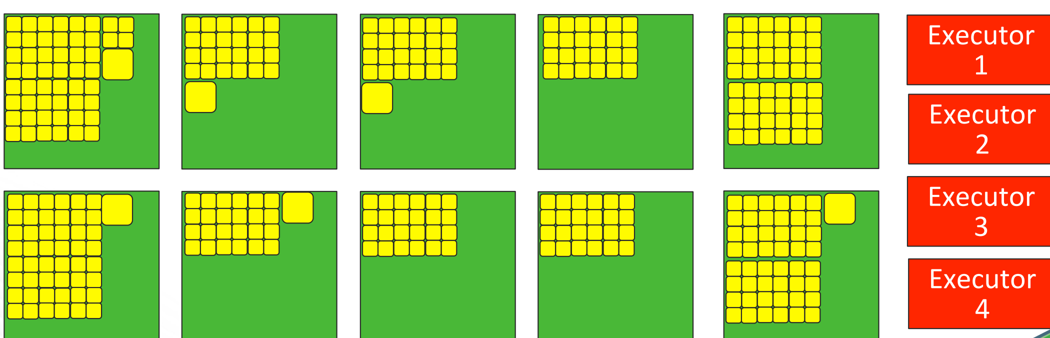


Spark on YARN and Resource Request Implications cont.

- Every machine has a finite amount of memory and CPU's available to it
- The larger the requested container, the harder it is to find a spot to allocate

Ex. Request 4 – 40gb RAM executors on a 10 node, 100gb RAM/machine cluster, 50% load

Becomes less obvious where to run



Knowledge Check

Questions

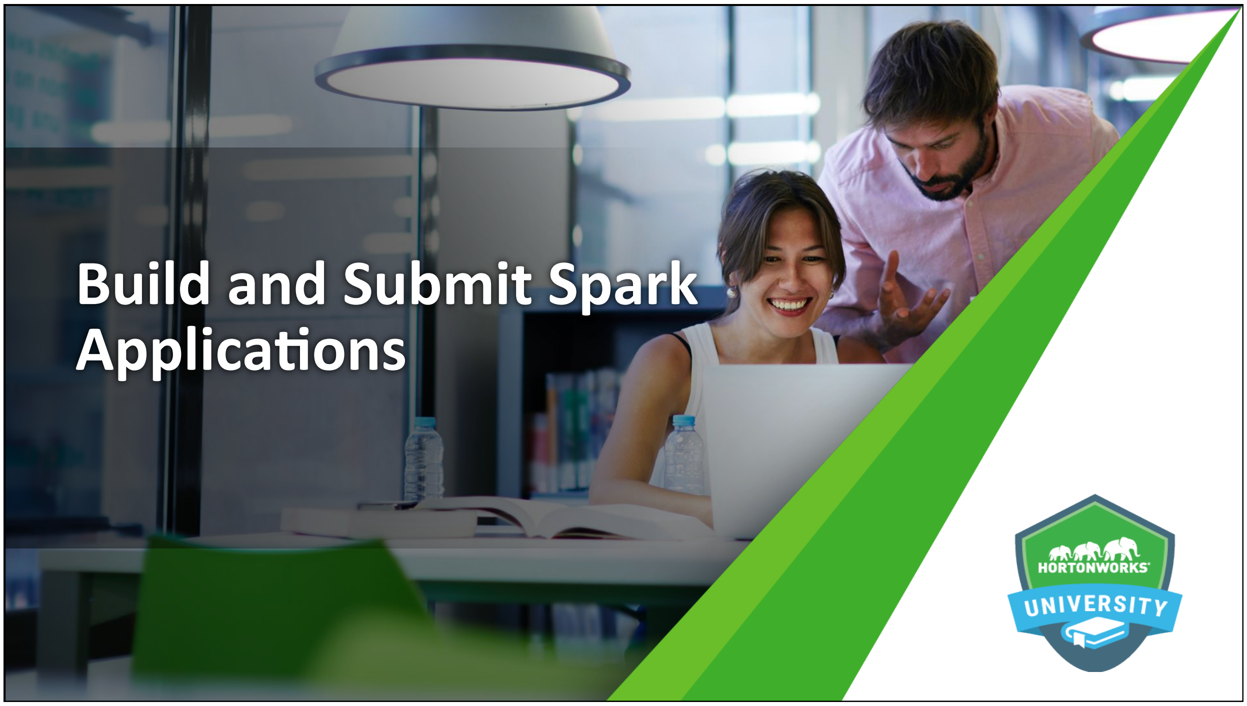
1. Why can mapPartitions be faster than map?
2. Why does preserving partition potentially make down stream operation faster?
3. Whats better, too many or to few partitions?
4. Is a lot of small executor, or fewer big ones ideal?

Summary

Summary

- mapPartitions() is similar to map() but operates at the partition instead of element level
- Controlling RDD parallelism before performing complex operations can result in significant performance improvements
- Caching uses memory to store data that is frequently used
- Checkpointing writes data to disk every so often, resulting in faster recovery should a system failure occur
- Broadcast variables allow tasks running in an executor to share a single, centralized copy of a data variable to reduce network traffic and improve performance
- Join operations can be significantly enhanced by pre-shuffling and pre-filtering data
- Executors are highly customizable, including number, memory, and CPU resources
- Spark SQL makes a lot of manual optimization unnecessary due to Catalyst

352



Build and Submit Spark Applications

Objectives

After completing this lesson, students should be able to:

- Create an application to submit to the cluster
- Describe client vs. cluster submission with YARN
- Submit an application to the cluster
- List and set important configuration items

354

Objectives

- Creating an Application to Submit to a Cluster

Zeppelin / REPLs vs. Spark Applications

- Zeppelin and REPLs allow for interactive manipulation, exploration, and testing
- Spark applications run as independent programs for production applications
  - Can be integrated into workflows managed by Falcon/Oozie
- The differences between them are minimal, making code reuse easy

Writing an Application to Submit to YARN

- Zeppelin and the REPLs take care of a few things automatically
  - Import the `SparkContext` and `SparkConf` libraries
  - Set up the main program
  - Create a Spark configuration object
  - Create and initialize a SparkContext instance
- For production applications, this must be coded by the developer
  - Can be accomplished in about five lines of code

Importing Libraries

- The user must code the import all the libraries used by the application
- All applications will need the `SparkContext` and `SparkConf` libraries in addition to basic libraries such as `sys` and `os`

```
import os
import sys
from pyspark import SparkContext, SparkConf
```

- To import other Spark libraries, its the same as any other application

```
from pyspark.sql import SQLContext
from pyspark.sql.types import Row, IntegerType
```

Creating a "main" Program

- The developer must set up the main program for the application

```
import os
import sys
from pyspark import SparkContext, SparkConf, SQLContext
if __name__ == "__main__":
   #Spark Programming
```


Creating a Spark Configuration

- The `SparkConf` configuration object is used by the context
  - It identifies the app name, resource manager, resources to request, etc.
- The developer must add the creation of the configuration to the application
- `SparkConf` supports pipelining as well as "setting" configuration properties

```
conf = SparkConf().setAppName("appName").setMaster("yarnMode")
conf.set('spark.executor.instances', '5')
conf.set('configuration', 'value')
```


Creating the SparkContext

- The `SparkContext` is used for the application to communicate to the cluster, request resources, and schedule tasks to be run
- The developer creates the context using the configuration object

```
sc = SparkContext(conf=conf)
```

- `SparkContext` has configurations that can be set after its been created

```
sc.setLogLevel("ERROR")
```

- Always stop the context at the end of the application
  - Ensures resources are properly released

```
sc.stop()
```


A Complete Application (Python)

```
import os
import sys
from pyspark import SparkContext, SparkConf

if __name__ == "__main__":

    conf = SparkConf()   .setAppName("appName").setMaster("yarnMode")

    sc = SparkContext(conf=conf)

    sc.textFile("dataFile.txt”)

    ## Spark Programming

    sc.stop()
```

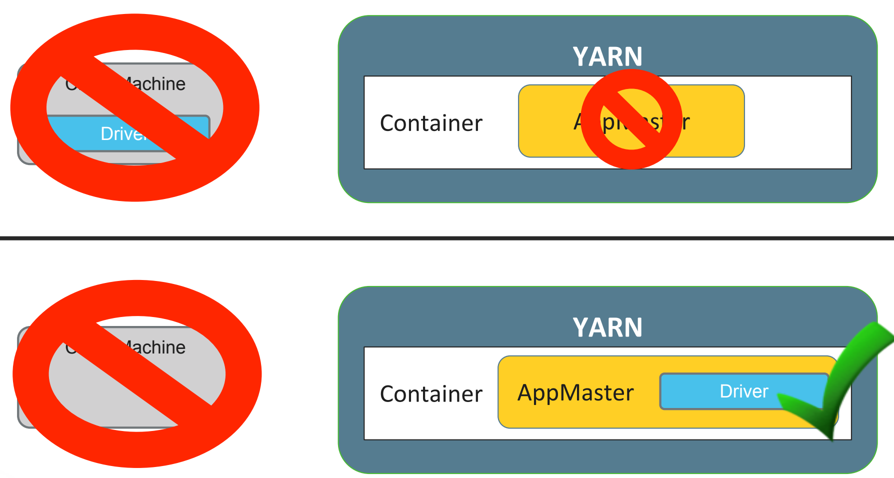
Objectives

- Creating an Application to Submit to a Cluster
- YARN Client vs. YARN Cluster

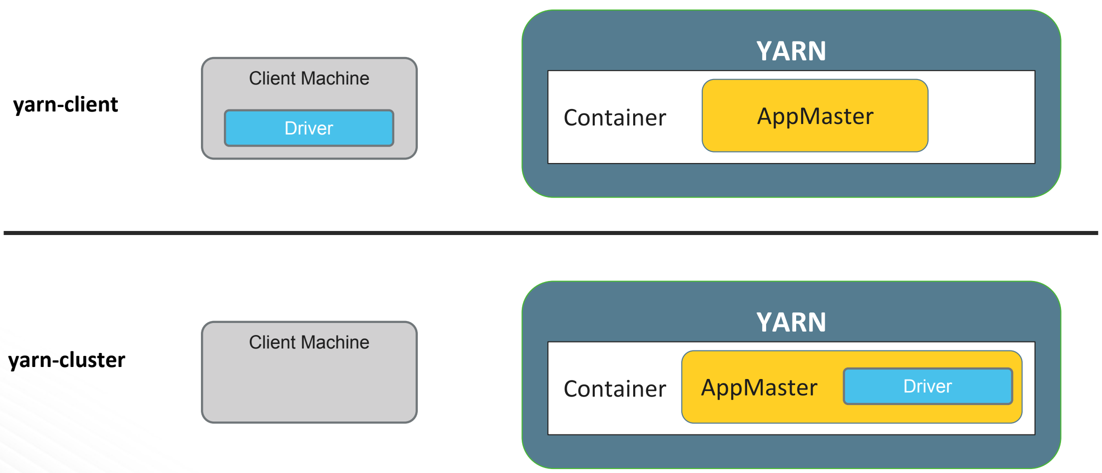
Spark Deployment Modes

yarn-client

yarn-cluster



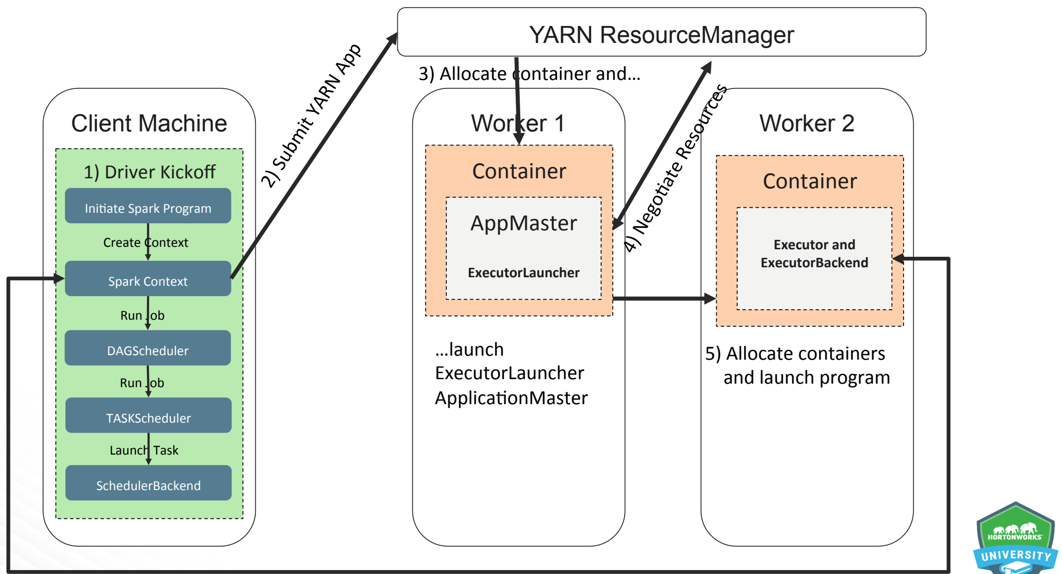
Spark Deployment Modes



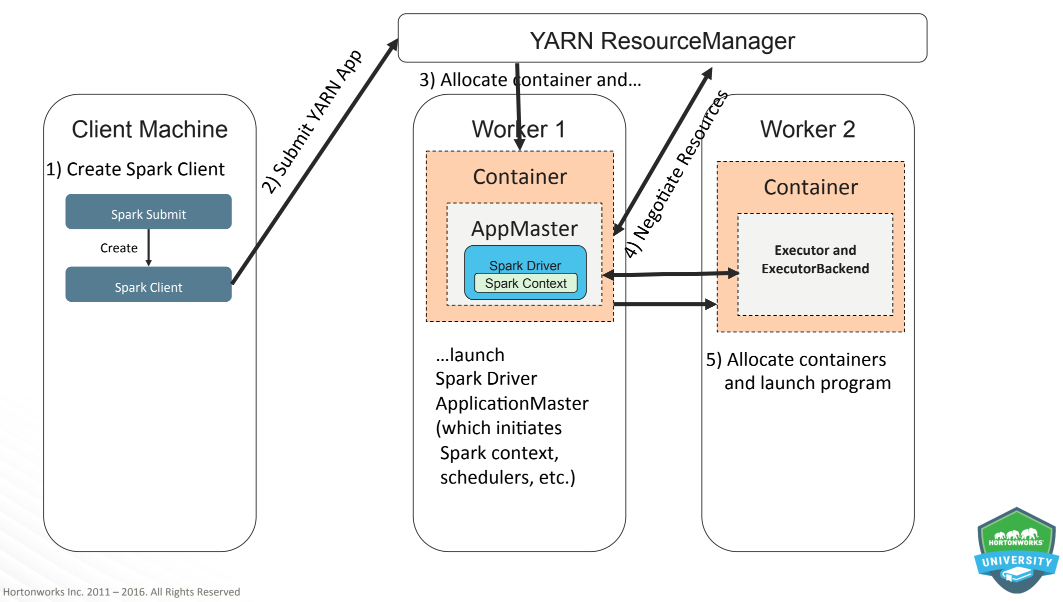
YARN Application Submission

- Spark YARN mode options:
  - `yarn-client`
  - `yarn-cluster`
- `yarn-client`
  - Developing applications
  - Testing of applications
  - REPLs and Zeppelin
- `yarn-cluster`
  - Running production applications

YARN Client Submission Process



YARN Cluster Submission Process



Objectives

- Creating an Application to Submit to a Cluster
- YARN Client vs. YARN Cluster
- Submitting an Application to YARN

Submitting an Application to YARN

- Spark uses the `spark-submit` command from the command line

```
spark-submit /path/to/sparkDemo.py
```

- Between `spark-submit` and the application file, the developer can add runtime configurations

```
--num-executors 2
--executor-memory 1g
--master yarn-cluster
--conf spark.executor.cores=2
```

- Arguments can be added after the file name

Using Different Versions of Python

- There are sometimes issue with the version of python you are using and what is configured on the cluster.

- Specifying the `PYSPARK_PYTHON` variable while submitting your application can fix this issue

```
PYSPARK_PYTHON=/usr/bin/python spark-submit \
   --master yarn-cluster sparkDemo.py
```


Application Submission Example

- Example of a `spark-submit` command:

```
spark-submit --master yarn-cluster --num-executors 4 \
   --executor-memory 8g /user/username/sparkDemo.py \
   /home/username/input.json /home/username/output.orc
```


Objectives

- Creating an Application to Submit to a Cluster
- YARN Client vs. YARN Cluster
- Submitting an Application to YARN
- Setting Important Configurations Items

Spark Configuration Hierarchy

- Can set the same configuration in multiple places
  – Where they are set will define which takes priority

Highest to least priority

1. Set inside the application
2. Set at runtime
3. Set in a configuration file passed to the application
4. Spark installation defaults located at `/etc/spark/conf/spark-defaults.conf`

- Documented at `spark.apache.org`

Setting Important Configuration Items at Runtime

Configurations with keywords

```
--num-executors 20
--executor-mem 8g
--executor-cores 2
--master yarn-client
--driver-memory 1g
```

Configuration set using `--conf key=value`

```
spark.shuffle.memoryFraction
spark.storage.memoryFraction
spark.default.parallelism
spark.speculation
```


Setting Important Configuration Items in Application

- These settings should always be set in the application
  – Don’t forget to register any custom classes with Kryo for non-Python applications

```
conf = SparkConf()
conf.set('spark.serializer',
   'org.apache.spark.serializer.KryoSerializer')
conf.set('spark.speculation','true')
```


Lab: Build and Submit YARN Applications

Knowledge Check

Questions

1. What components does the developer need to recreate when creating a Spark Application as opposed to using Zeppelin or a REPL?
2. What are the two YARN submission options the developer has?
3. What is the difference between the two YARN submission options?
4. When making a configuration setting, which location has the highest priority if the event of a conflict?
5. True or False: You should set your Python Spark SQL application to use Kryo serialization

Summary

Summary

- A developer must reproduce some of the back-end environment creation that Zeppelin and the REPLs handle automatically.
- The main differences between a `yarn-client` and `yarn-cluster` application submission is the location the Spark driver and `SparkContext`.
- Use `spark-submit`, with appropriate configurations, the application file, and necessary arguments, to submit an application to YARN.

Objectives

After completing this lesson, students should be able to:

- Describe the purpose of machine learning and some common algorithms used in it
- Describe the machine learning packages available in Spark
- Examine and run sample machine learning applications

Disclaimer

- This is not a Data Science class
- Fully utilizing the packages this lesson will discuss requires fundamental understandings of topics that go well beyond what will be covered
- Labs and suggested exercises will consist of pre-built scripts / applications that will demonstrate some of these topics in practice

- Machine Learning Basics

Objectives

Machine Learning Basics

- Machine learning attempts to find actionable patterns within data
- Creates a model, which is used to make predictions
- Two basic types of Machine Learning
  - Supervised
  - Unsupervised

Supervised Learning

- Most common type of machine learning
- A model is created that uses one or more variables to make a prediction, and then that prediction can be immediately tested to determine accuracy
- Two common types of predictions:
  - Classification: Yes or no, approve or reject, spam or safe, etc. - Will the flight depart on time?
  - Regression: What will the value be? - What time is the flight likely to depart?
- Breaks a dataset into two parts:
  - Training data: used to create the model
  - Testing data: used to determine model accuracy

Supervised Learning Example Dataset

Carrier	Airplane	Age	Airport	Time	Weather	StaffPerc	Sched	Actual
A	B	11	SFO	EarlyMorn	Clear	90	05:31	05:31
C	D	2	ORD	Morn	Windy	84	08:14	09:35
A	D	7	ATL	EarlyAft	Cloudy	100	12:05	12:05
D	D	14	ORD	Aft	Rain	100	15:21	15:45
B	A	4	JFK	EarlyEve	Stormy	94	17:00	19:20
C	B	6	BWI	Eve	Warnings	80	20:42	CANCEL
A	D	2	HDP	LateEve	Clear	100	22:00	22:00
E	D	10	STL	RedEye	Stormy	93	23:45	CANCEL
C	B	8	DAL	Aft	Rain	99	14:10	14:10
C	E	8	SJC	Morn	Clear	98	09:34	10:15

Thousands upon thousands of data points collected and available every day - massive historical data to work from

Terminology

- Each row in the dataset is called an "observation"
- Each column in the dataset is called a "feature"
- Columns selected for inclusion in the model are called "target variables"

Supervised Learning Example Workflow

- Randomly break data into two parts for training vs. test data
  – In Spark, extremely large datasets can be used due to availability of cluster resources
- Pick one or more variables to use to build model
  – For example: airplane age, weather, and airport
  – Pick too few and the model may not be accurate enough
  – Pick too many and the model is only accurate for the training data
- Run machine learning algorithm to build model based on those variables
- Run the model against the test data and see how accurately it predicts results
  – Then go back and alter variables, build new model, and test again until satisfied

Supervised Learning - Examining Results

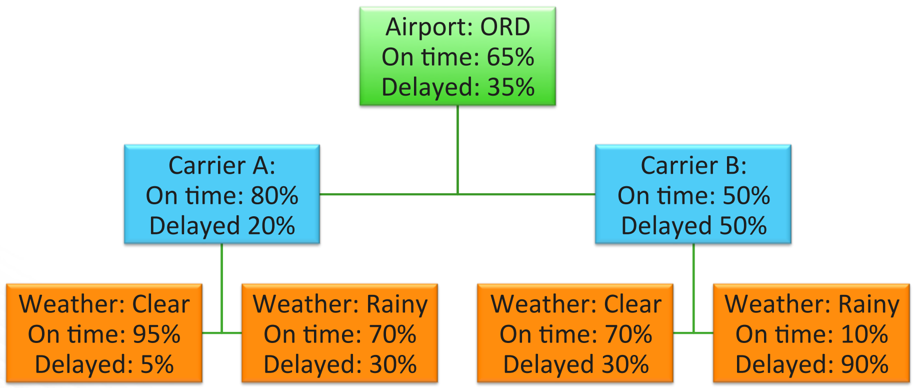
- For classification results, prediction is either correct or incorrect
  - Will the flight depart on time?
  - Percentage accuracy against testing data determines strength of the model
- For regression results, prediction will often be inexact, but better models produce closer predictions when compared to actual results on test data
  - What time will the flight leave? How far off is the prediction?
  - Minimal "sum of means squared error" determines strength of the model

Sum of Means Squared Error

- Simple example: four observations, two models with an average (mean) variance of 2
  - Model A variance by observation: 0, 4, 0, 4
  - Model B variance by observation: 1, 3, 2, 2
- Intuitively, even though Model A gets it exactly right more often, it also gets it more wrong consistently as well
- Sum of means squared takes each value and squares it, then adds them together
  - Larger variance values get an exponential penalty
  - Model A sum of means squared = 0 + 16 + 0 + 16 = 32
  - Model B sum of means squared = 1 + 9 + 4 + 4 = 18
- Thus, Model B is determined to be a better fit

Decision Tree Algorithm

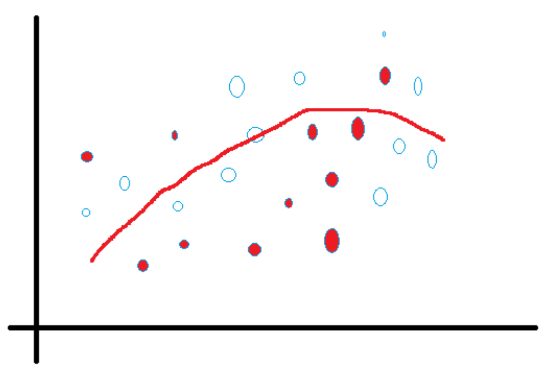
- Popular, multi-stage classification algorithm that classifies based on one variable, then drills deeper by adding another variable, and repeats process for additional variables



393

Classification Algorithms

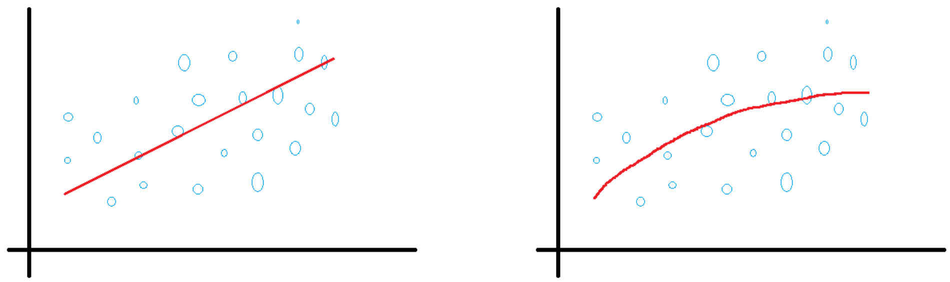
- Draw a line attempting to define the boundary between the two possible options



394

Linear Regression Algorithm

- Draw a line that has the best fit to the data



Unsupervised Learning

- Supervised learning is great as long as data is labeled, but what about data where the label is unknown?
- Example: Product reviews on social media and web sites
  - No explicit positive or negative label
  - How can we group them to determine whether general consensus is positive or negative?
- Goal is to find patterns in data that allow it to be labeled
  - Example: Group 1 = when review contains phrase X, it will also usually contain phrase Y
  - Upon evaluation, group that contains phrase Y are positive reviews
- Most common type is clustering

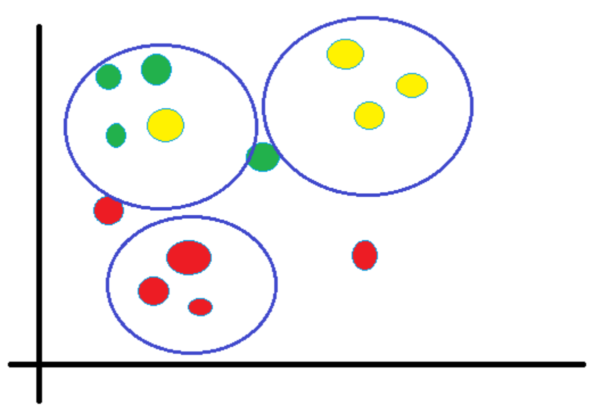
Unsupervised Learning Example Dataset

Phrase1	Phrase2	Phrase3
did not like	had a nice	it was ok
i loved this	awesome place to	will be back
would not recommend	will not return	did not like
would definitely recommend	i loved this	service was good
could not stand	would not recommend	had a nice
service was excellent	food was cold	not sure if
service was good	will be back	hard to find
was a dump	food was outstanding	might try again
food was cold	did not like	will not return
server was friendly	was not able	hard to find

Data is cleaned of extraneous phrases - search for patterns so that reviews can be grouped without knowing outcome

K-Means Algorithm

- Identify groupings that likely share the same label



Other Popular Algorithms

- Classification
  - Support Vector Machine (SVM)
  - Logistic Regression
  - Naïve Bayes
- Clustering
  - K-Nearest Neighbors
- Dimensionality Reduction / Decomposition
  - Help determine target variables when dataset contains large number of features
  - Principal Component Analysis (PCA)
  - Singular Value Decomposition (SVD)
- Collaborative Filtering / Recommendation
  - Used to predict results based on collaborative data
  - Alternating Least Squares

399

Objectives

- Machine Learning Basics
- Spark Machine Learning Libraries

400

Spark Machine Learning Overview

- Spark implementations of common learning algorithms and utilities
  - Allows traditional data science work to be performed on cluster-scale data
- Two packages available
  - `spark.mllib`: operates on RDDs
  - `spark.ml`: operates on DataFrames
- Both contain modules with various functions and sub-functions which provide powerful machine learning capabilities

`mllib` Modules

- classification
- clustering
- evaluation
- feature
- fpm
- linalg*
- optimization
- pmml
- random
- recommendation
- regression
- stat*
- tree*
- util

`ml` Modules

- attribute
- classification
- clustering
- evaluation
- feature
- param
- recommendation
- regression
- source.libsvm
- tree*
- tuning
- util

Spark Machine Learning Advantages

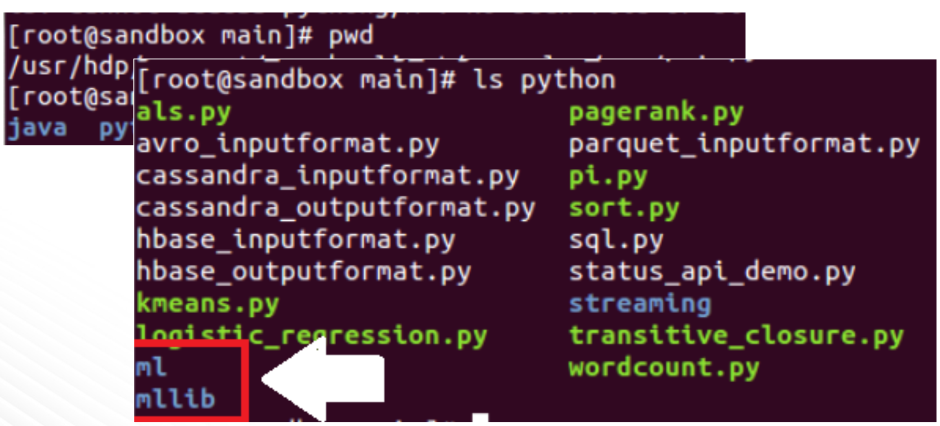
- Cluster-level data processing capabilities
  – Datasets not limited to what can fit into local memory
  – Built-in parallel processing over many machines at one time
- In-memory processing
  – Improved performance vs. older Hadoop machine learning libraries
- `ml` advantages over `mllib`
  – `ml` operates on DataFrames
  – Greater flexibility
  – Automatic performance enhancements via Catalyst
  – Create reusable machine learning pipelines

Objectives

- Machine Learning Basics
- Spark Machine Learning Libraries
- Machine Learning Sample Applications

Machine Learning Sample Applications

- Installed automatically when Spark is installed
  - `/usr/hdp/current/spark-client/examples/src/main/<language>/<mllib | ml>/`



Machine Learning Sample Application Files

```
[root@sandbox main]# ls python/ml/
aft_survival_regression.py
binarizer_example.py
bucketizer_example.py
cross_validator.py
dataframe_example.py
decision_tree_classification_example.py
decision_tree_regression_example.py
elementwise_product_example.py
gradient_boosted_tree_classifier_example.py
gradient_boosted_tree_regressor_example.py
index_to_string_example.py
kmeans_example.py
linear_regression_with_elastic_net.py
logistic_regression_with_elastic_net.py
multilayer_perceptron_classification.py
n_gram_example.py
normalizer_example.py
onehot_encoder_example.py
pca_example.py
polynomial_expansion_example.py
random_forest_classifier_example.py
random_forest_regressor_example.py
rformula_example.py
simple_params_example.py
simple_text_classification_pipeline.py
```

```
[root@sandbox main]# ls python/mllib/
binary_classification_metrics_example.py
correlations.py
decision_tree_classification_example.py
decision_tree_regression_example.py
fpgrowth_example.py
gaussian_mixture_model.py
gradient_boosting_classification_example.py
gradient_boosting_regression_example.py
isotonic_regression_example.py
kmeans.py
logistic_regression.py
multi_class_metrics_example.py
multi_label_metrics_example.py
naive_bayes_example.py
random_forest_classification_example.py
random_forest_regression_example.py
random_rdd_generation.py
ranking_metrics_example.py
recommendation_example.py
regression_metrics_example.py
sampled_rdds.py
word2vec.py
```

`mllib` Decision Tree Classification Example

```
  GNU nano 2.0.9 File: ...decision_tree_classification_example.py

from pyspark.mllib.tree import DecisionTree, DecisionTreeModel
from pyspark.mllib.util import MLUtils
# $example off$

if __name__ == "__main__":

    sc = SparkContext(appName="PythonDecisionTreeClassificationExample")

    # $example on$
    # Load and parse the data file into an RDD of LabeledPoint.
    data = MLUtils.loadLibSVMFile(sc, 'data/mllib/sample_libsvm_data.txt')
    # Split the data into training and test sets (30% held out for testing)
    (trainingData, testData) = data.randomSplit([0.7, 0.3])

    # Train a DecisionTree model.
    #  Empty categoricalFeaturesInfo indicates all features are continuous.
    model = DecisionTree.trainClassifier(trainingData, numClasses=2, catego$
                                         impurity='gini', maxDepth=5, maxBi$

    # Evaluate model on test instances and compute test error
    predictions = model.predict(testData.map(lambda x: x.features))
    labelsAndPredictions = testData.map(lambda lp: lp.label).zip(prediction$
    testErr = labelsAndPredictions.filter(lambda (v, p): v != p).count() / $
    print('Test Error = ' + str(testErr))
    print('Learned classification tree model:')
```

`ml` Logistic Regression Example

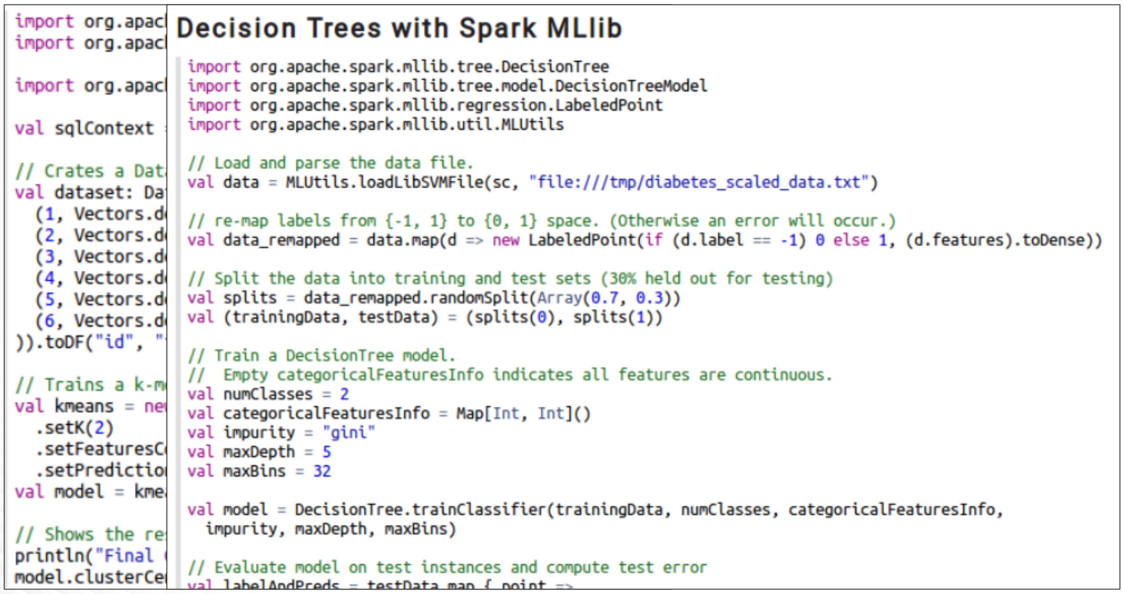
```
GNU nano 2.0.9 File: ...logistic_regression_with_elastic_net.py

if __name__ == "__main__":
    sc = SparkContext(appName="LogisticRegressionWithElasticNet")
    sqlContext = SQLContext(sc)

    # $example on$
    # Load training data
    training = sqlContext.read.format("libsvm").load("data/mllib/sample_lib$

    lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

    # Fit the model
    lrModel = lr.fit(training)

    # Print the coefficients and intercept for logistic regression
    print("Coefficients: " + str(lrModel.coefficients))
    print("Intercept: " + str(lrModel.intercept))
    # $example off$

    sc.stop()
```


K-Means Clustering E

```
GNU nano 2.0.9          File

import sys

import numpy as np
from pyspark import SparkCo
from pyspark.mllib.clusteri


def parseVector(line):
    return np.array([float(x


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: kmeans
        exit(-1)
    sc = SparkContext(appNam
    lines = sc.textFile(sys.
    data = lines.map(parseVe
    k = int(sys.argv[2])
    model = KMeans.train(dat
    print("Final centers: "
    print("Total Cost: " + s
```

```
GNU nano 2.0.9          File: python/ml/kmeans_example.py

from pyspark.ml.clustering import KMeans, KMeansModel
from pyspark.mllib.linalg import VectorUDT, _convert_to_vector
from pyspark.sql import SQLContext
from pyspark.sql.types import Row, StructField, StructType

"""
A simple example demonstrating a k-means clustering.
Run with:
  bin/spark-submit examples/src/main/python/ml/kmeans_example.py <input> <k>

This example requires NumPy (http://www.numpy.org/).
"""


def parseVector(line):
    array = np.array([float(x) for x in line.split(' ')])
    return _convert_to_vector(array)


if __name__ == "__main__":

    FEATURES_COL = "features"

    if len(sys.argv) != 3:
        print("Usage: kmeans_example.py <file> <k>", file=sys.stderr)
```


Zeppelin Machine Learning Lab Note



Lab: Machine Learning Walkthrough

Knowledge Check

Questions

1. What are two types of machine learning?
2. What are two types of supervised learning?
3. What do you call columns that are selected as variables to build a machine learning model?
4. What is a row of data called in machine learning?
5. What is the goal of unsupervised learning?
6. Name the two Spark machine learning packages.
7. Which machine learning package is designed to take advantage of flexibility and performance benefits of DataFrames?
8. Name two reasons to prefer Spark machine learning over other alternatives

Summary

Summary

- Spark supports machine learning algorithms running in a highly parallelized fashion using cluster-level resources and performing in-memory processing
- Supervised machine learning builds a model based on known data and uses it to predict outcomes for unknown data
- Unsupervised machine learning attempts to find grouping patterns within datasets
- Spark has two machine learning packages available
  - `mllib` operates on RDDs
  - `ml` operates on DataFrames
- Spark installs with a collection of sample machine learning applications


Thank You
HORTONWORKS
UNIVERSITY