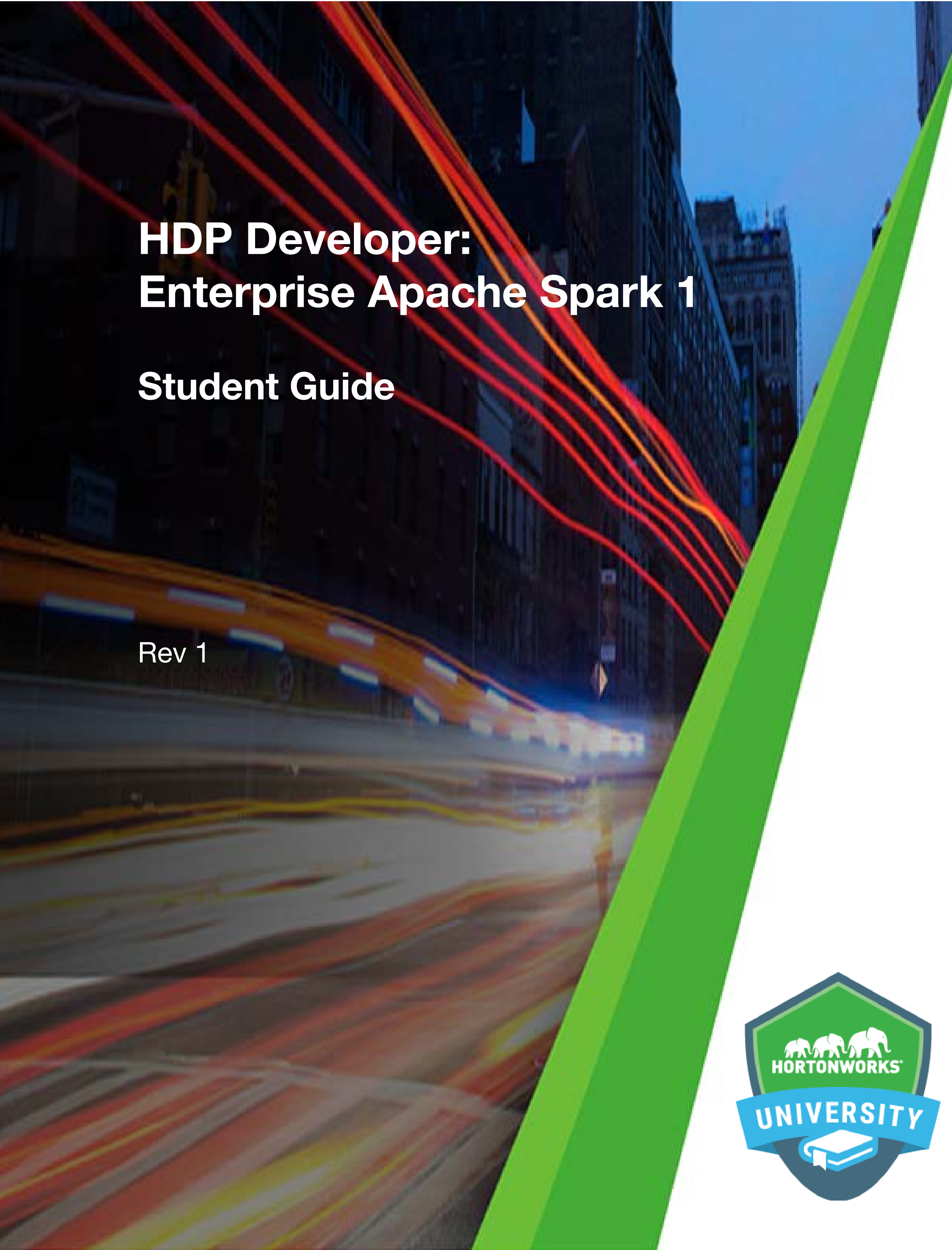


HDP Developer: Enterprise Apache Spark 1

Student Guide

Rev 1

Accumulo	HBase	Phoenix
Ambari	HDFS	Pig
Apache	Hive	Ranger
Atlas	Kafka	Slider
Beam	Karaf	Solr
Cassandra	Knox	Spark
CloudStack	Mahout	Sqoop
Falcon	MapReduce	Storm
Flink	Maven	Tez
Flume	Metron	Tomcat
Hadoop	MiniFi	WebHDFS
HAWQ	NiFi	YARN
HBase	Oozie	Zeppelin
HDFS	ORC	Zookeeper

Become a ***Hortonworks Certified Professional*** and establish your credentials:

• HDP Certified Developer: for Hadoop developers using frameworks like Pig, Hive, Sqoop and Flume.

• HDP Certified Administrator: for Hadoop administrators who deploy and manage Hadoop clusters.

• HDP Certified Developer: Java: for Hadoop developers who design, develop and architect Hadoop-based solutions written in the Java programming language.

• HDP Certified Developer: Spark: for Hadoop developers who write and deploy applications for the Spark framework.

How to Register: Visit www.examslocal.com and search for “Hortonworks” to register for an exam. The cost of each exam is $250 USD, and you can take the exam anytime, anywhere using your own computer. For more details, including a list of exam objectives and instructions on how to attempt our practice exams, visit http://hortonworks.com/training/certification/

Earn Digital Badges: Hortonworks Certified Professionals receive a digital badge for each certification earned. Display your badges proudly on your résumé, LinkedIn profile, email signature, etc.

Self Paced Learning Library

On Demand Learning

Hortonworks University Self-Paced Learning Library is an on-demand dynamic repository of content that is accessed using a Hortonworks University account. Learners can view lessons anywhere, at any time, and complete lessons at their own pace. Lessons can be stopped and started, as needed, and completion is tracked via the Hortonworks University Learning Management System.

Hortonworks University courses are designed and developed by Hadoop experts and provide an immersive and valuable real world experience. In our scenario-based training courses, we offer unmatched depth and expertise. We prepare you to be an expert with highly valued, practical skills and prepare you to successfully complete Hortonworks Technical Certifications.

Target Audience: Hortonworks University Self-Paced Learning Library is designed for those new to Hadoop, as well as architects, developers, analysts, data scientists, and IT decision makers. It is essentially for anyone who desires to learn more about Apache Hadoop and the Hortonworks Data Platform.

Duration: Access to the Hortonworks University Self-Paced Learning Library is provided for a 12-month period per individual named user. The subscription includes access to over 400 hours of learning lessons.

The online library accelerates time to Hadoop competency. In addition, the content is constantly being expanded with new material, on an ongoing basis.

Visit: http://hortonworks.com/training/class/hortonworks-university-self-paced-learning-library/

Table of Contents

HDP Overview for Developers

Lesson 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

What makes data big? Where did the phrase Big Data come from and what does it mean?

The term Big Data comes from the computational sciences. Specifically, it is used to describe scenarios where the volume and variety of data types overwhelm the existing tools to store and process it.

In 2001, the industry analyst Doug Laney described Big Data using the three V's of volume, velocity, and variety.

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.

Volume

Volume refers to the amount of data being generated. Think in terms of gigabytes, terabytes, and petabytes. Many systems and applications are just not able to store, let alone ingest or process, that much data.

Many factors contribute to the increase in data volume. This includes transaction-based data stored for years, unstructured data streaming in from social media, and the ever increasing amounts of sensor and machine data being produced and collected.

There are problems related to the volume of data. Storage cost is an obvious issue. Another problem is filtering and finding relevant and valuable information in large quantities of data that often contains not-valuable information.

You also need a solution to analyze data quickly enough in order to maximize business value today and not just next quarter or next year.

Velocity

Velocity refers to the rate at which new data is created.   Think in terms of megabytes per second and gigabytes per second.

Data is streaming in at unprecedented speed and must be dealt with in a timely manner in order to extract maximum value from the data. Sources of this data include logs, social media, RFID tags, sensors, and smart metering.

There are problems related to the velocity of data. These include not reacting quickly enough to benefit from the data. For example, data could be used to create a dashboard that could warn of imminent failure or a security breach. Failure to react in time could lead to service outages.

Another problem related to the velocity of data is that data flows tend to be highly inconsistent with periodic peaks. Causes include daily or seasonal changes or event-triggered peak loads. For example, a change in political leadership could cause an a peak in social media.

Variety

Variety refers to the number of types of data being generated. Varieties of data include structured, semi-structured, and unstructured data arriving from a myriad of sources. Data can be gathered from databases, XML or JSON files, text documents, email, video, audio, stock ticker data, and financial transactions.

There are problems related to the variety of data. This include how to gather, link, match, cleanse, and transform data across systems. You also have to consider how to connect and correlate data relationships and hierarchies in order to extract business value from the data.

Common Types of Data in Hadoop

Six types of data are commonly found in Hadoop. These include sentiment data, clickstream data, sensor or machine data, geographic/geolocation data, server log data, and text.

The type of Big Data that ends up in Hadoop typically fits into one of the following categories:

- **Sentiment:** Understand how your customers feel about your brand and products right now
- **Clickstream:** Capture and analyze website visitor's data trails and optimize your website
- **Sensor/Machine:** Discover patterns in data streaming automatically from remote sensors and machines
- **Geographic:** Analyze location-based data to manage operations where they occur
- **Server Logs:** Research log files to diagnose and process failures and prevent security breaches
- **Text:** Understand patterns in text across millions of web pages, emails and documents

Sentiment

Understand how your customers feel about your brand and products right now

Sentiment data is unstructured data containing opinions, emotions, and attitudes. Sentiment data is gathered from social media like Facebook and Twitter. It is also gathered from blogs, online product reviews, and customer support interactions.

Enterprises use sentiment analysis to understand how the public thinks and feels about something. They can also track how those thoughts and feelings change over time.

It is used to make targeted, real-time decisions that improve performance and improve market share. Sentiment data may be analyzed to get feedback about products, services, competitors, and reputation.

Clickstream

Capture and analyze website visitor's data trails and optimize your website

Clickstream data is the data trail left by a user while visiting a Web site. Clickstream data can be used to determine how long a customer stayed on a Web site, which pages they most frequently visited, which pages they most quickly abandoned, along with other statistical information.

This data is commonly captured in semi-structured Web logs.

Clickstream data is used, for example, for path optimization, basket analysis, next-product-to-buy analysis, and allocation of Web site resources.

Hadoop makes it easier to analyze, visualize, and ultimately change how visitors behave on your Web site.

Sensor/Machine

Discover Patterns in data streaming automatically from remote sensors and machines

A sensor is a converter that measures a physical quantity and transforms it into a digital signal. Sensor data is used to monitor machines, infrastructure, or natural phenomenon.

Sensors are everywhere these days. They are on the factory floor and they are in department stores in the form of RFID tags. Hospitals use biometric sensors to monitor patients and other sensors to monitor the delivery of medicines via intravenous drip lines. In all cases these machines stream low-cost, always-on data.

Hadoop makes it easier for you to rapidly collect, store, process, and refine this data. By processing and refining your data you can identify meaningful patterns that provide insight to make proactive business decisions.

Geographic

Analyze location-based data to manage operations where they occur

Geographic/geolocation data identifies the location of an object or individual at a moment in time. This data may take the form of coordinates or an actual street address.

This data might be voluminous to collect, store, and process; just like sensor data. In fact, geolocation data is collected by sensors.

Hadoop helps reduce data storage costs while providing value-driven intelligence from asset tracking. For example, you might optimize truck routes to save fuel costs.

Server Log

Research log files to diagnose and process failures and prevent security breaches

Server log data captures system and network operation information. Information technology organizations analyze server logs for many reasons. These include the need to answer questions about security, monitor for regulatory compliance, and troubleshoot failures.

Hadoop takes server-log analysis to the next level by speeding and improving log aggregation and data center-wide analysis. In many environments Hadoop can replace existing enterprise-wide systems and network monitoring tools, and reduce the complexity and costs associated with deploying and maintaining such tools.

Text

Understand patterns in text across millions of web pages, emails and documents

Text is often used for text-based data generated that doesn’t neatly fit into one of the above categories, as well as combinations of categories in order to find patterns across different text-based sources.

Introduction to HDP

Hadoop is a collection of open source software frameworks for the distributed storing and processing of large sets of data. Hadoop development is a community effort governed under the licensing of the Apache Software Foundation. Anyone can help to improve Hadoop by adding features, fixing software bugs, or improving performance and scalability.

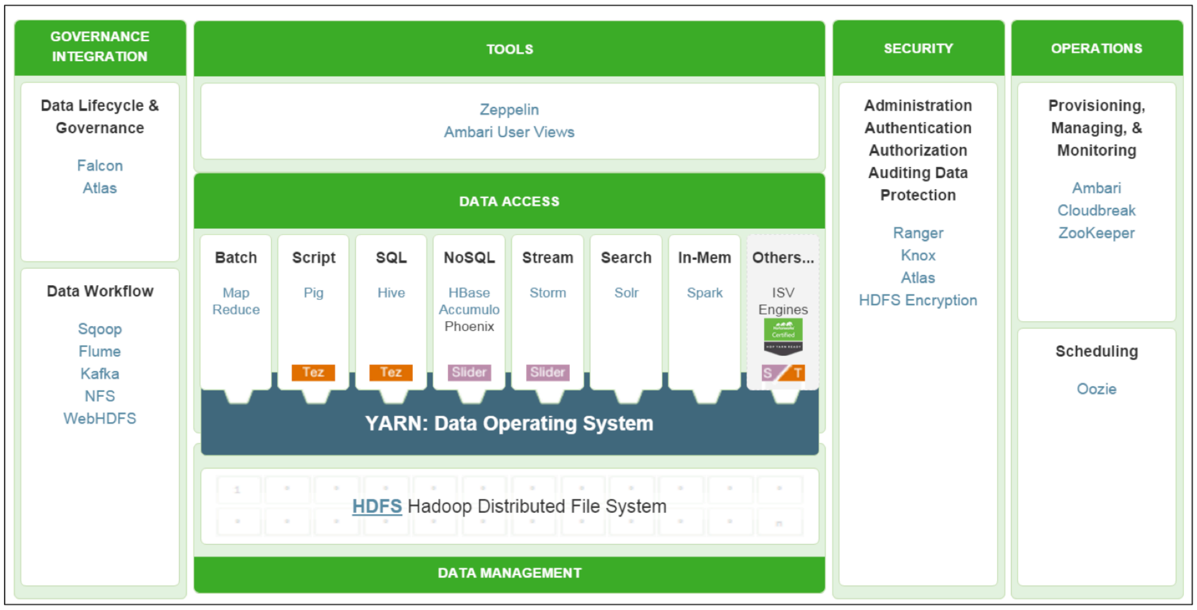
Hadoop clusters are scalable, ranging from a single machine to literally thousands of machines. It is also fault tolerant. Hadoop services achieve fault tolerance through redundancy.

Clusters are created using commodity, enterprise-grade hardware, which not only reduces the original purchase price, but potentially also reduces support costs too.

Hadoop also uses distributed storage and processing to achieve massive scalability. Large datasets are automatically split into smaller chunks, called blocks, and distributed across the cluster machines. Not only that, but each machine commonly processes its local block of data. This means that processing is distributed too, potentially across hundreds of CPUs and hundreds of gigabytes of memory.

HDP is an enterprise-ready collection of frameworks (sometimes referred to as the HDP Stack) that work within Hadoop that have been tested and are supported by Hortonworks for business clients.

Hortonworks Data Platform



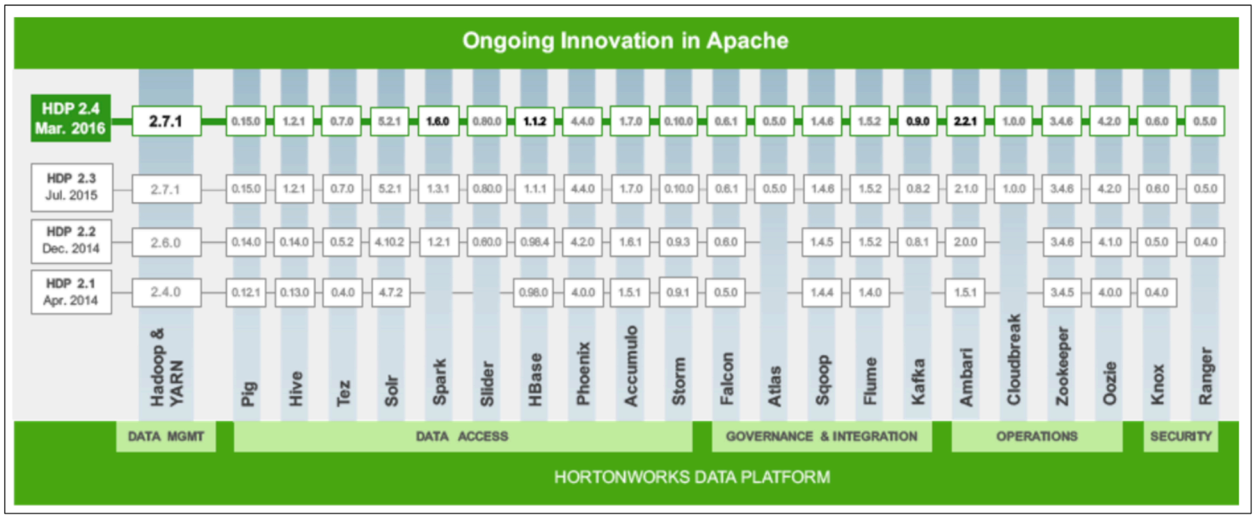
The Hortonworks Data Platform

Hadoop is not a monolithic piece of software. It is a collection of software frameworks. Most of the frameworks are part of the Apache software ecosystem. The picture illustrates the Apache frameworks that are part of the Hortonworks Hadoop distribution.

So why does Hadoop have so many frameworks and tools? The reason is that each tool is designed for a specific purpose. The functionality of some tools overlap but typically one tool is going to be better than others when performing certain tasks.

For example, both Apache Storm and Apache Flume ingest data and perform real-time analysis, but Storm has more functionality and is more powerful for real-time data analysis.

HDP Overview



	Hadoop & YARN	Pig	Hive	Tez	Solr	Spark	Slider	HBase	Phoenix	Accumulo	Storm	Falcon	Atlas	Sqoop	Flume	Kafka	Ambari	Cloudbreak	Zookeeper	Oozie	Knox	Ranger
HDP 2.4 Mar. 2016	2.7.1	0.15.0	1.2.1	0.7.0	5.2.1	1.6.0	0.80.0	1.1.2	4.4.0	1.7.0	0.10.0	0.6.1	0.5.0	1.4.6	1.5.2	0.9.0	2.2.1	1.0.0	3.4.6	4.2.0	0.6.0	0.5.0
HDP 2.3 Jul. 2015	2.7.1	0.15.0	1.2.1	0.7.0	5.2.1	1.3.1	0.80.0	1.1.1	4.4.0	1.7.0	0.10.0	0.6.1	0.5.0	1.4.6	1.5.2	0.8.2	2.1.0	1.0.0	3.4.6	4.2.0	0.6.0	0.5.0
HDP 2.2 Dec. 2014	2.6.0	0.14.0	0.14.0	0.5.2	4.10.2	1.2.1	0.60.0	0.98.4	4.2.0	1.6.1	0.9.3	0.6.0		1.4.5	1.5.2	0.8.1	2.0.0		3.4.6	4.1.0	0.5.0	0.4.0
HDP 2.1 Apr. 2014	2.4.0	0.12.1	0.13.0	0.4.0	4.7.2			0.98.0	4.0.0	1.5.1	0.9.1	0.5.0		1.4.4	1.4.0		1.5.1		3.4.5	4.0.0	0.4.0	

HDP Version Releases

The **Hortonworks Data Platform** (HDP) is an open enterprise version of Hadoop distributed by Hortonworks. It includes a single installation utility that installs many of the Apache Hadoop software frameworks. Even the installer is pure Hadoop. The primary benefit of HDP is that Hortonworks has put it through a rigorous set of system, functional, and regression tests to ensure that versions of any framework included in the distribution works seamlessly with other frameworks in a secure and reliable manner.

Because HDP is an open enterprise version of Hadoop, it is imperative that it uses the best combination of the most stable, reliable, secure, and current frameworks.

Data Management and Operations Frameworks

There are two **data management frameworks**: HDFS and YARN.

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

- **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

HDFS is a Java-based distributed file system that provides scalable, reliable, high-throughput access to application data stored across commodity servers. HDFS is similar to many conventional file systems. For example, it shares many similarities to the Linux file system. HDFS supports operations to read, write, and delete files. It supports operations to create, list, and delete directories.

YARN is a framework for cluster resource management and job scheduling. YARN is the architectural center of Hadoop that enables multiple data processing engines such as interactive SQL, real-time streaming, data science, and batch processing to co-exist on a single cluster.

The four **operations frameworks** are Apache Ambari, Apache ZooKeeper, Cloudbreak, and Apache Oozie.

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

Ambari is a completely open operational framework for provisioning, managing, and monitoring Hadoop clusters. It includes an intuitive collection of operator tools and a set of RESTful APIs that mask the complexity of Hadoop, simplifying the operation of clusters. The most visible Ambari component is the Ambari Web UI, a Web-based interface used to provision, manage, and monitor Hadoop clusters. The Ambari Web UI is the "face" of Hadoop management.

ZooKeeper is a coordination service for distributed applications and services. Coordination services are hard to build correctly, and are especially prone to errors such as race conditions and deadlock. In addition, a distributed system must be able to conduct coordinated operations while dealing with such things as scalability concerns, security concerns, consistency issues, network outages, bandwidth limitations, and synchronization issues. ZooKeeper is designed to help with these issues.

Cloudbreak is a cloud-agnostic tool for provisioning, managing, and monitoring on-demand clusters. It automates the launching of elastic Hadoop clusters with policy-based autoscaling on the major cloud infrastructure platforms including Microsoft Azure, Amazon Web Services, Google Cloud Platform, OpenStack, and Docker containers.

Oozie is a server-based workflow engine used to execute Hadoop jobs. Oozie enables Hadoop users to build and schedule complex data transformations by combining MapReduce, Apache Hive, Apache Pig, and Apache Sqoop jobs into a single, logical unit of work. Oozie can also perform Java, Linux shell, `distcp`, SSH, email, and other operations.

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.

Apache Pig is a high-level platform for extracting, transforming, or analyzing large datasets. Pig includes a scripted, procedural-based language that excels at building data pipelines to aggregate and add structure to data. Pig also provides data analysts with tools to analyze data.

Apache Hive is a data warehouse infrastructure built on top of Hadoop. It was designed to enable users with database experience to analyze data using familiar SQL-based statements. Hive includes support for SQL:2011 analytics. Hive and its SQL-based language enable an enterprise to utilize existing SQL skillsets to quickly derive value from a Hadoop deployment.

Apache HCatalog is a table information, schema, and metadata management system for Hive, Pig, MapReduce, and Tez. HCatalog is actually a module in Hive that enables non-Hive tools to access Hive metadata tables. It includes a REST API, named WebHCat, to make table information and metadata available to other vendors' tools.

Cascading is an application development framework for building data applications. Acting as an abstraction layer, Cascading converts applications built on Cascading into MapReduce jobs that run on top of Hadoop.

Apache HBase is a non-relational database. Sometimes a non-relational database is referred to as a NoSQL database. HBase was created for hosting very large tables with billions of rows and millions of columns. HBase provides random, real-time access to data. It adds some transactional capabilities to Hadoop, allowing users to conduct table inserts, updates, scans, and deletes.

Apache Phoenix is a client-side SQL skin over HBase that provides direct, low-latency access to HBase. Entirely written in Java, Phoenix enables querying and managing HBase tables using SQL commands.

Apache Accumulo is a low-latency, large table data storage and retrieval system with cell-level security. Accumulo is based on Google's Bigtable but it runs on YARN.

Apache Storm is a distributed computation system for processing continuous streams of real-time data. Storm augments the batch processing capabilities provided by MapReduce and Tez by adding reliable, real-time data processing capabilities to a Hadoop cluster.

Apache Solr is a distributed search platform capable of indexing petabytes of data. Solr provides user-friendly, interactive search to help businesses find data patterns, relationships, and correlations across petabytes of data. Solr ensures that all employees in an organization, not just the technical ones, can take advantage of the insights that Big Data can provide.

Apache Spark is an open source, general purpose processing engine that allows data scientists to build and run fast and sophisticated applications on Hadoop. Spark provides a set of simple and easy-to-understand programming APIs that are used to build applications at a rapid pace in Scala. The Spark Engine supports a set of high-level tools that support SQL-like queries, streaming data applications, complex analytics such as machine learning, and graph algorithms.

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

Apache Falcon is a data governance tool. It provides a workflow orchestration framework designed for data motion, coordination of data pipelines, lifecycle management, and data discovery. Falcon enables data stewards and Hadoop administrators to quickly onboard data and configure associated processing and management on Hadoop clusters.

WebHDFS uses the standard HTTP verbs GET, PUT, POST, and DELETE to access, operate, and manage HDFS. Using WebHDFS, a user can create, list, and delete directories as well as create, read, append, and delete files. A user can also manage file and directory ownership and permissions. Administrators can manage HDFS.

The HDFS NFS Gateway allows access to HDFS as though it were part of an NFS client's local file system. The NFS client mounts the root directory of the HDFS cluster as a volume and then uses local command-line commands, scripts, or file explorer applications to manipulate HDFS files and directories.

Apache Flume is a distributed, reliable, and highly-available service that efficiently collects, aggregates, and moves streaming data. It is a distributed service because it can be deployed across many systems. The benefits of a distributed system include increased scalability and redundancy. It is reliable because its architecture and components are designed to prevent data loss. It is highly-available because it uses redundancy to limit downtime.

Apache Sqoop is a collection of related tools. The primary tools are the import and export tools. Writing your own scripts or MapReduce program to move data between Hadoop and a database or an enterprise data warehouse is an error prone and non-trivial task. Sqoop import and export tools are designed to reliably transfer data between Hadoop and relational databases or enterprise data warehouse systems.

Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system. Kafka is often used in place of traditional message brokers like Java Messaging Service (JMS) or Advance Message Queuing Protocol (AMQP) because of its higher throughput, reliability, and replication.

Apache Atlas is a scalable and extensible set of core foundational governance services that enable an enterprise to meet compliance requirements within Hadoop and enables integration with the complete enterprise data ecosystem.

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

HDFS also contributes security features to Hadoop. HDFS includes file and directory permissions, access control lists, and transparent data encryption. Access to data and services often depends on having the correct HDFS permissions and encryption keys.

YARN also contributes security features to Hadoop. YARN includes access control lists that control access to cluster memory and CPU resources, along with access to YARN administrative capabilities.

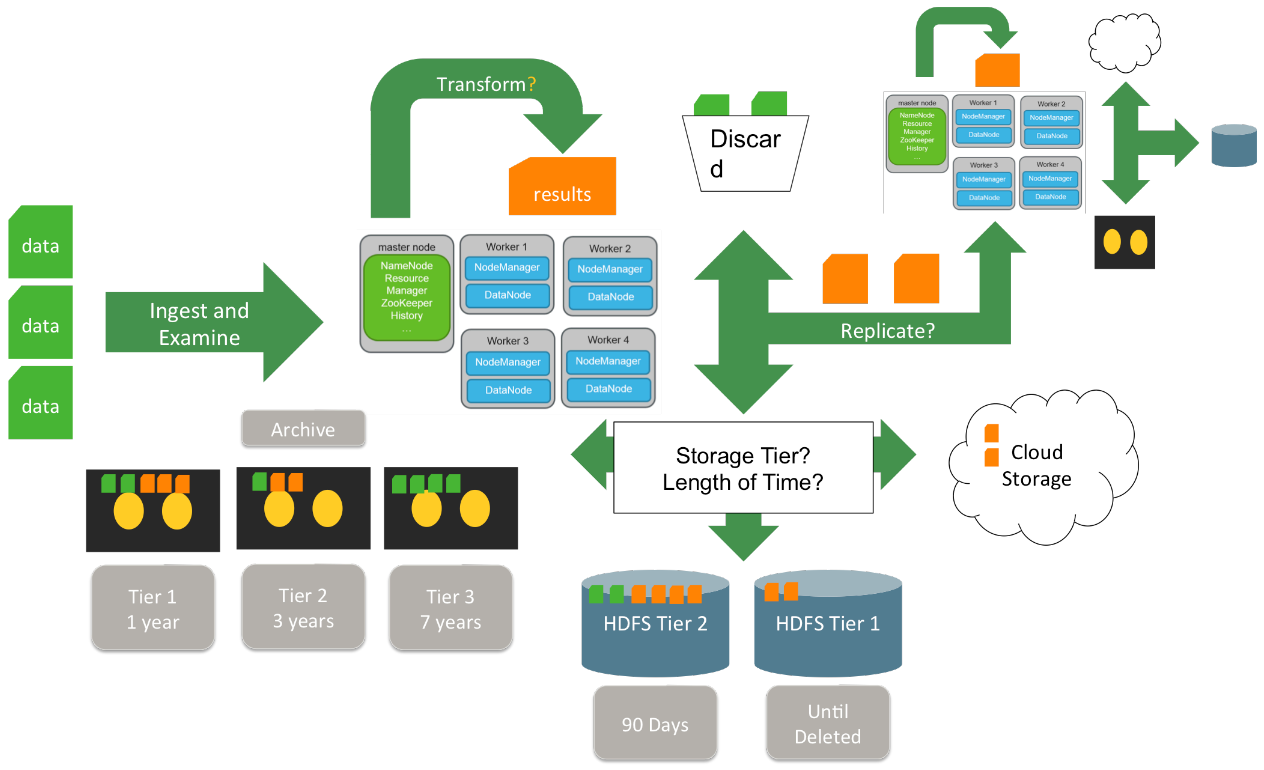
Hive can be configured to control access to table columns and rows.

Falcon is a data governance tool that also includes access controls that limit who may submit automated workflow jobs on a Hadoop cluster.

Apache Knox is a perimeter gateway protecting a Hadoop cluster. It provides a single point of authentication into a Hadoop cluster.

Apache Ranger is a centralized security framework offering fine-grained policy controls for HDFS, Hive, HBase, Knox, Storm, Kafka, and Solr. Using the Ranger Console, security administrators can easily manage policies for access to files, directories, databases, tables, and columns. These policies can be set for individual users or groups and then enforced within Hadoop.

HDP and the Data Lifecycle



HDP supports the data lifecycle

Why does HDP need so many frameworks? Let's take a look at a simple data lifecycle example.

We start with some raw data and an HDP cluster. The first step in managing the data is to get it into the HDP cluster. We must have some mechanism to ingest that data – perhaps Sqoop, Flume, Spark Streaming, or Storm - and then another mechanism to analyze and decide what to do with it next. Does this data require some kind of transformation in order to be used? If so, ETL processes must be run, and those results generated into another file. Quite often, this is not a single step, but multiple steps configured into a data application pipeline.

The next decision comes in regard to whether to keep or discard the data. Not all data must be kept, either because it has no value (empty files, for example), or it is not necessary to keep once it has been processed and transformed. Thus some raw data can simply be deleted.

Data that must be kept requires additional decisions. For example, where will the data be stored, and for how long? Your Hadoop cluster might have multiple tiers of HDFS storage available, perhaps separated via some kind of node label mechanism. In the example, we have two HDFS storage tiers. Any data that is copied to tier 2 should be stored for 90 days. We have another, higher tier of HDFS Storage, and any data stored here should be kept until it is manually deleted.

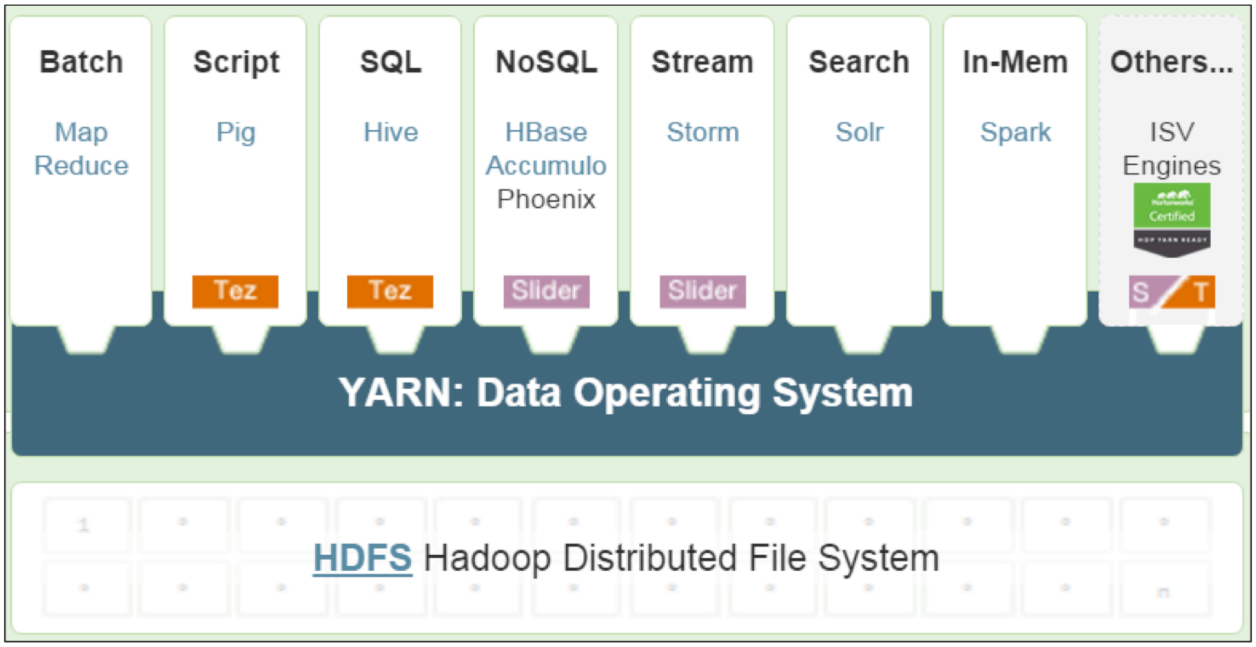
You may decide that some data should be archived rather than made immediately available via HDFS, and you can have multiple tiers of archives as well. In our example we have three tiers of archival storage, and data is kept for one, three, and seven years depending on where it is stored.

A third location where and data might end up is on some kind of cloud storage, such as AWS or Microsoft Azure.

Both raw data and transformed data might be kept anywhere in this storage infrastructure as result of having been input and processed by this HDP cluster. In addition, you may be working in a multi-cluster environment, in which case an additional decision is required. What data needs to be replicated between the clusters? If files need to be replicated to another HDP cluster, then once that cluster ingests in examines that data, this same kind of processes and decision mechanisms need to be employed. Perhaps additional transformation is required. Perhaps some files can be examined and deleted. For files that are to be kept, their location and length of retention must be decided, just as on the first cluster.

This is a relatively simple example of the kind of data lifecycle decisions that need to be made in an environment where the capabilities of HDP are being fully utilized. This can get significantly more complex with the addition of additional storage tiers, retention requirements, and geographically dispersed HDP clusters which must replicate data between each other, and perhaps with a central global cluster designed to do all final processing.

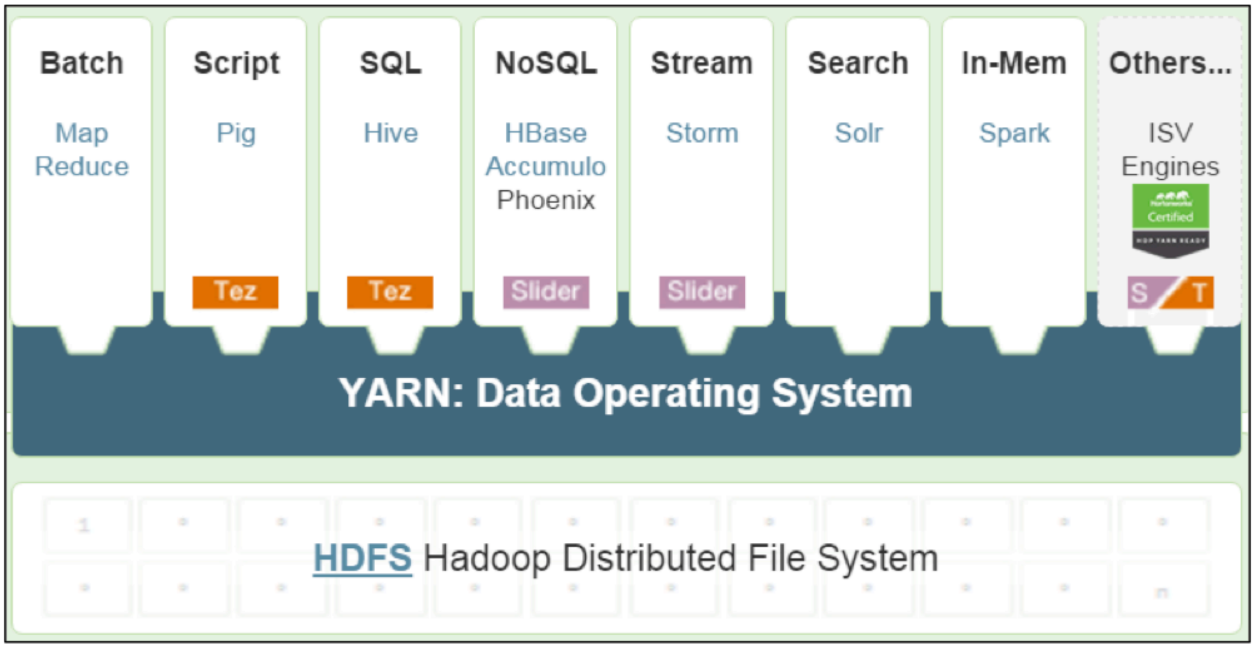
HDFS Overview



HDFS and YARN

The Hadoop Distributed File System (HDFS) and YARN (Yet Another Resource Negotiator) are part of the core of Hadoop and are installed when you install the Hortonworks Data Platform. In this section we will focus on HDFS, the distributed file system for HDP.

HDFS – The HDP File System



HDFS and YARN

HDFS is the basis for Hadoop's storage scalability and availability.

HDFS achieves scalability by taking large files and splitting them, by default, into 128-megabyte chunks called blocks. This block size is configurable. The file blocks are spread across the slave nodes in the cluster. For example, if a file is split into 10 blocks and you have 10 slave servers, all 10 blocks can be read in parallel. To achieve greater scalability for larger files, just add more slave nodes to the cluster.

HDFS achieves availability by automatically replicating each block of a file. By default, HDFS maintains three copies of the data, and ensures that those copies are not on the same slave nodes, or even the same server rack if the cluster is configured correctly. As an example, if a file is split into 10 blocks then the cluster would actually maintain 30 blocks.

A Hadoop cluster scales computation capacity, storage capacity, and I/O bandwidth by adding more slave nodes. HDFS scales by adding more machines with local disks, making NAS and SAN storage unnecessary for HDP environments.

HDFS Command Line Interaction

```
hdfs dfs –command [args]
```

Developers can interact with HDFS directly via the command line using the `hdfs dfs` command and appropriate arguments. If a developer has previous Linux command line experience, the `hdfs dfs` commands will be familiar and intuitive to use. The most common use for command line usage is manual data ingestion and file manipulation.

Example commands include:

- `-cat`: display file content (uncompressed)
- `-text`: just like cat but works on compressed files
- `-mkdir`: create a directory in HDFS
- `-put`, `-get`: copies files from the local file system to the HDFS and vice versa.
- `-mv`: moves files
- `-ls`, `-rm`: list and remove files/directories (adding `-R` makes listing/removal recursive)
- `-chgrp`, `-chmod`, `-chown`: changes file permissions
- `-stat`: statistical info for any given file (block size, number of blocks, file type, etc.)

Additional information can be obtained by `hdfs dfs` at the command line with no arguments or options, or by viewing online documentation.

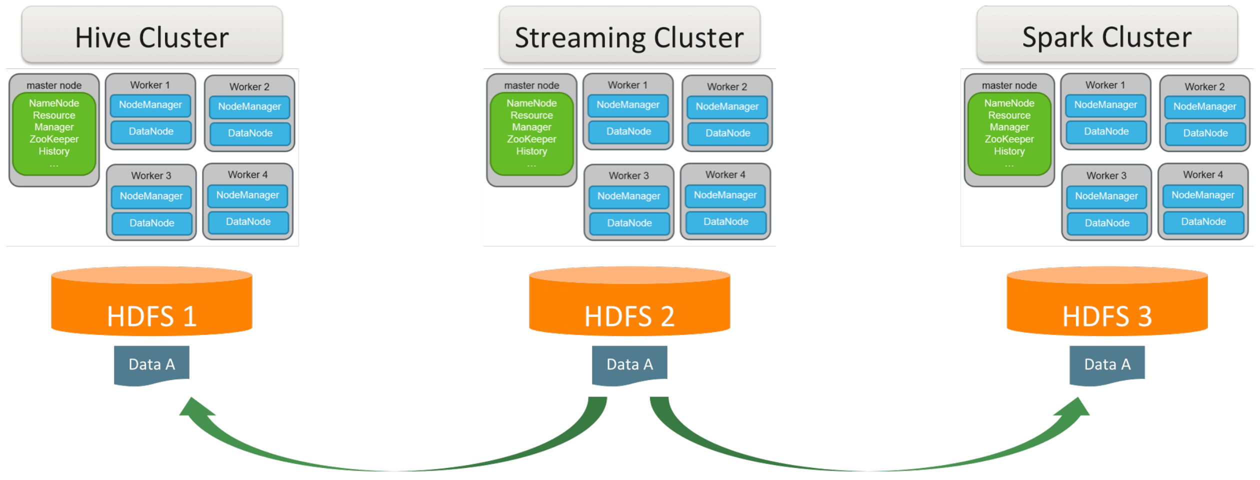
The sequence of command below creates a directory, copies a file from the local file system to the new directory in HDFS, and then lists the contents of the directory:

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

HDFS implements a permissions model for files and directories that shares much from the POSIX model:

- Each file and directory is associated with an owner and a group. The file or directory has separate permissions for the user that is the owner, for other users that are members of the group, and for all other users.
- For files, the `r` permission is required to read the file and the `w` permission is required to write or append to the file.
- For directories, the `r` permission is required to list the contents of the directory, the `w` permission is required to create or delete files or directories, and the `x` permission is required to access a child of the directory.

YARN Overview

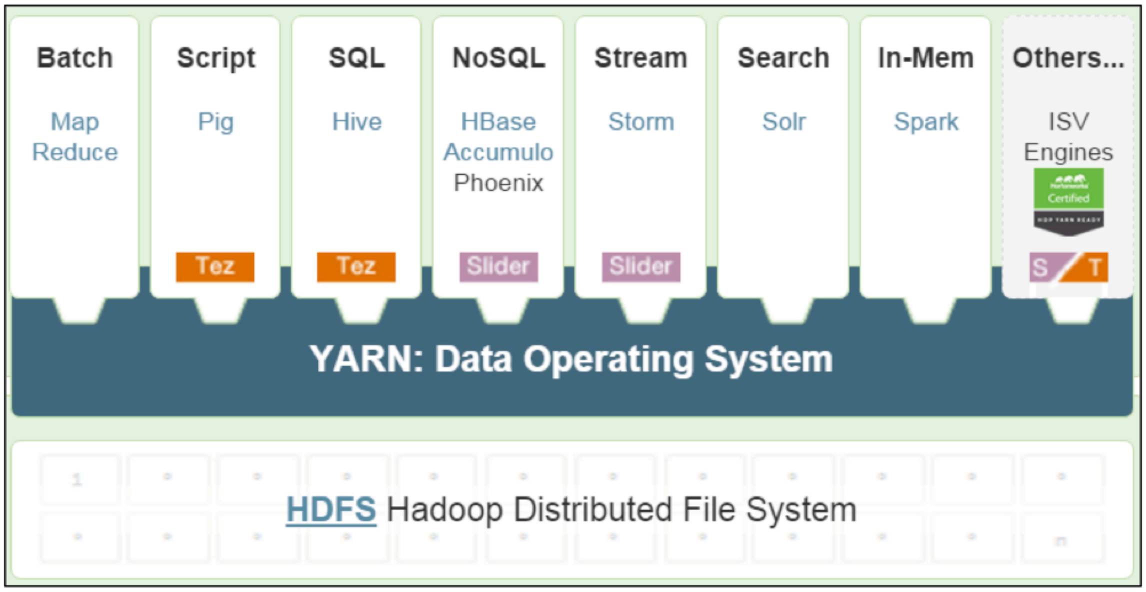


Hadoop applications without YARN

Why is YARN so important to Spark? Let's take a look at a sample enterprise Hadoop deployment. Without a central resource manager to ensure good behavior between applications, it is necessary to create specialized, separate clusters to support multiple applications. This, in turn, means that when you want to do something different with the data that application was using, it is necessary to copy that data between clusters. This introduces inefficiencies in terms of network, CPU, memory, storage, general datacenter management, and data integrity across Hadoop applications.

YARN as a resource manager mitigates this issue by allowing different types of applications to access the same underlying resources pooled into a single data lake. Since Spark runs on YARN, it can join other Hadoop applications on the same cluster, enabling data and resource sharing at enterprise scale.

YARN – The HDP Operating System



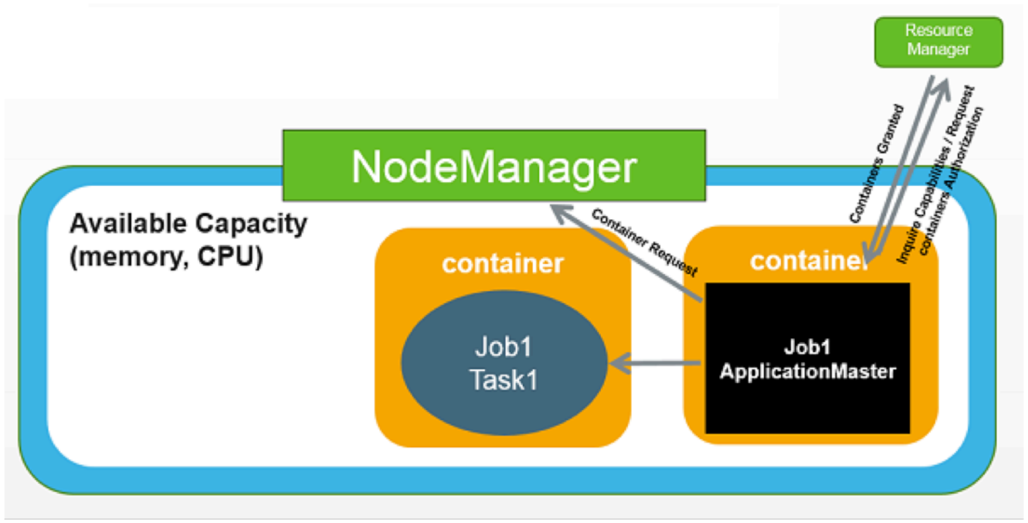
HDFS and YARN

YARN (unofficially "Yet Another Resource Negotiator") is the computing framework for Hadoop. If you think about HDFS as the cluster file system for Hadoop, YARN would be the cluster operating system. It is the architectural center of Hadoop.

A computer operating system, such as Windows or Linux, manages access to resources, such as CPU, memory, and disk, for installed applications. In similar fashion, YARN provides a managed framework that allows for multiple types of applications – batch, interactive, online, streaming, and so on – to execute on data across your entire cluster. Just like a computer operating system manages both resource allocation (which application gets access to CPU, memory, and disk now, and which one has to wait if contention exists?) and security (does the current user have permission to perform the requested action?), YARN manages resource allocation for the various types of data processing workloads, prioritizes and schedules jobs, and enables authentication and multitenancy.

YARN Resource Containers

To understand the YARN architecture, it helps to understand YARN *resource containers*.



YARN resource containers

Every slave node in a cluster is comprised of resources such as CPU and memory. The abstract notion of a resource container is used to represent a discreet amount of these resources. Cluster applications run inside one or more containers. Containers are managed and scheduled by YARN.

A container's resources are logically isolated from other containers running on the same machine. This isolation provides strong application multi-tenancy support.

Applications are allocated different-sized containers based on application-defined resource requests, but always within the constraints configured by the Hadoop administrator.

Knowledge Check

You can use the following questions and answers for self-assessment.

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 ____________________.

Answers

1) Name the three V's of big data.

Answer: Volume, Velocity, and Variety

2) Name four of the six types of data commonly found in Hadoop.

Answer: Sentiment, clickstream, sensor/machine, server, geographic, and text

3) Why is HDP comprised of so many different frameworks?

Answer: To allow for end-to-end management of the data lifecycle

4) What two frameworks make up the core of HDP?

Answer: HDFS and YARN

5) What is the base command-line interface command for manipulating files and directories in HDFS?

Answer: `hdfs dfs`

6) YARN allocates resources to applications via ____________________.

Answer: Containers

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

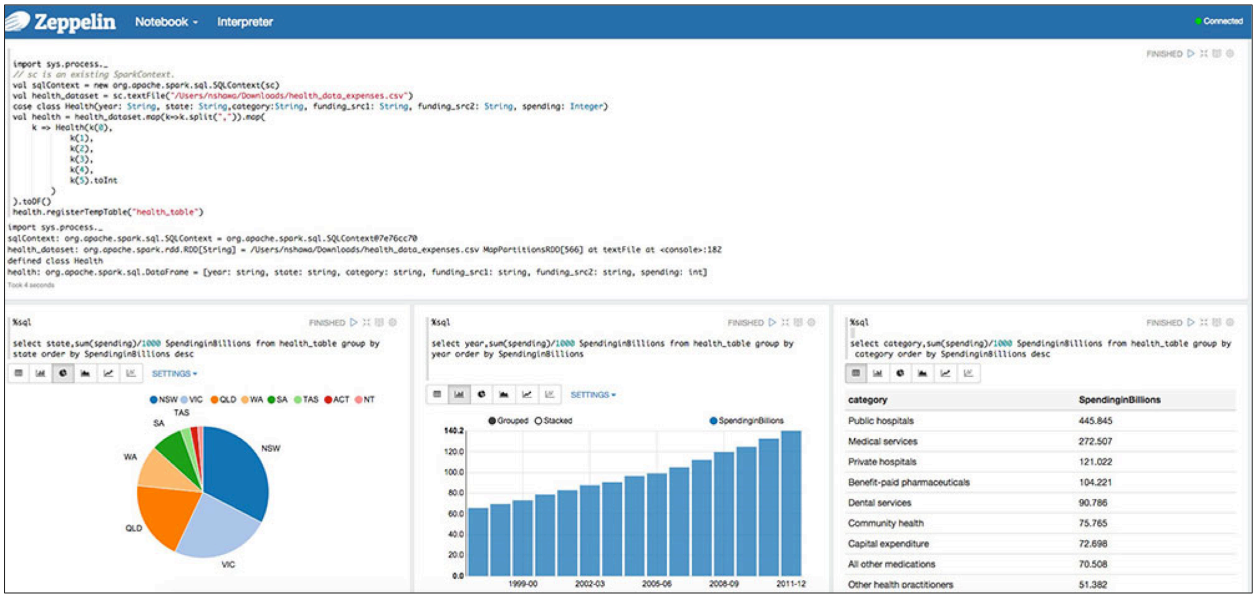
Zeppelin and Spark

Lesson Objectives

After completing this lesson, students should be able to:

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

Zeppelin Overview



```
import sys.process._
// sc is an existing SparkContext.
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
val health_dataset = sc.textFile("/Users/nshawa/Downloads/health_data_expenses.csv")
case class Health(year: String, state: String,category:String, funding_src1: String, funding_src2: String, spending: Integer)
val health = health_dataset.map(k=>k.split(",")).map(
    k => Health(k(0),
              k(1),
              k(2),
              k(3),
              k(4),
              k(5).toInt
        )
).toDF()
health.registerTempTable("health_table")
```

```
import sys.process._
sqlContext: org.apache.spark.sql.SQLContext = org.apache.spark.sql.SQLContext@7e76cc70
health_dataset: org.apache.spark.rdd.RDD[String] = /Users/nshawa/Downloads/health_data_expenses.csv MapPartitionsRDD[566] at textFile at <console>:182
defined class Health
health: org.apache.spark.sql.DataFrame = [year: string, state: string, category: string, funding_src1: string, funding_src2: string, spending: int]
Took 4 seconds
```

```
%sql
select state,sum(spending)/1000 SpendinginBillions from health_table group by
state order by SpendinginBillions desc
```

```
%sql
select year,sum(spending)/1000 SpendinginBillions from health_table group by
year order by SpendinginBillions
```

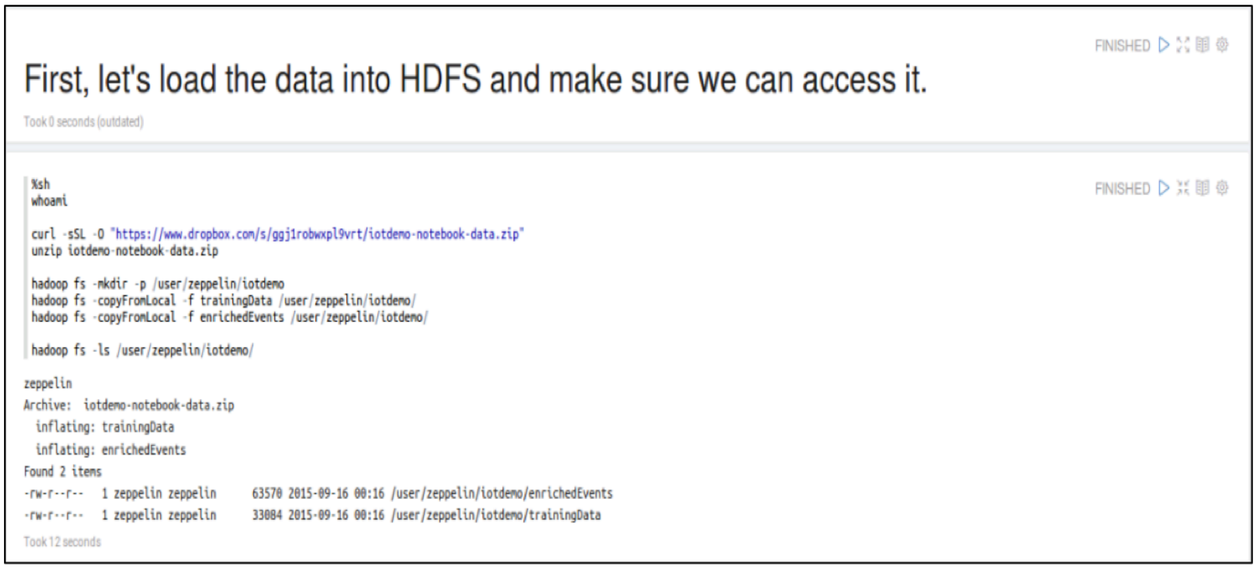
```
%sql
select category,sum(spending)/1000 SpendinginBillions from health_table group by
 category order by SpendinginBillions desc
```

category	SpendinginBillions
Public hospitals	445.845
Medical services	272.507
Private hospitals	121.022
Benefit-paid pharmaceuticals	104.221
Dental services	90.786
Community health	75.765
Capital expenditure	72.698
All other medications	70.508
Other health practitioners	51.382

Apache Zeppelin

Apache Zeppelin is a web-based notebook that enables interactive data analytics on top of Spark. It supports a growing list of programming languages, such as Python, Scala, Hive, SparkSQL, shell, and markdown. It allows for data visualization, report generation, and collaboration.

Major Zeppelin Functions



First, let's load the data into HDFS and make sure we can access it.

Took 0 seconds (outdated)

```
%sh
whoami

curl -sSL -O "https://www.dropbox.com/s/ggj1robwxpl9vrt/iotdemo-notebook-data.zip"
unzip iotdemo-notebook-data.zip

hadoop fs -mkdir -p /user/zeppelin/iotdemo
hadoop fs -copyFromLocal -f trainingData /user/zeppelin/iotdemo/
hadoop fs -copyFromLocal -f enrichedEvents /user/zeppelin/iotdemo/

hadoop fs -ls /user/zeppelin/iotdemo/
```

```
zeppelin
Archive:  iotdemo-notebook-data.zip
  inflating: trainingData
  inflating: enrichedEvents
Found 2 items
-rw-r--r--   1 zeppelin zeppelin      63570 2015-09-16 00:16 /user/zeppelin/iotdemo/enrichedEvents
-rw-r--r--   1 zeppelin zeppelin      33084 2015-09-16 00:16 /user/zeppelin/iotdemo/trainingData
```

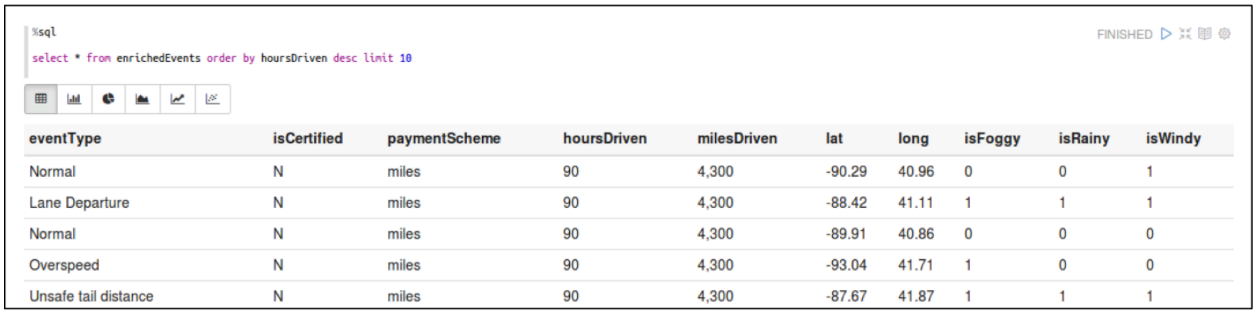
Took 12 seconds

Zeppelin

Zeppelin has four major functions: **data ingestion**, **discovery**, **analytics**, and **visualization.** It comes with built-in examples that demonstrate these capabilities. These examples can be reused and modified for real-world scenarios.

Data Visualization

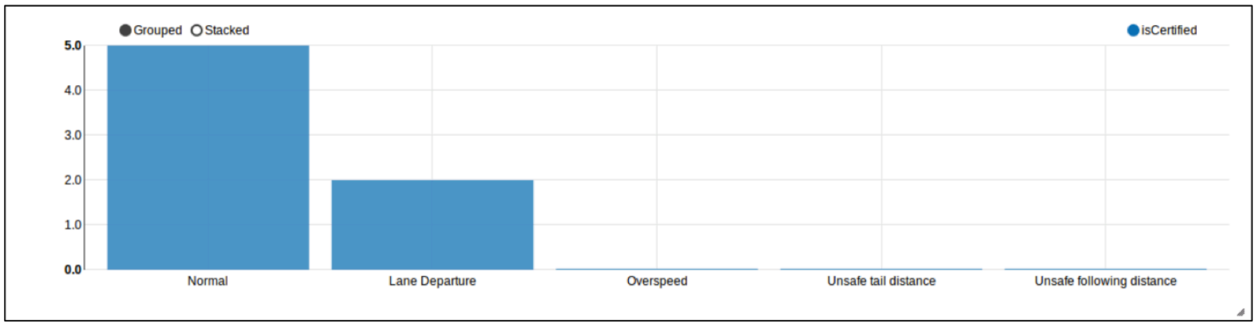
Zeppelin comes with several built-in ways to interactively view and visualize data including table view, column charts, pie charts, area charts, line charts, and scatter plot charts – illustrated below:



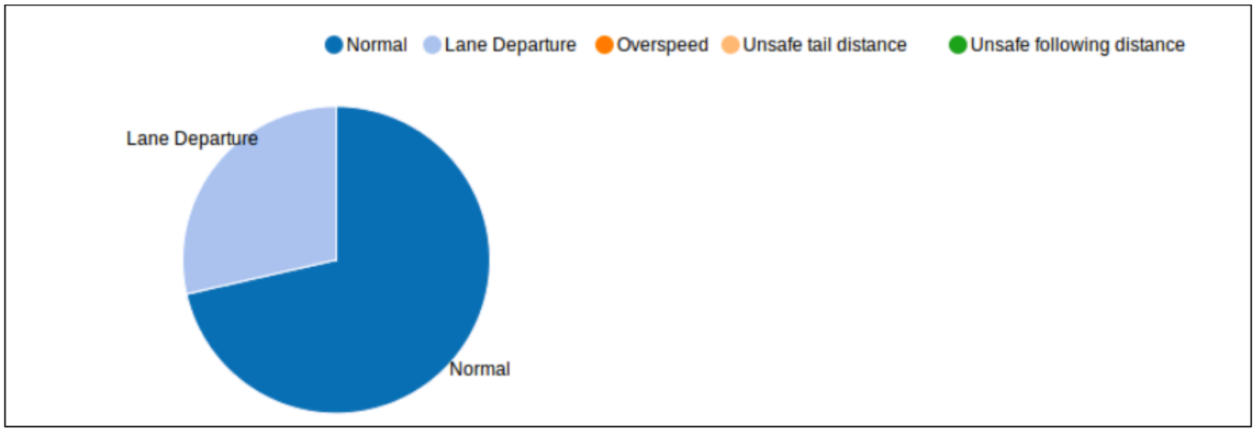
```
%sql
select * from enrichedEvents order by hoursDriven desc limit 10
```

eventType	isCertified	paymentScheme	hoursDriven	milesDriven	lat	long	isFoggy	isRainy	isWindy
Normal	N	miles	90	4,300	-90.29	40.96	0	0	1
Lane Departure	N	miles	90	4,300	-88.42	41.11	1	1	1
Normal	N	miles	90	4,300	-89.91	40.86	0	0	0
Overspeed	N	miles	90	4,300	-93.04	41.71	1	0	0
Unsafe tail distance	N	miles	90	4,300	-87.67	41.87	1	1	1

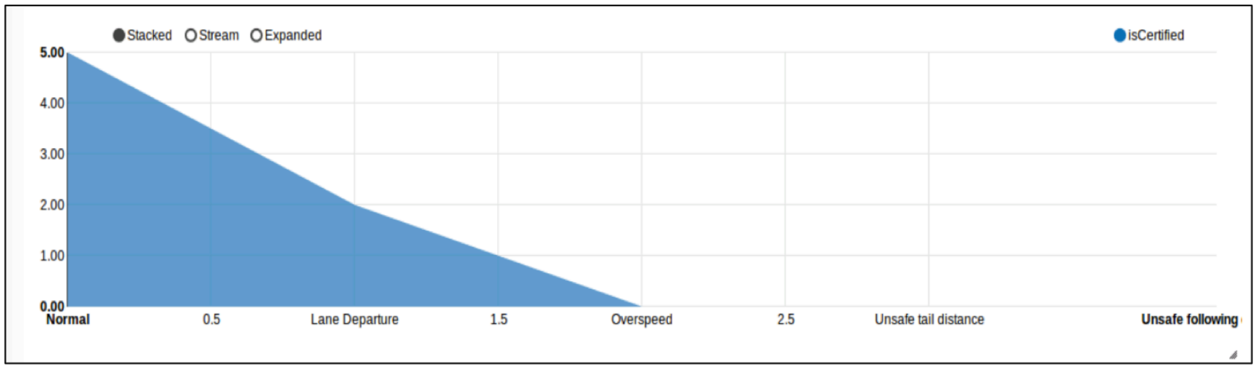
Zeppelin table view



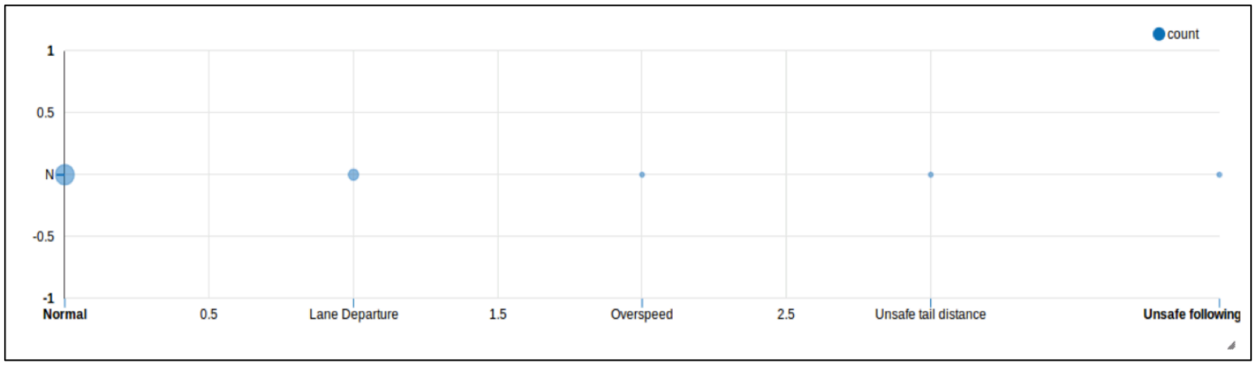
Zeppelin column chart



Zeppelin pie chart



Zeppelin line chart



Zeppelin scatter plot chart

Zeppelin Technical Preview

At the time of this writing, Zeppelin is in its final technical preview, and will likely be generally available and production-ready soon after this course is released.

As a general rule, Hortonworks University courses avoid the inclusion of technologies that are in technical preview, however in this case an exception was made based on the following reasons:

- *Nearly* all of the labs in this class could be run either from the command line or Zeppelin with identical steps, so the use of Zeppelin does not interfere with learning in any way.
- Despite its technical preview status, Zeppelin is the best solution available today in HDP for key functionalities such as data visualization and collaboration supporting multiple languages (for example, both Python and Scala.)
- When Zeppelin comes out of technical preview - which may have already happened by the time this is read - by using it now you will already have significant hands-on experience.

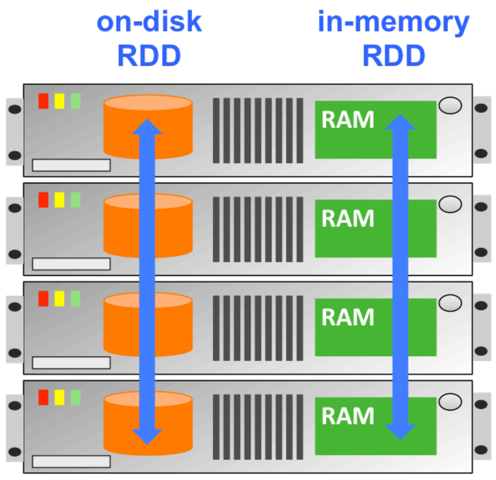
Spark Overview

Spark Introduction

Spark is a platform that allows large-scale, cluster-based, in-memory data processing. It enables fast, large-scale data engineering and analytics for iterative and performance-sensitive applications. It offers development APIs for Scala, Java, Python, and R. In addition, Spark has been extended to support SQL-like operations, streaming, and machine learning as well.

Spark is supported by Hortonworks on HDP and is YARN compliant, meaning it can leverage datasets that exist across many other applications in HDP.

Spark RDDs – Scalability and Performance

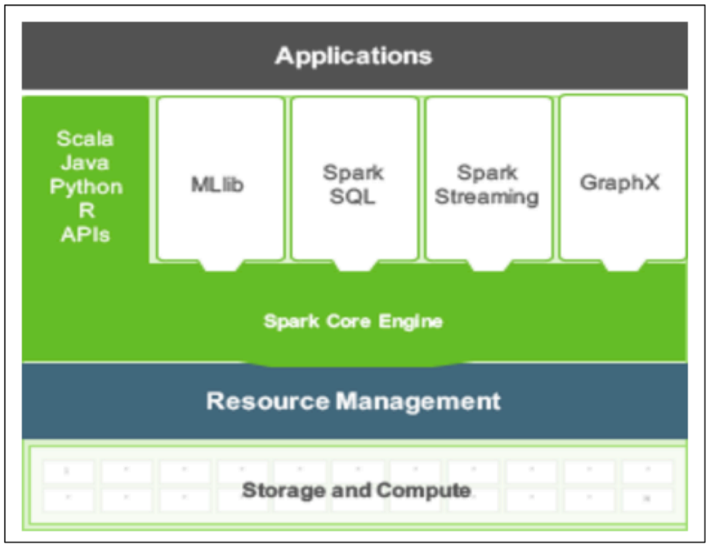


Spark RDDs

To leverage Hadoop's horizontal scalability, Spark processes data in a **Resilient Distributed Dataset**, called an **RDD**. An RDD is a fault-tolerant collection of data elements. An RDD is created by starting with a file on disk, or a collection of data in memory in a driver program. Each RDD is distributed across multiple nodes in a cluster. This enables parallel processing across the nodes.

Allowing RDDs to reside and be processed in memory dramatically increases performance especially when a dataset needs to be manipulated through multiple stages of processing. The data in an RDD can be transformed and analyzed very rapidly.

Spark High-Level Tools



Spark tools

Coming from the Spark project, Spark Core supports a set of four high-level tools that support SQL-like queries, streaming data applications, a machine learning library (Mlib), and graph algorithms (GraphX). In addition, Spark also integrates with a number of other HDP tools, such as Hive for SQL-like operations and Zeppelin for graphing / data visualization.

Spark and HDP

The following Spark versions align with various HDP versions:

HDP 2.4.0 – Spark 1.6.0

HDP 2.3.5 – 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 the labs in this class, we use Spark 1.6.0 on HDP 2.4.0.

Spark REPLs and Application Architecture

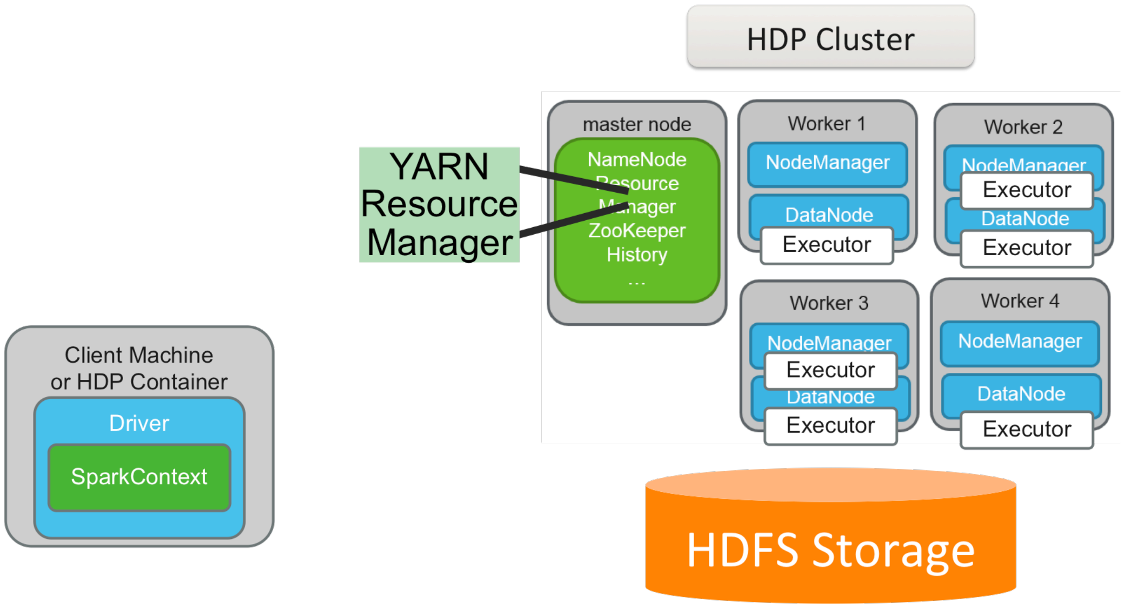
Spark REPL Shells

REPL stands for "**Read, Evaluate, Print, Loop.**" A REPL takes a single user input, evaluates it, and returns the result. There are two REPLs available in Spark – one for Python and one for Scala. These are also sometimes referred to as "Spark shell" applications.

In addition to helping one learn Spark, the shells are great for debugging and exploring data. They can also be used for building applications.

In order to start the Python shell, the user needs to enter "`pyspark`" on the command line. In order to start the Scala shell, the user needs to enter "`spark-shell`" on the command line.

Enterprise Spark Application Components in HDP



Spark Application Components

There are five core components of an enterprise Spark application in HDP. They are the Driver, `SparkContext`, YARN ResourceManager, HDFS Storage, and Executors.

When using a REPL, the driver and `SparkContext` will run on a client machine. When deploying an application as a cluster application, the driver and `SparkContext` can also run in a YARN container. In both cases, Spark executors run in YARN containers on the cluster.

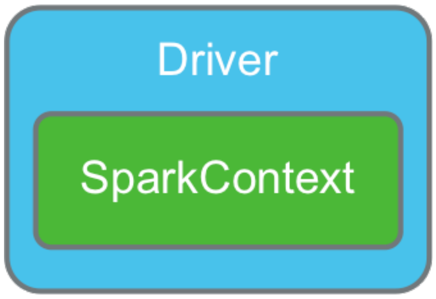
Spark Driver

The Spark driver contains the `main()` Spark program that manages the overall execution of a Spark application. It is a JVM that creates the `SparkContext` which then communicates directly with Spark. It is also responsible for writing/displaying and storing the logs that the `SparkContext` gathers from executors.

Spark shell REPLs are examples of Spark driver programs.

IMPORTANT: The Spark driver is a single point of failure for a YARN client application. If the driver fails, the application will fail. This is mitigated when deploying applications using YARN cluster.

SparkContext



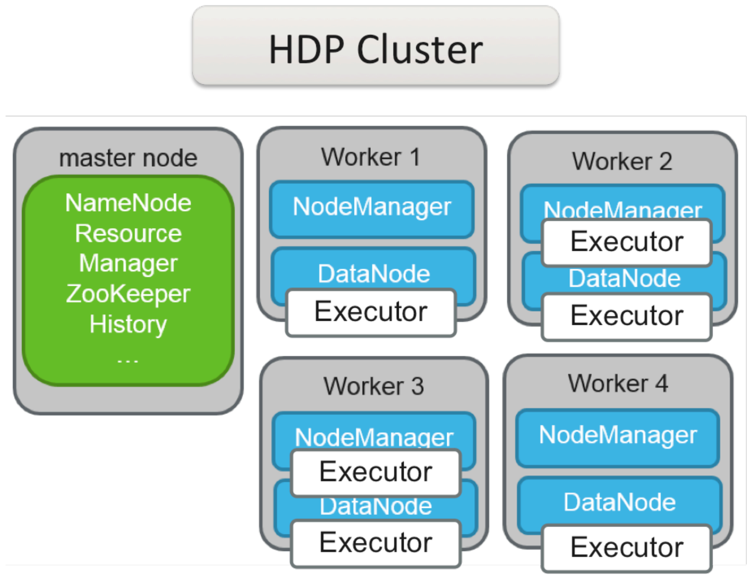
`SparkContext`

For an application to become a Spark application, an instance of the `SparkContext` class must be instantiated. The `SparkContext` contains all code and objects required to process data in the cluster, and works with the YARN `ResourceManager` to get requested resources for the application. It is also responsible for scheduling tasks for Spark executors. The `SparkContext` checks in with the executors to report work being done and provide log updates.

A `SparkContext` is automatically created and named `sc` when a REPL is launched. The following code is executed at start up for `pyspark`:

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

Spark Executors



Spark Executors

The **Spark executor** is the component that performs the map and reduce tasks of a Spark application, and is sometimes referred to as a Spark “worker.” Once created, executors exist for the life of the application.

NOTE:

In the context of Spark, the `SparkContext` is the "master" and executors are the "workers." However, in the context of HDP in general, you also have "master" nodes and "worker" nodes. Both uses of the term worker are correct - in terms of HDP, the worker (node) can run one or more Spark workers (executors). When in doubt, make sure to verify whether the worker being described is an HDP node or a Spark executor running on an HDP node.

Spark executors function as interchangeable work spaces for Spark application processing. If an executor is lost while an application is running, all tasks assigned to it will be reassigned to another executor. In addition, any data lost will be recomputed on another executor.

Executor behavior can be controlled programmatically. Configuring the number of executors and their available resources can greatly increase performance of an application when done correctly.

Knowledge Check

You can use the following questions and answers for self-assessment.

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?

Answers

1) Name the tool in HDP that allows for interactive data analytics, data visualization, and collaboration with Spark.

Answer: Zeppelin

2) What programming languages does Spark currently support?

Answer: Scala, Java, Python, and R

3) What is the primary benefit of running Spark on YARN?

Answer: Access to datasets shared across the cluster with other HDP applications

4) Name the five components of an enterprise Spark application running in HDP.

Answer: Driver, `SparkContext`, YARN, HDFS, and executors.

5) Which component of a Spark application is responsible for application workload processing?

Answer: Executor

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

Lesson Objectives

After completing this lesson, students should be able to:

- ✓ 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

Introduction to RDDs

Resilient Distributed Datasets (RDDs)

The Resilient Distributed Dataset (RDD) is an *immutable, distributed, and resilient* collection of elements that can be operated on in parallel.

An RDD is composed of one or more partitions that can be distributed across nodes in the cluster. These partitions are immutable, meaning when changes occur, new RDDs are created rather than having those changes written to existing RDDs. They are also resilient, meaning if an RDD partition fails, it can be recreated on another node.

For HDFS files, the RDD partitions will be aligned with the file's blocks to maximize parallelism and other HDP infrastructure benefits.

Creating RDDs Programmatically from Simple Lists

Users can programmatically create RDDs from lists of number or text values using the `sc.parallelize()` API. This is useful for learning Spark and distributing local collections of data to the cluster. For example:

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

This creates an RDD that contains four numeric values. To confirm the RDD was created and contains the correct information, we can look at the contents using `collect()`:

```
rddNumList.collect()
```

This returns the following output:

```
[5,7,11,14]
```

The same process works on lists of text. For example:

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

Creating RDDs Programmatically using Variables as Input

RDDs can also take variable values as input. To demonstrate, we start by creating a variable named `maryFile` and assigning it a text string value:

```
maryFile = ("Mary had a little lamb")
```

Then we use `parallelize()` to turn the variable contents into an RDD, and `collect()` to view the results:

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

NOTE:

The brackets `[ ]` around the variable name prevents the variable input from being read one character at a time. If we had used parentheses `( )` instead, the `collect` function would have returned:

```
['M', 'a', 'r', 'y', ' ', 'h', 'a', 'd', . . . , 'l',
'a', 'm', 'b']
```

In this example, we are creating the list as a variable first:

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

Creating Simple RDDs from Text Files

Another common way to create an RDD is to load it from one or more text files using `sc.textFile()`. These files can be located on local storage, HDFS, or other cloud or network storage locations. For example, to create an RDD from a text file on your local drive:

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

Or to create an RDD from a file on HDFS:

```
rddHDFS = sc.textFile("hdfs://namenodename:8020/HDFSpath/filename.txt")
```

Multiple files can be combined as part of a single RDD using either a comma-separated list of individual files, the wildcard character, or a combination of the two. For example, to create an RDD based on two specific files:

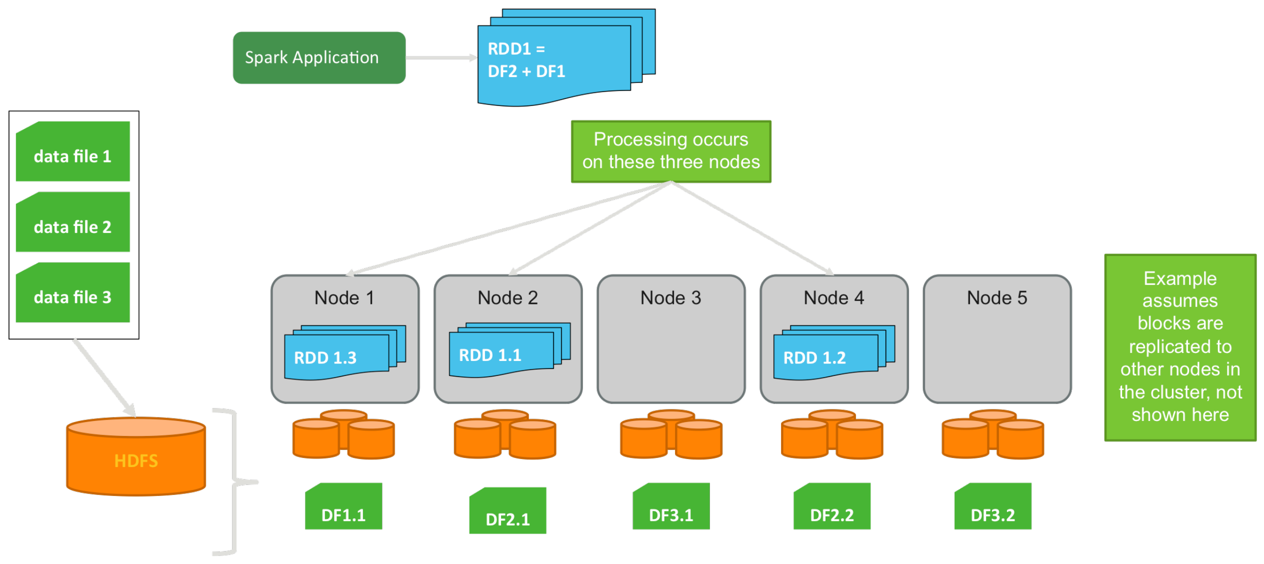
```
rddComma = sc.textFile("fileLocation/file1.txt,fileLocation/file2.txt")
```

Or to create an RDD that contains all `.txt` files that meet wildcard requirements in a given location:

```
rddWild = sc.textFile("fileLocation/*.txt")
```

Wildcards and comma-separated lists can be combined in any configuration.

From Data Files to HDFS to RDD



Data being transferred to HDFS and converted to RDDs

In this simple example, we have a small cluster which has been loaded with three data files, and will walk through their input into HDFS, followed by having two of them used to create a single RDD, and begin to demonstrate the power of parallel datasets.

The first file in the example is small enough to fit entirely in a single 128 MB HDFS block – so data file 1 is made up of only one HDFS block (labeled DF1.1) which is written to Node 1. This would be replicated by default to two other nodes, which are not shown in the image.

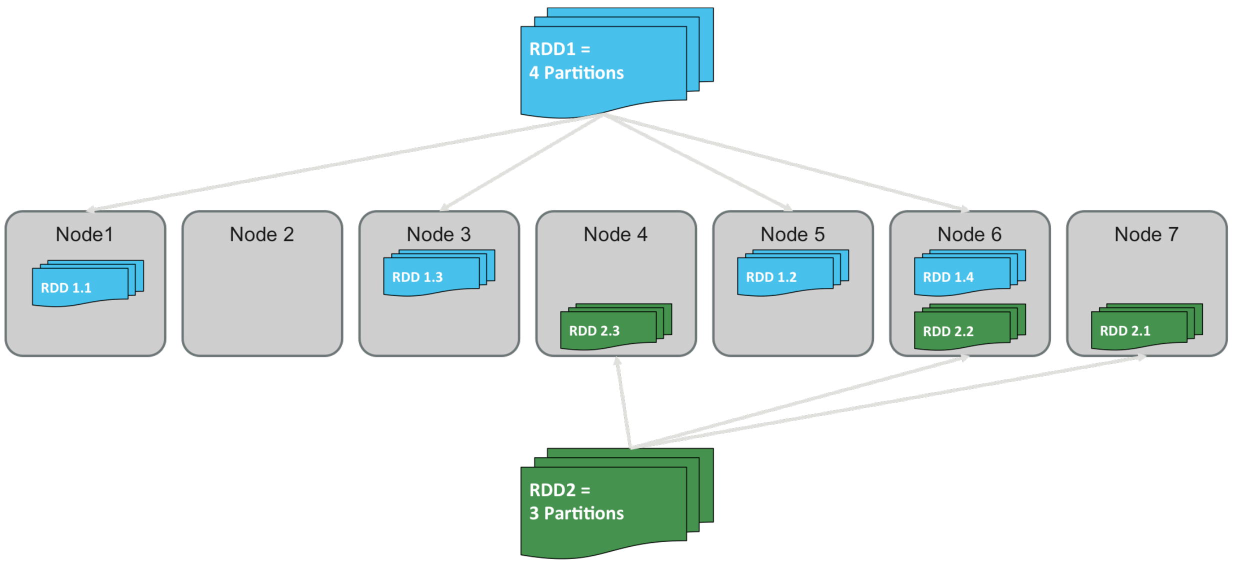
Data files 2 and 3 take up two HDFS blocks each. In our example, these four data blocks are written to four different HDFS nodes. Data file 2 is represented in HDFS by DF2.1 and DF 2.2 (written to node 2 and node 4, respectively). Data file 3 is represented in HDFS by DF3.1 and DF3.2 (written to node 3 and node 5, respectively). Again, not shown in the image, each of these blocks would be replicated multiple times across nodes in the cluster.

Next we write a Spark application that initially defines an RDD that is made up of a combination of the data in data files 1 and 2. In HDFS, these two files are represented by three HDFS blocks on nodes 1, 2, and 4. When the RDD partitions are created in memory, the nodes that will be used will be the same nodes that contain the data blocks. This improves performance and reduces network traffic that would result from pulling data from one node's disk to another node's memory.

The three data blocks that represent these two files that were combined by the Spark application are then copied into memory on their respective nodes and become the partitions of the RDD. DF2.1 is written to an RDD partition we have labeled RDD 1.1. DF2.2 is written to an RDD partition we have labeled RDD 1.2. DF1.1 is written to an RDD partition we have labeled RDD 1.3.

In our example, one RDD was created from two files (which are split across three HDFS data nodes), which exist in memory as three partitions that a Spark application can then continue to use.

Multiple RDDs in a Cluster



A hypothetical cluster that has two RDD's. Each RDD is composed of multiple partitions, which are distributed across the cluster.

RDD Characteristics

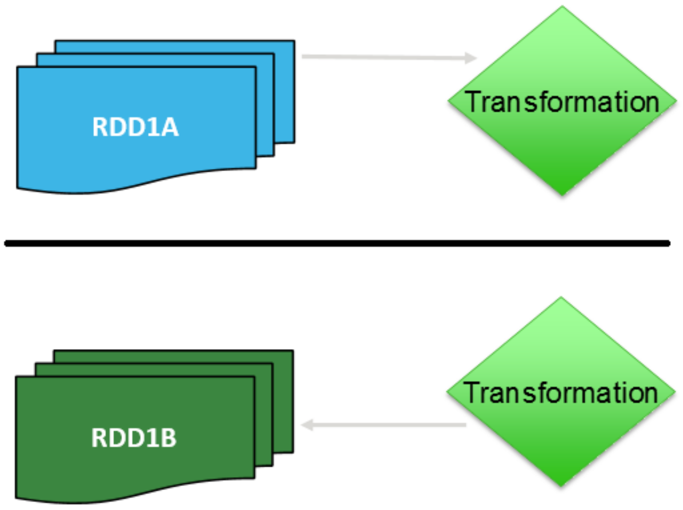
RDDs can contain any type of ***serializable*** element, meaning those that can be converted to and from a byte stream. Examples include: `int`, `float`, `bool`, and sequences/iteratives like `arrays`, `lists`, `tuples`, and `strings`. Element types in an RDD can be mixed as well. For example, an `array` or `list` can contain both `string` and `int` values. Furthermore, RDD types are converted implicitly when possible, meaning there is no need to explicitly specify type during RDD creation.

NOTE:

Non-serializable elements (for example: objects created with certain third-party JAR files or other external resource) cannot be made into RDDs.

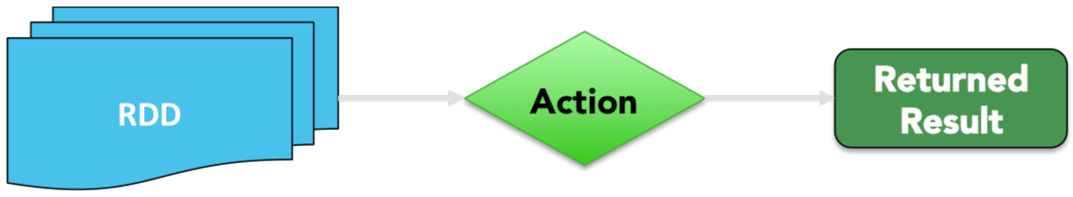
RDD Operations

Once an RDD is created, there are two operations that can be performed on it: **Actions** and **Transformations.**



Transformations apply a function to RDD elements and create new RDD partitions based on the output

A *Transformation* takes an existing RDD, applies a function to the elements of the RDD, and creates a new RDD comprised of the transformed elements.



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

An *action* returns a result of a function applied to the elements of an RDD in the form of screen output, a file write, etc.

Spark Programming Basics

Before we further explore transformations and actions, let's go through a brief overview of functional programming and the use of anonymous functions.

Introduction to Functional Programming

Spark makes heavy use of functional programming, but what does that mean? An easy way to think about functional programming is to define it as programming that does not rely on data outside of the function being executed, nor does it modify data that exists outside of the current function.

First let's take a look at an example of non-functional programming. In the following function, we define a variable value outside of our function, then pull that value into our function and modify it. Note the dependence on, and the writing to, a variable that exists external to the function itself:

```
varValue = 0
def unfunctionalCode():
  global varValue
  varValue = varValue + 1
```

Now let's take a look at the same basic example, but this time written using functional programming principles. In this example, the variable value is instantiated as part of calling the function itself, and only the value within the function is modified.

```
def functionalCode(varValue):
  return varValue + 1
```

All Spark transformations are based on functional programming.

Functional Programming Implications in Spark

The need to operate on different partitions of the same RDD in parallel makes functional programming a requirement for Spark. Therefore, Spark programming has the following characteristics:

- **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

Anonymous functions, also known as **Lambda functions**, are used heavily in Spark programming. They start with a declaration of the anonymous function by using the keyword `lambda`. Then, the programmer provides the variable name that will be used in the function body, followed by a colon. To the right of the colon will be the function body, which can be a function definition or a call to yet another function.

Here is an example using "`z`" as the anonymous function variable selected and applying `z+1` to all elements of an RDD as an argument to the Spark `map` function:

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

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

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

Basic Spark Transformations

`map()`

The map transformation applies a function supplied as its argument to each element of an RDD.

In the example below, we have a list of numbers that make up an RDD. We then apply the map function and instruct Spark to run the anonymous function (using `z` as the variable name for each element) `z+1`, an immediately print the output to the screen:

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

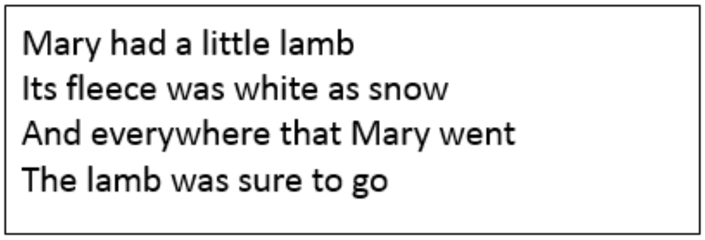
NOTE:

the second line of code in this example did not define a new RDD. If further transformations were necessary, the second line of code would need to be rewritten as follows:

```
rddAnon = rddNumList.map(lambda z: z + 1)
rddAnon.collect()
[6, 8, 12, 15]
```

Maps can apply to strings as well. Here is an example that starts by reading a file from HDFS called "`mary.txt`":

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



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

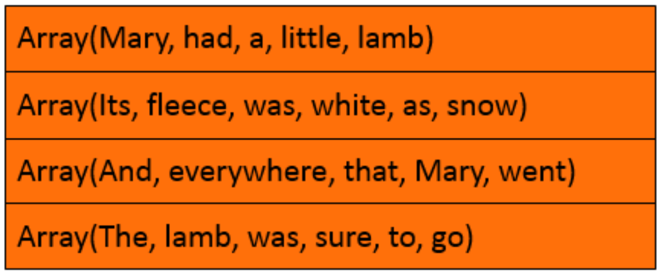
Contents of the mary.txt file and initial RDD

RDDs created using the `textFile` method treat newline characters as characters that separate elements. Thus, since the file had four lines, the file as shown in the image would have four elements in the RDD.

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

A `map()` transformation is then called. The goal of the `map` transformation in this scenario is to take each element, which is a `string` containing multiple words, and break it up into an `array` that is stored in a new RDD for further processing. The `split` function takes a `string` and breaks it into `arrays` based on the delimiter passed into `split()`.

The result is an RDD that still only has four elements, but now those elements are `arrays` rather than monolithic `strings`.


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)

Contents of rddMary converted into arrays using map()

`flatMap()`

The `flatMap` function is similar to map, with the exception that it performs an extra step to break down (or *flatten*) the component parts of elements such as `arrays` or other sequences into individual elements after running a `map` function.

`map()` is a one-to-one transformation: one element comes in, one element comes out. Using `map()`, four line elements were converted into four array elements, but we still started and ended with the same number of elements. The `flatMap()` function, on the other hand, is a one-to-many transformation: one element may go in, but many can come out.

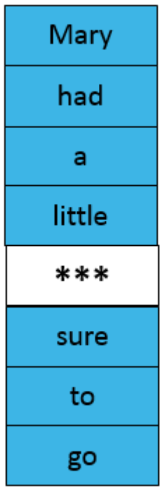
Let's compare using the previous `map()` illustration.

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

If we run the same code, replacing `map()` with `flatMap()`, the output is returned as a single list of individual elements rather than four lists of elements that were originally separated by the line break.

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

This time, each word is treated as its own element, resulting in 22 elements instead of 4. Again, it is easiest to think about `flatMap()` as a map operation followed by a flatten operation, in a single step.



Contents of rddMary converted into individual elements using `flatMap()`

filter()

The filter function is used to remove elements from an RDD that don't meet a certain criteria, or put another way, `filter()` keeps elements in an RDD based on a predicate. If the predicate returns `true` (the filter criteria are met), the record is passed on to the transformed RDD.

In the example below, we have an RDD composed of four elements and want to filter out any element whose value is greater than ten (or, in other words, keep any value ten or less). Notice that the initial RDD is being created using the `sc.parallelize` API:

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

This could have been performed using any standard mathematical operation.

`Filter` is not limited to working with numbers. It can work with `strings` as well. Let's use an example RDD consisting of the list of months below:

```
months = ["January", "March", "May", "July", "September"]
rddMonths = sc.parallelize(months)
```

We then use `filter`, with an anonymous function that uses the `len` function to count the number of characters in each element, and then filter out any that contain five or fewer characters.

```
rddMonths.filter(lambda name: len(name) > 5).collect()
['January', 'September']
```

Again, any available function that performs evaluations on text strings or arrays could be used to filter for a given result.

distinct()

The `distinct` function removes duplicate elements from an RDD. Consider the following RDD:

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

Notice that the numbers 5 and 14 are listed twice. If we just wanted to see each element only listed one time in our output, we could use `distinct()` as follows:

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

Notice that 5 and 14 now only appear once in the results.

Basic Spark Actions

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

The `collect` function returns the entire RDD on which it is run against. In addition, the `first` function returns just the first element in an RDD. The `take` function allows for the specification of a number of elements, and returns that number of elements when executed. Below are some examples of these functions in action:

```
rddNumList = sc.parallelize([5, 7, 11, 14])
rddNumList.collect()
[5, 7, 11, 14]
rddNumList.first()
5
rddNumList.take(2)
[5, 7]
```

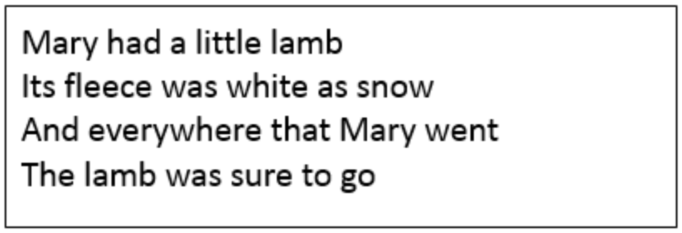
count()

`count()` returns the number of elements in the RDD. Here is an example:

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

In the case of a file that contains lines of text, `count()` would return the number of lines in the RDD, as in the following example:

```
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

The count function applied to rddMary returns 4, a count for each line

`reduce()`

`reduce()` aggregates the elements of an RDD using a function that takes two arguments and returns one, iterating through them until only a single value remains. For example:

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

rddNumList.reduce(lambda a,b: a+b)

...

37
```

Because of the parallelized nature of RDDs, the reduce function does not necessarily process RDD elements in a particular order. Therefore, the function used with `reduce()` should be both commutative and associative.

Commutative means "to move around" - which implies that the order things are done in should not matter. For example, 5+7 = 7+5, or 5*7 = 7*5. A function that is not commutative can return different results each time it is run. For example, 5 / 7 does not equal 7 / 5, so a function that employed division would not be commutative.

Associative refers to the way numbers are grouped using parenthesis. For example, 5 + (7 + 5) equals (5 + 7) + 5, or 5 * (7 * 5) = (5 * 7) * 5. A function that is not associative can return different results each time it is run. For example 5 * (7 + 5) does not equal (5 * 7) + 5, so this would not be associative.

saveAsTextFile()

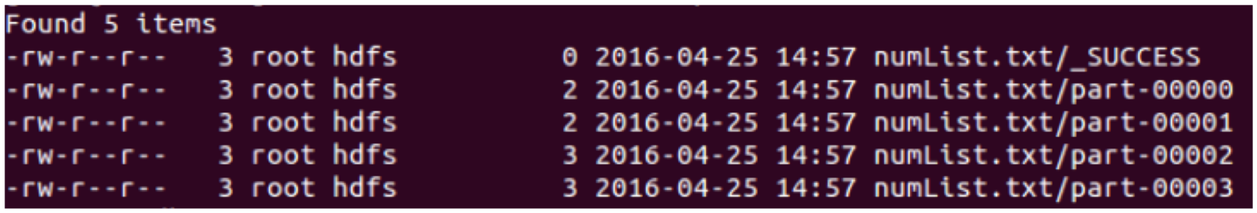
The `saveAsTextFile` function writes the contents of RDD partitions to a specified location (such as `hdfs://` for HDFS, or `file:/` for local file system, and so forth) and directory as a set of text files. For example:

```
rddNumList.saveAsTextFile("hdfs://desiredLocation/foldername")
```

The contents of the RDD in the example would be written to a specific a directory in HDFS.

Success can be verified using typical tools from a command line or GUI. In the case of our example, we could use the `hdfs dfs -ls` command to verify it had been written successfully:

```
$ 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
```

Using `saveAsTextFile()`, each RDD partition is written to a different text file by default

The output would look like the screenshot shown, with each RDD partition being written to a different text file by default. The files could be copied to the local file system and then be read using a standard text editor / viewer such as `nano`, `more`, `vi`, etc.

Transformations vs Actions: Lazy Evaluation

Transformations are *lazy*, meaning they do not actually perform any computations until an action is performed. Lazy evaluation prevents the processing of unneeded data. When a transformation alone is applied to an RDD, initially nothing happens. The RDD just keeps track of all the transformations applied and will execute them when it's necessary.

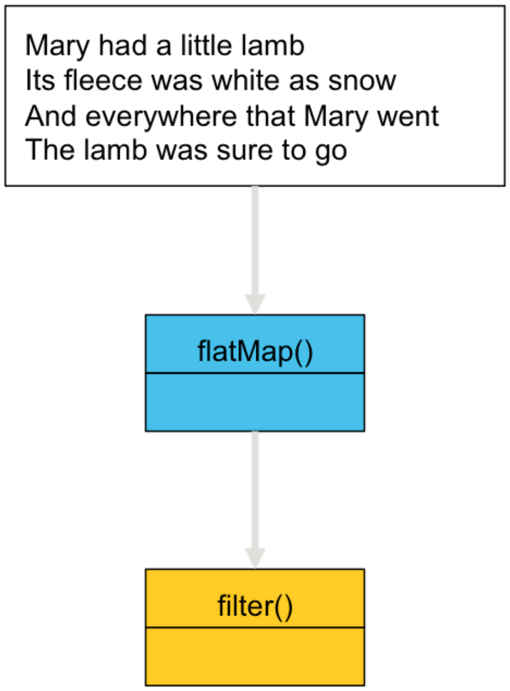
Let's look at an example:

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

In the above code, two transformations were applied (`flatMap` and `filter`), but it was not until the `count` action is called that the RDD runs through the various transformations.

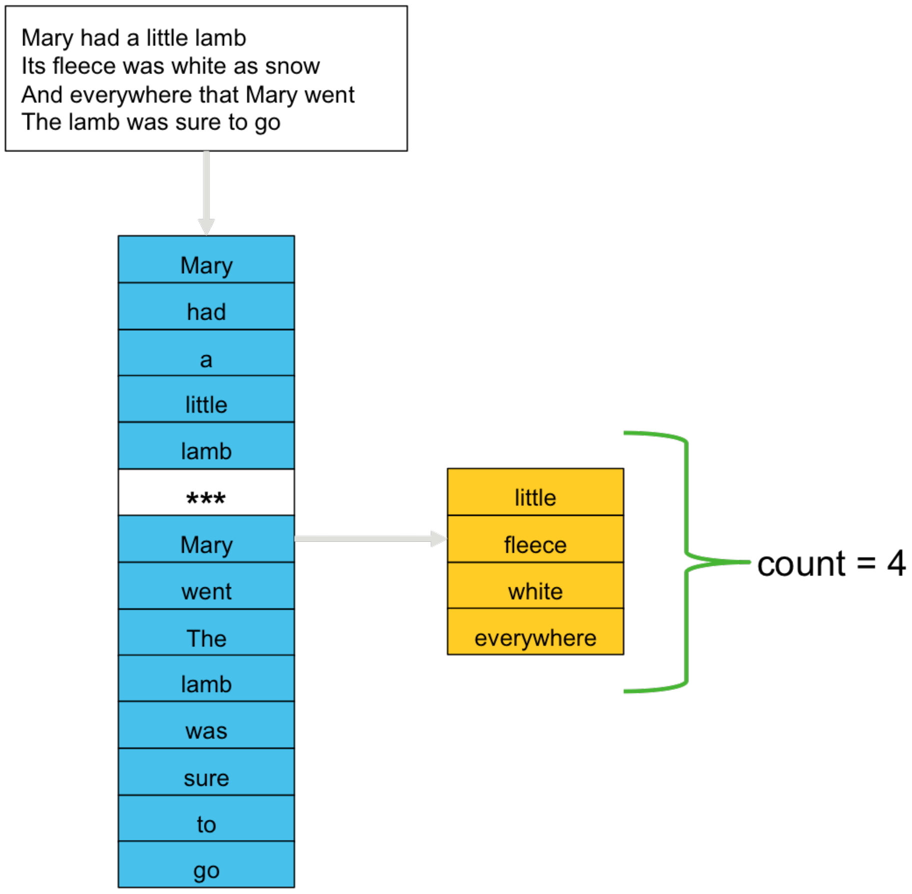
Lazy Evaluation Visualized

Here is picture of what is happening behind the scenes when building transformations, prior to executing actions.



Transformations prior to an action being executed

As the visual indicates, when performing transformations on an RDD, it just saves the recipe of what it is supposed to do when needed.



When the action is called, the data is pushed through the transformations so that the result can be calculated

Only at the end, when an action is called, will the data be pushed through the recipe to create the desired outcome.

RDD Special Topics

We'll finish this lesson by looking at some additional RDD programming topics not already discussed, including some functions that work with multiple RDDs, the creation and use of named functions, and some basic numeric operations available for appropriate RDDs.

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

To combine two RDDs, use the `union` function. This transformation is applied to an RDD and takes a second RDD as an argument, resulting in a third RDD that represents a combination of the two. For example, take the following two RDDs:

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

Let's say we want to create a single RDD that is made up of the contents of the first RDD followed by the contents of the second RDD. The union function would accomplish this with the following code:

```
rddCombined = rddNumList.union(rddNumList2)
```

We can then use `collect()` to view our new RDD:

```
rddCombined.collect()
[5, 7, 11, 14, 2, 4, 5, 14, 21]
```

The `intersection` function is a transformation in which only the values that exist in both RDDs are output. You can create such an RDD and verify it worked using the following code:

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

NOTE:

If there was no need to use the union or intersected RDDs for future purposes, the results above could have been obtained in each case with the following single lines of code:

```
rddNumList.union(rddNumList2).collect()
```

...and...

```
rddNumList.intersection(rddNumList2).collect()
```

Named Functions

Custom functions can be defined and named, then used as arguments to other functions. A custom function should be defined when the same code will be used multiple times throughout the program, or if a function argument will take more than a single line of code, making it too complex for an anonymous function.

The following example evaluates a number to determine if is 90 or greater. If true, it returns the text string "`A`" and if false it returns "`Not an A`".

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

In the REPL, the number of `tabs` matter. For example, in line 2, you have to `tab` once before typing the line, and in line 3 you must `tab` twice. In line 4, you have to `tab` once, and in line 5, twice.

The custom named function `gradeAorNot` can then be passed as an argument to another function - for example, `map()`.

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

The named function could also be used as the function body in an anonymous function. The following example results in equivalent output to the code above:

```
rddGrades.map(lambda grade: gradeAorNot(grade)).collect()
```

Numeric Operations

Numeric operations can be performed on RDDs, including `mean`, `count`, `stDev`, `sum`, `max`, and `min`, as well as a `stats` function that collects several of these values with a single function. For example:

```
rddNumList = sc.parallelize([5, 7, 11, 14])
rddNumList.stats()
(count: 4, mean: 9.25, stdev: 3.49…, max: 14, min: 5)
```

The individual functions can be called as well:

```
rddNumList.min()
5
```

IMPORTANT:

To double check the output of the Spark `stdev()` in Excel, use Excel's `stdevp` function rather than the `stdev` function. Excel's `stdev` function assumes that the entire population is unknown, and thus makes adjustments to outputs based on assumed bias. The `stdevp` function assumes the entire dataset (the `p` stands for "`population`") is fully represented and does not make a bias correction. Thus, Excel's `stdevp` function is more similar to Spark's `stdev` function.

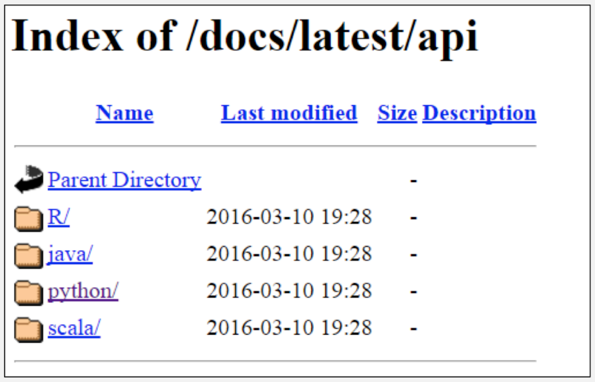
REFERENCE:

There are many other Spark APIs available at:

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

`<version>` can be an actual version number, such as "`1.4.0`" or "`1.6.1`", or alternatively you can use "`latest`" to view documentation on the newest release of Spark. For example:

http://spark.apache.org/docs/latest/api/



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	-	

From there, click on the appropriate programming language to view the documentation.

Knowledge Check

You can use the following questions and answers as a self-assessment.

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) Which transformation will take all of the words in a text object and break each of them down into a separate element in an RDD?

5) **True or False:** The `count` action returns the number of lines in a text document, not the number of words it contains.

6) What is it called when transformations are not actually executed until an action is performed?

7) **True or False:** The `distinct` function allows you to compare two RDDs and return only those values that exist in both of them

8) **True or False:** Lazy evaluation makes it possible to run code that "performs" hundreds of transformations without actually executing any of them

Answers

1) What does RDD stand for?

Answer: Resilient Distributed Dataset

2) What two functions were covered in this lesson that create RDDs?

Answer: `sc.parallelize()` and `sc.textfile()`

3) **True or False:** Transformations apply a function to an RDD, modifying its values

Answer: **False.** Transformations result in new RDDs being created. In Spark, data is immutable.

4) Which transformation will take all of the words in a text object and break each of them down into a separate element in an RDD?

Answer: `flatmap()`

5) **True or False:** The `count` action returns the number of lines in a text document, not the number of words it contains.

Answer: **True**

6) What is it called when transformations are not actually executed until an action is performed?

Answer: Lazy evaluation

7) **True or False:** The `distinct` function allows you to compare two RDDs and return only those values that exist in both of them

Answer: **False.** The `intersection` function performs this task. The `distinct` function would remove duplicate elements, so that each element is only listed once regardless of how many times it appeared in the original RDD.

8) **True or False:** Lazy evaluation makes it possible to run code that "performs" hundreds of transformations without actually executing any of them

Answer: **True**

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

Pair RDDs

Lesson Objectives

After completing this lesson, students should be able to:

- ✓ Use Core RDD functions and create your own function
- ✓ Perform common operations on Pair RDDs

Pair RDD Introduction

Key-value Pair RDDs (a.k.a. "Pair RDDs") are made up of elements comprised of key-value pairs. By creating Pair RDDs in Spark, developers gain access to additional API functionality. This provides the developer with the capability to directly interact with RDD data in much the same way they might interact with data using Hive on Spark, Spark on Hive, or Spark SQL.

Creating Pair RDDs

Let's look at some ways to create a pair RDD.

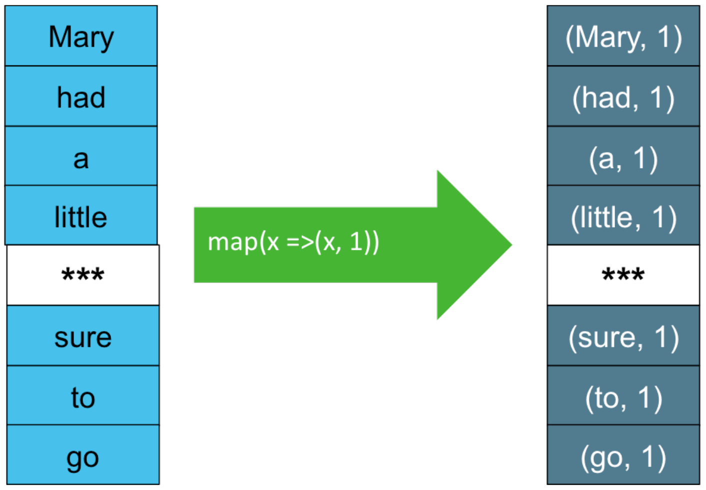
map()

The first two lines of the example below should look very familiar. We are first creating an RDD from a file called "`mary.txt`" and then calling a `flatMap`. Once each word is now its own element, key-value pairs can be created.

```
rdd = sc.textFile("filelocation/mary.txt")
rddFlat = rdd.flatMap(lambda line: line.split(' '))
```

A `map` transformation can be used to create the key-value pair. In this process, a function is passed to `map()` to create a tuple. In the example below, we create an anonymous function which returns a tuple of `(word,1)`.

```
kvRdd = rddFlat.map(lambda word: (word,1))
kvRdd.collect()
```



Mapping words to tuples

The illustration visually demonstrates what happens when the map function is applied on the initial elements, each consisting of a single word.

`keyBy()`

The `keyBy` API creates key-value pairs by applying a function on each data element. The function result becomes the key, and the original data element becomes the value in the pair.

For example:

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

Additional example:

```
rddThreeWords = sc.parallelize(["cat","A","spoon"])
keyByRdd2 = rddThreeWords.keyBy(len)
keyByRdd2.collect()
[(3,'cat'),(1,'A'),(5,'spoon')]
```

`zipWithIndex()`

The `zipWithIndex` function creates key-value pairs by assigning the index, or numerical position, of the element as the value, and the element itself as the key.

For example:

```
rddThreeWords = sc.parallelize(["cat","A","spoon"])
zipWIRdd = rddThreeWords.zipWithIndex()
zipWIRdd.collect()
[('cat', 0), ('A', 1), ('spoon', 2)]
```

zip()

The `zip` function creates key-value pairs by taking elements from one RDD as the key and elements of another RDD as the value. It has the following syntax:

```
keyRDD.zip(valueRDD)
```

The API assumes the two RDDs have the same number of elements. If not, it will return an error.

```
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 Operations

`mapValues()`

The `mapValues` function performs a defined operation on the values in a Pair RDD while leaving the keys unchanged. For example:

```
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()`

Some common functions to call when working with Pair RDDs include `keys()`, `values()`, and `sortByKey()`. We will explore them using the `rddMapVals` Pair RDD we created in the previous example:

```
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()` - returns a list of just the keys in the RDD without any values.

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

- `values()` - returns a list of just the values in the RDD without any keys.

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

- `sortByKey(ascending=False)` - sorts the RDD alphanumerically by key. By default, will sort from smallest to largest value (`ascending=True`). If `ascending` is explicitly set to `False`, it orders from largest to smallest.

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

IMPORTANT:

Without creating a `PairRDD` prior to using these functions, they will not work as expected.

Reordering Key-Value Pairs using `map()`

To reorder the placement of elements in key-value pairs, you can use pattern matching available via the `map` function. The example below demonstrates how to take a simple Pair RDD and switch the order such that the key is now the value, and the value becomes the key:

```
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')]
```

This can be useful when you want to use a function that is designed to work on a key (for example, the `sortByKey` function), but have it applied to a value instead.

For more complex key-value pairs (for example, keys that contain lists of values instead of just a single value) more complex patterns can be invoked in the anonymous function. For example:

```
rddKeyTwoVal = sc.parallelize([("cat", (0,1)), ("spoon", (2,3))])

rddK2VReorder = rddKeyTwoVal.map(lambda (key, (val1, val2)) : ((key, val1) ,
val2))
rddK2VReorder.collect()
[(('cat', 0), 1),  (('spoon', 2), 3)]
```

Virtually no limit exists for how key-value pairs can be rearranged in an RDD.

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

There are many other functions available for Pair RDD transformations and actions. We will go back to the `keyByRdd` example created earlier to discuss `lookup()`, `countByKey()`, and `collectAsMap()`. As a reminder, here is how the RDD was created:

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

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

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

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

- `countByKey()` - returns a count of the number of times each key appears in the RDD (in our example, there were no duplicate keys, so each is returned as `1`).

```
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)}
```

Note that these actions did not require us to also specify `collect()` in order to view the results.

`reduceByKey()`

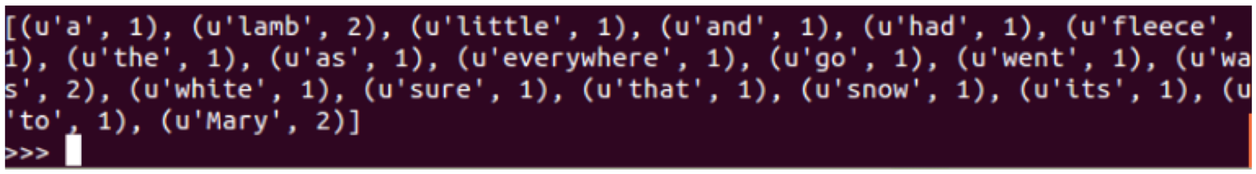
The `reduceByKey` function performs a reduce operation on all elements of a key/value pair RDD that share a key. For our example here, we'll return to `kvRdd` that was created using the following code:

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

As an example of `reduceByKey()`, take a look at the following code:

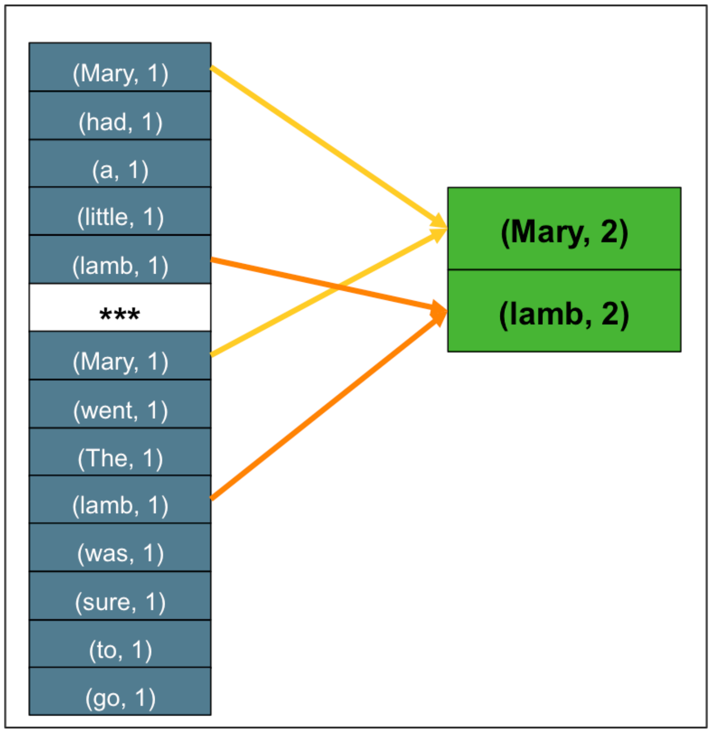
```
kvReduced = kvRdd.reduceByKey(lambda a,b: a+b)
```

The `reduceByKey` function goes through the elements and if it sees a key that it hasn't already encountered, it adds it to the list and records the value as-is. If a duplicate key is found, `reduceByKey` performs a function on the key values and keeps the number of keys to one. In our example, then, the anonymous function "`lambda a,b: a+b`" only kicks in if a duplicate key is found. If so, the anonymous function tells `reduceByKey` to take the values of the two keys (`a` and `b`) and add them together to compute a new value for the now-reduced key. The actual function being performed is up to the developer, but incrementally adding values would be a fairly common task.



```
[(u'a', 1), (u'lamb', 2), (u'little', 1), (u'and', 1), (u'had', 1), (u'fleece',
1), (u'the', 1), (u'as', 1), (u'everywhere', 1), (u'go', 1), (u'went', 1), (u'wa
s', 2), (u'white', 1), (u'sure', 1), (u'that', 1), (u'snow', 1), (u'its', 1), (u
'to', 1), (u'Mary', 2)]
>>>
```

Observe that the keys 'Mary', 'was', and 'lamb' have been reduced



The duplicate keys are reduced ('was' not shown)

Visually, then, what is happening is that the elements of the RDD are being recorded and passed to the new `kvReduced` RDD, with the exception of two keys - 'Mary' and 'lamb' - which were both found twice. All other values remain unchanged, but now 'Mary' and 'lamb' are reduced to single key, each with a value of 2.

`groupByKey()`

Grouping values by key allow us to aggregate values based on a key. In order to see this grouping, the results must be turned into a list before being collected.

For example, let's again use our `kvRdd` example created with the following code:

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

Next, we will use `groupByKey` to group all values that have the same key into an iterable object (that is, on its own, unable to be viewed) and then use a `map` function to define these grouped elements into a readable 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])]
```

If we had simply generated output using `groupByKey` alone, as below:

```
kvGroupByKey = kvRdd.groupByKey()
```

… the output would have looked something like this:

```
[(u'a', <pyspark.resultiterable.ResultIterable object at 0xde8450>), (u'lamb',
<pyspark.resultiterable.ResultInterable object at 0xde8490>),…(u'Mary',
<pyspark.resultiterable.ResultIterable object at 0xde8960>)]
```

This tells you that the results are an object that allows iteration, but does not display the individual elements by default. Using `map` to list the elements, performed the necessary iteration to see the desired formatted results.

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

NOTE:

The `groupByKey` and `reduceByKey` functions have significant overlap and similar capabilities, depending on how the called function is defined by the developer. When either is able to get the desired output, it is better to use `reduceByKey()` as it is more efficient over large datasets.

`flatMapValues()`

Like the `mapValues` function, the `flatMapValues` function performs a function on Pair RDD values, leaving the keys unchanged. However, in the event it encounters a key that has multiple values, it flattens those into individual key-value pairs, meaning no key will have more than one value, but you will end up with duplicate keys in the RDD. Let's start with the RDD we created in the `groupByKey()` example:

```
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])]
```

Notice that both the `'lamb'` and `'Mary'` keys contain a multiple key values in a list. Next, let's create an RDD that flattens those key-value pairs using the `flatMapValues` function:

```
rddFlatMapVals = kvGroupByKey.flatMapValues(lambda val: val)
rddFlatMapVals.collect()
[(u'a', 1), (u'lamb', 1), (u'lamb', 1), (u'little', 1), … (u'Mary',1),
(u'Mary',1)]
```

Now all key-value pairs have only a single value, and both `'lamb'` and `'Mary'` exist as duplicated keys in the RDD.

NOTE:

In the example above, the anonymous function simply returns the original unedited value. However, like the `mapValues` function, `flatMapValues()` can be configured to modify the keys as it flattens out the key-value pairs. For example, if we had defined the anonymous function as: `(lambda val: val + 1)`, then each of the values would have been returned as `2` instead of `1`.

`subtractByKey()`

The `subtractByKey` function will return key-value pairs containing keys not found in another RDD. This can be useful when you need to identify differences between keys in two RDDs. Here is an example:

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

The key-value pair that had a key of `'cradle'` was the only one returned because both RDDs contained key-value pairs that had key values of `'cat'` and `'spoon'`.

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

Notice that `('A', 1)` was not returned as part of the result. This is because `subtractByKey()` is only evaluating matches in the first RDD (the one that precedes the function) as compared to the second one. It does not return all keys in either RDD that are unique to that RDD. If you wanted to get a list of all unique keys for both RDDs using `subtractByKey()`, you would need to run the operation twice - once as shown, and then again, swapping out the two RDD values in the last line of code. For example, this code would return the unique key values for `zipWIRdd`:

```
zipWIRdd.subtractByKey(rddSong).collect()
[('A', 1)]
```

If needed, you could store these outputs in two other RDDs, then use another function to combine them into a single list as desired.

Pair RDD Joins

Spark provides support for inner, full outer, left outer, and right outer joins when working with Pair RDDs. Here is an example of using `leftOuterJoin()`:

```
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))]
```

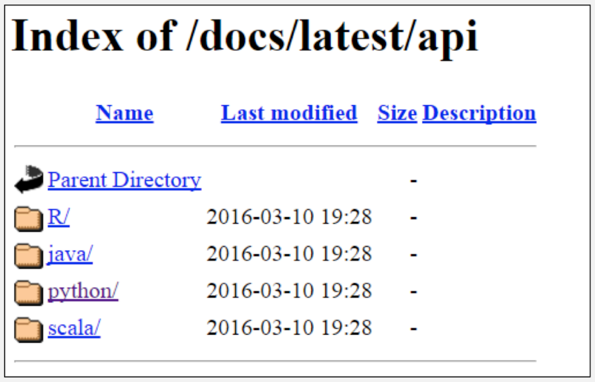
REFERENCE:

There are many other Spark APIs available at:

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

`<version>` can be an actual version number, such as "`1.4.0`" or "`1.6.1`", or alternatively you can use "`latest`" to view documentation on the newest release of Spark. For example:

http://spark.apache.org/docs/latest/api/



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	-	

From there, click on the appropriate programming language to view the documentation.

Apache Spark documentation

Knowledge Check

You can use the following questions and answers as a self-assessment.

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?

Answers

1) An RDD that contains elements made up of key-value pairs is sometimes referred to as a ________________.

Answer: Pair RDD

2) Name two functions that can be used to create a Pair RDD.

Answer: `map()`, `keyBy()`, `zipWithIndex()`, `zip()`

3) **True or False:** A key can have a value that is actually a list of many values.

Answer: True

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?

Answer: First use `map()` to reorder the key-value pair so that the key is now the value. Then use `sortByKey()` to sort. Finally, use `map()` again to swap the keys and values back to their original positions.

5) You determine either `reduceByKey()` or `groupByKey()` could be used in your program to get the same results. Which one should you choose?

Answer: `reduceByKey()`, *because* it is more efficient - especially on large datasets.

6) How can you use `subtractByKey()` to determine *all* of the unique keys across two RDDs?

Answer: Run it twice, switching the order of the RDDs each time.

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.

Spark Streaming

Lesson Objectives

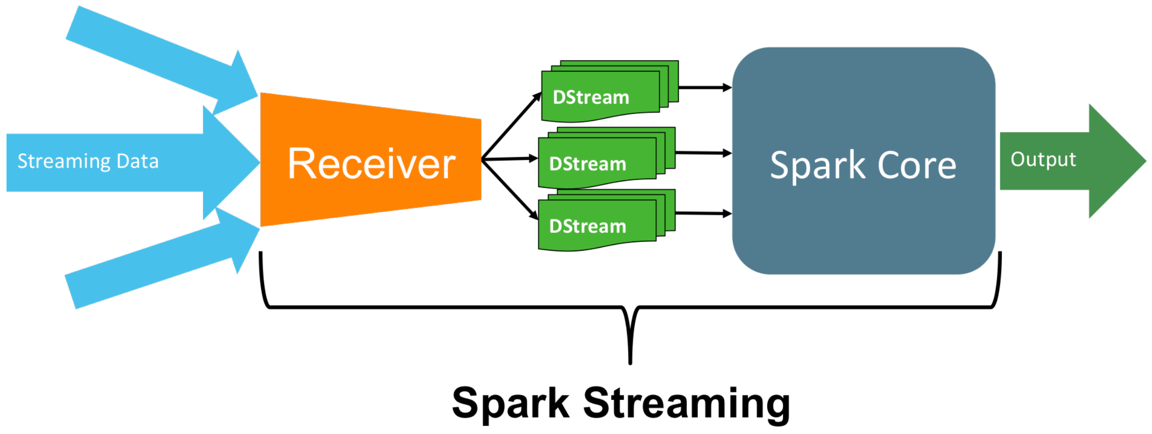
After completing this lesson, students should be able to:

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

Spark Streaming Overview

What is Spark Streaming?

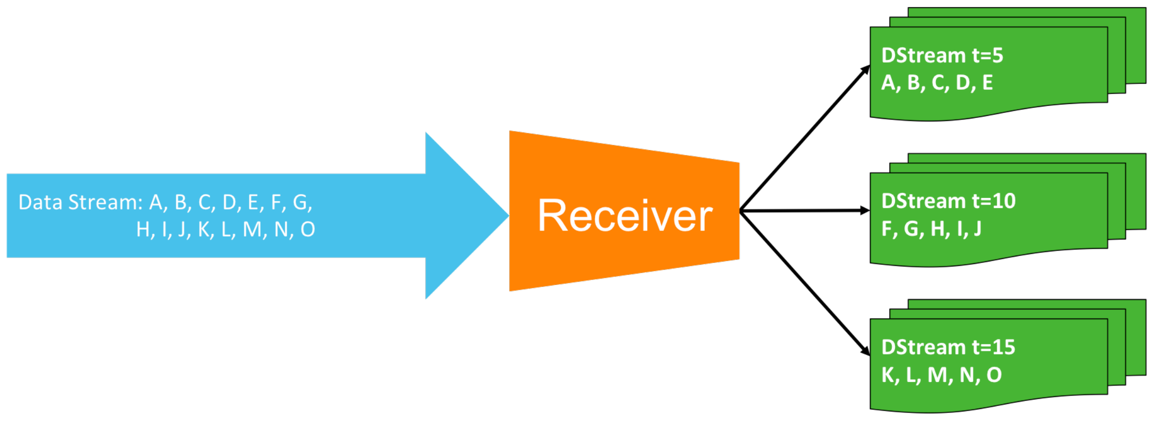
As the name implies, Spark Streaming is a tool that enables a developer to utilize the Spark platform to ingest and process streaming data. Spark Streaming is composed of a **receiver**, which collects streaming data in small amounts called micro-batches, and then makes them available for processing via a collection of specialized RDDs called a **DStream.** These DStreams can then be manipulated and further processed via the Spark Streaming API, which is an extension of **Spark Core**, to produce various output types.



Spark streaming

DStreams

A DStream is a collection of one or more specialized RDDs divided into discrete chunks based on a time interval. When a streaming source communicates with Spark Streaming, the receiver caches information for a specified time period, after which the data is converted into a DStream and made available for further processing. Each discrete time period (in the example pictured, every five seconds) is a separate DStream.



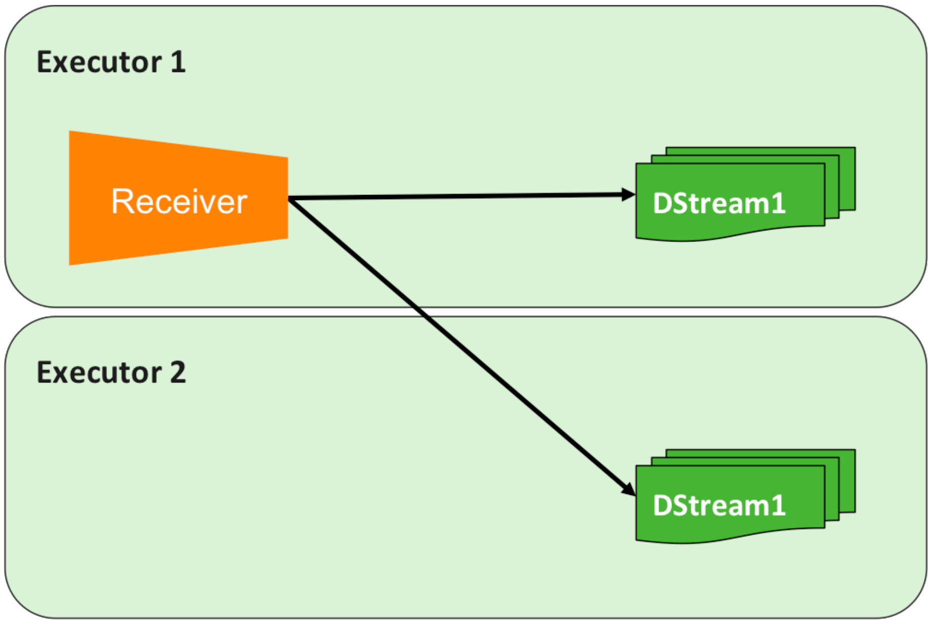
DStreams

DStream vs. RDD

DStreams have some differences compared to traditional RDDs. The **first**, and perhaps most consequential difference, is that DStreams contain data and physically exist in memory from the moment of creation. Traditional RDDs, on the other hand, are sets of instructions that don't actually contain in-memory data until an action is performed. The **second** difference is that, by default, DStreams only exist in memory until the next batch of data is ready to be processed. A **third**, difference is terminology: while you perform transformations on both RDDs and DStreams, RDDs create results via "*actions*" whereas DStreams result in similar "*outputs*."

DStream Replication

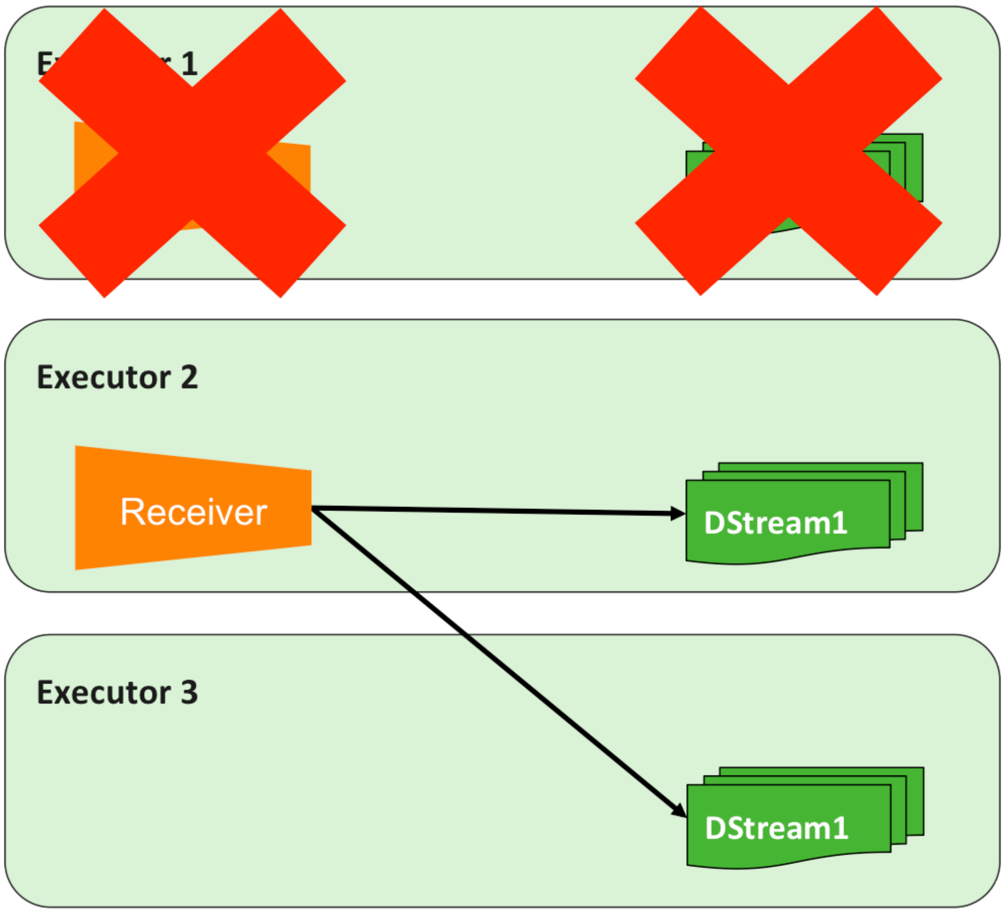
DStreams are fault tolerant, meaning they are written to a minimum of two executors at the moment of creation. The loss of a single executor will not result in the loss of the DStream.



Receiver duplicates data to two executors by default

Receiver Availability

By default, receivers are highly available. If the executor running the receiver goes down, the receiver will be immediately restarted in another executor.



A failed receiver will be restarted on another executor

Spark Streaming performs micro-batching rather than true bit-by-bit streaming. Collecting and processing data in batches can be more efficient in terms of resource utilization, but comes at a cost of latency and the risk that small amounts of data could be lost. Spark Streaming can be configured to process batch sizes as small as one second, which takes approximately another second to process, for a two-second delay from the moment the data is received until a response can be generated. This introduces a small risk of data loss, which can be mitigated by the use of reliable receivers (available in the Scala and Java APIs only at the time of this writing) and intelligent data sources.

Receiver Reliability

By default, receivers are "unreliable." This means there is:

- 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. A reliable receiver implements a handshake mechanism that acknowledges that data has been received and processed. Assuming the data source is also intelligent, it will wait to discard the data on the other side until this acknowledgement has been received, which also means it can retransmit it in the event of data loss. Custom receivers are available in the Scala and Java Spark Streaming APIs only, and are not available in Python. For more information on creating and implementing custom / reliable receivers, please refer to Spark Streaming documentation.

Streaming Data Source Examples

Spark Streaming can receive data from multiple sources, both basic and advanced types. Basic sources include monitoring HDFS directories and ingesting any new files that are created as streamed data, text streaming via a TCP socket connection, and even setting up queues of traditional RDDs that can be used to test a streaming application. Advanced sources include Apache Kafka, Amazon Kinesis, Apache Flume, and MQTT.

REFERENCE:

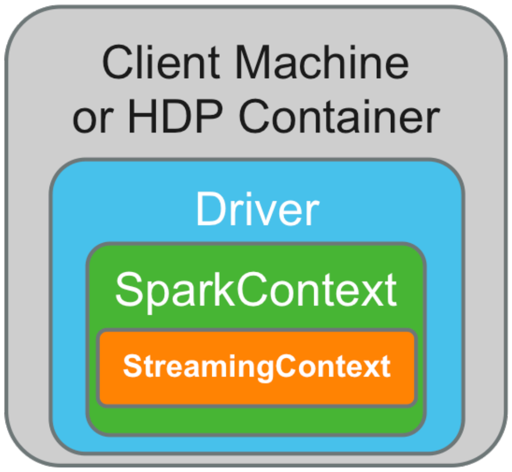
Additional data sources are available via the Scala and Java APIs. Please refer to the Spark Streaming documentation for additional information.

Visit http://spark.apache.org/documentation.html

Basic Streaming

`StreamingContext`

Spark Streaming extends the Spark Core architecture model by layering in a `StreamingContext` on top of the `SparkContext`. The StreamingContext acts as the entry point for streaming applications. It is responsible for setting up the receiver and enables real-time transformations on DStreams. It also produces various types of output.



The `StreamingContext`

Modifying REPL CPU Cores

When running streaming applications, it is usually beneficial to utilize multiple CPU cores. By default, Spark REPLs run programs on the local machine - which, in this course, is named "sandbox" - and are assigned to a single CPU core. As a general recommendation, a Spark Streaming application should have a number of cores equal to the number of DStreams being ingested plus a core for the receiver itself. Thus, if you have 20 DStreams, you should have 21 CPU cores available for best performance. This behavior can be changed by modifying the `MASTER` environment variable, which in the REPL can be done at launch. For example, to utilize two cores in `pyspark`:

```
# pyspark --master local[2]
```

Launch StreamingContext

To launch the `StreamingContext`, you first need to import the `StreamingContext` API. In `pyspark`, the code to perform this operation would be:

```
from pyspark.streaming import StreamingContext
```

Next, you create an instance of the `StreamingContext`. When doing so, you supply the name of the `SparkContext` in use, as well as the time interval (in seconds) for the receiver to collect data for micro-batch processing. When using a REPL, the `SparkContext` will be named `sc` by default. For example, when creating an instance of `StreamingContext` named `ssc`, in the `pyspark` REPL, with a micro-batch interval of one second, you would use the following code:

```
ssc = StreamingContext(sc, 1)
```

IMPORTANT:

This operation will return an error if the `StreamingContext` API has not been imported.

Both the name of the `SparkStreaming` instance and the time interval can be modified to fit your purposes. Here is an example of creating a `StreamingContext` instance with a 10-second micro-batch interval:

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

IMPORTANT:

While multiple instances of `StreamingContext` can be defined, only a single instance of `SparkContext` can run per JVM. Once running, another instance will fail to launch. In fact, once the current instance has been stopped, it cannot be launched again in the same JVM. Thus, while the REPL is a useful tool for learning and perhaps testing Spark Streaming applications, in production it would be problematic because every time a developer wanted to test a slightly different application, it would require stopping and restarting the REPL itself.

Streaming from HDFS Directories and TCP Sockets

To create a stream by monitoring an HDFS directory and ingesting any new files that appear, choose a variable name for the DStream, then call the `textFileStream()` function from the `StreamingContext` API and supply the full path to the appropriate HDFS directory. For example:

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

NOTE:

If the HDFS directory exists on the cluster the application is attached to, only the path needs to be provided. If it exists on a separate cluster, prepend the path with `"hdfs:/nodename:8020/…"`

To create a stream by monitoring a TCP socket source, choose a variable name for the DStream, then call the `socketTextStream()` function and supply the source hostname or IP address (whichever is appropriate in your situation) and the port number to monitor. For example:

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

Notice that in both examples, the name of the `StreamingContext` instance had to be specified before calling the function. Otherwise, the application would have tried to call this from the default `SparkContext`, and since these functions to not exist outside of Spark Streaming, an error would have been returned.

Output to Console and to HDFS

To print streaming output to the console / terminal window, in Python use the "pretty print" function, `pprint()`. In Scala/Java the same function is simply named `print()`. For example:

- Python: `DSvariableName.pprint()`
- Scala/Java: `DSvariableName.print()`

When printing output to the console, we suggest setting the log level for the `SparkContext` to "`ERROR`" in order to reduce screen output. Otherwise, lots of information besides what is being streamed will appear and clutter up the screen. To do this, use the `SparkContext setLogLevel` function as follows:

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

To save the output as a time-stamped text file on HDFS, use the `saveAsTextFiles` function. Make sure the application has `write` permissions to the HDFS directory selected. The syntax for this operation is:

```
DSVariable.saveAsTextFiles("HDFSlocation/prefix", "optionalSuffix")
```

The output will be generated on a per-DStream basis, with whatever you select as the prefix prepending the rest of the file name, which will be `"-<timestamp>."` In addition, you can choose to add a suffix to each file as well, which will appear at the end of the file. Prefixes and suffixes are useful, particularly if multiple data streams are being written to the same directory.

It is also noteworthy that these outputs are not exclusive. You can choose to output to the console *and* to HDFS without issue.

Starting and Stopping the Streaming Application

With streaming applications, all operations must be defined before the stream is started. Thus, any interactivity benefit of the REPL is lost. Once all transformations and outputs have been defined, launch a streaming application simply by calling the `SparkContext` instance and using the start function. The application can be stopped in the same fashion using the `stop` function.

For example:

- When ready: `ssc.start()`
- When finished: `ssc.stop()`

In a REPL, the streaming application will also stop if the terminal it is running in is closed.

Simple Streaming Program Example Using a REPL

Here's an example of a simple streaming program created using the `pyspark` 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()
```

In the above application:

1. The `pyspark` REPL is called with the MASTER variable configured to use two CPU cores on the local machine
2. The `SparkContext` log level is set to "`ERROR`"
3. Then the `StreamingContext` API is imported
4. Then, an instance of `StreamingContext` named `sscFive` is created, with a micro-batch time interval of five seconds
5. A DStream is defined using the `textFileStream` function to monitor the local HDFS directory `/user/root/test/`
6. Those inputs are then output as text files with a prefix of "`name`" (no suffix is defined) to the local HDFS directory `/user/root/test/stream/`
7. Finally, the inputs are also output to the console using `pprint()`
8. Then the application is launched

Basic Streaming Transformations

DStream Transformations

Transformations allow modification of DStream data to create new DStreams with different output. DStream transformations are similar in nature and scope to traditional RDD transformations. In fact, many of the same functions in Spark Core also exist in Spark Streaming. The following functions should look familiar:

- `map()`
- `flatMap()`
- `filter()`
- `repartition()`
- `union()`
- `count()`
- `reduceByKey()`
- `join()`

Transformation using flatMap()

As an example, here is a Spark Streaming application that utilizes the `flatMap` function and prints output to the console.

```
# pyspark --master local[2]
>>> sc.setLogLevel("ERROR")
>>> from pyspark.streaming import StreamingContext
>>> ssc = StreamingContext(sc, 5)
>>> hdfsInputDS = ssc.textFileStream("someHDFSdirectory")
>>> flatMapDS = hdfsInputDS.flatMap(lambda line: line.split(" ")
>>> flatMapDS.pprint()
>>> ssc.start()
```

Notice that the first five lines of code are similar to the application example from the previous section. In this example, then, the original DStream transformed into a new DStream named `flatMapDS`, and the new DStream is output to the console.

Combining DStreams using `union()`

The union function also works in the same way as it does with traditional RDDs. In this example, we create two DStreams and combine them with `union()`. Notice that we used the same HDFS location in the example, so the output would simply be the same input written twice.

```
# pyspark --master local[2]
>>> sc.setLogLevel("ERROR")
>>> from pyspark.streaming import StreamingContext
>>> ssc = StreamingContext(sc, 5)
>>> input1 = ssc.textFileStream("/user/root/test/")
>>> input2 = ssc.textFileStream("/user/root/test/")
>>> combined = input1.union(input2)
>>> combined.pprint()
>>> ssc.start()
```

Create Key-Value Pairs

Like traditional RDDs, DStreams can be transformed into key-value pairs using `flatMap()` followed by `map()`. Here's an example:

```
# pyspark --master local[2]
>>> sc.setLogLevel("ERROR")
>>> from pyspark.streaming import StreamingContext
>>> ssc = StreamingContext(sc, 5)
>>> hdfsInputDS = ssc.textFileStream("someHDFSdirectory")
>>> kvPairDS = hdfsInputDS.flatMap(lambda line: line.split(" ").map(lambda
word: (word, 1))
>>> kvPairDS.pprint()
>>> ssc.start()
```

reduceByKey()

Also, like traditional RDDs, key-value pair DStreams can be reduced using the `reduceByKey` function. Here's an example:

```
# pyspark --master local[2]
>>> sc.setLogLevel("ERROR")
>>> from pyspark.streaming import StreamingContext
>>> ssc = StreamingContext(sc, 5)
>>> 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()
```

This coding pattern is often used to write word count applications.

Window Transformations

Stateful vs. Stateless Operations

By default, DStreams are discarded from memory when the next batch of data arrives. This behavior is fine assuming all operations / transformations performed are on single DStreams, which are not dependent on previous data. This mode of operation is referred to as working with "**stateless**" operations / transformations.

However, it is sometimes beneficial or necessary to perform transformations and gather output across overlapping time slices, or even across an entire collected dataset. For example, you might want to compute a running average every 15 seconds using the last 45 seconds worth of data. This is referred to as working with "**stateful**" operations / transformations.

Checkpointing

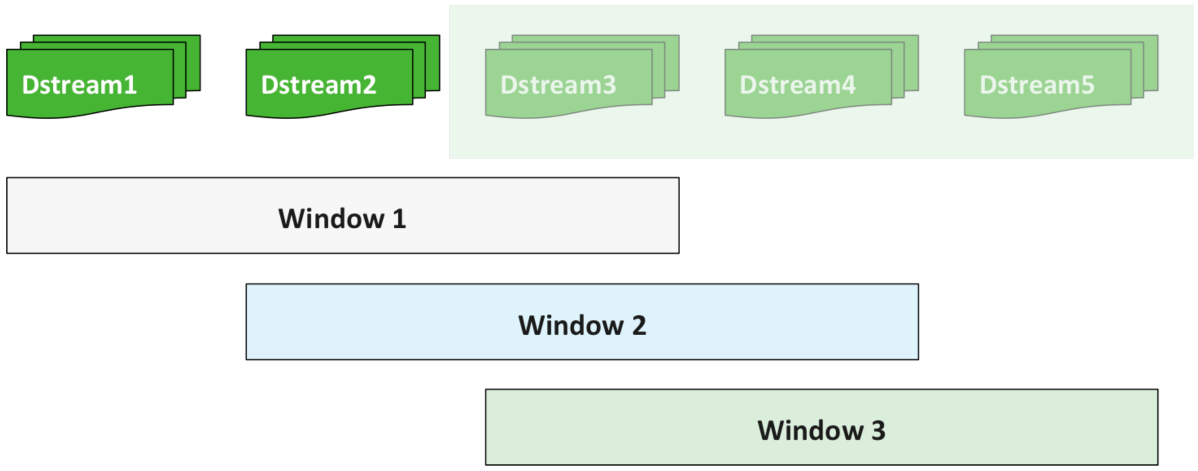
Checkpointing is used in stateful streaming operations to maintain state in the event of system failure. To enable checkpointing, you can simply specify an HDFS directory to write checkpoint data to using the `checkpoint` function. For example:

```
ssc.checkpoint("someHDFSdirectory")
```

Trying to write a stateful application without specifying a checkpoint directory will result in an error once the application is launched.

Streaming Window Functions

Window functions are stateful operations that run on a set of DStreams, combining the results of a group. In the diagram, each window is composed of the last 3 DStreams. The window length, which is the size measured in seconds, and the time interval to collect the results are set during the window creation. Both of these values must be a multiple of the `StreamingContext` micro-batch interval value. For example, if the micro-batch interval is set to five seconds, the window size and interval must be some multiple of five. If this were the case for the example pictured, each window would have a size of 15 seconds, and could be processed at either 5 or 10-second intervals.



Each window comprises the last three DStreams in this scenario

NOTE:

Technically, you could also process a 15-second window in 15-second intervals, however this is functionally equivalent to setting the `StreamingContext` interval to 15 seconds and not using the window function at all.

IMPORTANT:

For basic inputs, `window()` does not work as expected using `textFileStream()`. An application will process the first file stream correctly, but then lock up when a second file is added to the HDFS directory. Because of this, all course labs and examples will use the `socketTextStream` function.

Basic Window Transformations

Two basic window transformations can be performed using the `window` and `countByWindow` functions.

The `window` function takes the window length and collection interval as arguments, and returns a new DStream based on the values provided. Functionally, this is similar to the `union()` transformation, as the data across multiple DStreams is combined into a single unit. The difference is that with streaming, the `union` function only operates on DStreams created simultaneously, whereas the `window` function combines the same DStream created across multiple micro-batches. The basic syntax is `window(windowLength, interval)`. An example of creating a window of 30 seconds collected every 10 seconds would look like the following:

```
windowDS = streamingDS.window(30,10)
```

The `countByWindow` function is equivalent to the window function followed by the Spark Streaming count function, and returns a count of the number of elements in the stream. However, if the number of elements is large, this will be a more efficient way of generating a count. The arguments are the same as `window()`, and has the following syntax: `countByWindow(windowLength, interval)`.

Example:

```
windowCountDS = streamingDS.countByWindow(30,10)
```

Sample Window Application

Here's a sample application that uses `window()` to generate a count of all elements in a data stream, first separating them out using `flatMap()`. (Without `flatMap()`, the number of DStreams would be counted instead.) It generates this output every five seconds, using the last 15 seconds worth of DStream data. Notice that the DStream interval is set to one second for the `StreamingContext`:

```
# 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()
```

`reduceByKeyAndWindow()`

You can also work with key-value pair windows, and there are specialized functions designed to do that. One such example is `reduceByKeyAndWindow()`, which behaves similarly to the `reduceByKey` function discussed previously, but over a specified window and collection interval. For example, take a look at the following application:

```
# pyspark --master local[2]
>>> sc.setLogLevel("ERROR")
>>> from pyspark.streaming import StreamingContext
>>> ssc = StreamingContext(sc, 1)
>>> ssc.checkpoint("/user/root/test/checkpoint/")
>>> 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()
```

To generate the reduced key-value pair, the DStream is transformed using `flatMap()`, then converted to a key-value pair using `map()`. Then, the `reduceByKeyAndWindow` function is called.

Notice that the `reduceByKeyAndWindow` function actually takes two functions as arguments prior to the window size and interval arguments. The first argument is the function that should be applied to the DStream. The second argument is the *inverse* of the first function, and is applied to the data that has fallen out of the window. The value of each window is calculated incrementally as the window slides across DStreams without having to recompute the data across all of the DStreams in the window each time.

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?

Answers

1) Name the two new components added to Spark Core to create Spark Streaming.

Answer: Receivers and DStreams. `StreamingContext` is also an acceptable answer here.

2) If an application will ingest three streams of data, how many CPU cores should it be allocated?

Answer: Four - one for each stream, and one for the receiver.

3) Name the three basic streaming input types supported by both Python and Scala APIs.

Answer: HDFS text via directory monitoring, text via TCP socket monitoring, and queues of RDDs.

4) What two arguments does an instance of `StreamingContext` require?

Answer: The name of the `SparkContext` and the micro-batch interval.

5) What is the additional prerequisite for any stateful operation?

Answer: Checkpointing.

6) What two parameters are required to create a window?

Answer: Window duration and collection/sliding interval.

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

Lesson Objectives

After completing this lesson, students should be able to:

- ✓ List 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

Spark SQL Components

Spark SQL is a Spark module that enables optimized processing of structured data. It provides many advantages to the Spark developer, including automated performance improvements that generally mean better performance than applications built using Spark Core RDD APIs. Spark SQL also integrates seamlessly with Hive, which allows an enterprise to leverage investments already made in that platform (including data stored as well as knowledge and processes that have built over time) while simultaneously benefitting from Spark's improved performance via in-memory processing.

DataFrames

A DataFrame is data that has been organized into one or more columns, similar in structure to an SQL table, but is actually constructed from underlying RDDs. DataFrames can be created directly from RDDs, as well as from Hive tables and many other outside data sources.

There are three primary methods available to interact with DataFrames and tables in Spark SQL:

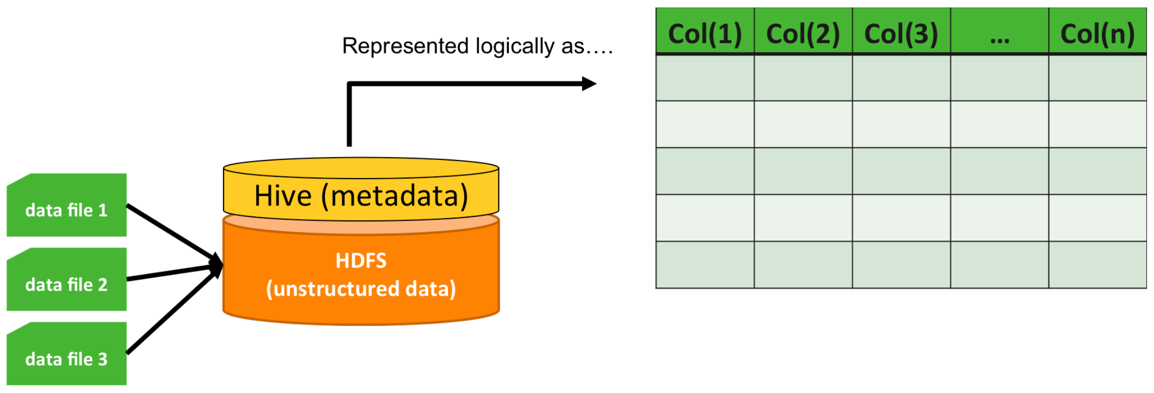
- The **DataFrames API**, which is available for Java, Scala, Python, and R developers
- The native **Spark SQL API**, which is composed of a subset of the SQL92 API commands
- The **HiveQL API.** Most of the HiveQL API is supported in Spark SQL.

Hive

Most enterprises that have deployed Hadoop are familiar with Hive which is the original data warehouse platform developed for Hadoop. It represents unstructured data stored in HDFS as structured tables using a metadata overlay managed by Hive's **HCatalog**, and can interact with those tables via HiveQL, it's SQL-like query language.

Hive is distributed with every major Hadoop distribution. Massive amounts of data are currently managed by Hive across the globe. Thus, Spark SQL's ability to integrate with Hive and utilize HiveQL capabilities and syntax provides massive value for the Spark developer.

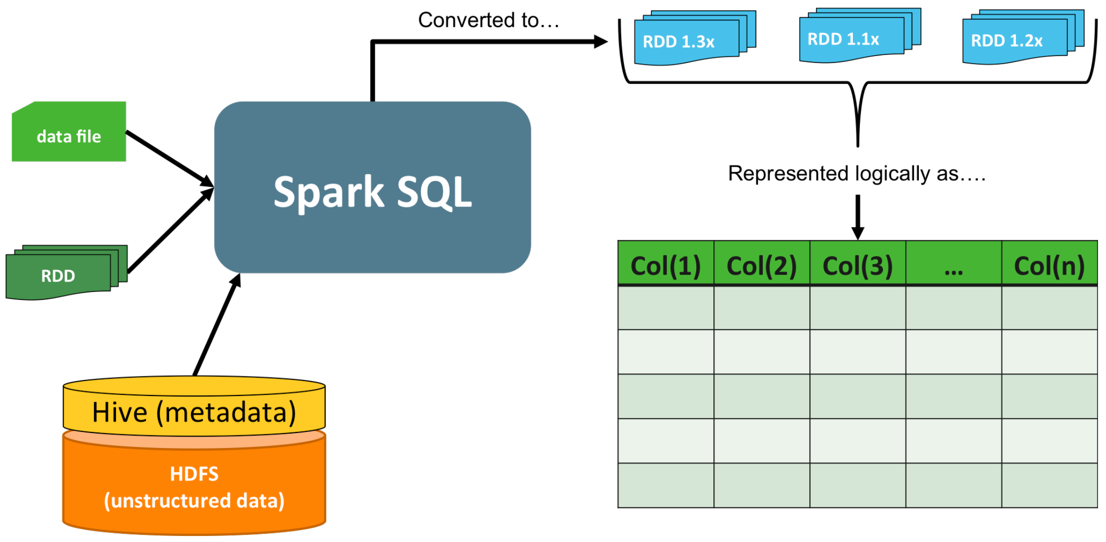
Hive Data Visually



Visual Display of Hive Data

Hive data starts as raw data that has been written to HDFS. Hive has a metadata component that logically organizes these unstructured data files into rows and columns like a table. The metadata layer acts as a translator, enabling SQL-like interactions to take place even though the underlying data on HDFS remains unstructured.

DataFrame Visually



Visual Display of DataFrame Data

In much the same way, a DataFrame starts out as something else - perhaps an ORC or JSON file, perhaps a list of values in an RDD, or perhaps a Hive table. Spark SQL has the ability to take these (and other) data sources and convert them into a DataFrame. As mentioned earlier, DataFrames are actually RDDs, but are represented logically as rows and columns. In this sense, Spark SQL behaves in a similar fashion to Hive, only instead of representing files on a disk as tables like Hive does, Spark SQL represents RDDs in memory as tables.

Spark SQL Contexts

Spark requires a Spark context, Spark Streaming requires a Spark Streaming context, and Spark SQL requires an SQL context. The SQL context, however, provides two contexts to choose from:

- The `SQLcontext`
  or
- The `HiveContext`.

In both cases, the default name usually given to the context is `sqlContext`, but this is at the discretion of the developer.

Spark SQL Contexts – Python

The basic code difference is which context gets imported at the beginning of the program. The options are:

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

or

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

Zeppelin and Spark REPLs both default to using a `HiveContext` with a name of `sqlContext`. For most of the code examples in this lesson, we will follow the same pattern. For Zeppelin, this becomes active when running code prepended by `%sql`.

Spark SQL Contexts – Scala

To create instances of either `SQLContext` or `HiveContext` using Scala you would use:

```
import org.apache.spark.sql._
val sqlContext = SQLContext(sc)
```

or

```
import org.apache.spark.sql.hive._
val sqlContext = HiveContext(sc)
```

SQLContext vs. HiveContext

When deciding which Spark SQL context to use, keep the following in mind:

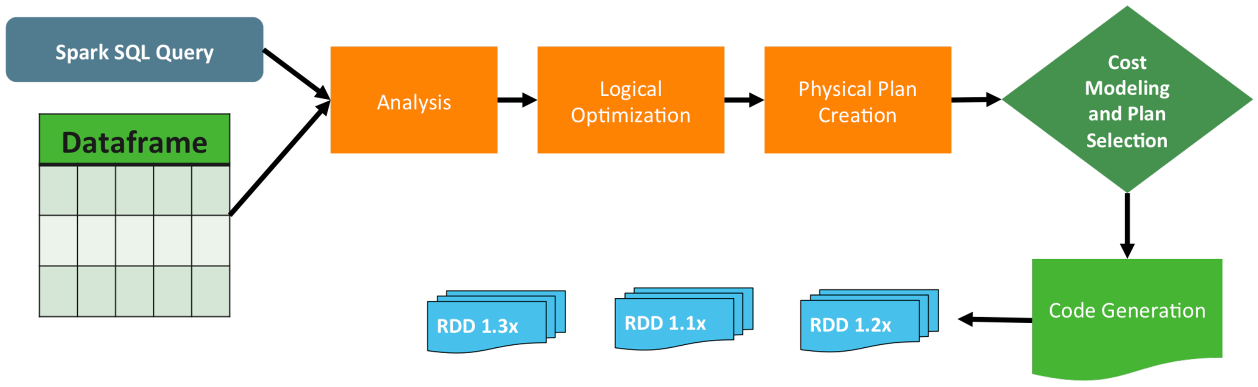
SQLContext is a standalone API that provides a limited, generic `SQL92` parser.

HiveContext contains all the capabilities of `SQLContext` because it is a superset of `SQLContext` - meaning it contains all of the `SQLContext` capabilities, and extends them. Those extensions include numerous additional operations available via the `HiveQL` parser, the ability to read and write data to and from Hive tables, and access to Hive's User Defined Function (UDF) capabilities. Furthermore, these capabilities are available whether or not you currently utilize Hive in your HDP cluster.

If your structured data needs are simple, the `SQLContext` has fewer dependencies and uses nominally fewer resources than `HiveContext` does, which can be an advantage if the limited `SQL92` API meets your needs.

Otherwise, `HiveContext` allows for far greater flexibility and extended capabilities. In general, or when in doubt, it is always safer to use `HiveContext`.

Catalyst Spark SQL Optimizer



Catalyst, The Spark SQL Optimizer

Spark SQL uses an optimizer called **Catalyst.** Catalyst accelerates query performance via an extensive built-in, extensible, catalog of optimizations that goes through a logical process and builds a series of optimized plans for executing a query. This is followed by an intelligent, cost-based modeling and plan selection engine, which then generates the code required to perform the operation.

This provides numerous advantages over core RDD programming. It is simpler to write an SQL statement to perform an operation on structured data than it is to write a series of `filter()`, `group()`, and other calls. Not only is it simpler, executing queries using Catalyst provides performance that matches or outperforms equivalent core RDD nearly 100% of the time.

Spark SQL make managing and processing structured data easier, and it provides performance improvements as well.

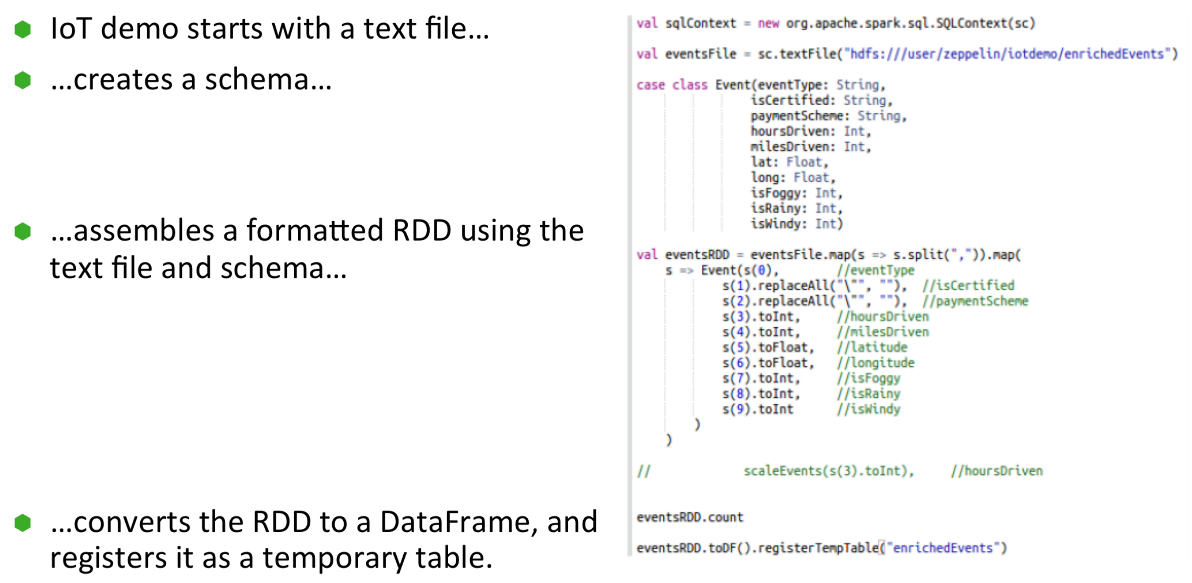
DataFrames, Tables and Contexts

Registering a DataFrame as a temporary table makes that table available for either DataFrames API or SQL interactions while operating in that specific context. Storing tables in Hive (and using `HiveContext`) makes them available across all contexts in a cluster.

DataFrames and Tables

To understand the relationship between DataFrames and tables, we will start with the IoT demo.

IoT Demo Create



- IoT demo starts with a text file…
- …creates a schema…
- …assembles a formatted RDD using the text file and schema…
- …converts the RDD to a DataFrame, and registers it as a temporary table.

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)

val eventsFile = sc.textFile("hdfs:///user/zeppelin/iotdemo/enrichedEvents")

case class Event(eventType: String,
                isCertified: String,
                paymentScheme: String,
                hoursDriven: Int,
                milesDriven: Int,
                lat: Float,
                long: Float,
                isFoggy: Int,
                isRainy: Int,
                isWindy: Int)

val eventsRDD = eventsFile.map(s => s.split(",")).map(
    s => Event(s(0),        //eventType
            s(1).replaceAll("\"", ""),  //isCertified
            s(2).replaceAll("\"", ""),  //paymentScheme
            s(3).toInt,     //hoursDriven
            s(4).toInt,     //milesDriven
            s(5).toFloat,   //latitude
            s(6).toFloat,   //longitude
            s(7).toInt,     //isFoggy
            s(8).toInt,     //isRainy
            s(9).toInt      //isWindy
        )
    )

//              scaleEvents(s(3).toInt),     //hoursDriven


eventsRDD.count

eventsRDD.toDF().registerTempTable("enrichedEvents")
```

IoT Demo Create

In this code, a CSV file is converted to an RDD named `eventsFile` using `sc.textFile`. Next, a schema named Event is created which labels each column and sets its type.

Then a new RDD named eventsRDD is generated which takes the content of the `eventFile` RDD and transforms the elements according to the Event schema, casting each column and reformatting data as necessary.

The code then counts the rows of `eventsRDD` - presumably as some kind of verification that the operation was successfully performed.

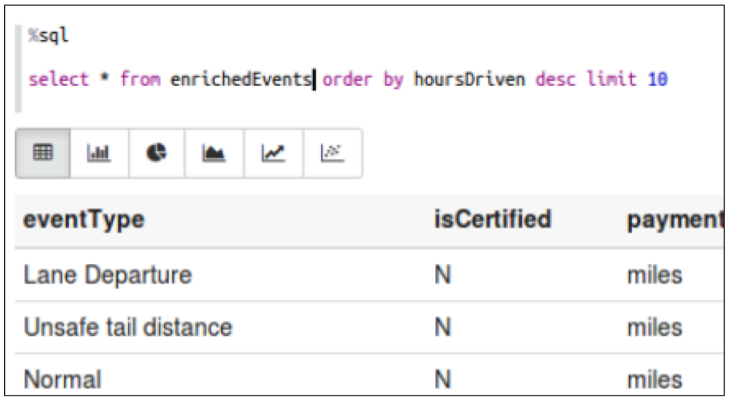
The final two steps are performed on a single line of code. First `eventsRDD` is converted to an unnamed DataFrame, which is then immediately registered as a temporary table named `enrichedEvents`.

IMPORTANT:

Some of the techniques shown here to format a text file for use in a DataFrame is beyond the scope of this class, but various references exist online on how to accomplish this.

IoT Demo Use

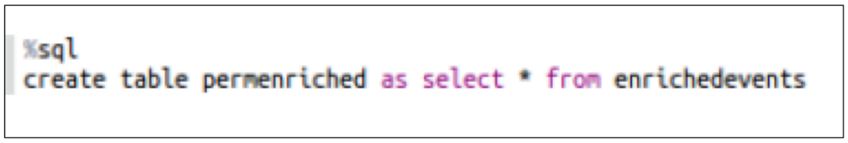
Once a DataFrame has been registered as a table, it is now available for direct manipulation by a Spark SQL context, using SQL commands.



```
%sql
select * from enrichedEvents order by hoursDriven desc limit 10
```

eventType	isCertified	payment
Lane Departure	N	miles
Unsafe tail distance	N	miles
Normal	N	miles

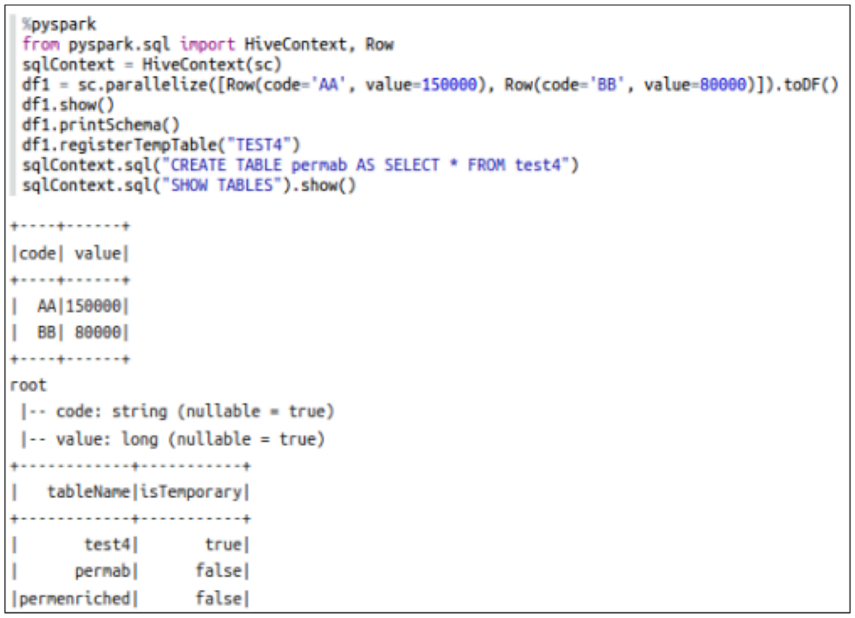
This temporary table can also be permanently converted to a permanent table in Hive. Making the table part of Hive's managed data has the added benefit of making the table available across multiple Spark SQL contexts.



```
%sql
create table permenriched as select * from enrichedevents
```

Python Row Objects

Another way to create a DataFrame is by creating a Row object. In the Python code shown, a Row object is created using `sc.parallelize()`.

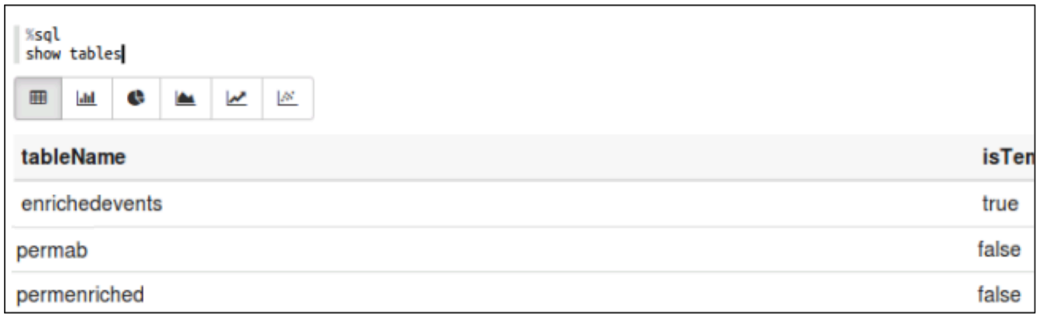


```
%pyspark
from pyspark.sql import HiveContext, Row
sqlContext = HiveContext(sc)
df1 = sc.parallelize([Row(code='AA', value=150000), Row(code='BB', value=80000)]).toDF()
df1.show()
df1.printSchema()
df1.registerTempTable("TEST4")
sqlContext.sql("CREATE TABLE permab AS SELECT * FROM test4")
sqlContext.sql("SHOW TABLES").show()

+----+------+
|code| value|
+----+------+
|  AA|150000|
|  BB| 80000|
+----+------+
root
 |-- code: string (nullable = true)
 |-- value: long (nullable = true)
+------------+-----------+
|   tableName|isTemporary|
+------------+-----------+
|       test4|       true|
|      permab|      false|
|permenriched|      false|
```

This Row object has an implied schema of two columns - named code and value - and there are two records for code AA with a value of 150000 and code BB with a value of 80000. Because we do not need to work with the RDD directly, we immediately convert this collection of Row objects to a DataFrame using `toDF()`. We then visually verify that the Row objects were converted to a DataFrame using `show()`, and that the schema was applied correctly using `printSchema()`.

The DataFrame is registered as a temporary table named `test4`, which is then converted to a permanent Hive table named `permab`. We then run `SHOW TABLES` to view the tables available for SQL to manipulate.

`SHOW TABLES` returned two Hive tables - `permab` and `permenriched` - and the temporary table `test4`. However, if we return to our other SQL context and run `SHOW TABLES`, we see the two Hive tables, but we **do not** see `test4`.



```
%sql
show tables
```

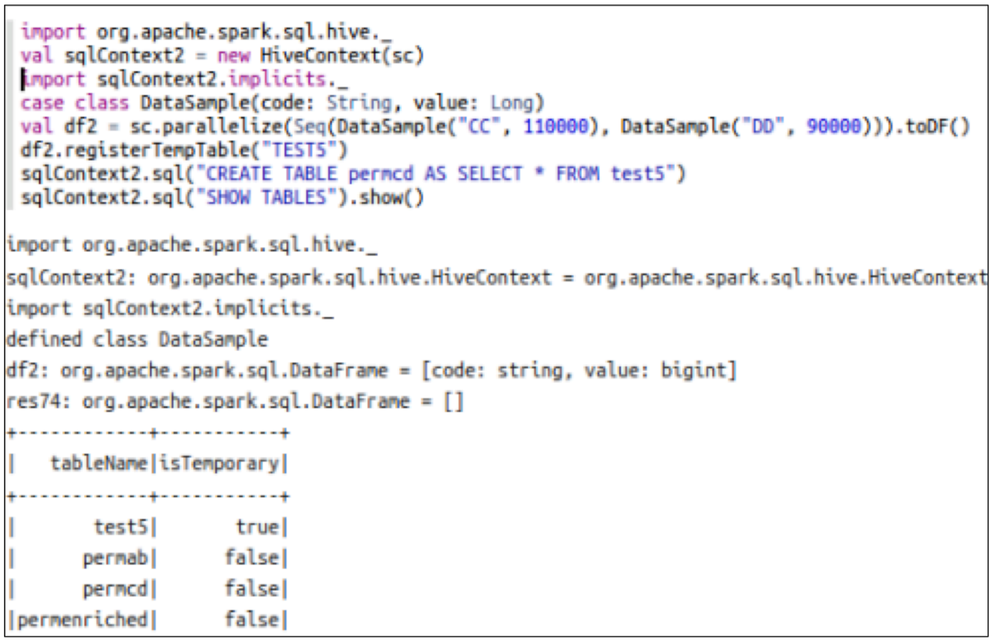
tableName	isTen
enrichedevents	true
permab	false
permenriched	false

In addition, this SQL context can still see the `enrichedevents` temporary table, which was not visible to our other SQL context.

KEY TAKE-AWAY:

if you want to make a table available to all SQL contexts across the cluster and not just the current SQL context, you must convert it to a permanent Hive table.

Scala Row Objects



```
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|
```

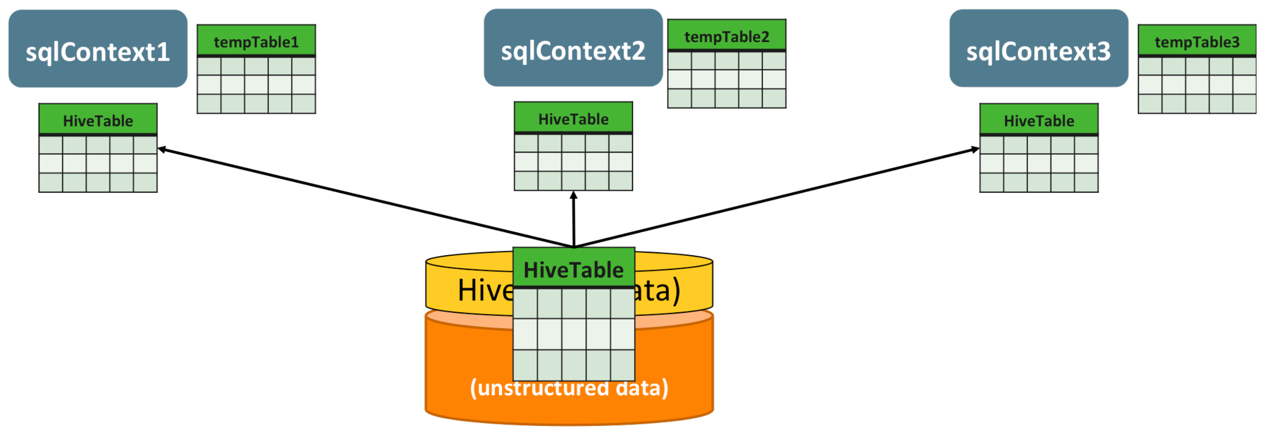
The code in this screenshot performs an operation similar to the IoT example, but on a smaller scale, and using Row objects in Scala rather than Python. Some of the key differences between this and the previous example include the definition of the `DataSample` schema class prior to creation of the `df2` RDD, with a different syntax during creation of the RDD.

As in the previous example, this RDD is immediately converted to a DataFrame, then registered as a temporary table which is converted to a Hive table.

NOTE:

The only temporary table visible when `SHOW TABLES` is executed is the temporary table created by this instance of the SQL context. All of the Hive tables are returned, as per previous examples.

DataFrames and Tables Summary



A key concept when writing multiple applications and utilizing multiple Spark SQL contexts across a cluster, is that registering a temporary table makes it available for either DataFrame API or SQL interactions while operating in that specific context.

However, those DataFrames and tables are **only available** within the Spark SQL context in which they were created.

To make a table visible across Spark SQL contexts, you should store that table permanently in Hive, which makes it available to any `HiveContext` instance across the cluster.

Create and Save DataFrames and Tables

There are various methods for creating and saving DataFrames and Tables, including:

- Converting RDDs to DataFrames
- Creating DataFrames Programmatically in Python and Scala
- Registering DataFrames as Temporary Tables
- Making DataFrames Available Across Contexts
- Creating DataFrames from Existing Hive Tables
- Storing DataFrames Permanently HDFS
- Using Data Stored in HDFS to Create DataFrames

Converting 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:

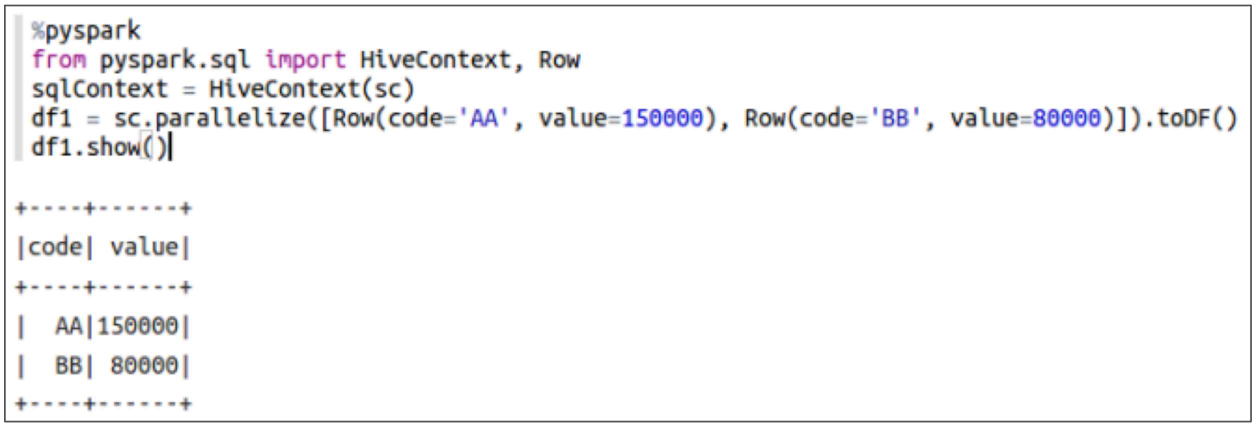
```
toDF():   dataframeX = rddName.toDF()

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

If an RDD is properly formatted but lacks a schema, in Python `createDataFrame()` can be used to infer the schema on DataFrame creation. (Scala lacks an easy, "on the fly" way to accomplish this.)

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

Creating DataFrames Programmatically in Python

Before an RDD can be converted to a DataFrame, a properly structured RDD must be created. In Python this can be accomplished by creating an RDD of Row elements in which the schema is defined on an element-by-element basis. This RDD can then be converted to a DataFrame. For example:



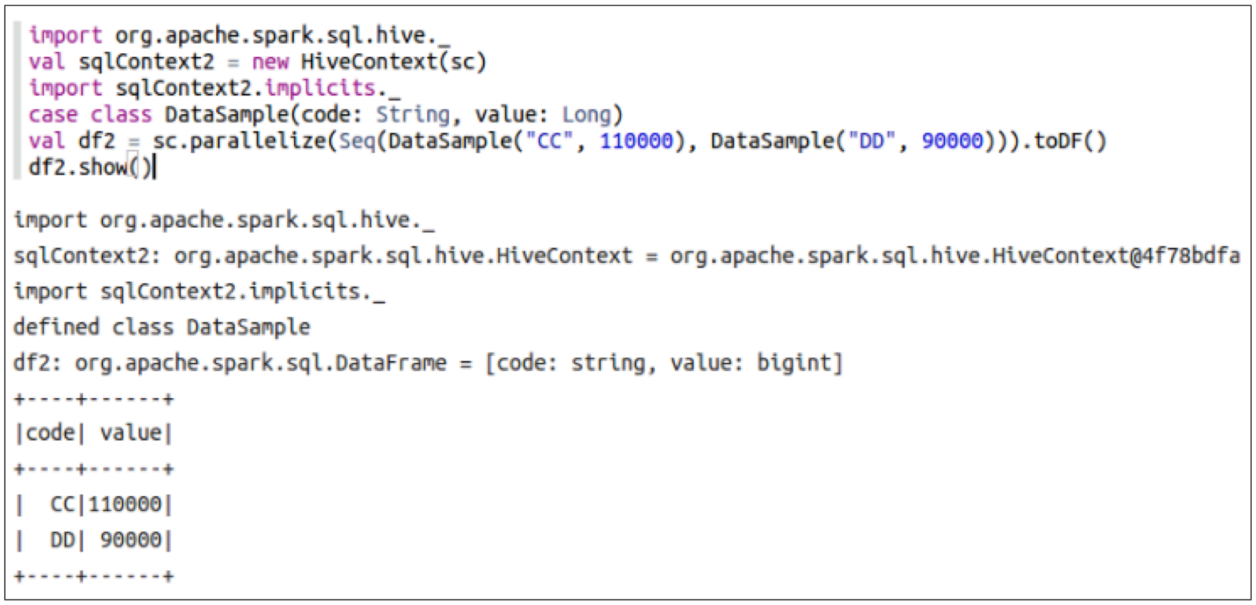
```
%pyspark
from pyspark.sql import HiveContext, Row
sqlContext = HiveContext(sc)
df1 = sc.parallelize([Row(code='AA', value=150000), Row(code='BB', value=80000)]).toDF()
df1.show()

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

Creating DataFrames Programmatically in Scala

To create an RDD that can be converted to a DataFrame using Scala, start by importing `sqlContext.implicits._` (assuming your Spark SQL context is named `sqlContext`), which assists in implicit type casting. (This is handled automatically in most cases when programming with Python.)

Then, create the schema you will be using with case class `[schemaName](columnName: datatype, columnName: datatype, ...`.).

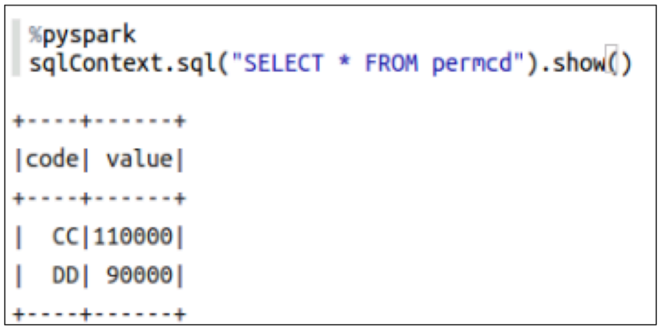
Then, create the DataFrame by creating an RDD that contains a sequential list of instances of this class and provide appropriate values, followed by `toDF()`. For example:



```
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.show()

import org.apache.spark.sql.hive._
sqlContext2: org.apache.spark.sql.hive.HiveContext = org.apache.spark.sql.hive.HiveContext@4f78bdfa
import sqlContext2.implicits._
defined class DataSample
df2: org.apache.spark.sql.DataFrame = [code: string, value: bigint]
+----+------+
|code| value|
+----+------+
|  CC|110000|
|  DD| 90000|
+----+------+
```

Using sqlContext.sql() and show()

Once a table is created, the DataFrames API can still be used to view and manipulate it using `sqlContext.sql("SQL command goes here")`. This assumes your Spark SQL context is named `sqlContext`. If you wish to see the output from the SQL command, append `show()` to the line of code. For example:



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

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

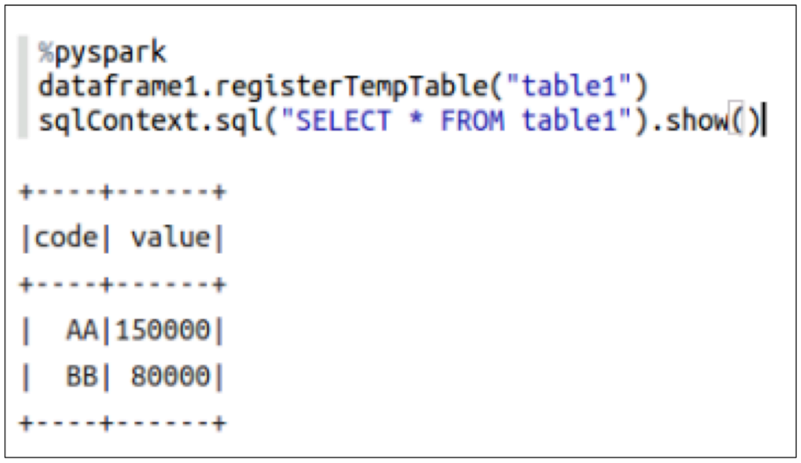
Using the show function to display the results of an sql() query

Registering DataFrames as Temporary Tables

By default, DataFrames are not available for direct SQL manipulation. Instead, a developer would need to use the DataFrames API to perform operations. To register a DataFrame to a table that can be directly manipulated by SQL, use `registerTempTable()`. For example:

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

This takes the DataFrame named `dataframe1` and registers it as a temporary table named `table1`. As the screenshot demonstrates, this table is now available for SQL manipulations:



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

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

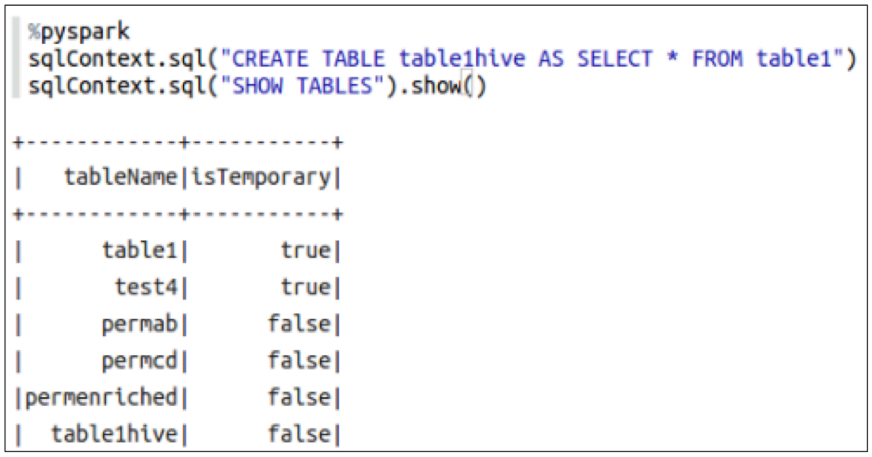
Manipulating a Temporary Tables with SQL

Making Tables Available Across Contexts with `CREATE TABLE`

Temporary tables are only available to the Spark SQL context that created them. One way to make a table visible across all Spark SQL contexts in the cluster is to create a Hive table and copy the contents of the temporary table to it. The code to accomplish this would be:

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

This creates a table named `table1hive` in Hive, copying all of the contents from temporary table `table1`. The following screenshot demonstrates that `table1hive` is now registered as a permanent table.



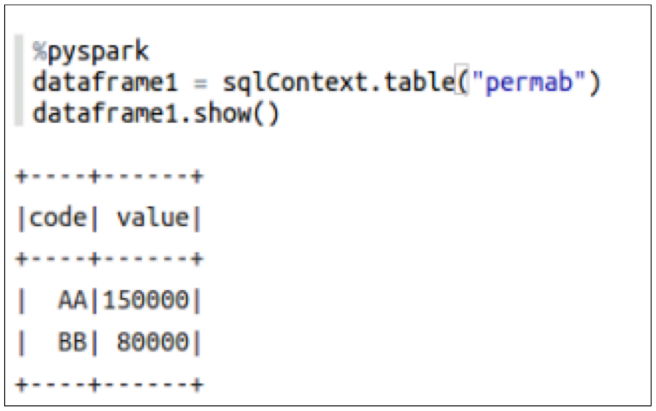
```
%pyspark
sqlContext.sql("CREATE TABLE table1hive AS SELECT * FROM table1")
sqlContext.sql("SHOW TABLES").show()

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

Using SHOW TABLES to Show that table1hive is Registered as a Permanent Table

Creating DataFrames from Existing Hive Tables

Just as DataFrames can be converted to tables, tables can also be converted to DataFrames, which can then be directly manipulated using the DataFrames API. This is accomplished by using `sqlContext.table()`, as in the example below:

```
dataframe1 = sqlContext.table("permab")
```



```
%pyspark
dataframe1 = sqlContext.table("permab")
dataframe1.show()

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

Creating a DataFrame from an Existing Hive Table

Saving DataFrames from HDFS

You can save DataFrames from data stored in HDFS using:

- `read()`
- `write()`

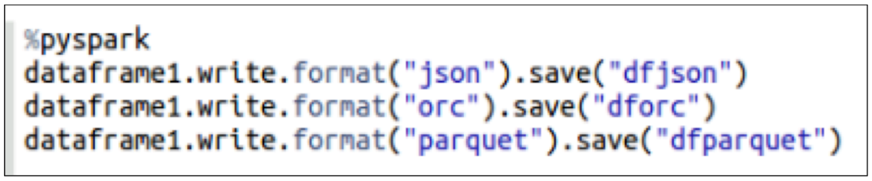
Saving DataFrames as Files using write()

DataFrames can be permanently stored in HDFS in several non-table formats as well using `write()`. Some of the common ones used include ORC, JSON, and parquet. The command to save the DataFrame contents to these file types follows the following syntax:

```
DataFrameName.write.format("selectedFormat").save("HDFSfolder")
```

Specific examples for JSON, ORC, and parquet files:

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



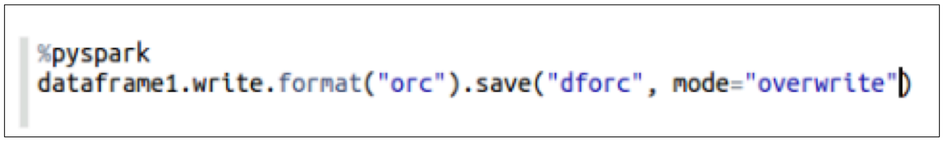
```
%pyspark
dataframe1.write.format("json").save("dfjson")
dataframe1.write.format("orc").save("dforc")
dataframe1.write.format("parquet").save("dfparquet")
```

Save Modes

By default, if a `write()` is used and the file already exists, an error will be returned. This is because of the default behavior of **save modes**. However, this default can be modified. Here are the possible values for save mode when writing a file and their definitions:

- `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

For example, if you were using the `write()` command from before to save an ORC file and you wanted the data to be overwritten / replaced if a file by the same name already existed, you would use the following code:



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

Creating DataFrames from Files using read()

DataFrames can be created from files of many structured file types, including ORC, parquet, and JSON files if they are properly formatted using `read()`.

REFERENCE:

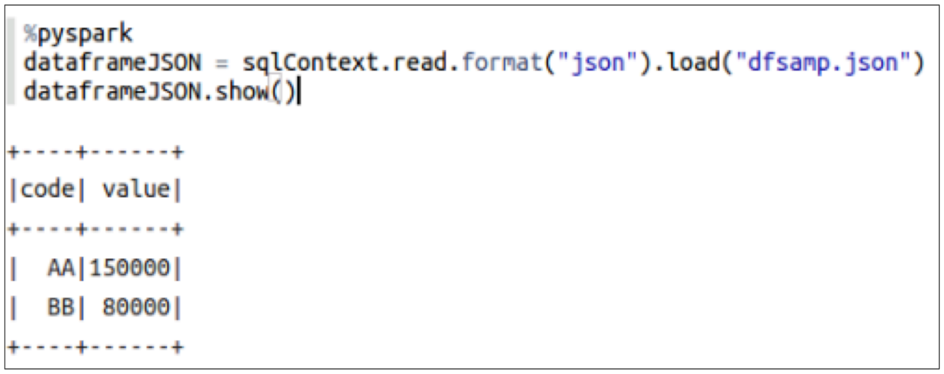
For a complete list of supported file types for direct import into DataFrames, please refer to Spark SQL documentation. (http://spark.apache.org/documentation.html)

The syntax is similar to the `write()` command used before, only `read()` is used, and an appropriate file is loaded. For example, to use the JSON file created earlier to create a DataFrame:

```
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|
+----+------+
```

Creating a DataFrame from Files using `read()`

NOTE:

If you peruse the documentation, you will note that some file formats have `read()` shortcuts - for example: `read.json` instead of `read.format("json")`. We do not demonstrate them in class because they are not consistent across all supported file types, however if a developer works primarily with JSON files on a regular basis, using the `read.json` shortcut may be beneficial.

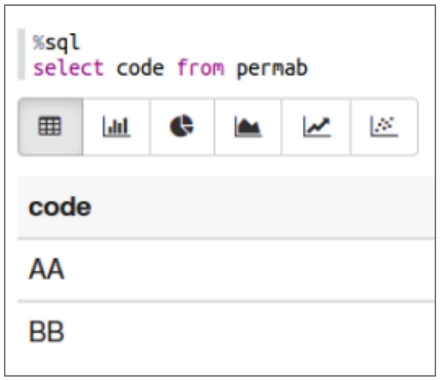
Manipulating DataFrames and Tables

You can manipulate SQL tables by using standard SQL commands via Spark SQL or from within the DataFrames API using `sqlContect.sql()`. In this class, we do not explicitly teach a significant number of SQL commands, as it is assumed most Spark developers have prior exposure to this topic.

DataFrames are manipulated using the DataFrames API.

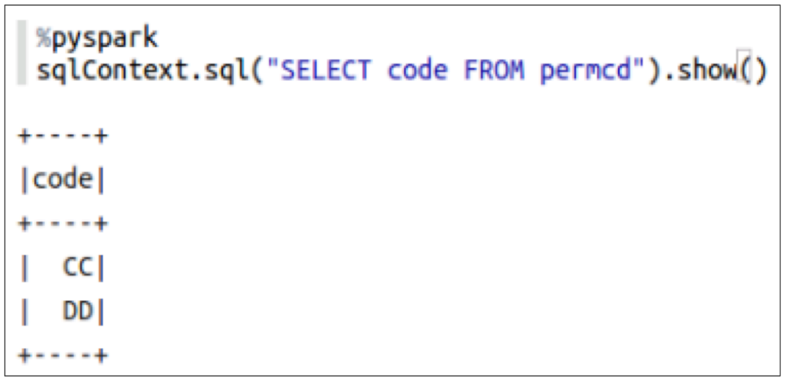
Manipulating SQL Tables

Tables can be manipulated using SQL and HiveQL commands via Spark SQL or within the DataFrames API using `sqlContext.sql()`. In both cases, queries are optimized using Catalyst, meaning highly optimized operations occur by default.



```
%sql
select code from permab
```

code
AA
BB



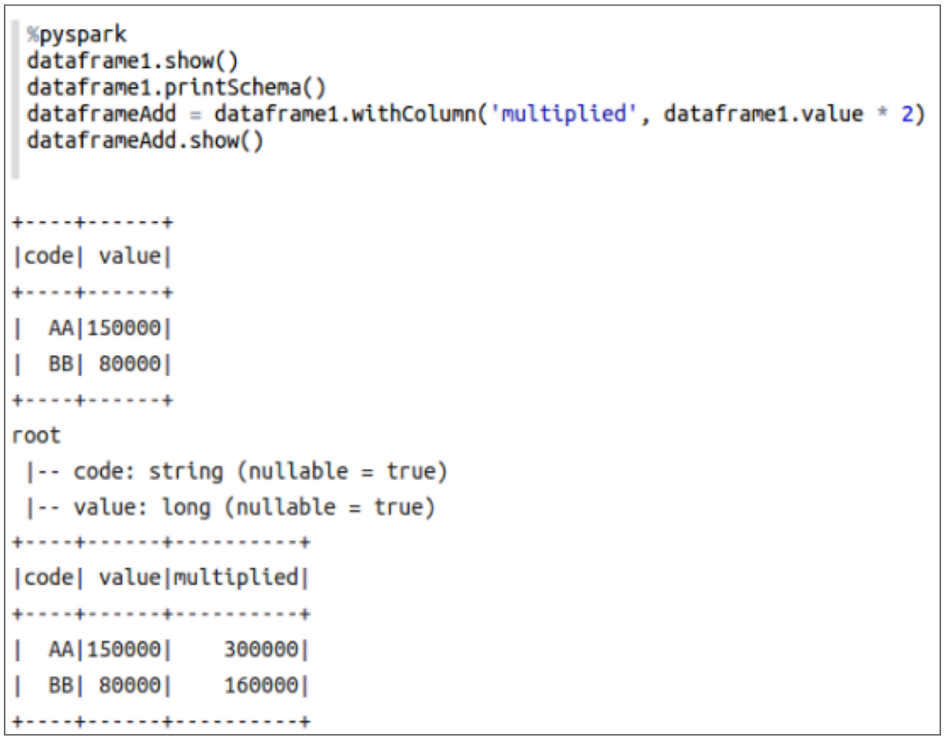
```
%pyspark
sqlContext.sql("SELECT code FROM permcd").show()

+----+
|code|
+----+
|  CC|
|  DD|
+----+
```

Methods for Manipulating SQL Tables

Manipulating DataFrames using the DataFrames API

In addition to `sqlContext.sql()`, the DataFrames API contains a number of functions that are designed to manipulate DataFrames (rather than tables) in a similar fashion to how SQL queries manipulate tables.

show(), printSchema(), and withColumn()



```
%pyspark
dataframe1.show()
dataframe1.printSchema()
dataframeAdd = dataframe1.withColumn('multiplied', dataframe1.value * 2)
dataframeAdd.show()


+----+------+
|code| value|
+----+------+
|  AA|150000|
|  BB| 80000|
+----+------+
root
 |-- code: string (nullable = true)
 |-- value: long (nullable = true)
+----+------+----------+
|code| value|multiplied|
+----+------+----------+
|  AA|150000|    300000|
|  BB| 80000|    160000|
+----+------+----------+
```

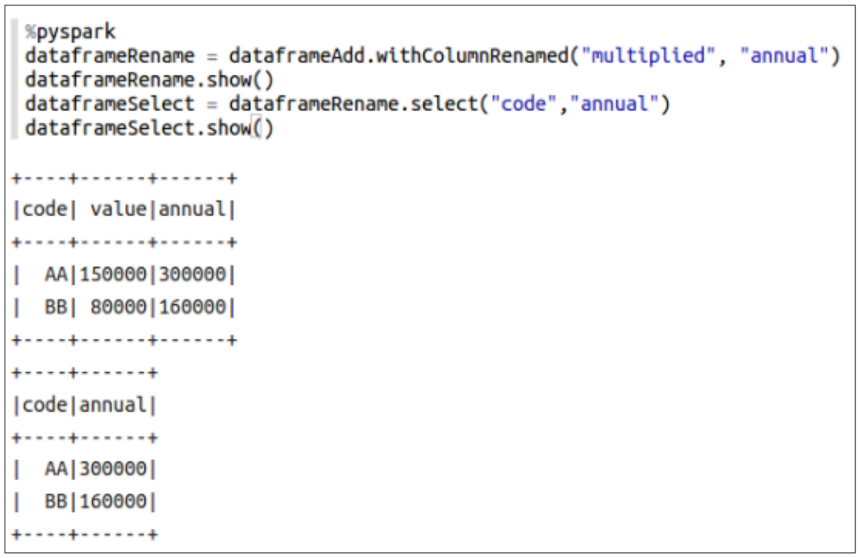
`show()`, `printSchema()`, and `withColumn()` Functions

The `show()` function displays the contents of a DataFrame, the output of an SQL command run within `sqlContext.sql()`, and is also required for on-screen display of several other functions that will be discussed.

The `printSchema()` function displays the schema for a DataFrame.

The `withColumn()` function returns a new DataFrame with a new column based on criteria you provide. In the example, a new column named multiplied was created, and the numbers in that column are the numbers in the value column multiplied by two.

withColumnRenamed() and select()



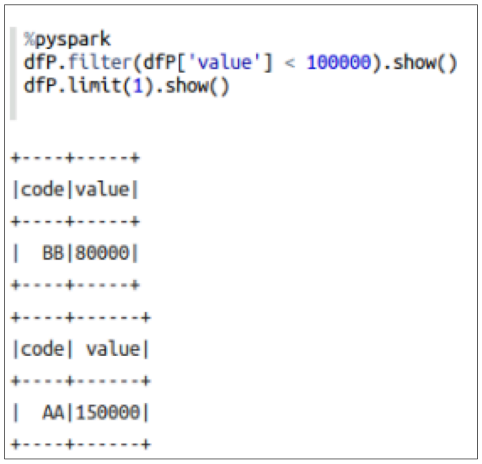
```
%pyspark
dataframeRename = dataframeAdd.withColumnRenamed("multiplied", "annual")
dataframeRename.show()
dataframeSelect = dataframeRename.select("code","annual")
dataframeSelect.show()

+----+------+------+
|code| value|annual|
+----+------+------+
|  AA|150000|300000|
|  BB| 80000|160000|
+----+------+------+
+----+------+
|code|annual|
+----+------+
|  AA|300000|
|  BB|160000|
+----+------+
```

`withColumnRenamed` and select() Functions

The `withColumnRenamed()` function returns a new DataFrame with a column renamed - in the example, from `multiplied` to `annual`.

The `select()` function returns a new DataFrame with only the explicitly named columns and their data.

filter() and limit()



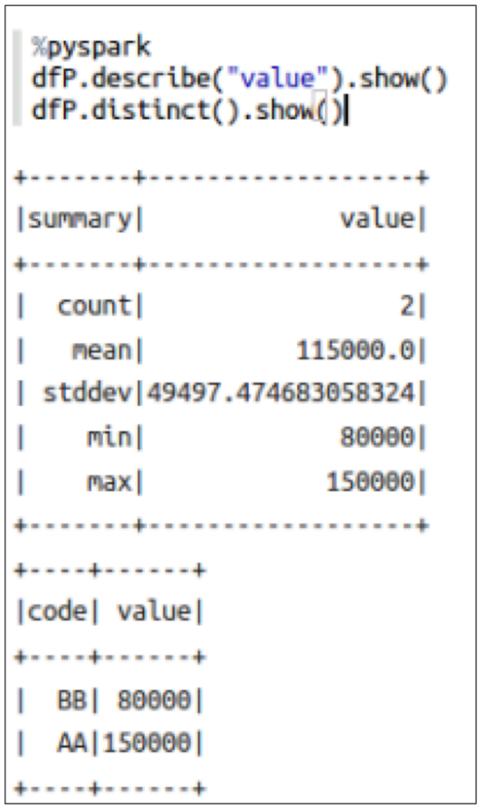
```
%pyspark
dfP.filter(dfP['value'] < 100000).show()
dfP.limit(1).show()

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

filter() and limit() Functions

The `filter()` function returns a DataFrame with only rows that have column values that meet a defined criteria - in the screenshot, only rows that had values less than 100,000 were returned.

The `limit()` function returns a DataFrame with the first *n* rows of the DataFrame. In the example, only the first row was returned.

describe() and distinct()



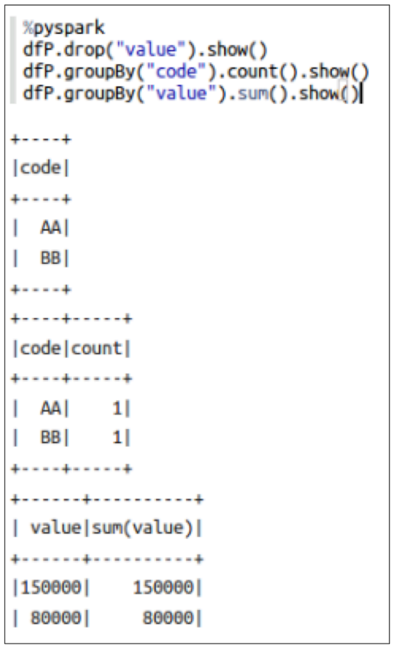
```
%pyspark
dfP.describe("value").show()
dfP.distinct().show()

+-------+-----------------+
|summary|            value|
+-------+-----------------+
|  count|                2|
|   mean|         115000.0|
| stddev|49497.474683058324|
|    min|            80000|
|    max|           150000|
+-------+-----------------+

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

`describe()` and `distinct()` Functions

The `describe()` function returns count, mean, standard deviation, minimum, and maximum values for named columns in a DataFrame. If no columns are named, all numerical columns will be used and reported on.

The `distinct()` function returns a new DataFrame of only the unique rows from the original DataFrame.

drop() and groupBy()



```
%pyspark
dfP.drop("value").show()
dfP.groupBy("code").count().show()
dfP.groupBy("value").sum().show()

+----+
|code|
+----+
|  AA|
|  BB|
+----+
+----+-----+
|code|count|
+----+-----+
|  AA|    1|
|  BB|    1|
+----+-----+
+------+----------+
| value|sum(value)|
+------+----------+
|150000|    150000|
| 80000|     80000|
```

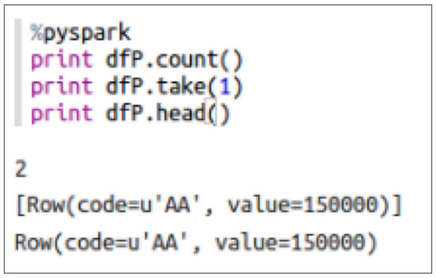
drop() and groupBy() Functions

The `drop()` function returns a DataFrame without specific volumes included. Think of it as the opposite of the `select()` function.

The `groupBy()` function groups rows by matching column values, and can then perform other functions on the combined rows such as `count()`.

The screenshot shows two examples. In the first one, the code column is grouped and the number of matching values are counted and displayed in a separate column. In the second one, the values column is scanned for matching values, and then the sums of the identical values are displayed in a separate column.

count(), take(), and head()



```
%pyspark
print dfP.count()
print dfP.take(1)
print dfP.head()

2
[Row(code=u'AA', value=150000)]
Row(code=u'AA', value=150000)
```

`count()`, `take()`, and `head()` Functions

The `count()` function returns the number of rows in the DataFrame as a result.

The `take()` function returns a number of rows in the DataFrame and returns them as Row objects.

The `head()` function returns the first row in the DataFrame as a Row object.

IMPORTANT:

In the screenshot provided, these functions are shown prepended with `print`. However, the print command is not required for Scala, nor is it required for Python when using the REPL.

Technically, these functions should probably not have required the `print` function in order to produce output either, but via trial and error testing we discovered that they worked when `print` was supplied in Zeppelin. In addition, at the time of this writing, a handful of pyspark functions did not operate correctly *at all* when run inside Zeppelin, even in conjunction with the `print` command.

Some examples include `first()`, `collect()`, and `columns()`. This is likely the result of a bug in the version of Zeppelin used to write this course material and may no longer be the case by the time you are reading this.

For additional DataFrames API functions, please refer to the online Apache Spark SQL DataFrames API documentation. Testing these pyspark functions without the print command will likely result in success in a future implementations.

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 an `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.

Answers

1) While core RDD programming is used with [structured/unstructured/both] data Spark SQL is used with [structured/unstructured/both] data.

Answer: Both / Structured

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.

Answer: False. The Catalyst optimizer means Spark SQL programs will generally outperform core RDD programs

3) True or False: A `HiveContext` can do everything that an `SQLContext` can do, but provides more functionality and flexibility.

Answer: True

4) True or False: Once a DataFrame is registered as a temporary table, it is available to any running sqlContext in the cluster.

Answer: False. Temporary tables are only visible to the context that created them.

5) Hive tables are stored [in memory/on disk].

Answer: On Disk

6) Name two functions that can convert an RDD to a DataFrame.

Answer: `toDF()` and `createDataFrame()`

7) Name two file formats that Spark SQL can use without modification to create DataFrames.

Answer: The ones discussed in class were ORC, JSON, and parquet files.

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

Data Visualization in Zeppelin

Lesson Objectives

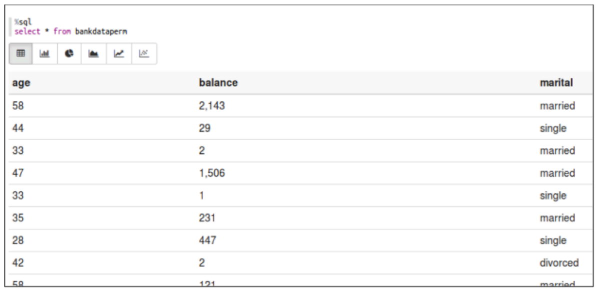
After completing this lesson, students should be able to:

- ✓ 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

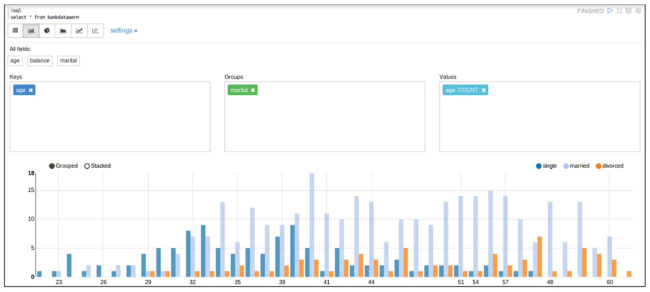
The usefulness of data in tabular format is limited from a human decision-making perspective. We simply lack the ability to look at hundreds of rows of varied data and make inferences or draw accurate conclusions. Thus tabular data on its own is limited in its ability to assist with real-world decision-making.

Data visualizations allow human beings to look at large amounts of consolidated data in a graphical form, allowing us to quickly make inferences and draw conclusions based on visual input alone. While several categories of data visualizations have become standard content for things like reports and dashboards, the possible types of data visualizations available are limited only by human imagination. Numerous programs and code libraries are available to produce them, and in addition, custom visualizations can easily be made using tools such as HTML and JavaScript.



```
%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
58	121	married



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

All fields: age balance marital

Keys: age

Groups: marital

Values: age COUNT

Data Visualizations

Data Visualization and Spark

The Spark project contains a module named GraphX. It is a Scala-only set of tools that enable a developer to programmatically produce relatively complex data visualizations in Spark.

In addition, however, Zeppelin comes with a feature-rich, easy to use, and highly extensible set of data visualization capabilities. Not only can Zeppelin produce data visualizations.

It enables a developer to:

- Collaborate with others via controlled notebook sharing
- Produce reports for external consumption by only sharing paragraph results
- Add manipulation tools to those published paragraphs so users without any code knowledge can be granted the ability to manipulate those results in real time for various purposes

Because of Zeppelin's direct integration with Spark, flexibility in terms of supported languages, and collaboration and reporting capabilities - the rest of this lesson will explore how to use this tool for greatest effect.

Keep in mind, however, that Zeppelin also supports HTML and JavaScript, and can also work with other data visualization libraries available to Python, Java, and other languages. If Zeppelin's built-in capabilities don't quite meet your needs, you always have the ability to expand on them.

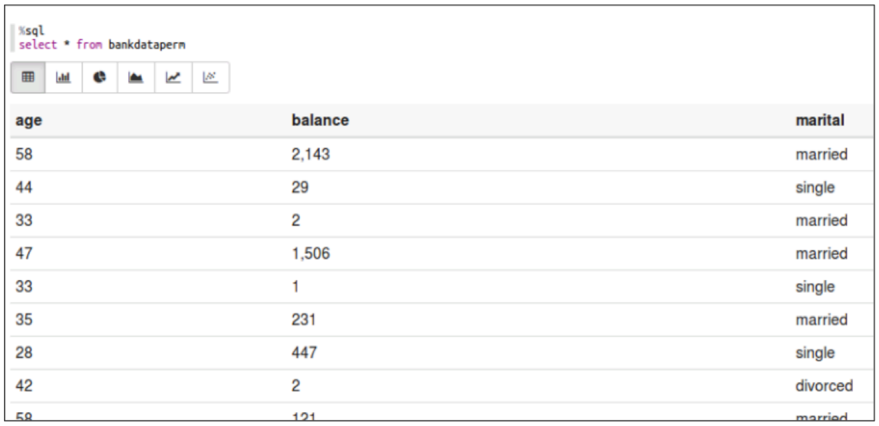
Data Exploration in Zeppelin

Zeppelin provides several different ways to visualize tables, and will provide these options by default whenever working with `%sql` binding in a note. The six views provided are:

- Table, which shows rows and columns much like a spreadsheet
- Bar chart
- Pie chart
- Area chart
- Line chart
- Scatter plot chart

Visualizations on Tables - %sql default

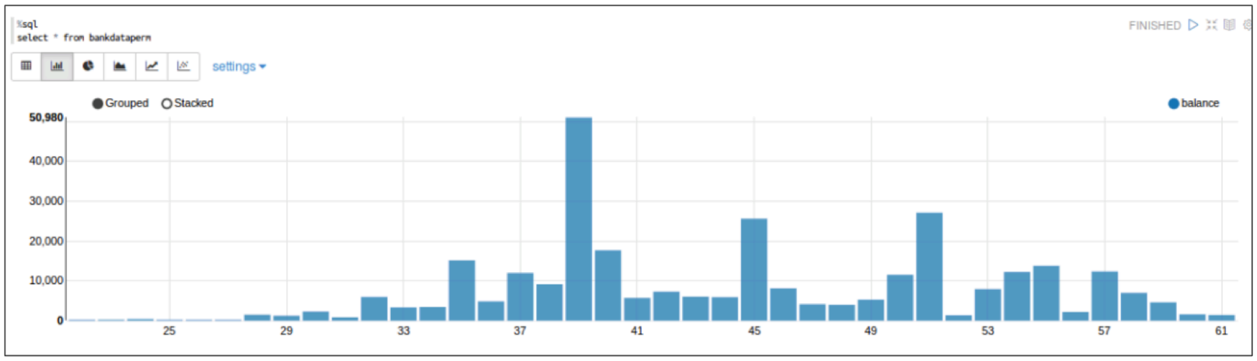
Table



```
%sql
select * from bankdatapern
```

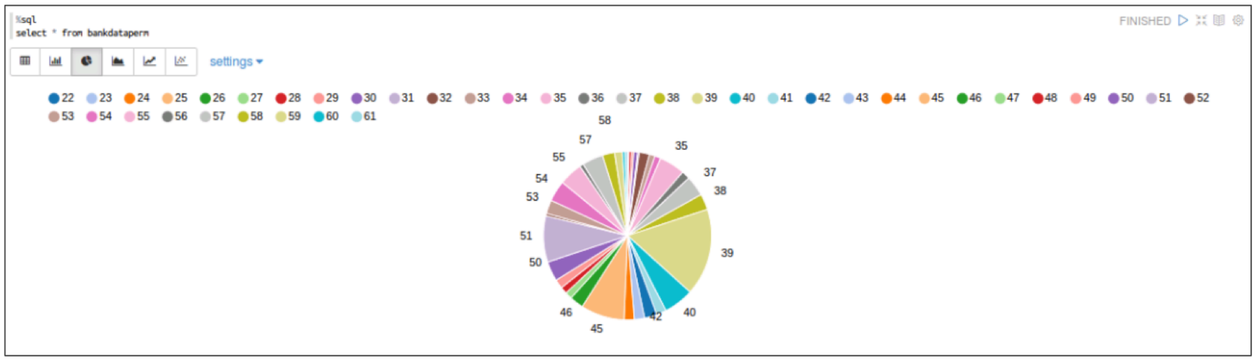
age	balance	marital
58	2,143	married
44	29	single
33	2	married
47	1,506	married
33	1	single
35	231	married
28	447	single
42	2	divorced
58	121	married

Bar Chart



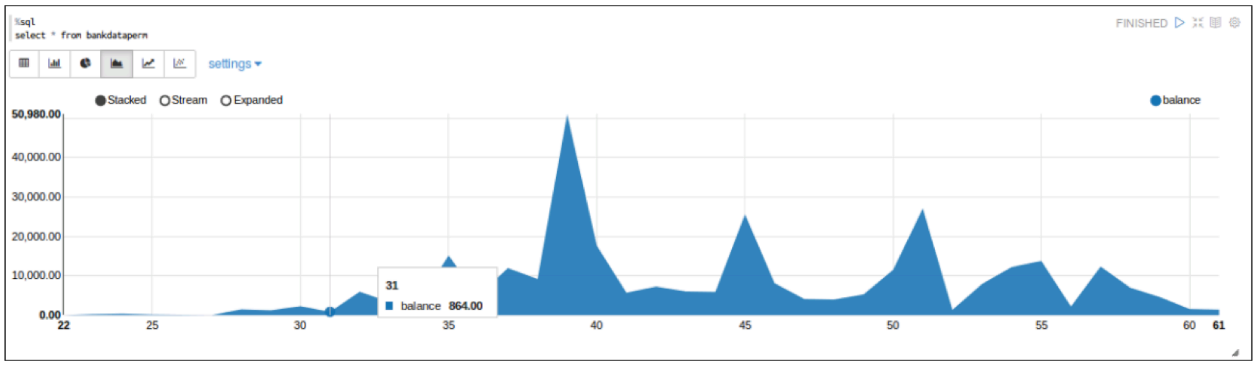
```
%sql
select * from bankdatapern
```

Pie Chart



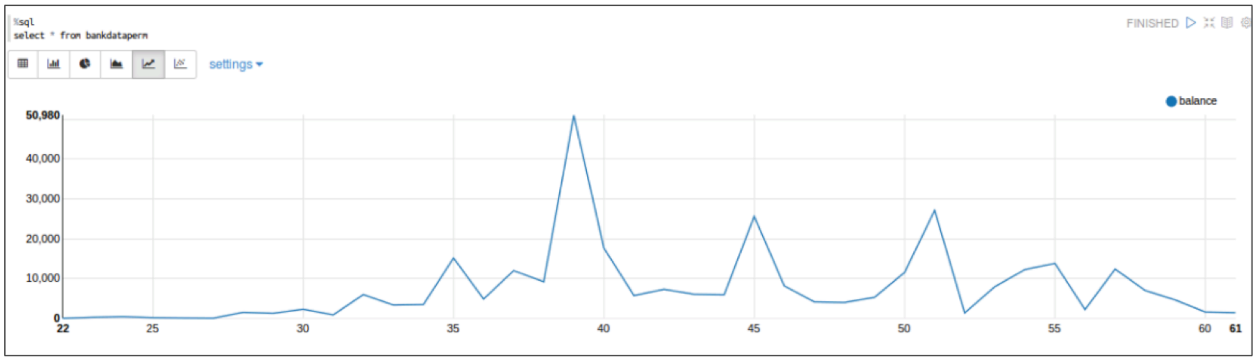
```
%sql
select * from bankdataperm
```

Area Chart



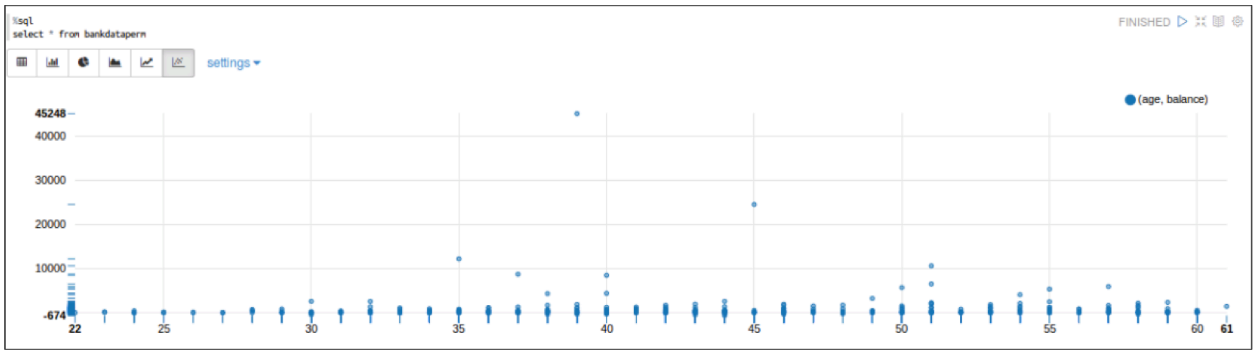
```
%sql
select * from bankdataperm
```

Line Chart



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

Scatter Plot Chart



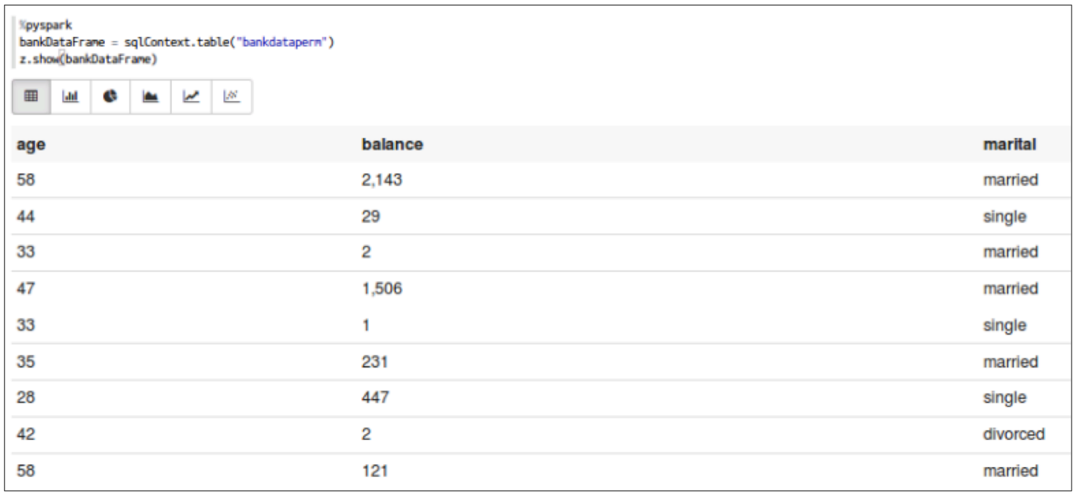
```
%sql
select * from bankdataperm
```

Visualizations on DataFrames

Zeppelin can also provide visualizations on DataFrames that have not been converted to SQL tables by using the following command:

```
z.show(DataFrameName)
```

This command tells Zeppelin to treat the DataFrame like a table for visualization purposes. Since a DataFrame is already formatted like a table, the command should work without issue on every DataFrame.



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

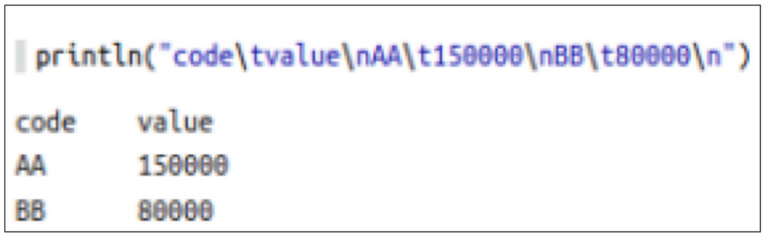
age	balance	marital
58	2,143	married
44	29	single
33	2	married
47	1,506	married
33	1	single
35	231	married
28	447	single
42	2	divorced
58	121	married

DataFrame Visualization in Zeppelin

Visualizations on Other Formatted Data

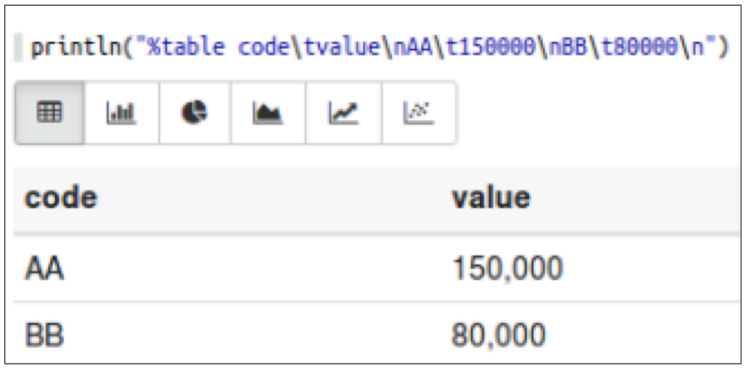
Zeppelin also comes with a `%table` indicator, which can be used to tell Zeppelin that any properly formatted data should be treated as a table. In the following example, we start by using the Scala `println()` command and create a simple table programmatically.

This screenshot shows what Zeppelin shows when %table is not part of the command.



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

code    value
AA      150000
BB      80000
```

Zeppelin then displays the content as a table, with supporting data visualizations available as below.



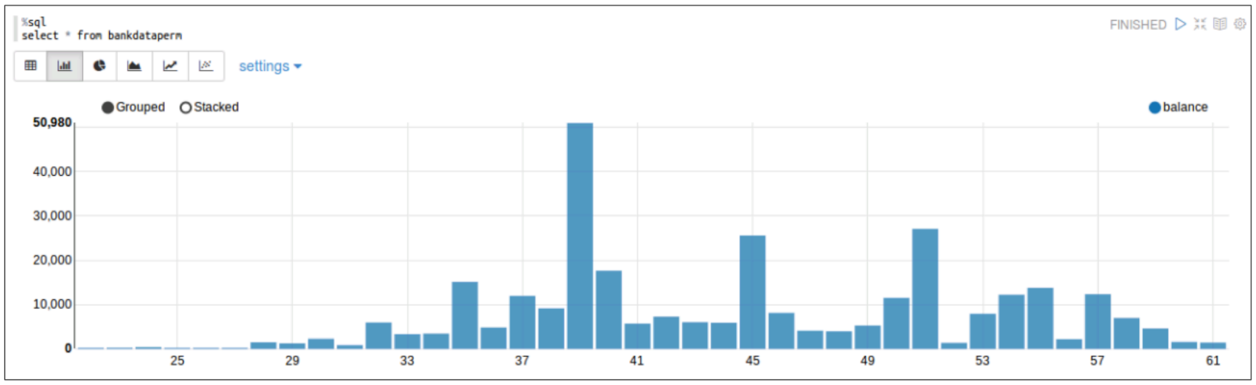
```
println("%table code\tvalue\nAA\t150000\nBB\t80000\n")
```

code	value
AA	150,000
BB	80,000

If the data is not formatted correctly, Zeppelin would simply return the string as a table name with no data.

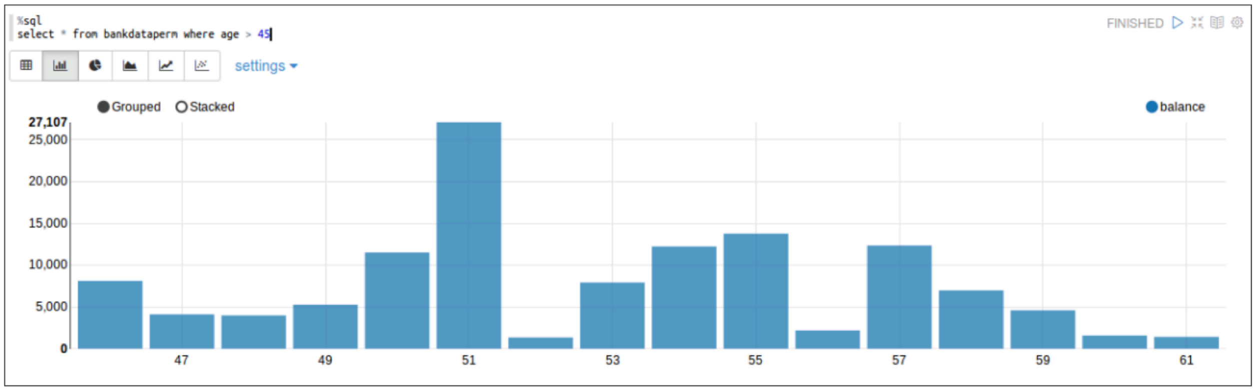
Interactive Programmatic Visualization

Zeppelin will update a visualization any time a new query is run in a paragraph.



```
%sql
select * from bankdatapern
```

For example, in the previous screenshot, the SQL command displays a visualization for all columns and rows.

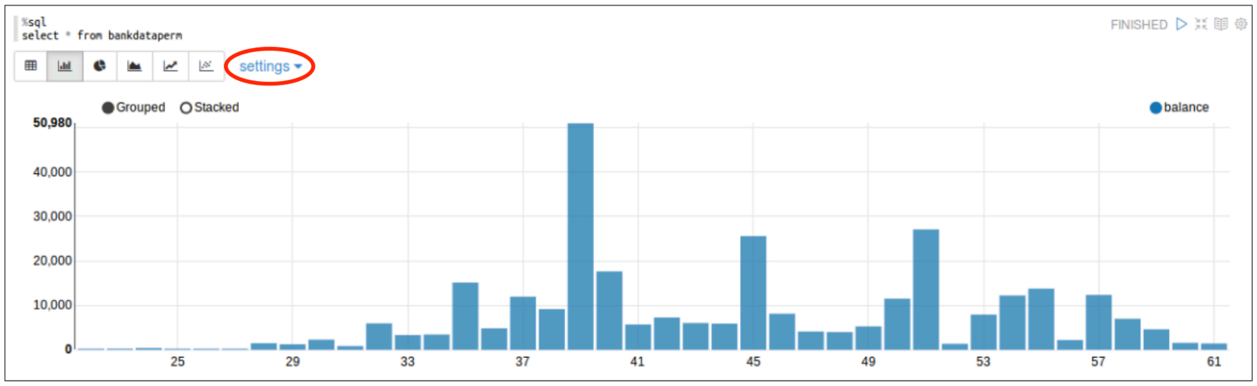


```
%sql
select * from bankdatapern where age > 45
```

However, in the second screenshot, the query was updated to only include rows with a value in the age column that exceeded 45.

Interactive Pivot Chart Visualization

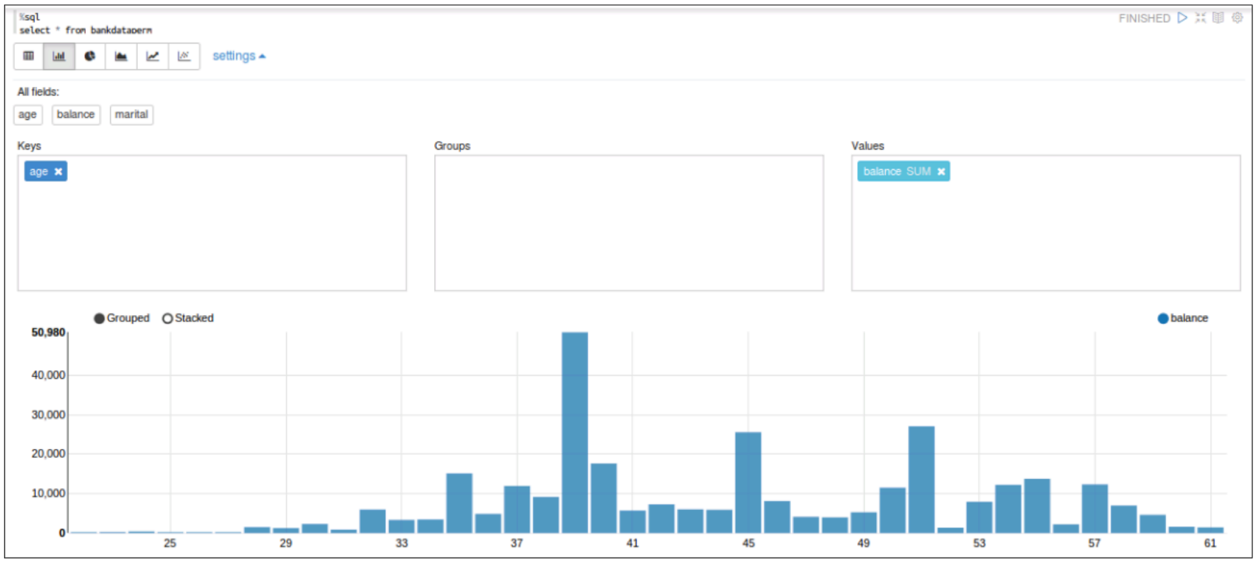
You can write and rewrite, and then rerun, SQL statements to get exactly the data you want. However, Zeppelin also provides a pivot chart capability that becomes available when you click settings to the right of the chart options.



```
%sql
select * from bankdatapern
```

This provides you with the ability to manipulate the chart output in a number of ways without requiring you to modify the initial query.

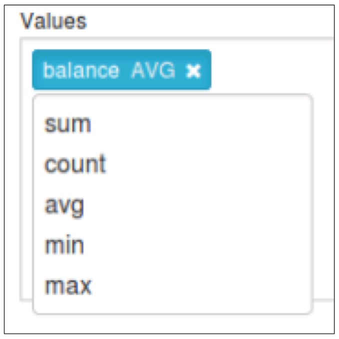
In the example above, we see that the default chart uses age as a key, and sums the balances for all persons of a given age as the value.



```
%sql
select * from bankdataoern
```

This was done automatically, without any grouping or sum command as part of the SQL statement itself.

Pivot Chart Value Options

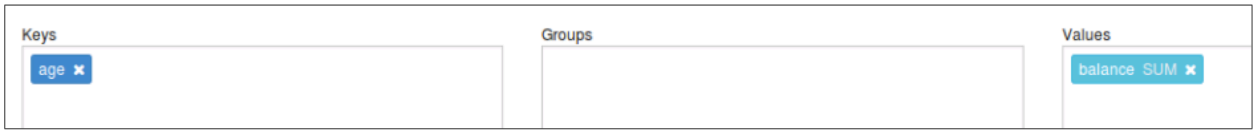


Pivot Chart – Value Options

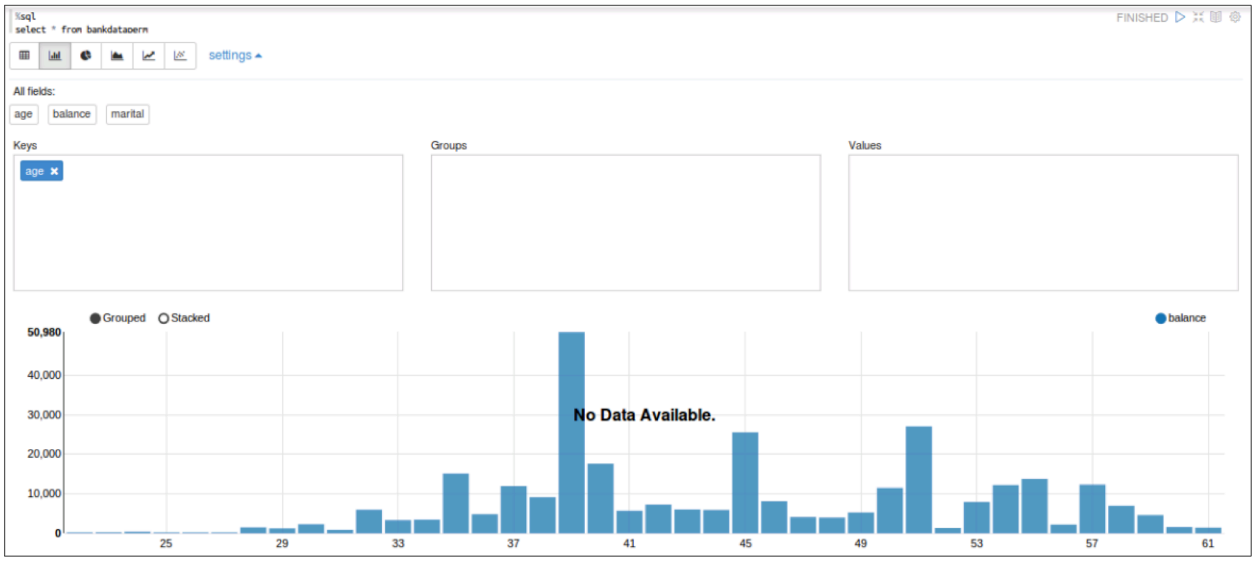
The pivot chart feature allows you to change the action performed on the Values column selected. Click on the box (in the screenshot, the one that says balance) and a drop-down menu of options appear which can be used to change the default value action. Options include SUM, AVG, COUNT, MIN, and MAX.

Pivot Chart Change Values or Keys

Any column in the table can be set as a Key or Value.

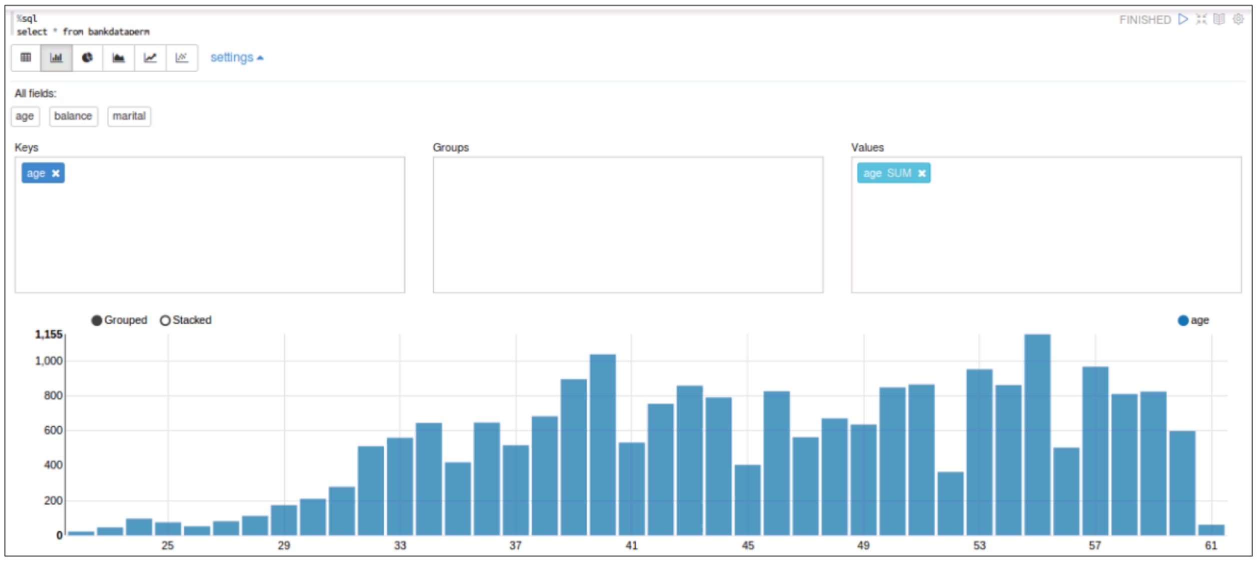


To remove a column, click the "x" to the top right of the name box and it will disappear.



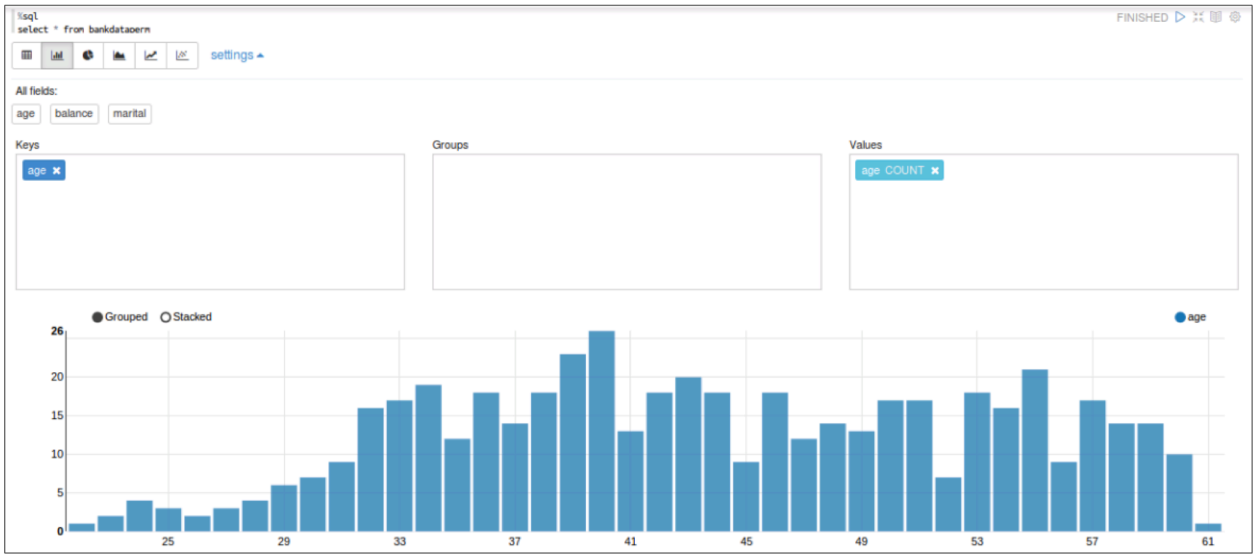
```
%sql
select * from bankdataperm
```

If either the Key or Value field is blank, the output indicates that there is no data available.



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

Then you simply drag and drop the field you want as a value into the appropriate box and the output refreshes to match.

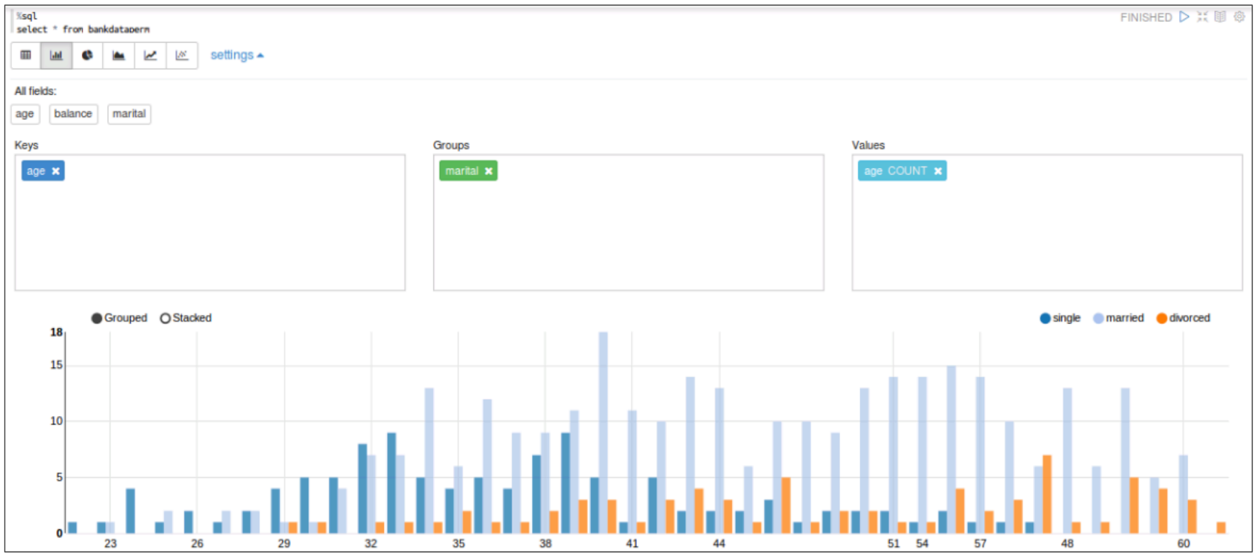


```
%sql
select * from bankdataperm
```

In this example, we elected to use the age column for both Keys and Values, and used the `COUNT` feature to count the number of individuals in each age category.

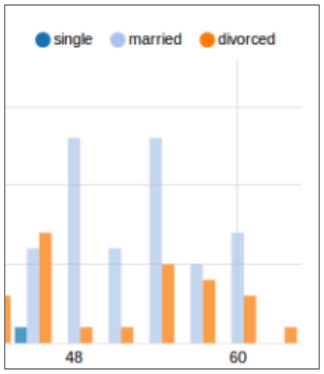
Pivot Chart Add Groups

In addition to Keys and Values, pivot charts also allow you to specify Group columns which can provide a further breakdown of the data.



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

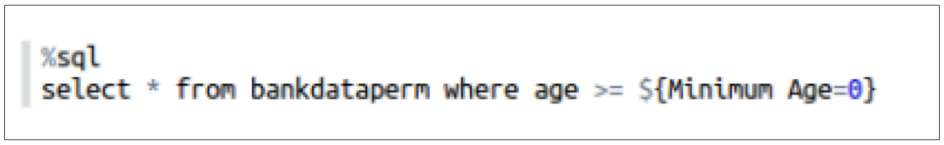
In this example, the marital column was defined as a grouping, and therefore every unique value in that column (married, single, or divorced) became its own bar color in the bar chart.



Dynamic Forms

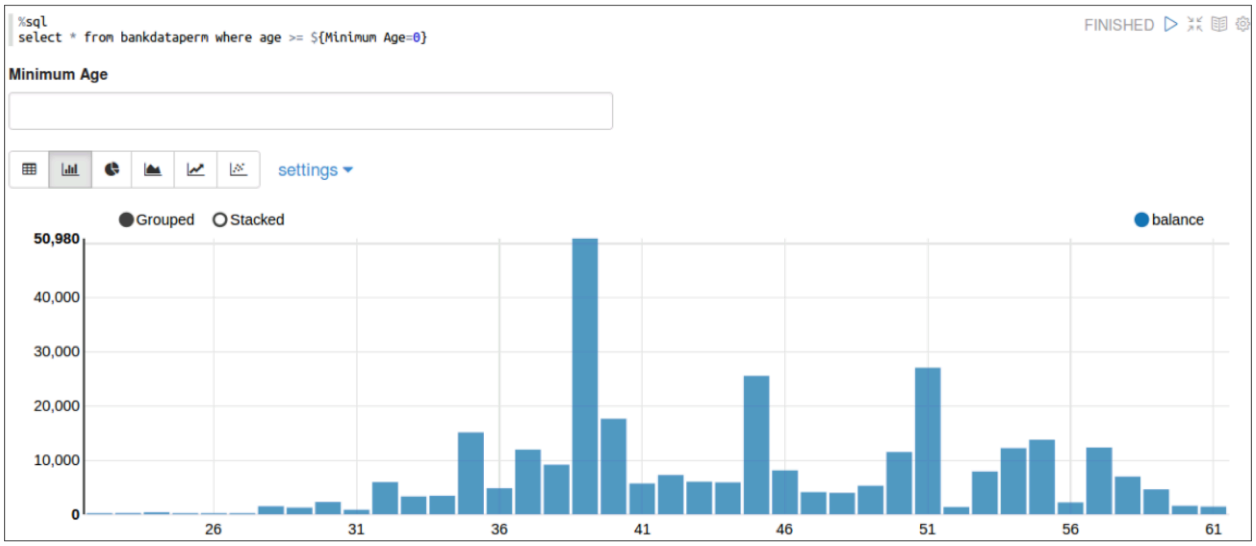
Dynamic Forms give you the ability to define a variable in the query or command and allow that value to be dynamically set via a form that appears above the output chart. These can be done in various programming languages. For SQL, you would use a `WHERE` clause and then specify the column name, some mathematical operator, and then a variable indicated by a dollar sign with the form name and the default value specified inside a pair of curly braces.

```
SELECT * FROM table WHERE colName [mathOp] ${LabelName=DefValue}
```



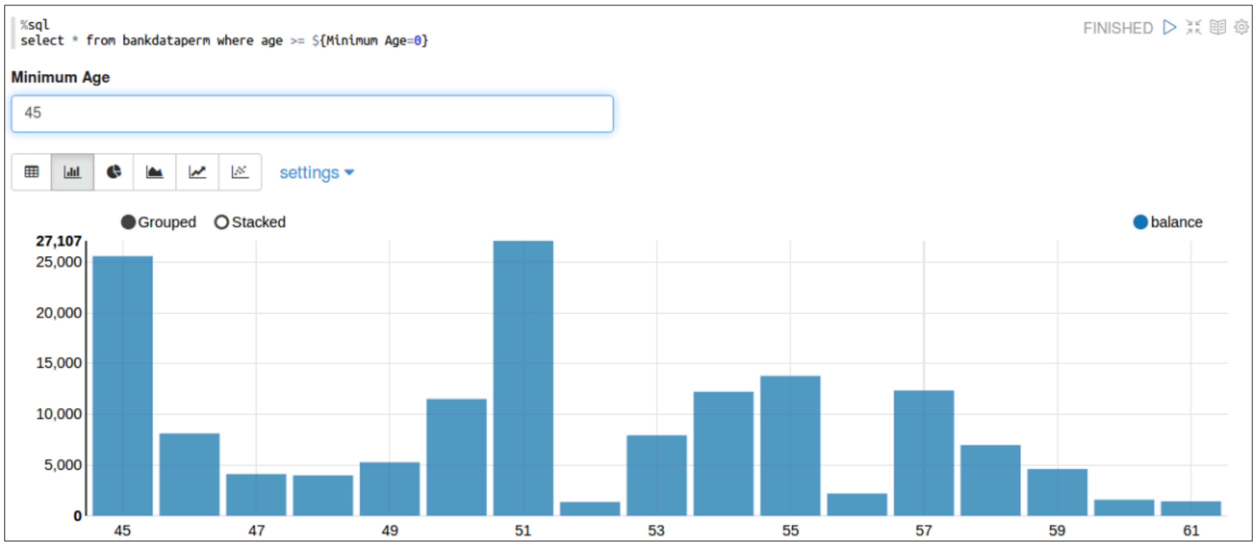
```
%sql
select * from bankdataperm where age >= ${Minimum Age=0}
```

In the following screenshot, we select the age column, where age is greater than or equal to the variable value, label the column Minimum Age and set the default value to 0 so that all values will appear by default.



```
%sql
select * from bankdataperm where age >= ${Minimum Age=0}
```

Then, in the resulting dynamic form, we set the minimum age to 45 and press enter, which results in the chart updating to reflect a minimum age of 45 in the output.



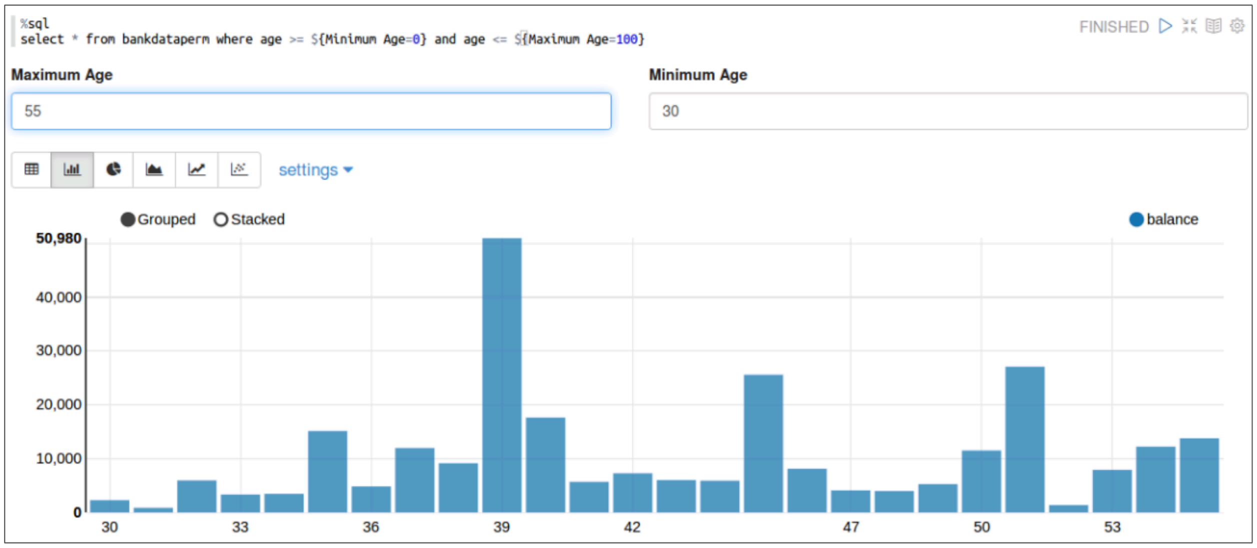
```
%sql
select * from bankdataperm where age >= ${Minimum Age=0}
```

Minimum Age: 45

Multiple Variables

Multiple variables can be included as dynamic forms.

In this example, the WHERE clause has been extended with an AND operator, so both a minimum age of 0 and a maximum age of 100 are set as defaults. The user then sets the minimum age value to 30 and the maximum age to 55 and presses enter, resulting in the underlying output changing to meet those criteria.



```
%sql
select * from bankdataperm where age >= ${Minimum Age=0} and age <= ${Maximum Age=100}
```

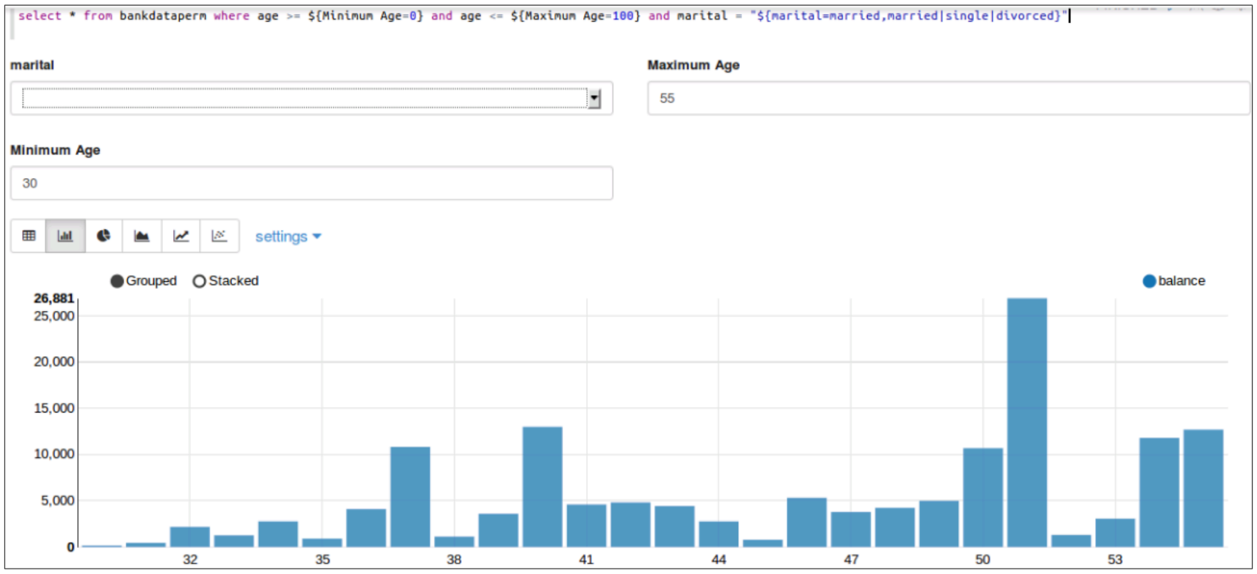
Maximum Age: 55

Minimum Age: 30

Select Lists

Dynamic forms can also include select lists (a.k.a. drop-down menus). The syntax for a select list within a `WHERE` clause would be:

```
... WHERE colName = "${LabelName=defaultLabel,opt1|opt2|opt3|…}"
```



```
select * from bankdataperm where age >= ${Minimum Age=0} and age <= ${Maximum Age=100} and marital = "${marital=married,married|single|divorced}"
```

marital:

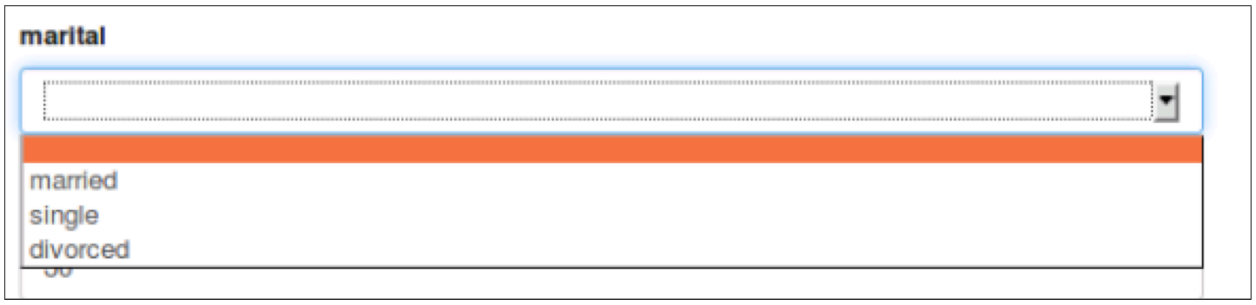
Maximum Age: 55

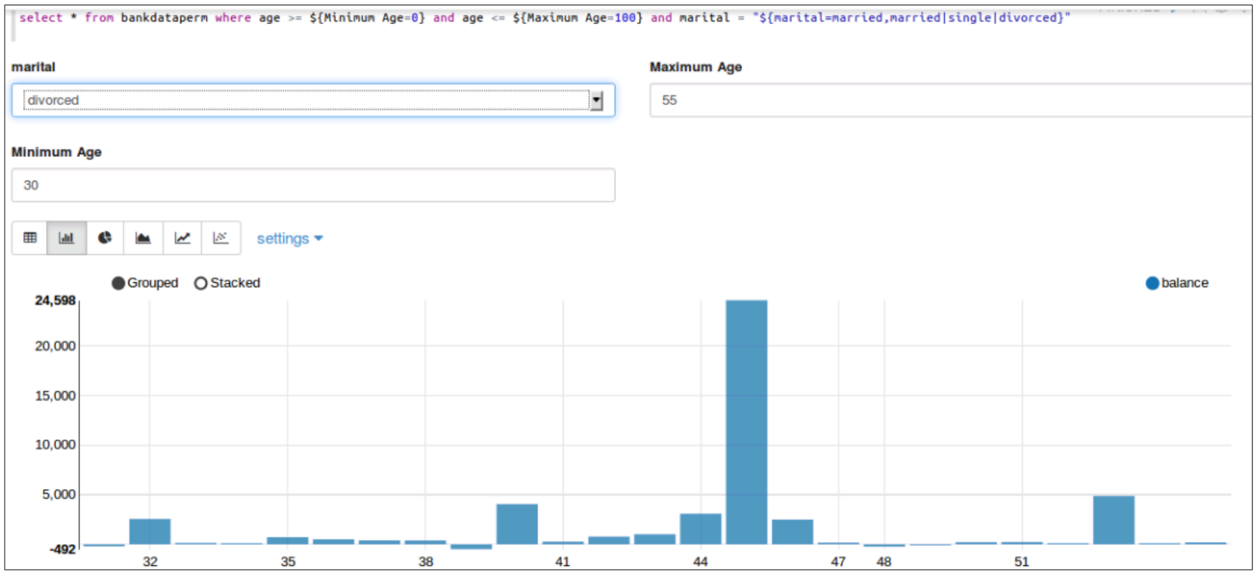
Minimum Age: 30

In the example shown, the marital column is specified, a variable created, and within the variable definition we specify the default value for marital = married.

Then, insert a comma, and provide the complete list of options you wish to provide separated by the | (pipe) character - in our case, married, single, and divorced.

This result is a new dynamic form, and the output will respond to changes in the drop-down menu.





```
select * from bankdataperm where age >= ${Minimum Age=0} and age <= ${Maximum Age=100} and marital = "${marital=married,married|single|divorced}"
```

marital: divorced

Maximum Age: 55

Minimum Age: 30

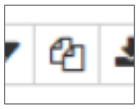
Collaboration and Sharing

You can use Notes to collaborate and share in Zeppelin.

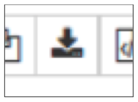
Cloning and Exporting Notes

Before sharing a note with others, it may be a good idea to make a copy of it. There are two easy ways to do this:

- **Clone**
  Makes a copy of the note in your Zeppelin notebook. You can clone a note by clicking on the button labeled "Clone the notebook" (when you hover over it with your mouse pointer) at the top of the note.

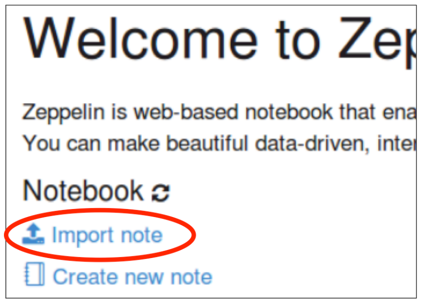


- **Export**
  Downloads a copy of the note to the local file system in JSON format. You can export a note by clicking on the button labeled "Export the notebook" at the top of the note.



Importing Notes

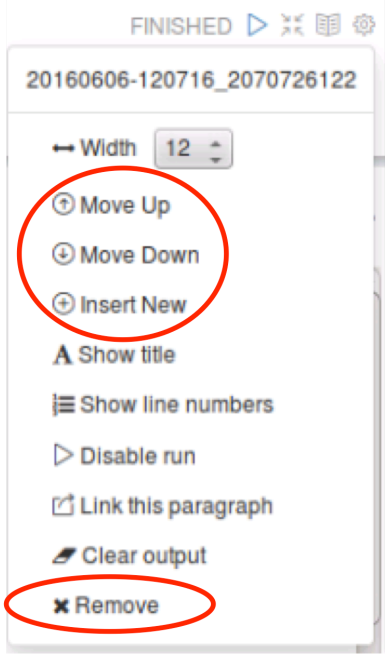
Exporting a note also gives you the ability to share that file with another developer, which they can then import into their own notebook from the Zeppelin landing page by clicking on "Import note."


Welcome to Zep
Zeppelin is web-based notebook that ena
You can make beautiful data-driven, inte
Notebook
Import note
Create new note

Note Cleanup

Often note development will be a series of trial and error approaches, comparing methods to pick the best alternative. This can result in a notebook that contains paragraphs that you don't want to keep, or don't want distributed to others for sake of clarity. Fortunately cleaning up a note prior to distribution is relatively easy.

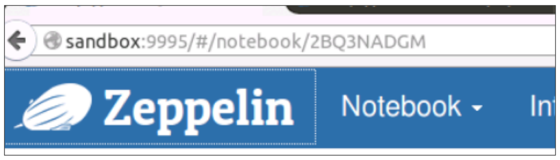
Individual paragraphs that are no longer needed can be removed/deleted from the note. In the paragraph, click on the settings button (gear icon) and select remove to delete it.



Paragraphs can also be moved up or down in the note and new paragraphs can be inserted (for example, to add comments in Markdown format describing the flow of the note).

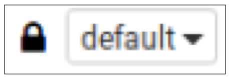
Interactive Note Sharing

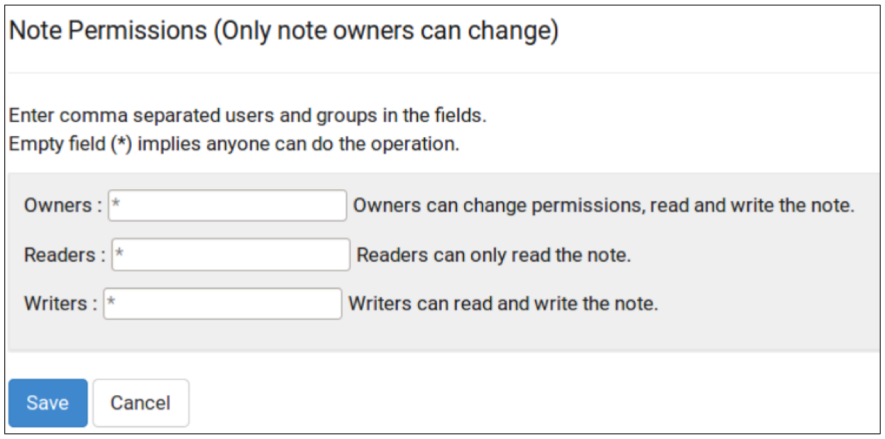
Zeppelin enables collaboration between developers by sharing the URL of the note. All connections using this URL are live, real-time connections, similar to joint editing of a Microsoft Office Online or Google Drive doc.



Note Access Control

By default, note permissions are open, meaning anyone with the note link can do anything with the note - including changing permissions, reading, and writing. To control what individuals can do with a shared note, click the **Note Permissions** (padlock) icon at the top-right corner of the note and set permissions accordingly. The exact format will vary based on the authentication mechanisms employed in your environment.





Note Permissions (Only note owners can change)

Enter comma separated users and groups in the fields.
Empty field (*) implies anyone can do the operation.

Owners : * Owners can change permissions, read and write the note.

Readers : * Readers can only read the note.

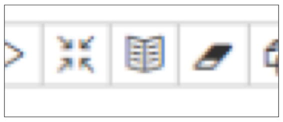
Writers : * Writers can read and write the note.

Save Cancel

Formatting Notes

Note owners can control all paragraphs at the note level, via a set of buttons at the top of the note. These controls include:

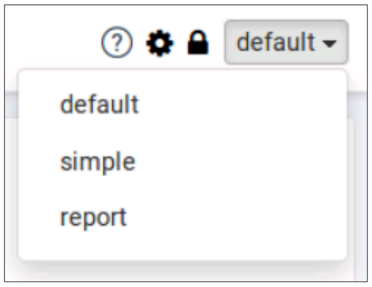
- Hide/Show all code via the button labeled "Show/hide the code,"
- Hide/Show all output via the button labeled "Show/hide the output" (which changes from an open book to a closed book icon based on the current setting), and
- Clear all output via the eraser icon button labeled "Clear output."



There are also two additional note views:

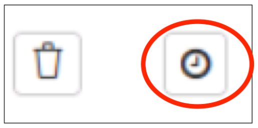
- The **Simple** view removes the note-level controls at the top of the note
- The **Report** view removes all note-level controls, as well as hides all code in the note, resulting in a series of outputs

These views can be selected by clicking the button labeled "**default**" at the top-right corner of the note and choosing the appropriate option from the resulting drop-down menu.

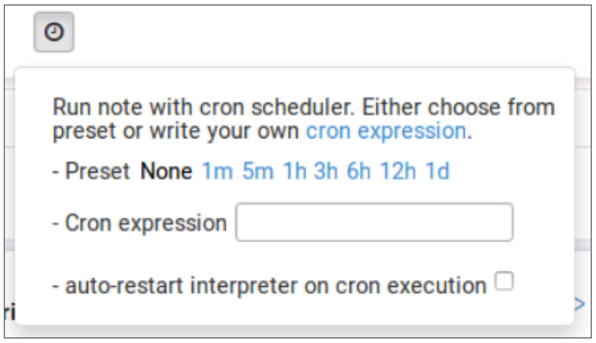


Automating Note Updates

An entire note can be played, paragraph-by-paragraph, by using the **Play** icon button labeled "**Run all paragraphs.**"



This operation can be scheduled to run on a regular basis using the scheduling feature, which is enabled by clicking the clock icon button labeled "Run scheduler." This allows you to schedule the note to run at regular intervals including every minute, every five minutes, every hour, and so on up to every 24 hours via preset links that can simply be clicked to activate. If these options are not granular enough for you, you can also schedule the note at a custom interval by supplying a `Cron` expression.



Run note with cron scheduler. Either choose from preset or write your own cron expression.

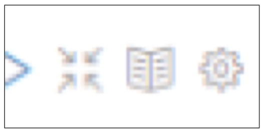
- Preset None 1m 5m 1h 3h 6h 12h 1d

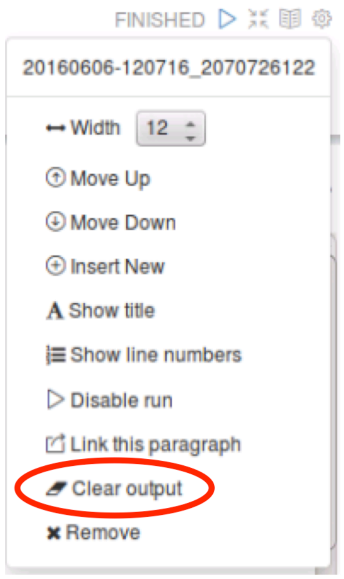
- Cron expression

- auto-restart interpreter on cron execution ☐

Paragraph Formatting

Paragraphs can also be formatted prior to distribution on an individual basis. These settings are available in the buttons menu at the top right of each paragraph, as well as underneath the settings menu (gear icon) button.





Formatting options that were also available at the note level include:

- Hide/Show paragraph code,
- Hide/Show paragraph output, and
- Clear paragraph output (only available under settings).

Paragraph Enhancements

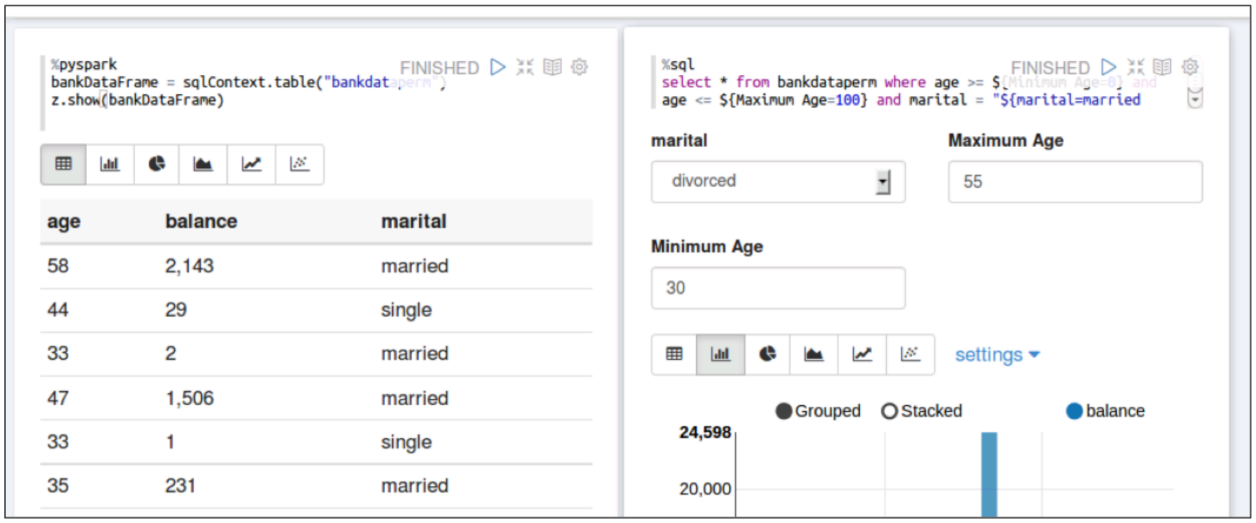
The visual appearance of paragraphs can be enhanced to support various collaboration goals. Such enhancements include:

- Setting paragraph width
- Showing paragraph title
- Showing line numbers

Width

Example:

Let's assume you want to create a dashboard within a Zeppelin note, showing multiple views of the same data on the same line.



```
%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

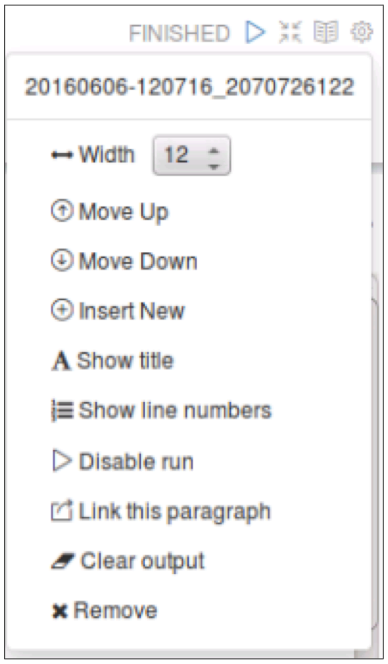
```
%sql
select * from bankdataperm where age >= ${Minimum Age=0} and
age <= ${Maximum Age=100} and marital = "${marital=married
```

marital: divorced

Maximum Age: 55

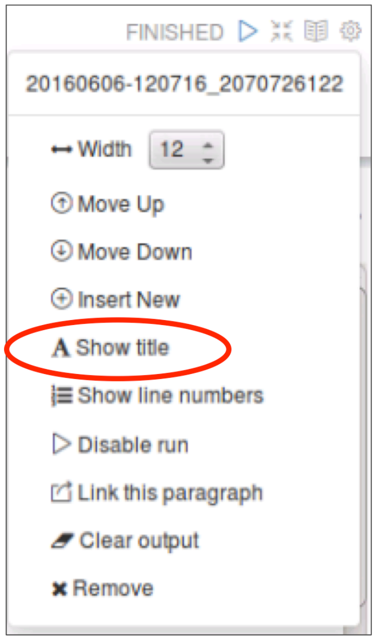
Minimum Age: 30

This can be accomplished by modifying the Width setting, found in paragraph settings. By default, the maximum width is used per paragraph, however, this can be modified so that two or more paragraphs will appear on the same line.

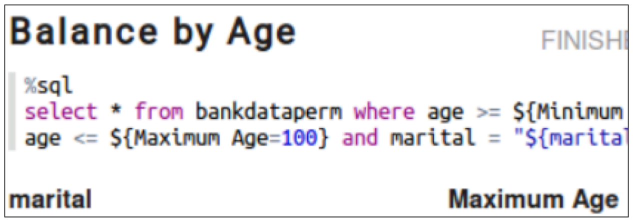


Show Title

Paragraphs can be given titles for added clarity when viewing output. To set a title, select **Show title** under paragraph settings. The default title is "Untitled."



Click on the title to change it, type the new title, and press the Enter key to set it.



Balance by Age

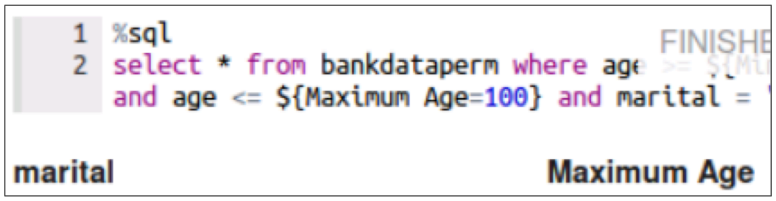
FINISHI

```
%sql
select * from bankdataperm where age >= ${Minimum
age <= ${Maximum Age=100} and marital = "${marita
```

marital Maximum Age

Line Numbers

Paragraphs displaying code can also be enhanced by showing line numbers for each line of code.

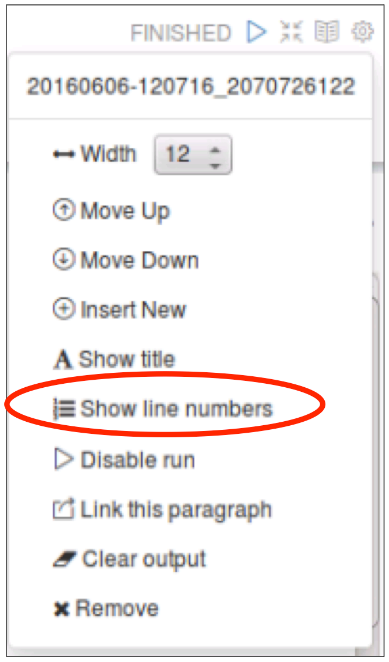


FINISHE

```
1 %sql
2 select * from bankdataperm where age >= ${Mi
  and age <= ${Maximum Age=100} and marital = '
```

marital Maximum Age

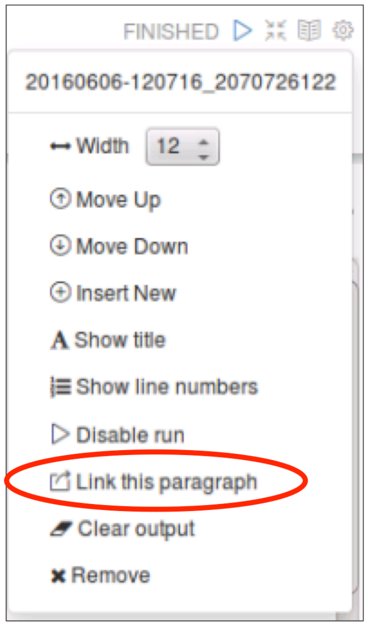
To turn on this feature, select **Show line numbers** under paragraph settings.



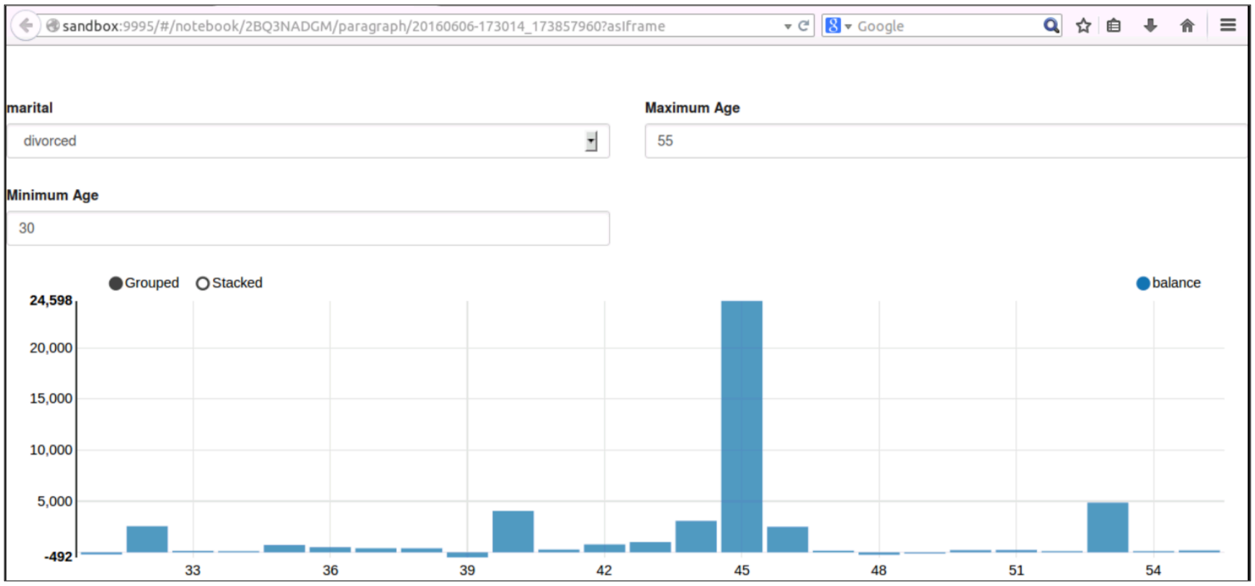
The numbers will appear to the left of the code lines. Lines that are wrapped based on the width of the paragraph will only be given a single number, even though on the screen they will appear as multiple lines.

Sharing Paragraphs

Individual paragraphs can be shared by generating a link, which can be used as an iframe or otherwise embedded in an external-to-Zeppelin report. To generate this URL, select **Link this paragraph** under paragraph settings.



This will automatically open the paragraph in a new browser tab, and the URL can be copied and pasted into whatever report or web page is needed.

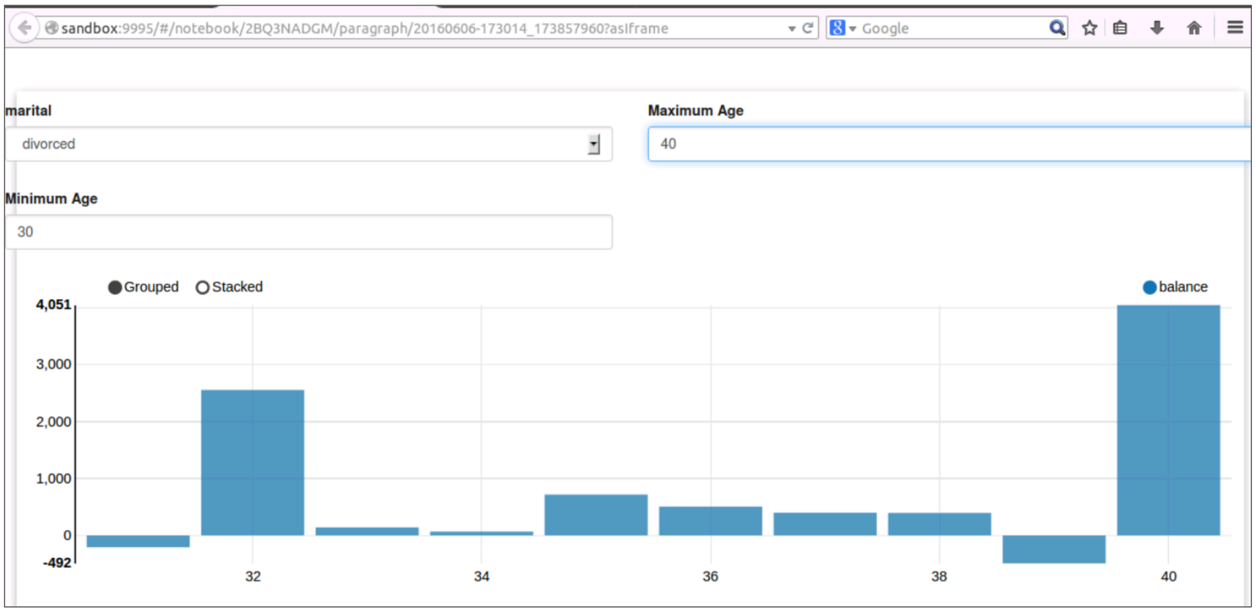


marital: divorced

Maximum Age: 55

Minimum Age: 30

If dynamic forms have been enabled for this note, anyone who modifies the form values will change the appearance of the paragraph output for everyone looking at the link. This can be a valuable tool if, for example, a marketing department wants to generate multiple outputs based on slight tweaks to the query. You can allow them to do this without giving them access to the entire note, and without the need to modify the code on the backend.


sandbox:9995/#/notebook/2BQ3NADGM/paragraph/20160606-173014_173857960?asIframe
Google
● Grouped
○ Stacked
● balance
4,051
3,000
2,000
1,000
0
-492
32
34
36
38
40

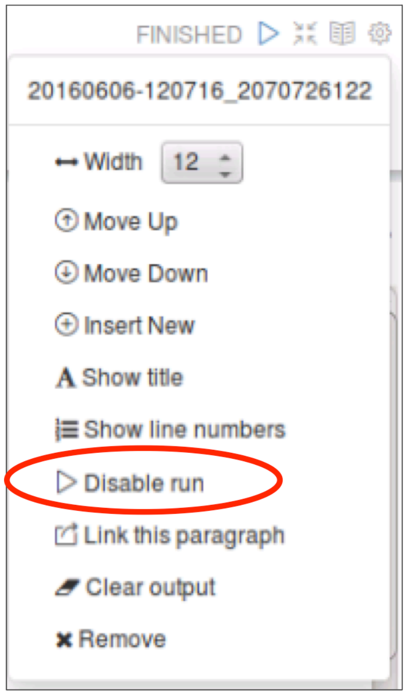
marital: divorced

Maximum Age: 40

Minimum Age: 30

Disabling Paragraph Output Changes

In some cases, you may want to freeze a paragraph's output in its current form. To do so, select the **Disable run** under paragraph settings.



Any changes to the code, as well as changes to dynamic forms input, will not change the output presented as long as the Disable run option is selected.

Knowledge Check

Use the following questions to assess your understanding of the concepts presented in this lesson.

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?

Answers

1) What is the value of data visualization?

Answer: Enable humans to make inferences and draw conclusions about large sets of data that would be impossible to make by looking at the data in tabular format.

2) How many chart views does Zeppelin provide by default?

Answer: Five

3) How do you share a copy of your note (non-collaborative) with another developer?

Answer: Export to JSON format, then they can import it.

4) How do you share your note collaboratively with another developer?

Answer: Give them the note URL

5) Which note view provides only paragraph outputs?

Answer: The Report view.

6) Which paragraph feature provides the ability for an outside person to see a paragraph's output without having access to the note?

Answer: Link the paragraph

7) What paragraph feature allows you to give outside users the ability to modify parameters and update the displayed output without using code?

Answer: Dynamic forms

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

Job Monitoring

Lesson Objectives

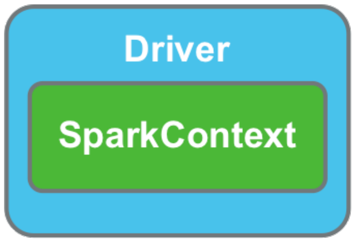
After completing this lesson, students should be able to:

- ✓ 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

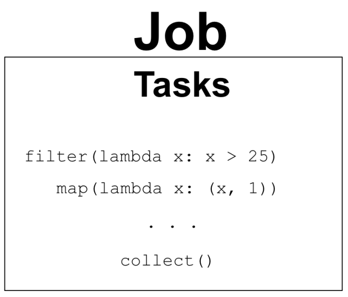
Spark Job Anatomy

Spark data processing operations can be spoken about in terms of jobs.

Jobs and Tasks



Spark applications require a Driver, which in turn loads and monitors the `SparkContext`. The `SparkContext` is then responsible for launching and managing Spark jobs. But what do we mean when we say job? When you type a line of code to use a Spark function, such as `flatMap()`, `filter()`, or `map()`, you are defining a Spark task which must be performed. A task is a unit of work, or "a thing to be done." When you put one or more tasks together with a resulting action task - such as `collect()` or `save()` - you have defined a Spark job. A job, then, is a collection of tasks (or things to be done) culminating in an action.



NOTE:

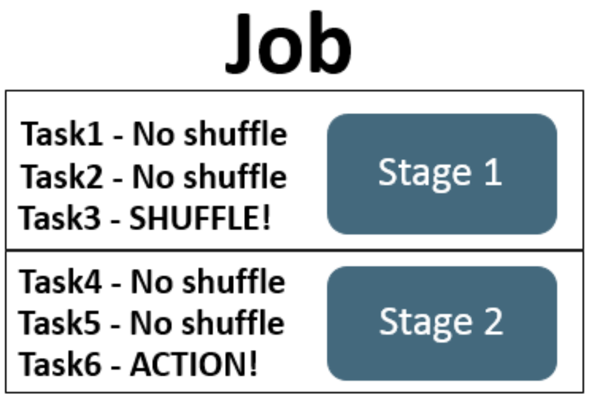
Not explicitly called out here: a Spark application can consist of one or more Spark jobs. Every action is considered part of a unique job, even if the action is the only task being performed.

Job Stages

A Spark job can be made up of several types of tasks. Some tasks don't require any data be moved from one executor to another in order to finish processing. This is referred to as a "**narrow**" operation, or one that does not require a data "**shuffle**" in order to execute.

Transformations that *do* require that data be moved between executors are called "**wide**" operations, and that movement of data is called a **shuffle.**

When executed, Spark will evaluate tasks that need to be performed and break up a job at any point where a shuffle will be required. While non-shuffle operations can happen somewhat asynchronously if needed, a task that follows a shuffle *must* wait for the shuffle to complete before executing.

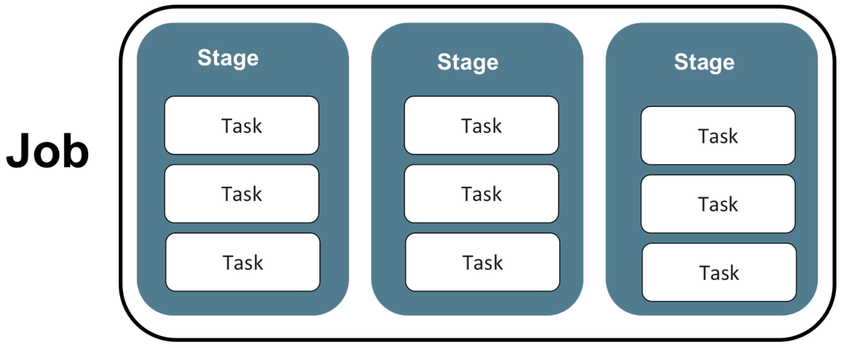


This break point, where processing must complete before the next task or set of tasks is executed, is referred to as a **stage.** A stage, then, can be thought of as "a logical grouping of tasks" or things to be done. A **shuffle** is a task requiring that data between RDD partitions be aggregated (or combined) in order to produce a desired result.

Job Anatomy Summary

To put it all together:

- A Spark job is a collection of tasks that culminate in an action.
- Tasks are a unit of work, or a thing to be done. Each transformation and action is a separate task.
- Because shuffled data must be complete before a following task can begin, jobs are divided into stages based on these shuffle boundaries.



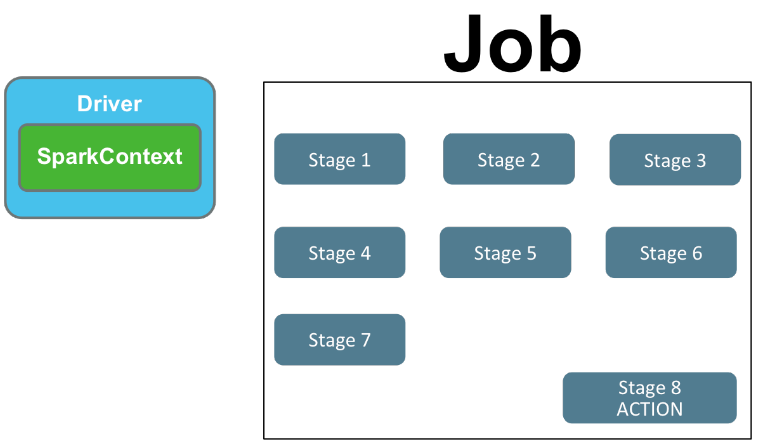
Anatomy of a Spark Job

Parallel Execution

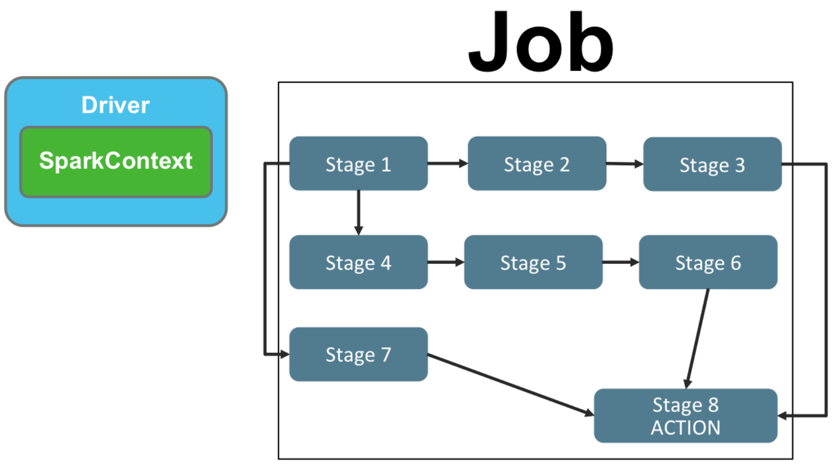
Spark jobs are automatically optimized via parallel execution at different levels.

DAGs and Stages

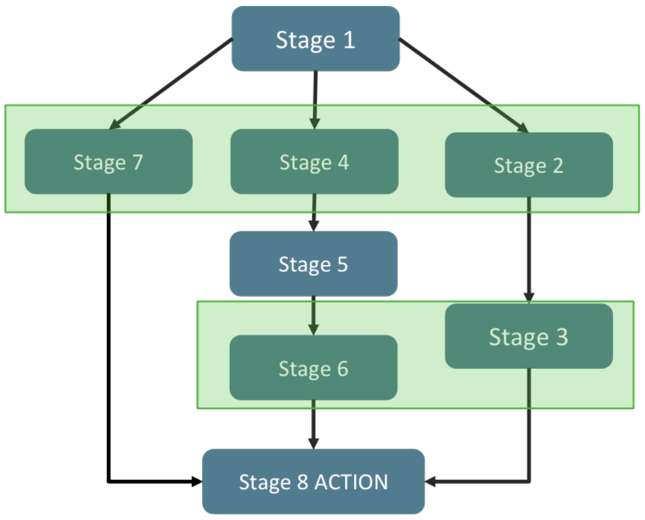
A job may have several stages (in this example, the job is divided into eight stages).



However, not all stages are dependent on one another. For example, in this job Stage 1 has to run first, but once it has completed there are three other stages (two, four, and seven) that can begin execution.



Operating in this fashion is known as a **Directed Acyclic Graph**, or **DAG**. A DAG is essentially a logical ordering of operations (in the case of our discussion here, Spark stages) based on dependencies. Since there is no reason for Stage 7 to wait for all of the previous six stages to complete, Spark will go ahead and execute it immediately after Stage 1 completes, along with Stage 2 and Stage 4.

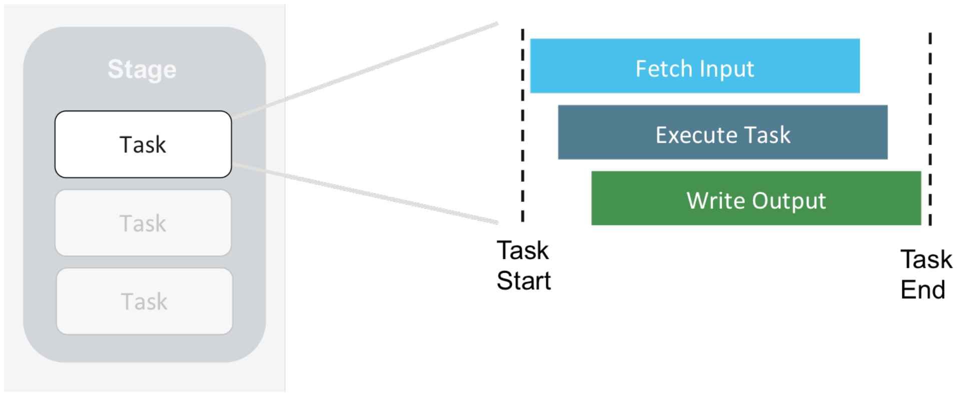


This parallel operation based on logical dependencies allows, in some cases, for significantly faster job completion across a cluster compared to platforms that require stages to complete, one at a time, in order.

The tracking and managing of these stages and their dependencies is managed by a Spark component known as a **DAG Scheduler**. It is the DAG scheduler that tells Spark which stages (sets of tasks) to execute and in what order. The DAG Scheduler ensures that dependencies are met, and any dependent stages have completed prior to the next stage completing.

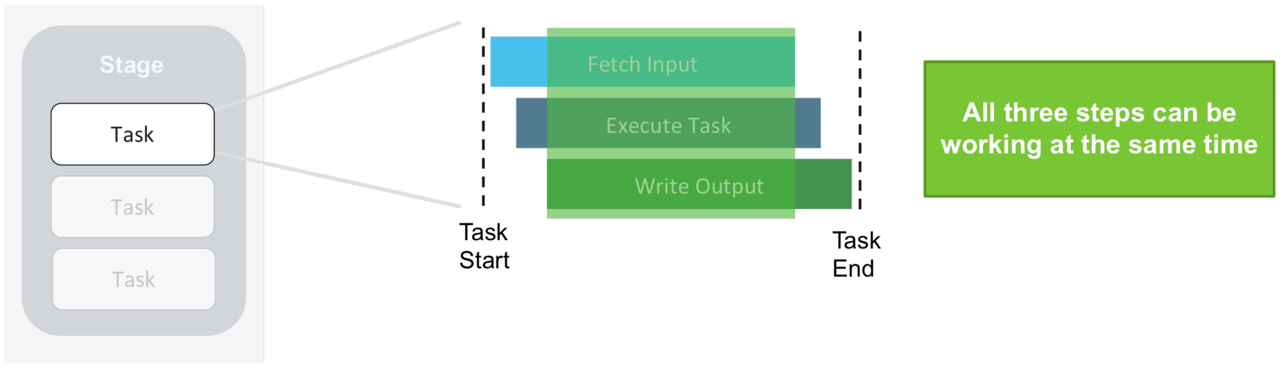
Task Steps

A task is actually a collection of three separate steps. When a task is first scheduled, it must first fetch the data it will need - either from an outside source, or perhaps from the results of a previous task. Once the data has been collected, the operation that the task is to do on that data can execute. Finally, the task produces some kind of output, either as an action, or perhaps as an intermediate step for a task to follow.



Task Steps: Fetch Input, Execute Task, Write Output

Tasks can begin execution once data has started to be collected. There is no need for the entire set of data to be loaded prior to performing the task operation. Therefore, execution begins as soon as the first bits of data are available, and can continue in parallel while the rest of the data is being fetched.

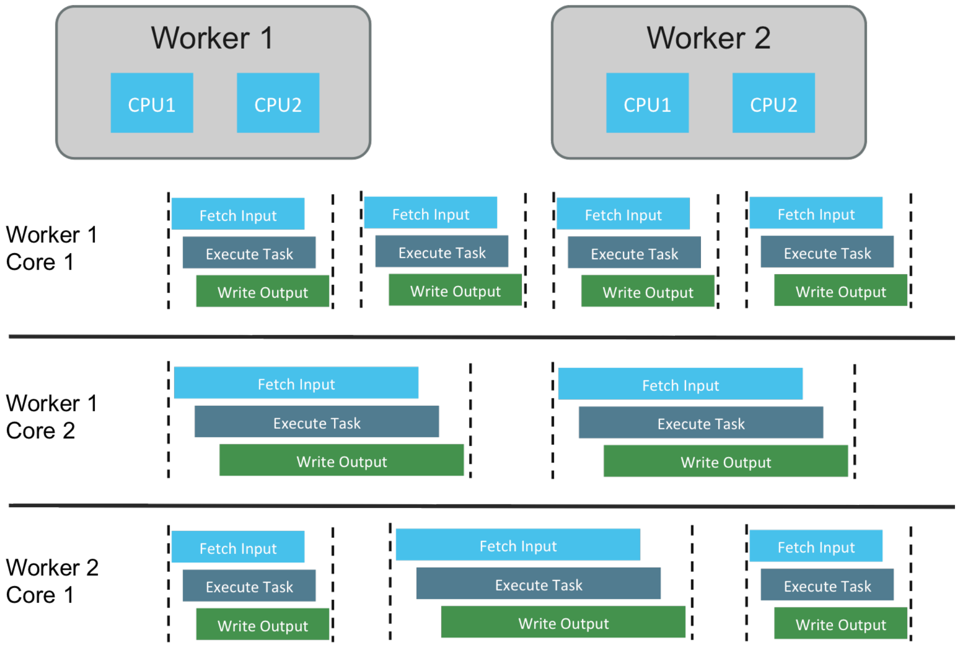


Parallel Task Step Execution

Furthermore, the output production step can begin as soon as the first bits of data have been transformed, and can theoretically be happening while the operation is being executed *and* while the rest of the data is being fetched. In this manner, all three steps of a task can be running at the same time, with the execute phase starting shortly after the fetch begins, and the output phase starting shortly after the execute phase begins. In terms of completion, the fetch will always complete first, but the execute can finish shortly thereafter, with the output phase shortly after that.

Tasks and CPU Cores

Tasks are independent units of work. Therefore, CPU cores can be operating on different tasks at different times without issue. If an executor has more than one CPU core available, it can work on multiple tasks at the same time, further increasing the parallelization of job tasks and decreasing the amount of time it takes to complete a Spark job.



CPU Cores on Each Node can Process Tasks Independently

Spark Application UI

Now that we've explored the anatomy of a Spark job and understand how they are executed on the cluster, let's take a look at monitoring those jobs and their components via the Spark Application UI.

Spark Application UI

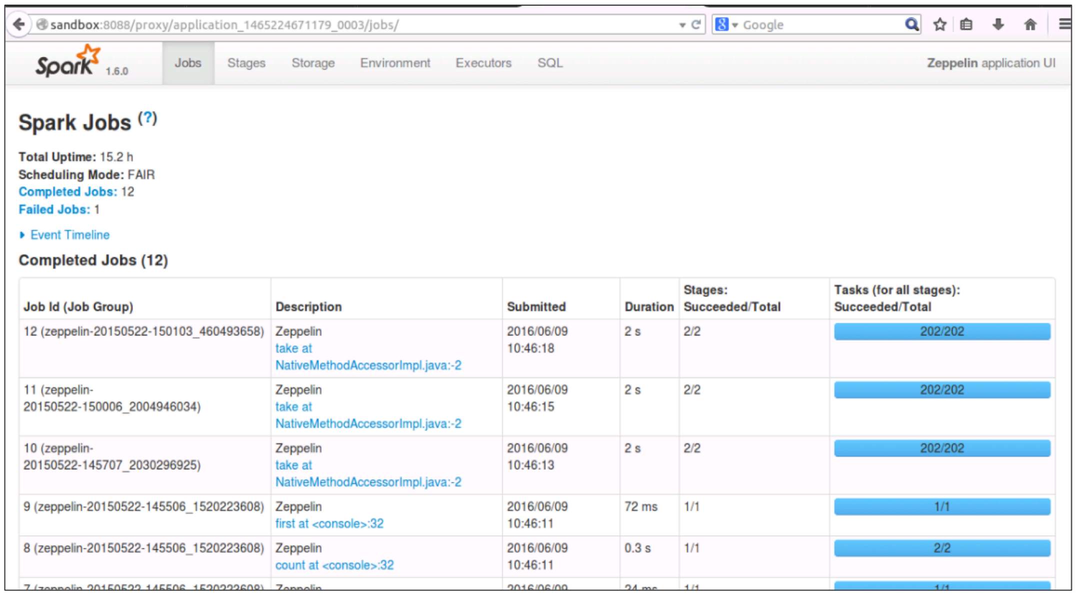
The **Spark Application UI** is a web interface generated by a `SparkContext`. It is therefore available for the life of the `SparkContext`. Once the `SparkContext` has been shut down, the Spark Application UI will no longer be available.

You access the Spark Application UI by default via your Driver node at port 4040.

Every `SparkContext` instance manages a separate Spark Application UI instance. Therefore, if multiple `SparkContext` instances are running on the same system, multiple Spark Application UIs will be available. Since they cannot share a port, when a `SparkContext` launches and detects an existing Spark Application UI, it will generate its own version of the monitoring tool at the next available port number, incremented by 1. Therefore, if you are running Zeppelin and it has created a Spark Application UI instance at port 4040, and then you launch an instance of the PySpark REPL in a terminal on the same machine, the REPL version of the monitoring site will exist at port 4041 instead of 4040. A third `SparkContext` would create the UI at port 4042, and so on.

Once a `SparkContext` is exited, that port number becomes available. Therefore, if you exited Zeppelin (using port 4040) and opened another REPL, it would create its Spark Application UI at port 4040. The two older `SparkContext` instances would keep the port numbers they had when they started.

Application UI Landing Page



Spark Jobs (?)

Total Uptime: 15.2 h
Scheduling Mode: FAIR
Completed Jobs: 12
Failed Jobs: 1

▸ Event Timeline

Completed Jobs (12)

Job Id (Job Group)	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
12 (zeppelin-20150522-150103_460493658)	Zeppelin take at NativeMethodAccessorImpl.java:-2	2016/06/09 10:46:18	2 s	2/2	202/202
11 (zeppelin-20150522-150006_2004946034)	Zeppelin take at NativeMethodAccessorImpl.java:-2	2016/06/09 10:46:15	2 s	2/2	202/202
10 (zeppelin-20150522-145707_2030296925)	Zeppelin take at NativeMethodAccessorImpl.java:-2	2016/06/09 10:46:13	2 s	2/2	202/202
9 (zeppelin-20150522-145506_1520223608)	Zeppelin first at <console>:32	2016/06/09 10:46:11	72 ms	1/1	1/1
8 (zeppelin-20150522-145506_1520223608)	Zeppelin count at <console>:32	2016/06/09 10:46:11	0.3 s	1/1	2/2

Spark Application UI Landing Page

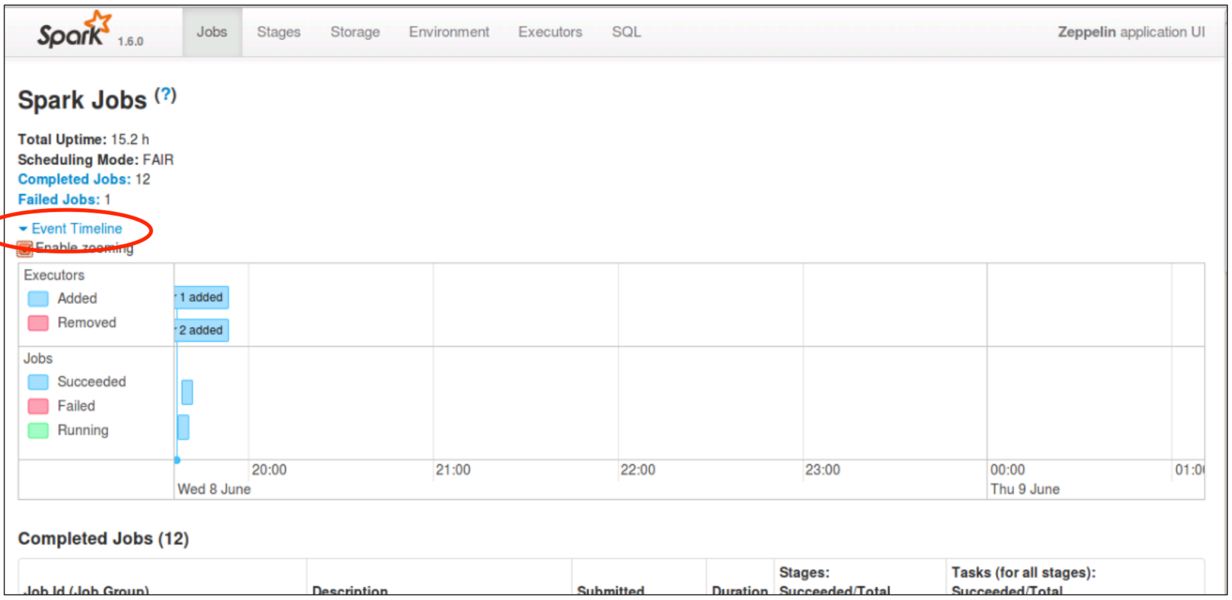
The Spark UI landing page opens up to a list of all of the Spark jobs that have been run by this SparkContext instance. You can see information about the number of jobs completed, as well as overview information for each job in terms of ID, description, when it was submitted, how long it took to execute, how many stages it had and how many of those were successful, and the number of tasks for all of those stages (and how many were successful.)

Clicking on a job description link will result in a screen providing more detailed information about that particular job.

NOTE:

The URL - which was typed as "`sandbox:4040`" - was redirected to port 8088. Port 8088 is the YARN ResourceManager UI, which tracks and manages all YARN jobs. This means that in this instance, Zeppelin has been configured to run on (and be managed resource-wise by) YARN.

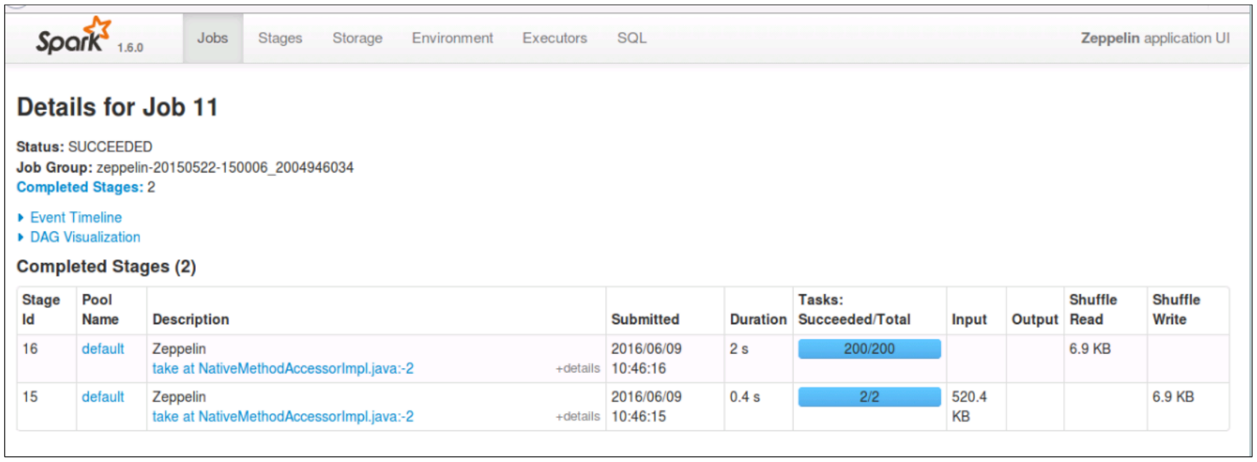
Landing Page Events Timeline



Spark Application UI Landing Page Events Timeline

On the Spark Application UI landing page, you will notice a link called Event Timeline. Clicking on this link results in a visualization that shows executors being added and removed from the cluster, as well as jobs being tracked and their current status. Enabling zoom allows you to see more granular detail, which can be particularly helpful if a large number of jobs have been executed over a long period of time for this SparkContext instance.

Job View



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

Details for Job 11

Status: SUCCEEDED
Job Group: zeppelin-20150522-150006_2004946034
Completed Stages: 2

- Event Timeline
- DAG Visualization

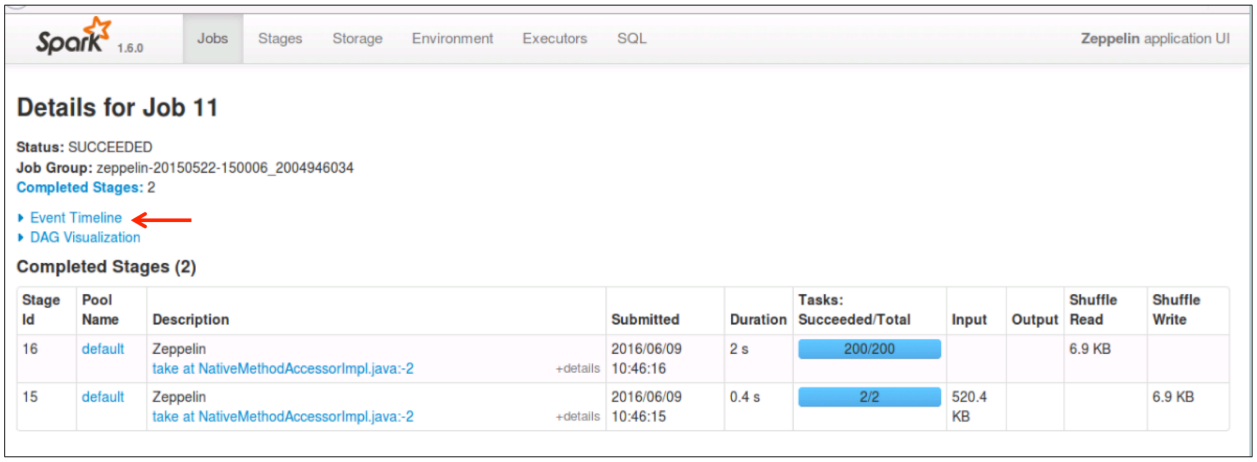
Completed Stages (2)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
16	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:16	2 s	200/200			6.9 KB	
15	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:15	0.4 s	2/2	520.4 KB			6.9 KB

S[ark Application UI Job View

Clicking on a job description on the Jobs landing page takes you to the "Details for Job *nn*" page (in our example Job 11). Here you can see more specific information about the job stages, including description, when they were submitted, how long they took to run, how many tasks succeeded out of how many attempted, the size of the input and output of each stage, how much data shuffling occurred between stages.

Job Events Timeline



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

Details for Job 11

Status: SUCCEEDED
Job Group: zeppelin-20150522-150006_2004946034
Completed Stages: 2

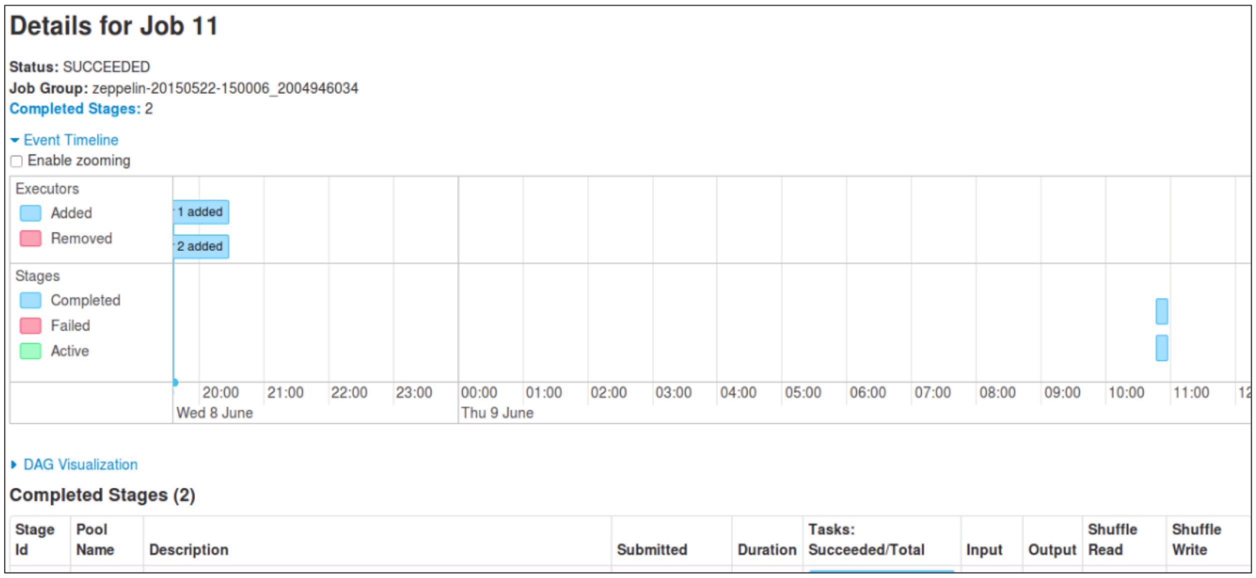
- Event Timeline
- DAG Visualization

Completed Stages (2)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
16	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:16	2 s	200/200			6.9 KB	
15	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:15	0.4 s	2/2	520.4 KB			6.9 KB

Event Timeline on Spark Application UI Job View

Clicking on the Event timeline once again results in a visualization very similar to the one on the landing page, but this time only for the stages of that particular job instead of all jobs.



Details for Job 11

Status: SUCCEEDED
Job Group: zeppelin-20150522-150006_2004946034
Completed Stages: 2

Event Timeline
Enable zooming

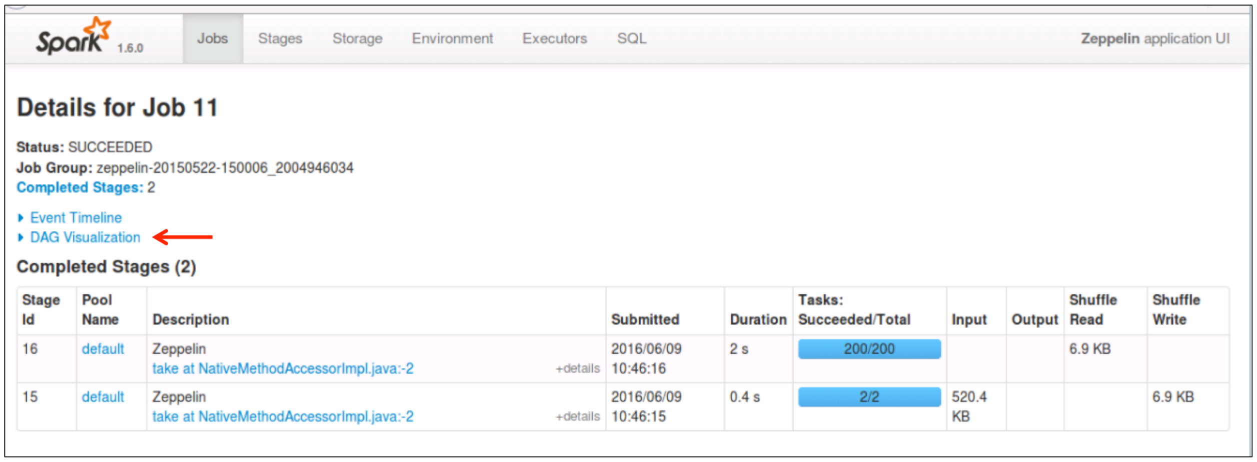
DAG Visualization

Completed Stages (2)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write

Spark Application UI Job Events Timline

Job DAG



Details for Job 11

Status: SUCCEEDED
Job Group: zeppelin-20150522-150006_2004946034
Completed Stages: 2

Event Timeline
DAG Visualization

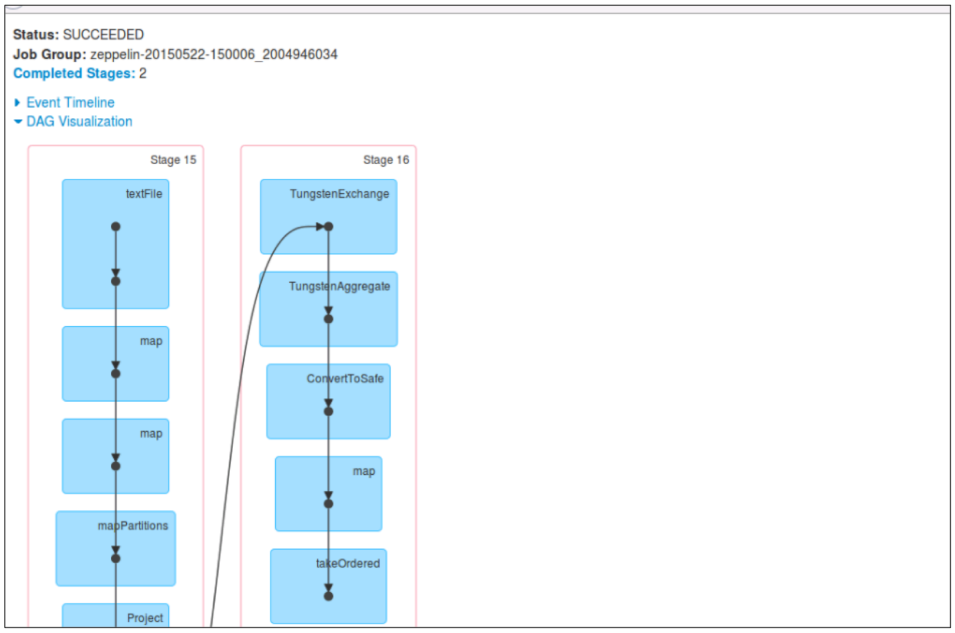
Completed Stages (2)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
16	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:16	2 s	200/200			6.9 KB	
15	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:15	0.4 s	2/2	520.4 KB			6.9 KB

DAG Link on Spark Application UI Job View

In addition, a new link is available on this screen: *DAG Visualization*.

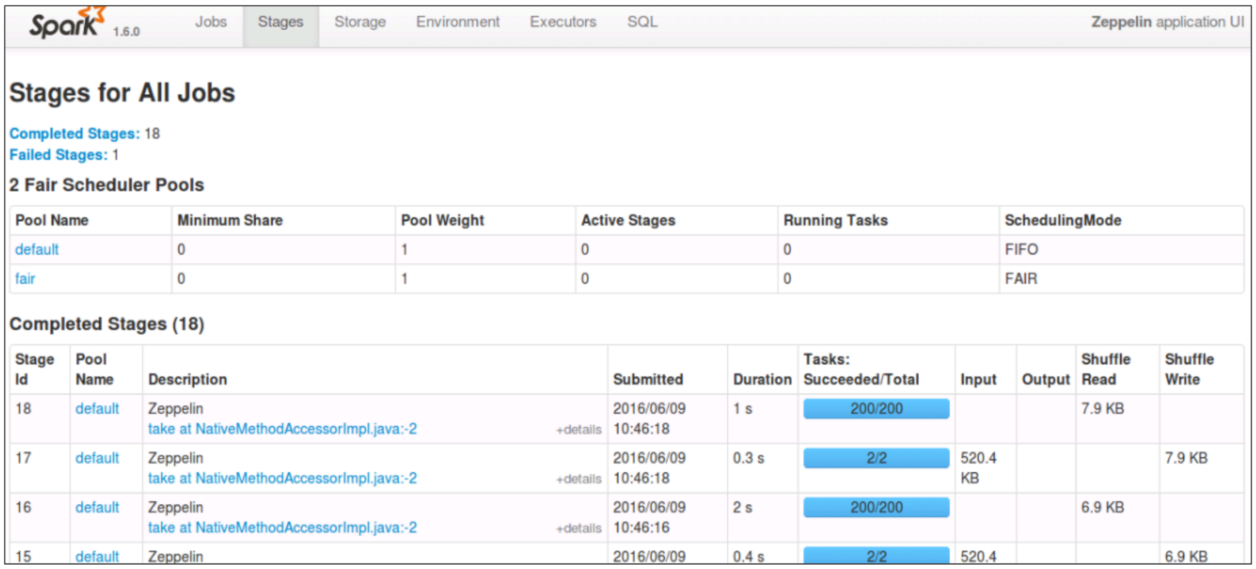
Clicking the **DAG Visualization** link results in a visual display of the stages (red outline boxes) and the flow of tasks each contains, as well as the dependencies between the stages.



Status: SUCCEEDED
Job Group: zeppelin-20150522-150006_2004946034
Completed Stages: 2

Spark Application UI Job DAG

Stage View



Stages for All Jobs

Completed Stages: 18
Failed Stages: 1

2 Fair Scheduler Pools

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
default	0	1	0	0	FIFO
fair	0	1	0	0	FAIR

Completed Stages (18)

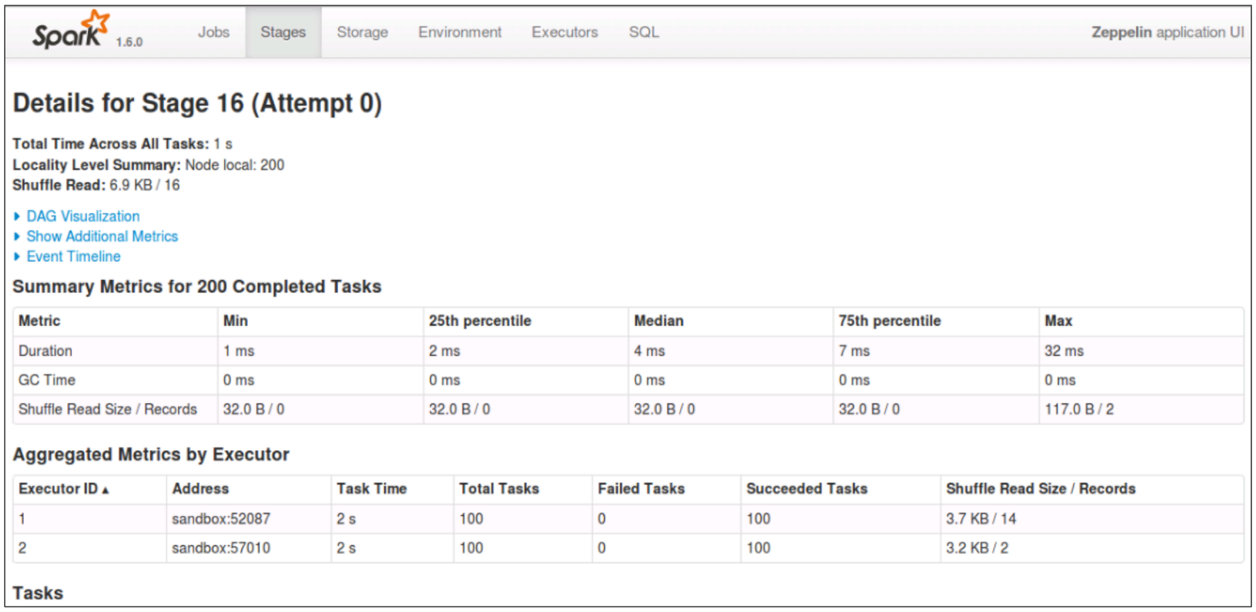
Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
18	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:18	1 s	200/200			7.9 KB	
17	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:18	0.3 s	2/2	520.4 KB			7.9 KB
16	default	Zeppelin take at NativeMethodAccessorImpl.java:-2 +details	2016/06/09 10:46:16	2 s	200/200			6.9 KB	
15	default	Zeppelin	2016/06/09	0.4 s	2/2	520.4			6.9 KB

Spark Application UI Stage View

At the top of the window, to the right of the Jobs tab you will see a tab called Stages.

Clicking on the **Stages** tab will result in a screen similar to the Jobs landing page, but instead of tracking activity at the job level it tracks it at the stage level, providing pertinent high-level information about each stage.

Stage Detail



Details for Stage 16 (Attempt 0)

Total Time Across All Tasks: 1 s
Locality Level Summary: Node local: 200
Shuffle Read: 6.9 KB / 16

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 ms	2 ms	4 ms	7 ms	32 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	32.0 B / 0	32.0 B / 0	32.0 B / 0	32.0 B / 0	117.0 B / 2

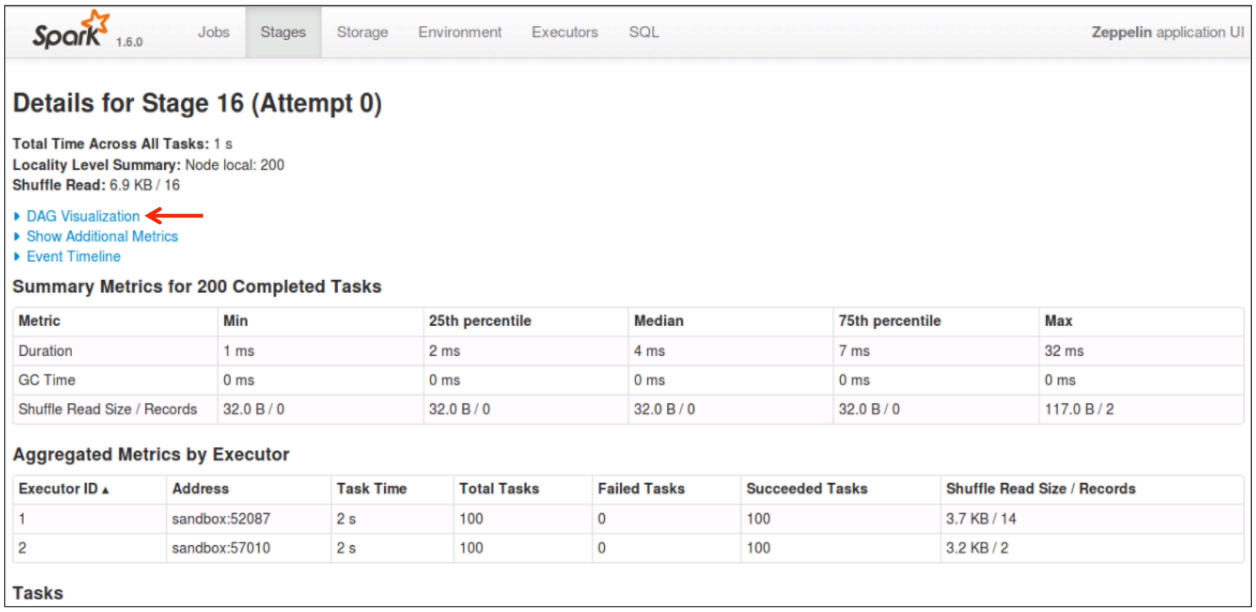
Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	sandbox:52087	2 s	100	0	100	3.7 KB / 14
2	sandbox:57010	2 s	100	0	100	3.2 KB / 2

Tasks

Spark Application UI Stage Detail

Stage DAG



Details for Stage 16 (Attempt 0)

Total Time Across All Tasks: 1 s
Locality Level Summary: Node local: 200
Shuffle Read: 6.9 KB / 16

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 ms	2 ms	4 ms	7 ms	32 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	32.0 B / 0	32.0 B / 0	32.0 B / 0	32.0 B / 0	117.0 B / 2

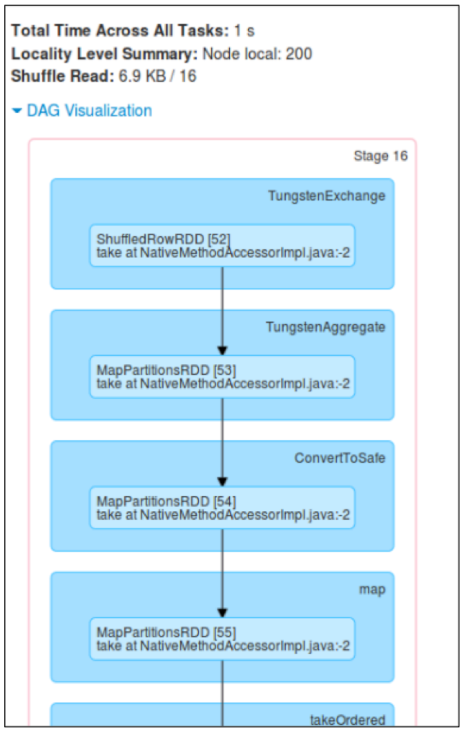
Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	sandbox:52087	2 s	100	0	100	3.7 KB / 14
2	sandbox:57010	2 s	100	0	100	3.2 KB / 2

Tasks

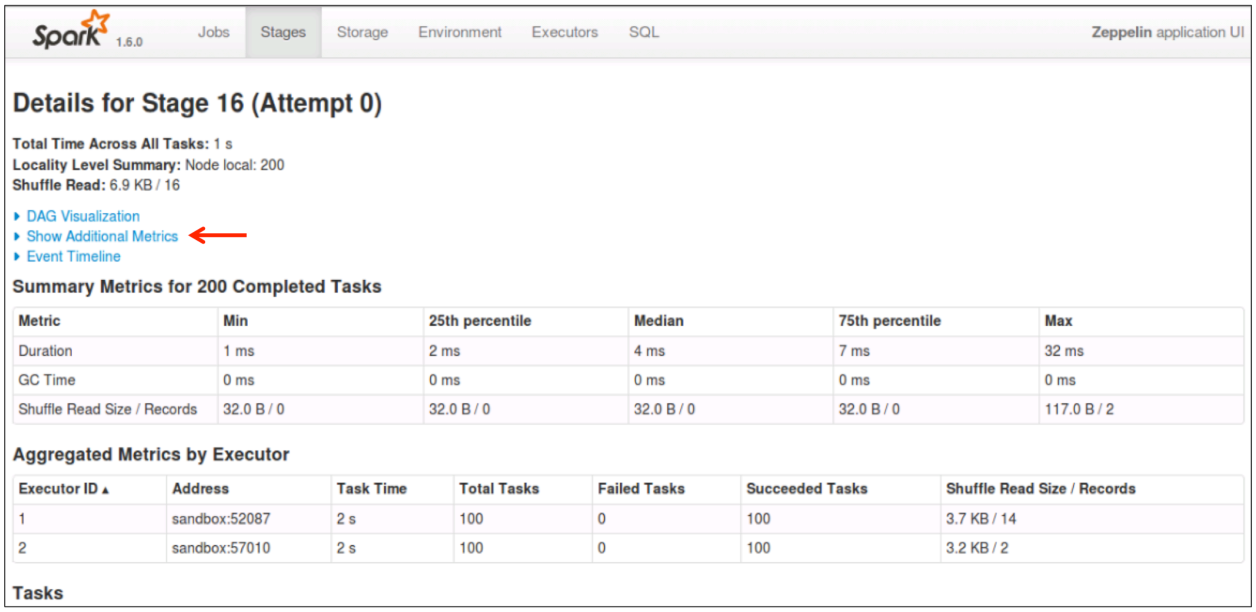
DAG Visualization Link

Clicking on the DAG Visualization link from this page results in a visual display of the operations within the stage in a DAG formatted view.


Total Time Across All Tasks: 1 s
Locality Level Summary: Node local: 200
Shuffle Read: 6.9 KB / 16

DAG Visualization

Stage Additional Metrics



Details for Stage 16 (Attempt 0)

Total Time Across All Tasks: 1 s
Locality Level Summary: Node local: 200
Shuffle Read: 6.9 KB / 16

Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 ms	2 ms	4 ms	7 ms	32 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	32.0 B / 0	32.0 B / 0	32.0 B / 0	32.0 B / 0	117.0 B / 2

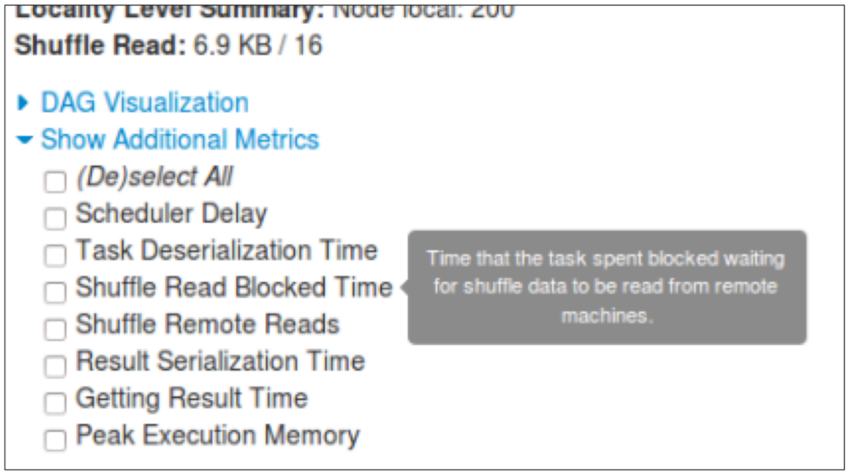
Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	sandbox:52087	2 s	100	0	100	3.7 KB / 14
2	sandbox:57010	2 s	100	0	100	3.2 KB / 2

Tasks

Clicking on the Show Additional Metrics link allows you to customize the display of information collected in the table below. Hovering over the metric will result in a brief description of the information that metric can provide.

This can be particularly useful when troubleshooting and determining the root cause of performance problems for an application.



Locality Level Summary: Node local: 200

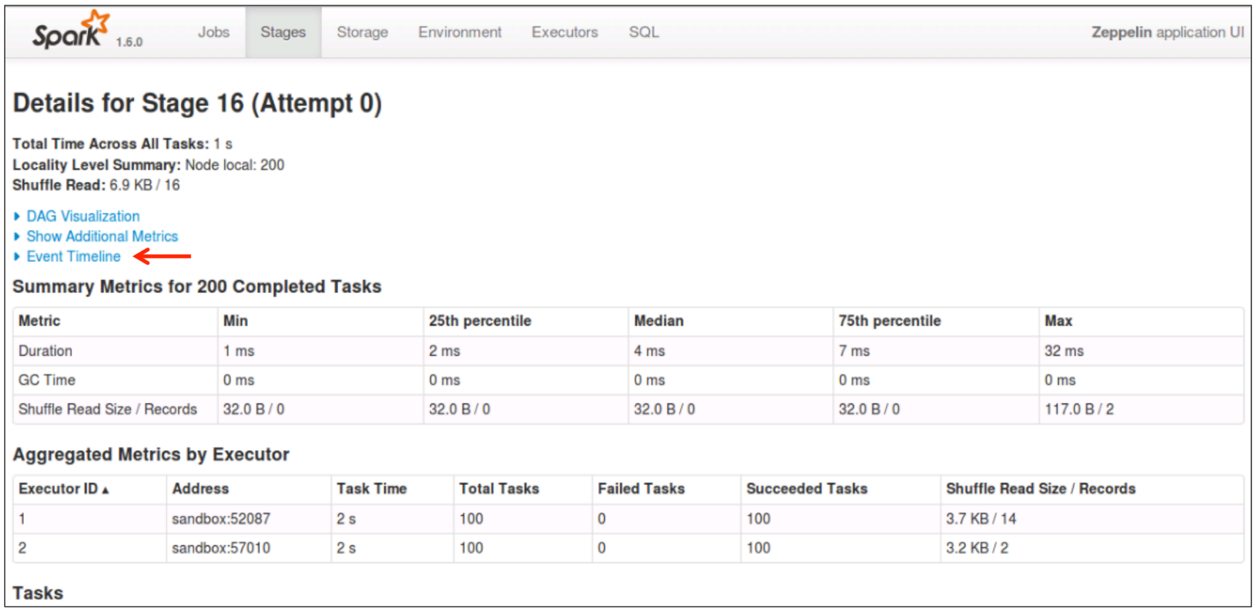
Shuffle Read: 6.9 KB / 16

- DAG Visualization
- Show Additional Metrics
  - [ ] *(De)select All*
  - [ ] Scheduler Delay
  - [ ] Task Deserialization Time
  - [ ] Shuffle Read Blocked Time
  - [ ] Shuffle Remote Reads
  - [ ] Result Serialization Time
  - [ ] Getting Result Time
  - [ ] Peak Execution Memory

Time that the task spent blocked waiting for shuffle data to be read from remote machines.

Stage Additional Metrics

Stage Event Timeline



Details for Stage 16 (Attempt 0)

Total Time Across All Tasks: 1 s
Locality Level Summary: Node local: 200
Shuffle Read: 6.9 KB / 16

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 ms	2 ms	4 ms	7 ms	32 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	32.0 B / 0	32.0 B / 0	32.0 B / 0	32.0 B / 0	117.0 B / 2

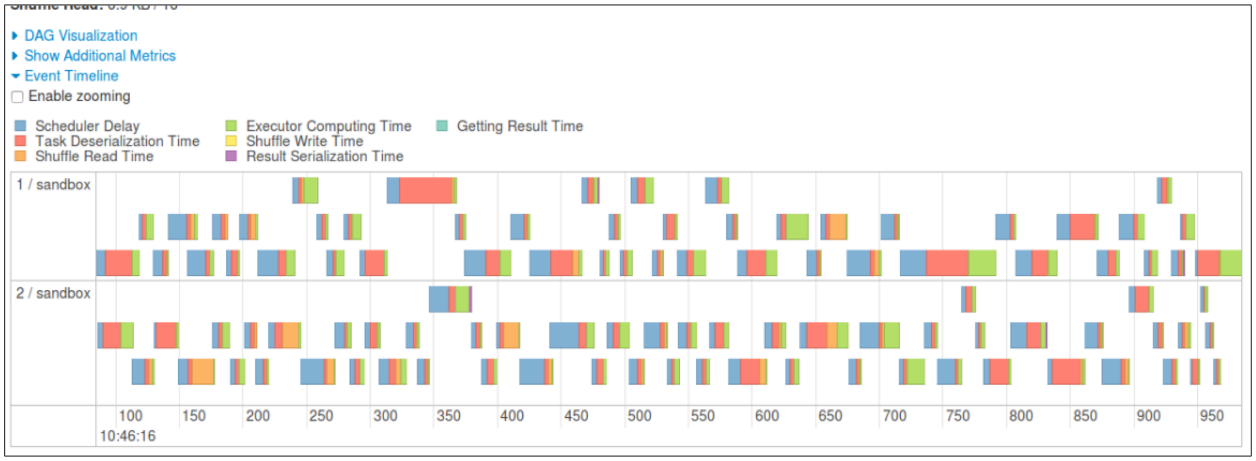
Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	sandbox:52087	2 s	100	0	100	3.7 KB / 14
2	sandbox:57010	2 s	100	0	100	3.2 KB / 2

Tasks

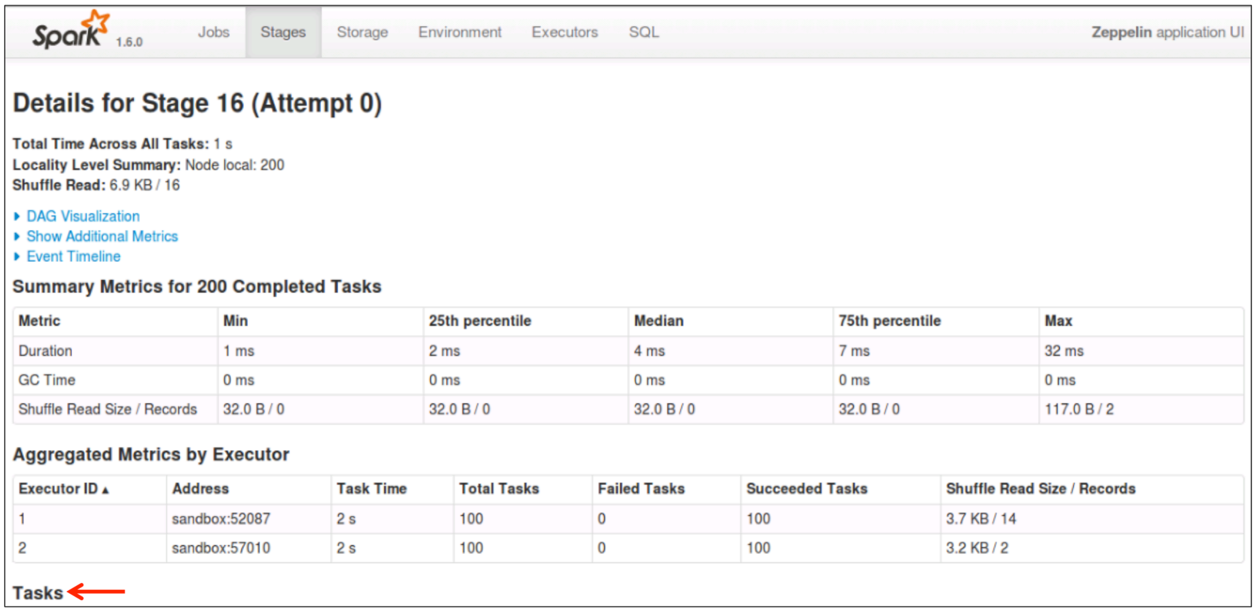
Stage Event Timeline Link

The **Stage Detail** page also provides an **Event Timeline visualization**, which breaks down tasks and types of tasks performed across the executors it utilized.



Stage Event Timeline

Stage Task List



Details for Stage 16 (Attempt 0)

Total Time Across All Tasks: 1 s
Locality Level Summary: Node local: 200
Shuffle Read: 6.9 KB / 16

- DAG Visualization
- Show Additional Metrics
- Event Timeline

Summary Metrics for 200 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	1 ms	2 ms	4 ms	7 ms	32 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Shuffle Read Size / Records	32.0 B / 0	32.0 B / 0	32.0 B / 0	32.0 B / 0	117.0 B / 2

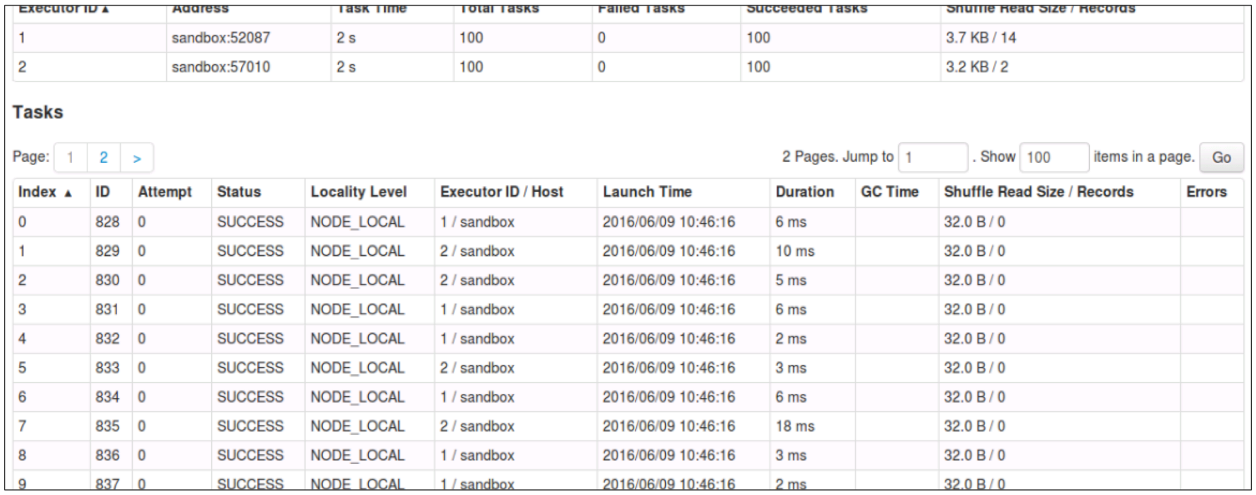
Aggregated Metrics by Executor

Executor ID ▴	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	sandbox:52087	2 s	100	0	100	3.7 KB / 14
2	sandbox:57010	2 s	100	0	100	3.2 KB / 2

Tasks

Task List

At the bottom of the Stage Detail page is a textual list of the tasks performed, as well as various information on them including the ID, status, executor and host, and duration.



Executor ID ▲	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Shuffle Read Size / Records
1	sandbox:52087	2 s	100	0	100	3.7 KB / 14
2	sandbox:57010	2 s	100	0	100	3.2 KB / 2

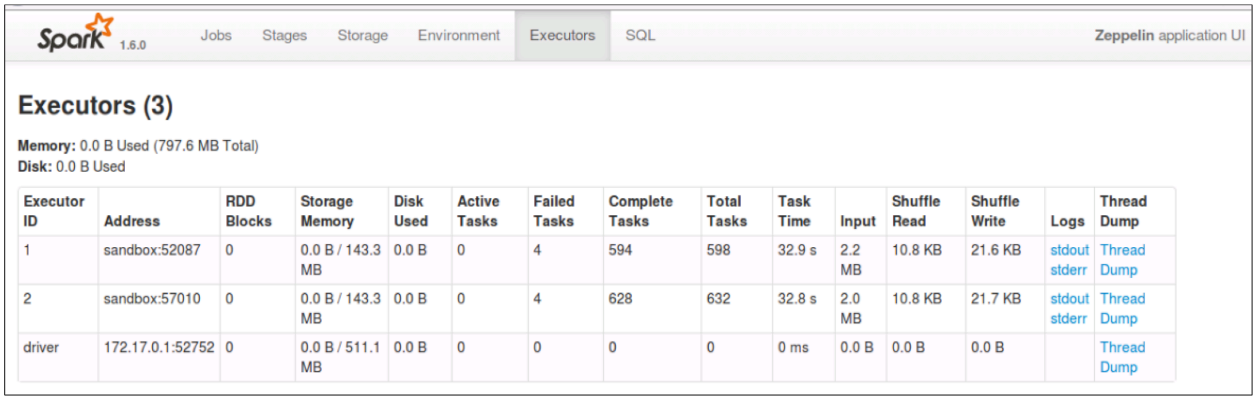
Tasks

Page: 1 2 > 2 Pages. Jump to 1 . Show 100 items in a page. Go

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Shuffle Read Size / Records	Errors
0	828	0	SUCCESS	NODE_LOCAL	1 / sandbox	2016/06/09 10:46:16	6 ms		32.0 B / 0	
1	829	0	SUCCESS	NODE_LOCAL	2 / sandbox	2016/06/09 10:46:16	10 ms		32.0 B / 0	
2	830	0	SUCCESS	NODE_LOCAL	2 / sandbox	2016/06/09 10:46:16	5 ms		32.0 B / 0	
3	831	0	SUCCESS	NODE_LOCAL	1 / sandbox	2016/06/09 10:46:16	6 ms		32.0 B / 0	
4	832	0	SUCCESS	NODE_LOCAL	1 / sandbox	2016/06/09 10:46:16	2 ms		32.0 B / 0	
5	833	0	SUCCESS	NODE_LOCAL	2 / sandbox	2016/06/09 10:46:16	3 ms		32.0 B / 0	
6	834	0	SUCCESS	NODE_LOCAL	1 / sandbox	2016/06/09 10:46:16	6 ms		32.0 B / 0	
7	835	0	SUCCESS	NODE_LOCAL	2 / sandbox	2016/06/09 10:46:16	18 ms		32.0 B / 0	
8	836	0	SUCCESS	NODE_LOCAL	1 / sandbox	2016/06/09 10:46:16	3 ms		32.0 B / 0	
9	837	0	SUCCESS	NODE_LOCAL	1 / sandbox	2016/06/09 10:46:16	2 ms		32.0 B / 0	

Stage Task Listing

Executor View



Executors (3)

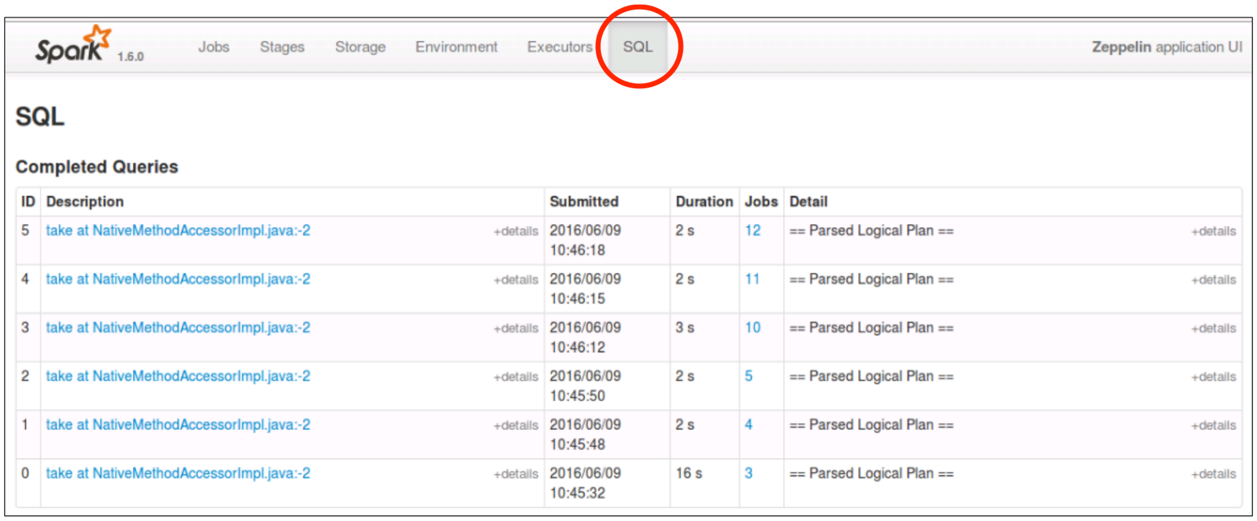
Memory: 0.0 B Used (797.6 MB Total)
Disk: 0.0 B Used

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

Spark Application UI Executor View

The last standard tab (visible regardless of what kind of jobs have been performed) is the **Executor tab.** This shows information about the executors that have been used across all Spark jobs run by this SparkContext instance, as well as providing links to logs and thread dumps, which can be used for troubleshooting purposes as well.

SQL View



SQL

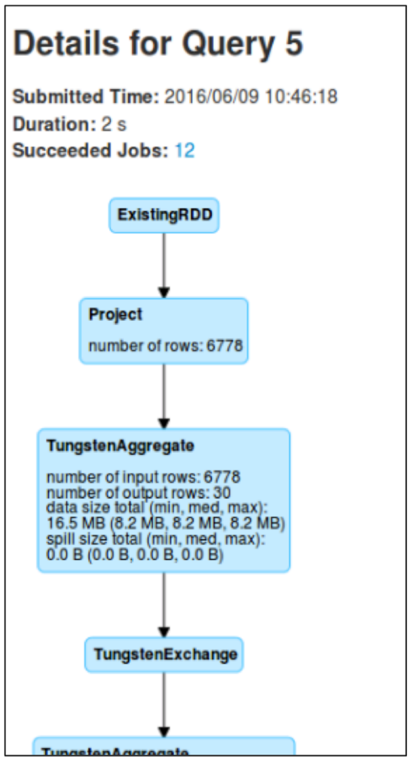
Completed Queries

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

Spark Application UI SQL View

When you run a Spark job that uses one of the Spark modules, another tab appears at the top of the window that provides module-specific types of information for those jobs. In this screenshot, we see that a Spark SQL job has been executed, and that a tab labeled SQL has appeared at the top-right. The information provided is in terms of queries rather than jobs (although the corresponding Spark job number is part of the information provided.)

SQL Visual Query Details


Details for Query 5

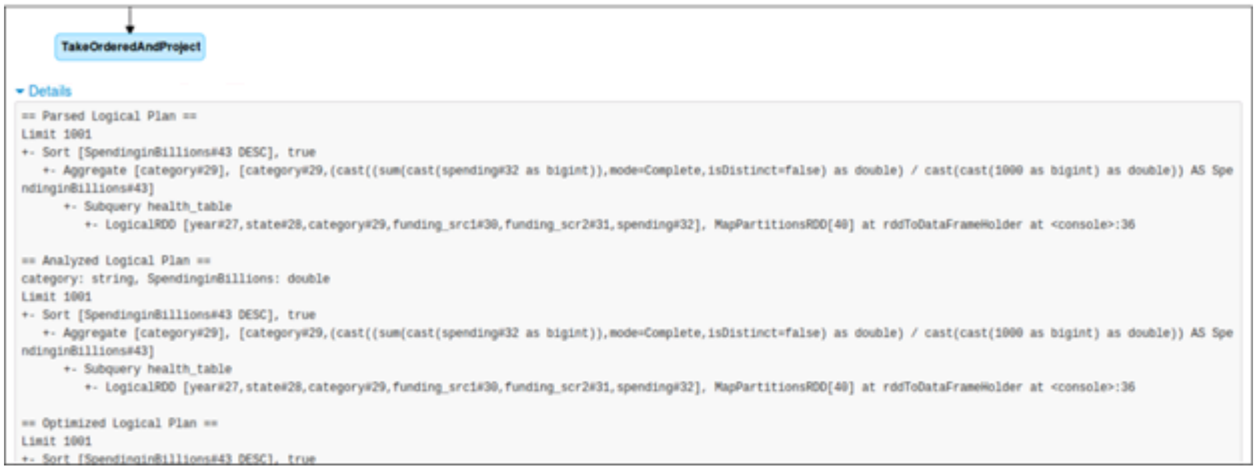
Submitted Time: 2016/06/09 10:46:18
Duration: 2 s
Succeeded Jobs: 12

ExistingRDD
Project
number of rows: 6778
TungstenAggregate
number of input rows: 6778
number of output rows: 30
data size total (min, med, max):
16.5 MB (8.2 MB, 8.2 MB, 8.2 MB)
spill size total (min, med, max):
0.0 B (0.0 B, 0.0 B, 0.0 B)
TungstenExchange
TungstenAggregate

SQL Query Details – Visual

Clicking on a query description will take you to a Details for Query "*X*" page, which will show a DAG of the operations performed as part of that query.

SQL Text Query Details

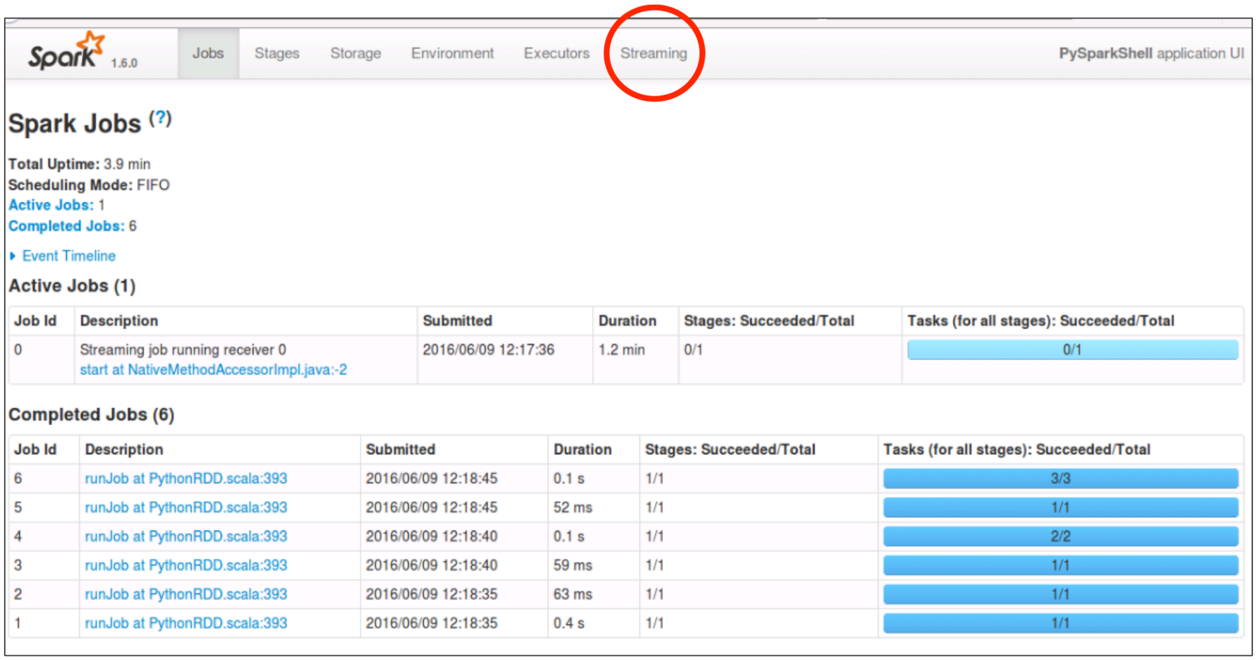


```
== Parsed Logical Plan ==
Limit 1001
+- Sort [SpendinginBillions#43 DESC], true
   +- Aggregate [category#29], [category#29,(cast((sum(cast(spending#32 as bigint)),mode=Complete,isDistinct=false) as double) / cast(cast(1000 as bigint) as double)) AS Spe
ndinginBillions#43]
      +- Subquery health_table
         +- 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
```

SQL Query Details – Text

Below the query DAG, there is a Details link which - when clicked - provides a text-based view of the details of the query.

Streaming Tab



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

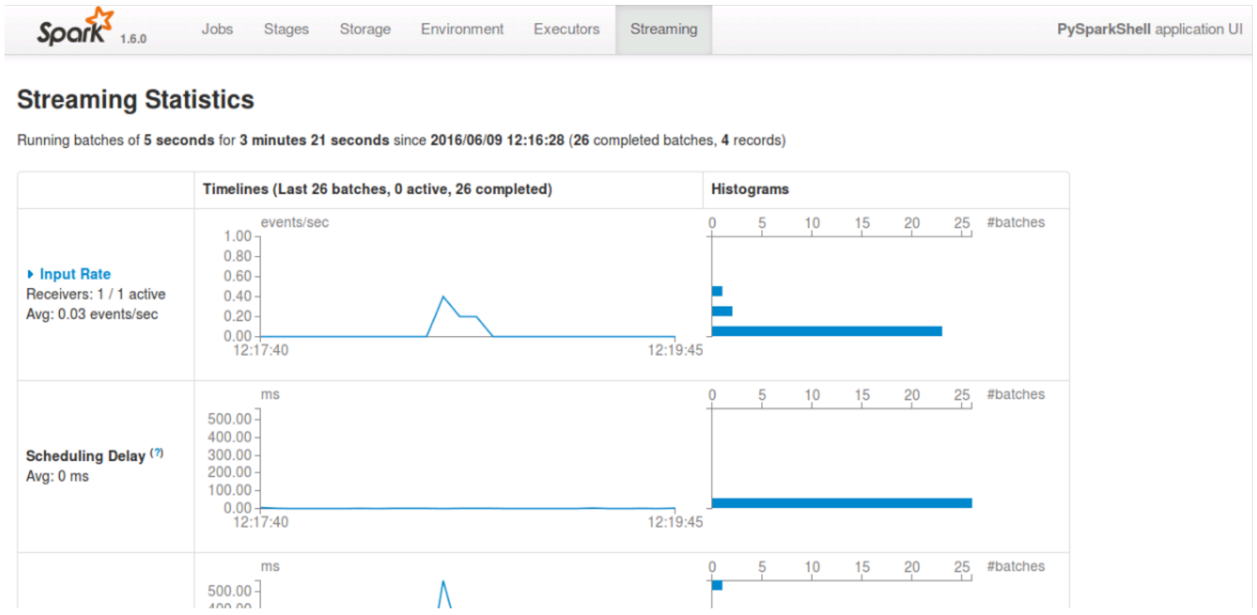
Completed Jobs (6)

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

Spark Application UI Streaming Tab

When running Spark Streaming jobs, a Streaming tab will appear in the Spark Application UI. In this view we can see details about each job Spark runs, equating to a DStream collection and processing.

Streaming View



Streaming Statistics

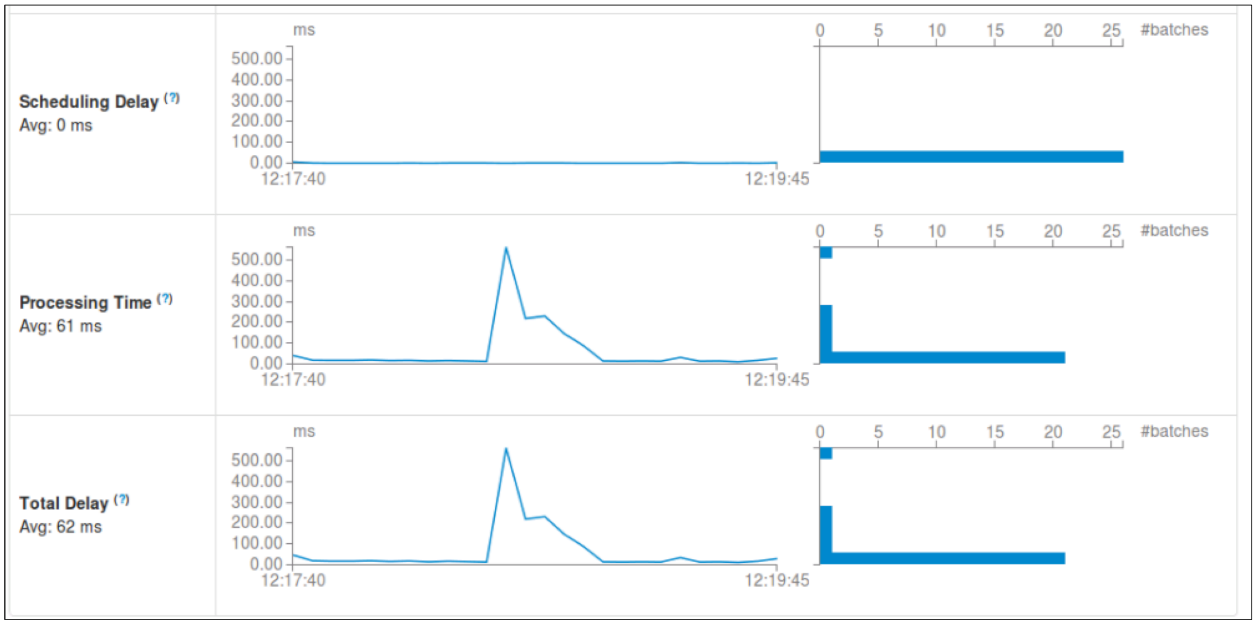
Running batches of **5 seconds** for **3 minutes 21 seconds** since **2016/06/09 12:16:28** (**26** completed batches, **4** records)

	Timelines (Last 26 batches, 0 active, 26 completed)	Histograms
Input Rate Receivers: 1 / 1 active Avg: 0.03 events/sec		
Scheduling Delay (?) Avg: 0 ms		

Streaming Job Statistics

Clicking on a streaming job description link results in a page that shows streaming statistics charts for a number of different metrics. Shown here are input rate and scheduling delay. Input Rate is a link that can be clicked to expand the metrics window and show the rate per receiver, if multiple receivers are in use. In our example, only one receiver was in use and active.

Additional Streaming Charts

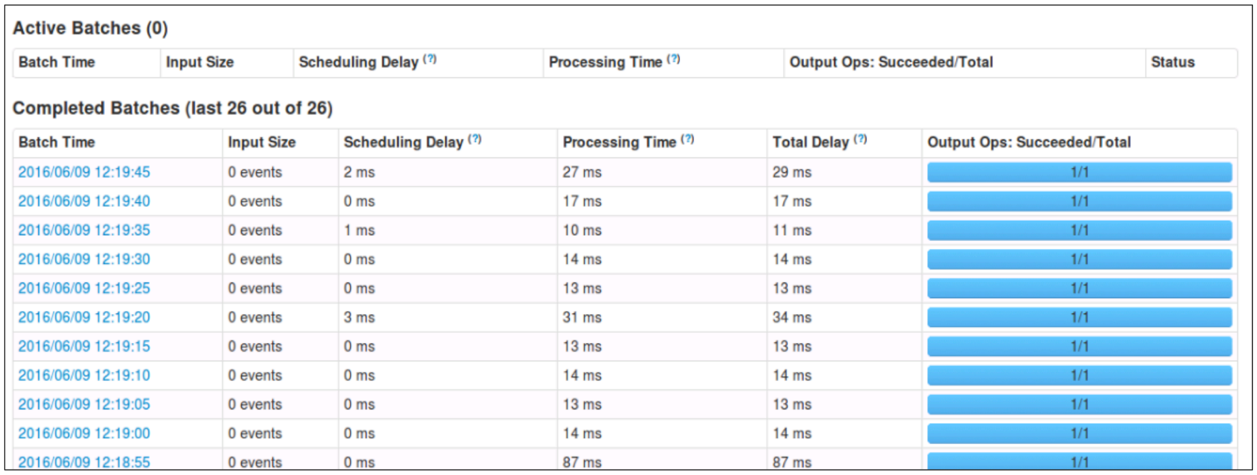


Additional Streaming Charts

Additional Spark Streaming charts include scheduling delay, processing time, and total delay.

When you need to understand what the cause of slowness might be, the charts on this page can be particularly useful when troubleshooting performance issues with a Spark Streaming job.

Streaming Batches



Active Batches (0)

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

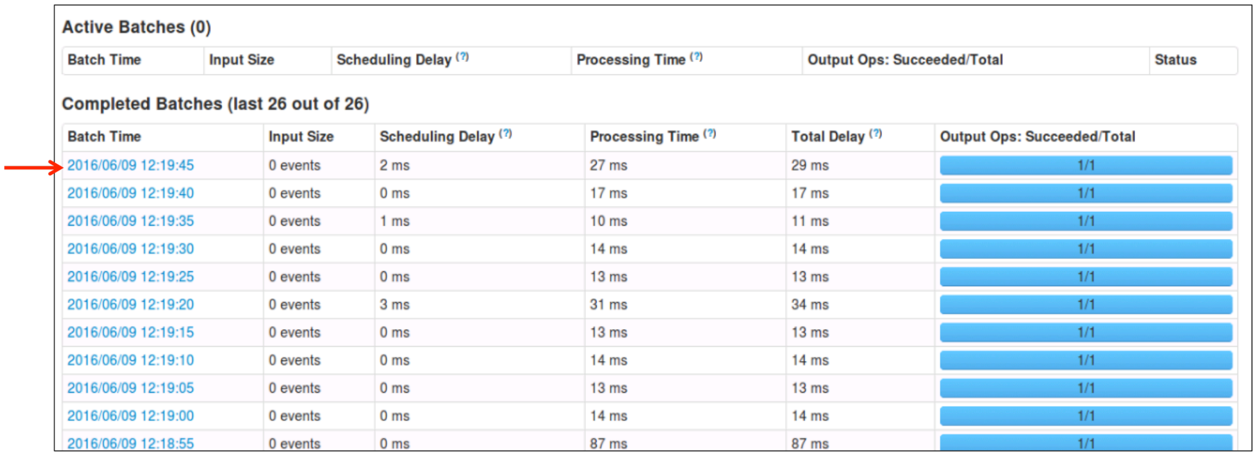
Completed Batches (last 26 out of 26)

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

Streaming Batch Statistics

Beneath the charts is a list of batches, with statistics available about each individual batch and whether the output operation was successful.

Batch Detail



Active Batches (0)

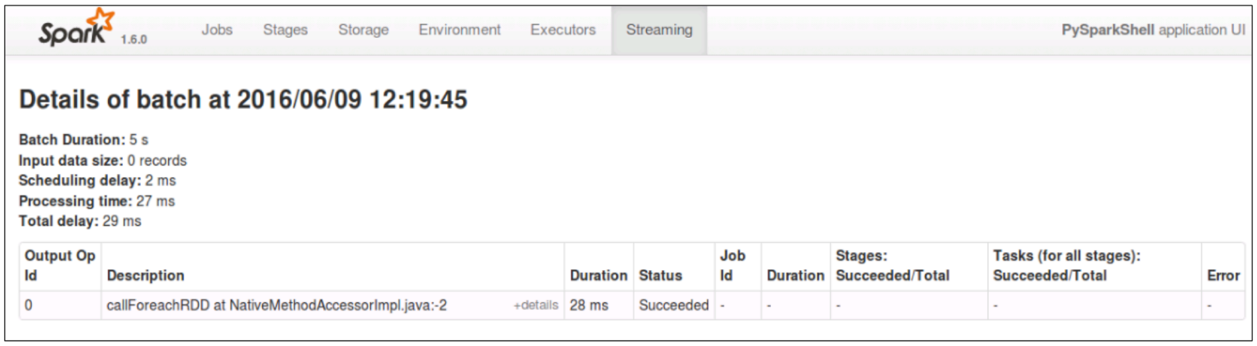
Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)	Output Ops: Succeeded/Total	Status

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

Batch Detail Link

Clicking on the batch time link results in a batch details page, where additional information may be found.



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	-	-	-	-	-

Streaming Batch Detail

Knowledge Check

You can use the following questions to assess your understanding of the concepts presented in this lesson.

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?

Answers

1) Spark jobs are divided into ____________, which are logical collections of ______________.

Answer: Stages, tasks

2) A job is defined as a set of tasks that culminates in a _______________.

Answer: Action

3) What Spark component organizes stages into logical groupings that allow for parallel execution?

Answer: DAG Scheduler

4) What is the default port used for the Spark Application UI?

Answer: 4040

5) If two SparkContext instances are running, what is the port used for the Spark Application UI of the second one?

Answer: 4041

6) As discussed in this lesson, what tabs in the Spark Application UI only appear if certain types of jobs are run?

Answer: SQL and Streaming

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.

Performance Tuning

Lesson Objectives

After completing this lesson, students should be able to:

- ✓ Explain why `mapPartitions` usually performs better than map
- ✓ Describe how to repartition RDDs and how this can improve performance
- ✓ Explain the different caching options available
- ✓ Describe how checkpointing can reduce recovery time in the event of loosing an executor
- ✓ Describe situations where broadcasting increases runtime efficiencies
- ✓ Detail the options available for configuring executors
- ✓ Explain the purpose and function of YARN

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

As a reminder, mapper transformations are narrow operations that benefit from partitions being operated on independently from each other.

The `mapPartitions` API is a special kind of map transformation that can be used when both the inputs and outputs are iterable. It operates at the RDD partition level. The `map` API, in contrast, operates at the element level. To see why that can matter, let's look at an example:

You want to initialize a database connection when mapping an RDD that contains 2,000,000 elements and is spread across four RDD partitions. If we use the `map` API to accomplish this, each element will require a database connection to be made which results in 2,000,000 initializations. However, if we use the `mapPartitions` API, each whole partition gets sent to the mapper's function at once, resulting in only four database connection initializations. Once initialized, the elements in the partition can then be iterated through and transformed. This reduction in the number of initializations that must take place can result in significantly improved performance.

As a simple (but not perfect, since this would be better done with a reducer) example of how this works, take a look at the following code:

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

The first line creates `rdd1` as an iterable list of numbers, and tells Spark to split this into two partitions. The next line uses `mapPartitions()` to sum the values in each partition and return the individual results. If these results needed to be further combined, the output could have been saved as a new RDD by changing the second line as follows:

```
rdd2 = rdd1.mapPartitions(lambda x: [sum(x)])
```

IMPORTANT:

The brackets [ ] around `sum(x)` are required in this example because the input *and* output of `mapPartitions()` must be iterable. Without the brackets to keep the individual partition values separate, the function would attempt to return a number rather than a list of results, and as such would fail. If the total sum was needed, you would need to perform an additional operation on rdd2 (from the modification to line 2 above) in order to compute it such as the operation below:

```
rdd1.mapPartitions(lambda x: [sum(x)]).reduce(lambda a,b: a+b)  # or
just rdd1.reduce(lambda a,b: a+b) in the first place if not trying
to explain mapPartitions
```

RDD Parallelism

The cornerstone of performance in Spark centers around the concepts of narrow and wide operations. How RDDs are partitioned, initially and via explicit changes, can make a significant impact on performance.

Inherent Parallelism – `parallelize()`

If a dataset has no parent, such as from an `sc.parallelize()` operation, then the total number of CPU cores across all the executor YARN containers for the full Spark application is used unless the value is less than two (the minimum number of partitions). The `spark.default.parallelism` property could be set (as described in http://spark.apache.org/docs/latest/configuration.html) for each application to override this.

Spark's goal of aligning the RDD's number of partitions to that of the cores available is based on the intention of making sure no resources are idle when an operation on the RDD is being performed.

For small datasets running in very resource-intensive Spark applications, this could mean partitions that are very shallow - or in other words, partitions that have a small number of elements, or in extreme cases, no elements. Depending on circumstances, this can either improve or hurt performance.

The parallelize operation can take an optional parameter to make the number of partitions larger or smaller than the default value for the application.

```
sc.defaultParallelism
4
rdd1 = sc.parallelize((1,2,3,4,5,6))
rdd1.getNumPartitions()
4
rdd1 = sc.parallelize((1,2,3,4,5,6),numSlices=8)
rdd1.getNumPartitions()
8
rdd1 = sc.parallelize((1,2,3,4,5,6),2)
rdd1.getNumPartitions()
2
```

Inherent Parallism – `textFile()`

Default parallelism aligns with the number of blocks the file(s) take up on the HDFS when using `textFile`. If the file is only one block in size, the RDD will be created with the minimum size of two partitions. This can be increased by providing a second optional argument declaring the preferred number of partitions. This number cannot be less than the number of HDFS blocks, but this will not surface and error. Instead, it will just be created with the number of partitions equal to the number of blocks.

This is primarily due to the fact that the HDFS default block size of 128MB is being easily accounted for as an RDD partition size. Furthermore, Spark can more quickly create the RDD by reading a single HDFS block and creating two, or more, RDD partitions from it than it can read from multiple HDFS partitions (usually on separate worker nodes) to create a single RDD partition.

Here is an example of creating an RDD from a one-block HDFS file.

```
rdd1 = sc.textFile("statePopulations.csv")
rdd1.getNumPartitions()
2
rdd1 = sc.textFile("statePopulations.csv", minPartitions=4)
rdd1.getNumPartitions()
4
rdd1 = sc.textFile("statePopulations.csv", 1)
rdd1.getNumPartitions()
1
```

Notice that it defaults to two partitions, but can be created with only one since the number of blocks used is also one.

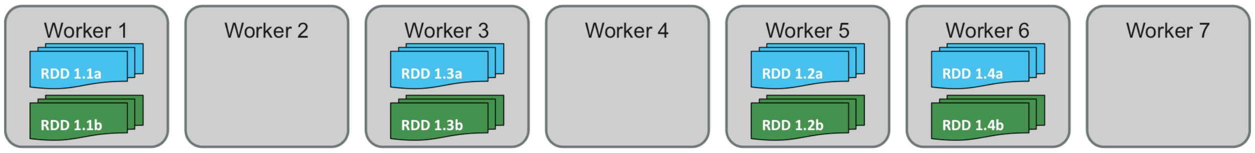
Here is an example of creating an RDD from a five-block HDFS file.

```
rdd2 = sc.textFile("/proto/2000.csv")
rdd2.getNumPartitions()
5
rdd2 = sc.textFile("/proto/2000.csv",10)
rdd2.getNumPartitions()
10
rdd2 = sc.textFile("/proto/2000.csv",minPartitions=2)
rdd2.getNumPartitions()
5
```

Notice that trying to reduce the number of partitions to a value smaller than the number of blocks does not produce an error but simply sets it to the minimum size based on the number of blocks.

As with parallelize, the goal is still to have enough RDD partitions to allow for all executors to be working during operations; especially narrow ones. The reverse can happen with a small HDFS file compared with parallelize. A small file will take up few partitions. Generally speaking, you should increase this number on smaller files to be more inline with the number of cores available.

Narrow Dependencies/Operations

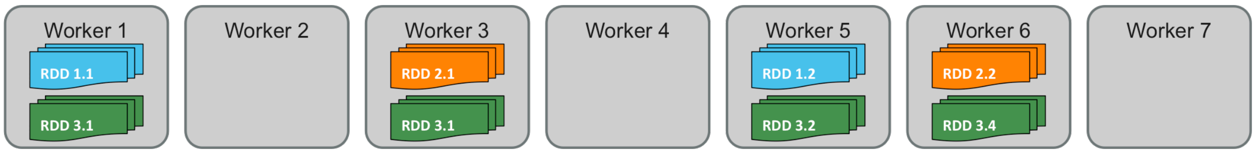


Narrow Dependencies/Operations with `map()`, `flatMap()` *or* `filter()`

Narrow operations can be executed locally and do not depend on any outside the current element. Examples of narrow operations are `map()`, `flatMap()`, `union()`, and `filter()`.

The picture above depicts examples of how narrow operations work. As visible in the picture above, there are no interdependencies between partitions.

Transformations maintain the partitioning of the largest parent RDD for the operation. For single parent RDD transformations, including `filter()`, `flatMap()`, and `map()`, the resulting RDD has the same number of partitions as the parent RDD.

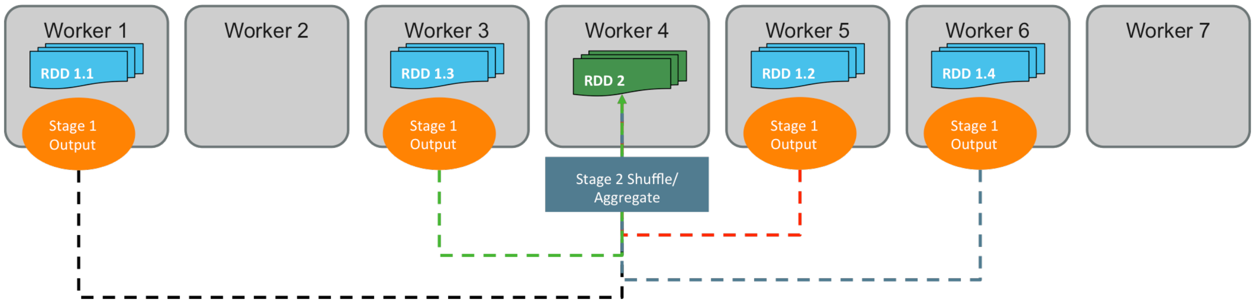


Narrow Dependencies/Operations with `union()`

For combining transformations such as `union()`, the number of resulting partitions will be equal to the total number of partitions from the parent RDDs.

Wide Dependencies/Operations

Wide operations occur when shuffling of data is required. Examples of wide operations are `reduceByKey()`, `groupByKey()`, `repartition()`, and `join()`.



Wide Dependencies/Operations

Above is an example of a wide operation. Notice that the child partitions are dependent on more than one parent partition. This should help explain why wide operations separate stages. The child RDD cannot exist completely unless all the data from the parent partitions have finished processing.

The above example shows the four RDD1 partitions reducing to a single RDD2 partition, but in reality multiple RDD2 partitions would have been generated, each one pulling a different subset of data from each of the RDD1 partitions. The diagram shows the logical combination rather than a physical result.

All shuffle-based operations' outputs use the number of partitions that are present in the parent with the largest number of partitions. In the previous diagram this would have resulted in RDD2 being spread across four partitions. Again, it was shown as a single partition to help visualize what is happening and to prevent an overly complicated diagram. The developer can specify the number of partitions the transformation will use, instead of defaulting to the larger parent. This is shown by passing a `numPartitions` as an optional parameter, as shown in the following two versions of the same operation.

```
reduceByKey(lambda c1,c2: c1+c2, numPartitions=4)
```

or simply

```
reduceByKey(lambda c1,c2: c1+c2, 4)
```

Controlling Parallelism

The following RDD transformations allow for partition-number changes: `distinct()`, `groupByKey()`, `reduceByKey()`, `aggregateByKey()`, `sortByKey()`, `join()`, `cogroup()`, `coalesce()`, and `repartition()`.

Generally speaking, the larger the number of partitions, the more parallelization the application can achieve. There are two operations for manually changing the partitions without using a shuffle-based transformation: `repartition()` and `coalesce()`.

A `repartition()` operation will shuffle the entire dataset across the network. A `coalesce()` just shuffles the partitions that need to be moved. Coalesce should only be used when reducing the number of partitions. Examples:

```
Use reparition() to change the number of partitions to 500:
rdd.repartition(500)

Use coalesce() to reduce the number of partitions to 20:
rdd.coalesce(20)
```

Changing Parallelism During a Transformation

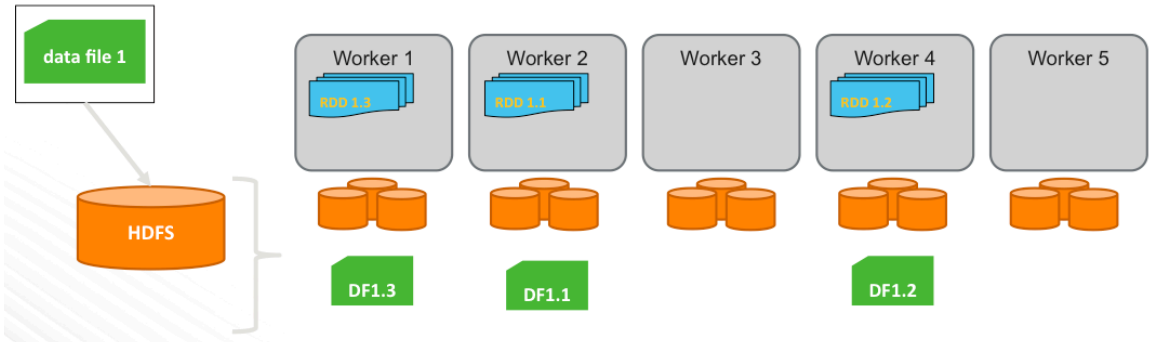
The code below represents a simple application that is summing up population counts by state.

```
sc.textFile("statePopulations.csv").map(lambda line:
line.split(",")).map(lambda rec: (rec[4],int(rec[5]))).reduceByKey(lambda
c1,c2: c1+c2, numPartitions=2).collect()
```

The following series of diagrams is an example of what would be going on with the RDD partitions of data.

In the first line of code, a file is read in.

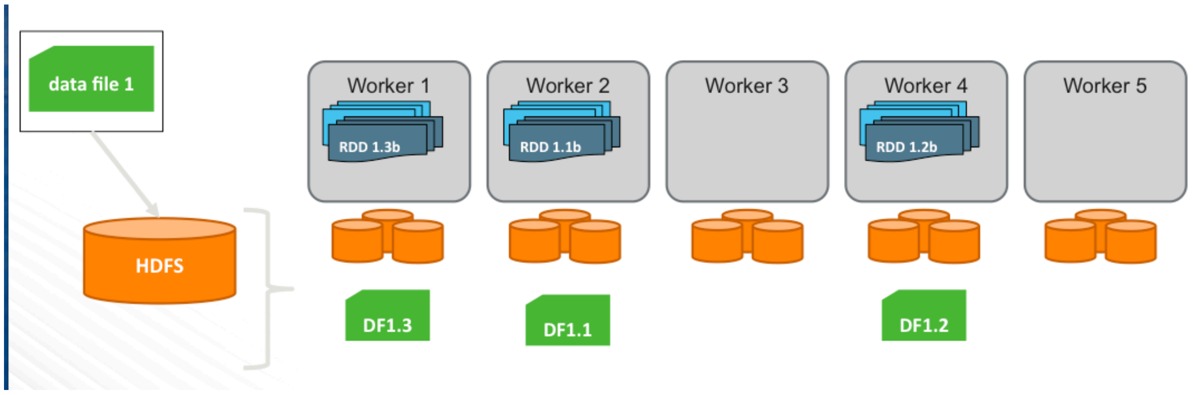
```
sc.textFile("statePopulations.csv")
```



The number of partitions defaults to the number of blocks the file takes up on the HDFS. Here, if the file takes up three blocks on HDFS it is represented by a three-partition RDD spread across three worker nodes.

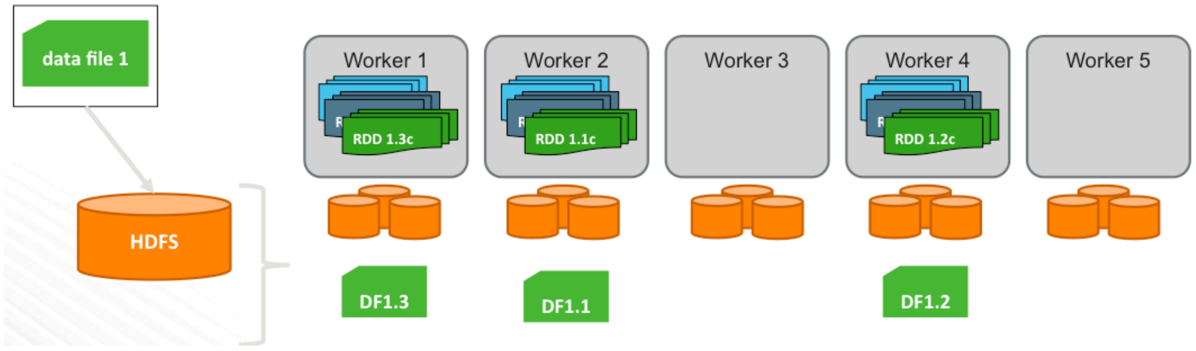
In the first map operation that is splitting the CSV record into attributes, no data needed to be referenced from another partition to perform the map transformation.

```
.map(lambda line: line.split(",")) \
```



The same is true for the next map that is creating a PairRDD for each row's particular state and population count.

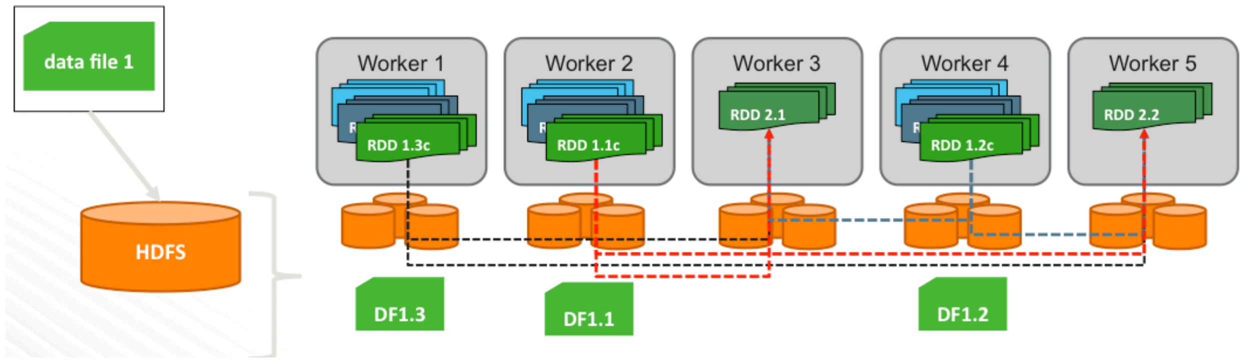
```
.map(lambda rec: (rec[4],int(rec[5])))
```



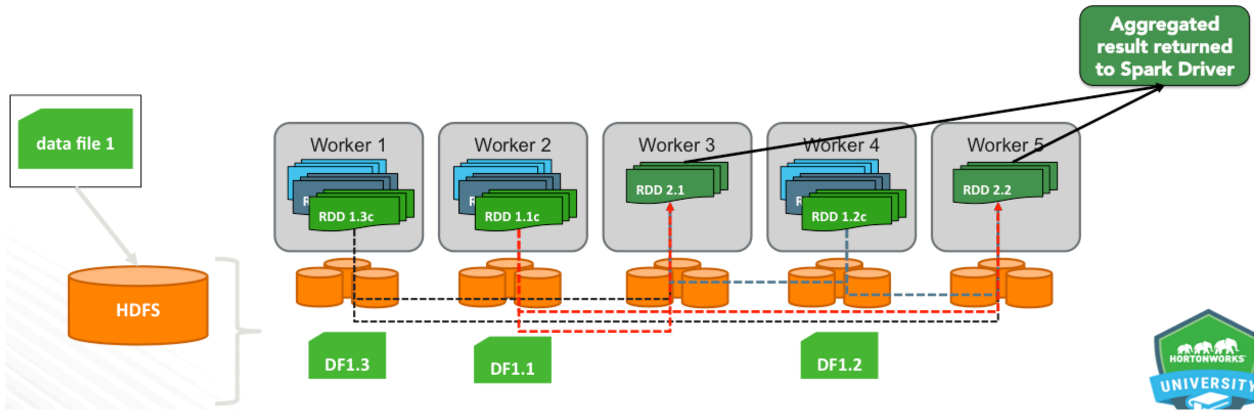
In the `reduceByKey` transformation that calculates final population totals for each state, there is an explicit reduction in the number of partitions.

The reduction of partitions is controlled by an optional `numPartitions` argument that `reduceByKey()` takes.

```
.reduceByKey(lambda c1,c2 : c1+c2, numPartitions=2)
```



Notice also, when doing a `reduceByKey()`, the same key may be present in multiple partitions that are output from the map operation. When this happens, the data is required to shuffle.

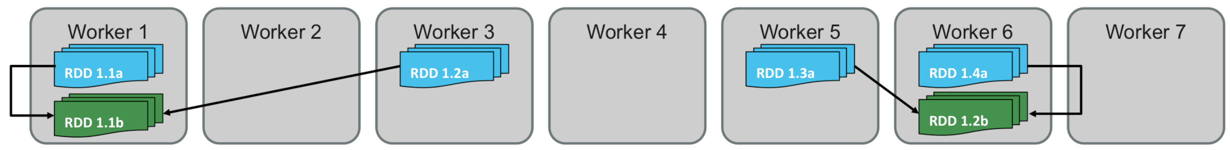


Finally, at the end, the `collect()` returns all the results from the two partitions in the `reduceByKey` operation to the driver.

Changing Parallelism without a Transformation

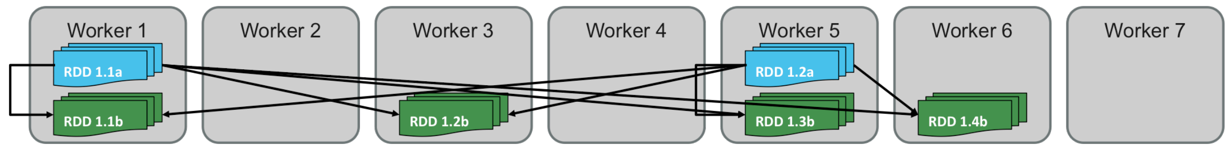
Changing the level of parallelism is a very common performance optimization task. These diagrams below illustrate the difference between `repartition()` and `coalesce()`.

Coalesce – Ex: From four partitions to two



Repartition – Ex: From two partitions to four

RDD size remains the same – only number of partitions changes



Whenever reducing the number of partitions, always use `coalesce()`, as it minimizes the amount of network shuffle. A `repartition()` is required if the developer is going to increase the number of partitions.

PairRDD Parallelism – Hashed Partitions

PairRDDs can be partitioned in such a way that their keys are leveraged when determining how to partition data. This means that a PairRDD could be created such that all pairs for a given key wind up in the same partition. When this is done, the resulting partitions are referred to as "hashed" partitions. Simply reading an HDFS file or calling `parallelize()` on a non-RDD dataset does not guarantee that partitions will be hashed partitions. The same is true for `map()` operations that create PairRDDs.

On the other hand, operations such as `partitionBy()`, `cogroup()`, `join()`, `reduceByKey()` and `sortByKey()` create hashed partitions by default. The `HashPartitioner` component (which must be explicitly imported and called when coding in Scala) guarantees identical keys go to the same partition. The following demo code explores this concept

```
rdd1 = sc.parallelize(range(3),1).map(lambda x:(x,'X'))
rdd1.getNumPartitions()
1
rdd1.collect()
[(0, 'X'), (1, 'X'), (2, 'X')]

rdd2 = sc.parallelize((1,2,3),1).map(lambda y:(y,'Y'))
rdd2.getNumPartitions()
1
rdd2.collect()
[(1, 'Y'), (2, 'Y'), (3, 'Y')]

rdd3 = rdd1.union(rdd2)
rdd3.getNumPartitions()
2
rdd3.collect()
[(0, 'X'), (1, 'X'), (2, 'X'), (1, 'Y'), (2, 'Y'), (3, 'Y')]

rdd3.glom().collect()
[[(0, 'X'), (1, 'X'), (2, 'X')], [(1, 'Y'), (2, 'Y'), (3, 'Y')]]
rdd3.glom().collect()  # make an array for each partition
[[(0, 'X'), (1, 'X'), (2, 'X')], [(1, 'Y'), (2, 'Y'), (3, 'Y')]]

rdd4 = rdd3.partitionBy(2)
rdd4.glom().collect()  # this one shows all of same key in same part
[[(0, 'X'), (2, 'X'), (2, 'Y')], [(1, 'X'), (1, 'Y'), (3, 'Y')]]
```

This concept allows many operations to skip shuffle steps knowing all the keys are in a single location, and thus increasing the performance. When partitioning data manually, by specifying number of partitions, be aware of how many executors are being used and try to have at least one partition per executor.

This can result in performance improvements, particularly when implementing joins.

Preserving Hashed Partitions

Once data is using hashed partitioning, maintaining this partitioning is important to help downstream operations run faster. Examples of operations that preserve partitioning are `mapValues()`, `flatMapValues()`, `filter()`, `reduceByKey()`, `groupByKey()`, and `join()`.

These operations don't modify keys in hashed partitions. Spark assumes the data is properly partitioned and uses this assumption to speed up downstream operations. For example, when using `reduceByKey()`, all the values for each key are aggregated. If the data is properly partitioned beforehand, Spark can assume that all the instances of a single key are in the same partition, on the same machine. With this knowledge, Spark can avoid doing a dataset wide shuffle, and instead all the aggregations can be performed locally.

Continuing the same hash partitioning demo from the previous section, the following example shows that the two partitions from rdd4 are still observed after performing a filter operation.

```
rdd4.glom().collect()
[[(0, 'X'), (2, 'X'), (2, 'Y')], [(1, 'X'), (1, 'Y'), (3, 'Y')]]

rdd4.filter(lambda kvp: kvp[0]>=2).glom().collect()
[[(2, 'X'), (2, 'Y')], [(3, 'Y')]]
```

Partitioning Optimization

There is no perfect formula, but the general rule of thumb is that too many partitions is better than too few. Since each dataset and each use case can be very different from all others, experimentation is required to find the optimum number of partitions for each situation.

The more partitions you have, the less time it will take to process each. Eventually, you will reach a point of diminishing returns. Spark schedules its own tasks within the executors it has available. This scheduling of tasks takes approximately 10-20ms in most situations. Spark can efficiently run tasks as fast as 200ms without any trouble. With experimentation, you could have tasks run for even less time than this, but tasks should take at least 100ms to execute to prevent the system from spending too much time scheduling instead of executing tasks.

A simple and novel approach to identifying the best number of partitions is to keep increasing the number by 50% until performance stops improving. Once that occurs, find the mid-point between the last two partition sizes and execute the application in production there. If anything changes regarding the executors (number available or sizing characteristics) then the tuning exercise should be executed again.

It is optimal to have the number of partitions be just slightly smaller than a multiple of the number of overall executor cores available. This is to ensure that if multiple waves of tasks must be run that all waves are fully utilizing the available resources. The slight reduction from an actual multiple is to account for Spark internal activities such as speculative execution (a process that looks for performance outliers and reruns potentially slow or hung tasks). For example, if there are 10 executors with two cores (20 cores total), make RDDs with 39, 58, or 78 partitions.

Generally speaking, this level of optimization is for programmers who work directly with RDDs. Spark SQL and it's DataFrame API were created to be a higher level of abstraction that lets the developer focus on **what** needs to be done as opposed to exactly **how** those steps should be executed. Spark SQL's Catalyst optimizer eliminates the need for developers to focus on this level of optimization in the code.

REFERENCE:

Spark SQL (as of 1.6.1) still has some outstanding unsupported Hive functionality that developers should be aware of. These are identified at `http://spark.apache.org/docs/latest/sql-programming-guide.html#unsupported-hive-functionality`. These shortcomings will likely be addressed in a future release.

Caching and Persisting

The first thing people think of when they hear Apache Spark is in-memory data. While this is technically accurate, data is only in-memory once you tell it to go to memory. This is why caching and persisting can improve performance. When these operations are used, they put data into memory. This makes applications that reuse this data much faster. Caching data should be used when an application is going to reuse the same dataset, like when doing an iterative application, or when creating multiple output files from the same input.

The concepts of "in-memory processing" and actual caching of datasets is often mixed together and can be confusing. "In-memory processing" applies to just about all programming models. Most languages/frameworks read data from a non-memory location and then bring the data into computer memory to perform processing. Conceptually, Spark is no different than any other similar platform.

The key difference for Spark is how it can perform a series of narrow tasks within a single stage. Since each partition can be acted upon independently from all other partitions in an RDD or DataFrame, Spark can perform each task on the data element(s) without having to write the data back to persistent storage (such as HDFS). This does not mean that Spark can always fit into memory completely during these transformations, and in those situations it can fall back to a local disk used by the executors as needed. This is still far faster than having to write this information to HDFS on completion of each task.

This concept actually works very well when applications are written so that no previous RDD is programmatically referenced again. For example, rddA > rddB > rddC > rddD. If a previously created RDD is referenced later -- for example, rddB is used to create a new rddE -- a consequence arises. In this case, Spark consults the underlying lineage/recipe, and then recreates rddA and rddB so it can create rddE. Caching rddB prior to creating rddE could increase overall performance.

A typical example might be an application where a "clean" file, reject file, and summary file are each created by processing the same original file. Caching the original file prior to performing the transformations would result in performance improvements.

Memory Representations and Limitations

The cache of an RDD is either stored in serialized or "raw" format. The raw format is the data just as it exists in memory and is the fastest to process. This comes at the cost of memory used - usually between two and ten times the amount of memory is needed compared to a more compact, serialized version of the cache. Serialization converts the memory object into a series of bytes, typically for the benefit of storing to a persistent store and/or to stream to another process.

Serialization can take up significantly less space than raw caching, but it has to go through the complimentary deserialization process to re-inflate objects back to their raw state for processing. This requires additional processing resources when the cached file is needed.

Serialization happens at the partition level rather than at the entire RDD level.

DataFrame objects act differently in regards to caching and are actually more efficient. DataFrames have a schema assigned to them and therefore can leverage an "in-memory columnar format" when caching the overall dataset. This approach ends up storing each column separately, which can aid performance by only looking at the columns needed for a given operation. Spark SQL also makes intelligent decisions on its own regarding compression and garbage collection. Because of this, it is recommended that `cache()` be leveraged to persist a DataFrame in memory.

There is a finite amount of memory allocated within an executor for caching of datasets. When an attempt is made to store more data in cache than physical memory allows, some type of control must be in place to help the system determine what to do. Spark fundamentally leverages a **Least Recently Used** (LRU) strategy to determine what to do. This system keeps track of when cached data was last accessed and evicts datasets from memory based on when they were last used.

If an operation tries to use cached data that for some reason was lost, the operation will attempt to use whatever is still in cache, but will recompute the lost data. As the data is recomputed, the data will be re-cached assuming space is available.

Caching Syntax

The functions for caching and persisting have the same names for RDDs and DataFrames.

- `persist()` - Developer can control caching storage level. Syntax: `persist(StorageLevel)`. Example: `persist(MEMORY_AND_DISK)`.
- `cache()` - Simple operation, equivalent to `persist(MEMORY_ONLY)`.
- `unpersist()` –remove data from cache.

To use caching, we must do a two things:

- The first is to import the library. Here is how to import the library for Scala and Python:

  Scala: `import org.apache.spark.storageLevel_`
  Python: `from pyspark import StorageLevel`

- Once the libraries are imported, we can then call the persist/cache operations.
  Here is an example of persisting our RDD "rdd"

  ```
  rdd.persist(StorageLevel)
  ```

The Spark SQL `SQLContext` object also features a `cacheTable(tableName)` method for any table that it knows by name and the complimentary `uncacheTable(tableName)` method. Additional helper methods `isCached(tableName)` and `clearCache()` are also provided.

NOTE:

In Python, cached objects will always be serialized, so it does not matter whether you choose serialization or not. When we talk about storage levels next, when using pyspark, `MEMORY_ONLY` is the same as `MEMORY_ONLY_SER`.

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

Use of the `persist` API is recommended for use with RDDs as it requires the developer to be completely aware of which "storage level" is best for a given dataset and its use case. The ultimate decision on which storage level to use is based on questions centered around serialization and disk usage.

First: should the cached data live in-memory or on disk. It may not sound like disk is a great choice, but remember that the RDD could be the result of multiple transformations that would be costly to reproduce if not cached. Additionally, the executor can very quickly get to data on its local disk when needed. This storage level is identified as `DISK_ONLY`.

If in-memory is a better choice, then the **second** question is whether raw or serialized caching (for Scala - as previously mentioned, Python automatically serializes cached objects) should be used.

Regardless of that answer, the **third** question to answer is should the cached data be rolled onto local disk if it gets evicted from memory or should it just be dropped? There still may be significant value from this data on local disk compared to having to recompute an RDD from the beginning.

Raw in-memory caching has an additional option to store the cached data for each partition in two different cluster nodes. This storage level allows for some additional levels of resiliency should an executor fail.

Spill over to disk if cache runs out of RAM?

- Yes
  - Serialize the data?
    - Yes – then use MEMORY_AND_DISK_SER
    - No
      - Keep two copies?
        - Yes – then use MEMORY_AND_DISK_2
        - No – then use MEMORY_AND_DISK
- No
  - Serialize the data?
    - Yes – then use MEMORY_ONLY_SER
    - No
      - Keep two copies?
        - Yes – then use MEMORY_ONLY_2
        - No – then use MEMORY_ONLY

NOTE:

There is also an experimental storage level identified as `OFF_HEAP` which is most similar to `MEMORY_ONLY_SER` except that, as the name suggests, the cache is stored off of the JVM heap.

Again, while there are many choices above which indicate some level of testing will be required to find the optimal storage level, the use of `persist()` with RDD caching and specifically identifying the best storage level possible is recommended. For Spark SQL, continue to use `cache(DataFrame)` or `cacheTable(table)` and let the Catalyst optimizer determine the best options.

Caching Example

Here is an example where an RDD is reused more than once:

```
from pyspark import StorageLevel
ordersRrdd = sc.textFiles("/orders/received/*")
ordersRdd.persist(StorageLevel.MEMORY_ONLY_SER)
ordersRdd.map(…).saveAsTextFile("/orders/reports/valid.txt")
ordersRdd.filter(…).saveAsTextFile(”/orders/reports/filtered.txt")
ordersRdd.unpersist()
```

Choosing a Storage Level

While there is no perfect rule of thumb to follow, the following are generally considered best practices for persisting data:

- If the RDD fits in memory, use the default `MEMORY_ONLY`, as it will be the fasted option for processing. (Again, in Python coding, serialization is automatic, therefore this setting is identical to `MEMORY_ONLY_SER`.)
- If RDDs don’t fit in memory, try `MEMORY_ONLY_SER` with a fast serialization library. Doing this uses more CPU, so use efficient serialization like Kryo (described later).
- If the RDDs don’t fit into memory even in serialized form, consider the time to compute this RDD from parent RDDs vs the time to load it from disk. In some cases, re-computing an RDD may sometimes be faster than reading it from disk. The best way to decide which one is better is to try them both and see what happens.

Another option is **replicated storage.** In replicated storage, the data is replicated on two nodes, instead of one. 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 the total memory of the system.

DataFrame objects are actually more efficient when left to their defaults due to Catalyst optimizations. Use `cache()` rather than `persist()` when working with DataFrames.

Serialization Options

For Scala

For JVM-based languages (Java and Scala), Spark attempts to strike a balance between convenience (allowing you to work with any Java type in your operations) and performance. It provides two serialization libraries: Java serialization (by default), and Kryo serialization.

Kryo serializing is significantly faster and more compact than the default Java serialization, often as much as 10 times faster. The reason Kryo wasn't set as default is because initially Kryo didn't support all serializable types and users would have had to register custom classes.

Since these issues have been addressed in recent versions of Spark. Kryo serialization should always be used.

For Python

In Python, applications use the Pickle library for serializing, unless you are working with DataFrames or tables, in which case the Catalyst optimizer manages serialization. The optimizer converts the code into Java byte code, which - if left unspecified - will use the Java default serialization. Thus, Python DataFrame applications will still need to specify the use of Kryo serialization.

Kryo Serialization

To implement Kryo Serialization in your application, include the following in your configuration:

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

This works for DataFrames as well as RDDs since the `SQLContext` (or `HiveContext`) are passed the `SparkContext` in their constructor method.

If using the pyspark or spark-shell REPLs, add `-conf spark.serializer=org.apache.spark.serializer.KryoSerializer` as a command-line argument to these executables.

Checkpointing

Since Spark was initially built for long-running, iterative applications, it keeps track of an RDD's recipe or lineage. This provides reliability and resilience, but as the number of transformations performed increases and the lineage grows, an application can run into problems. The lineage can become too big for the object allocated to hold everything. When the lineage gets too long, there is a possibility of a stack overflow.

Also, when a worker node dies, any intermediate data stored on the executor has to be re-computed. If 500 iterations were already performed, and part of the 500th iteration was lost, the application has to re-do all 500 iterations. That can take an incredibly long time, and can become inevitable given a long enough running application processing a large amount of data.

Spark provides a mechanism to mitigate these issues: **checkpointing.**

About Checkpointing

When checkpointing is enabled, it does two things: data checkpointing and metadata checkpointing. (Checkpointing is not yet available for Spark SQL).

Data checkpointing – Saves the generated RDDs to reliable storage. As we saw with Spark Streaming window transformations, this was a requirement for transformations that combined data across multiple batches. To avoid unbounded increases in recovery time (proportional to the dependency chain), intermediate RDDs of extended transformation chains can be periodically checkpointed to reliable storage (typically HDFS) to shorten the number of dependencies in the event of failure.

Metadata checkpointing – Saves the information defining the streaming computation to fault-tolerant storage like HDFS. This is used to recover from failure of the node running the driver of the streaming application.

Metadata includes:

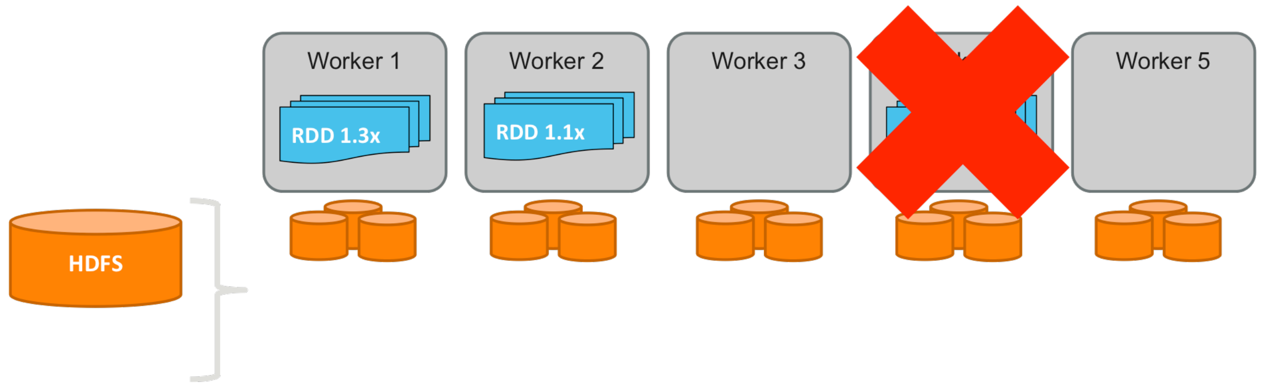
- Configuration - The configuration that was used to create the streaming application.
- DStream operations - The set of DStream operations that define the streaming application.
- Incomplete batches - Batches whose jobs are queued but have not completed yet.

When a checkpoint is initialized, the lineage tracker is "reset" to the point of the last checkpoint.

When enabling checkpointing, consider the following:

- Checkpointing is performed at the RDD level, not the application level
- Checkpointing is not supported in DataFrames or Spark SQL
- There is a performance expense incurred when pausing to write the checkpoint data, but this is usually overshadowed by the benefits in the event of failure
- Checkpointed data is not automatically deleted from the HDFS. The user needs to manually clean up the directory when they're positive that data will no longer be required.

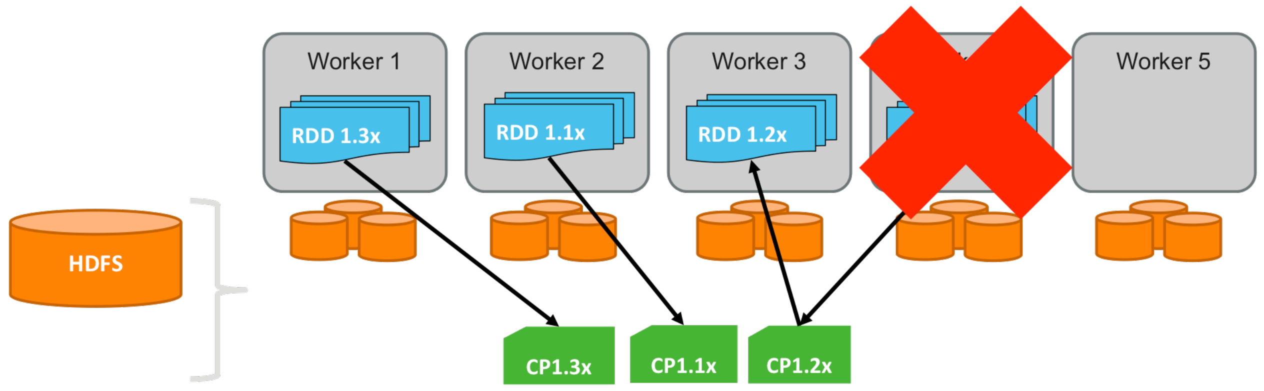
Node Loss Without Checkpointing



Without Checkpointing, all processes must be repeated – potentially thousands of transformations – if a node is lost

This is a typical application that iterates and has a lineage of "*n*" number of RDDs. There is no checkpointing enabled. If an in-use node fails during processing, all processing steps up to that point must be repeated from the beginning. This might be hundreds, or even thousands of transformations, which can result in significant time lost to reprocessing.

Node Loss With Checkpointing



With Checkpointing, only processes performed since the last checkpoint must be repeated

In this example, we have the same application with checkpointing enabled. We can see that every nth iteration, data is being permanently stored to the HDFS. This may not seem intuitive at first as one might ask why we should save data to the HDFS when it is not needed. In the case that a worker node goes down, instead of trying to redo all previous transformations (which again, can number in the thousands), the data can be retrieved from HDFS and then processing can continue from the point of the last checkpoint.

This example shows that checkpointing can be viewed as a sort of insurance for events such as this. Instead of simply hoping a long-running application will finish without any worker failures, the developer makes a bit a performance tradeoff up front in choosing to pause and write data to HDFS from time to time.

Checkpointing vs. Caching

It is very common for developers to become confused with checkpointing and caching when first learning about the two concepts. While they initially sound like they do similar things, that is not the case.

Checkpointing saves a permanent copy of intermediate data to HDFS. Lineage is updated, so the current RDD only has to go as far back as the checkpointed data in case of worker failure.

Caching saves a temporary copy of the data to local disk or RAM. Lineage is preserved from the base data, and thus caching has no effect on recomputation. When an application uses data from a cached RDD, it is aware the data is cached and attempts to use the data cached first. If the data cached is lost for some reason, the data will then be recomputed from base data.

When to Use Checkpointing

Checkpointing is typically used in applications doing more than a few iterations and almost all real-world Spark Streaming applications. It is also frequently used in machine learning (MLlib) applications.

Implementing Checkpointing

To implement checkpointing, the developer must specify a location for the checkpoint directory before using the checkpoint function. Here is an example:

```
sc.setCheckpointDir("hdfs://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")
```

This code generates a checkpoint every fifth iteration of the RDD operation.

Broadcast Variables

A broadcast variable is a read-only variable cached once in each executor that can be shared among tasks. It cannot be modified by the executor. The ideal use case is something more substantial than a very small list or map, but also not something that could be considered “Big Data”.

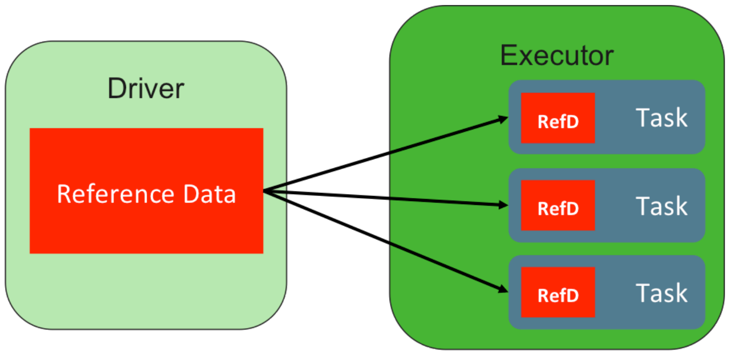
Broadcast variables are implemented as wrappers around collections of simple data types. They are not intended to wrap around other distributed data structures such as RDDs and DataFrames.

The goal of broadcast variables is to increase performance by not copying a local dataset to each task that needs it and leveraging a broadcast version of it. This is not a transparent operation in the codebase - the developer has to specifically leverage the broadcast variable name.

Spark uses concepts from P2P torrenting to efficiently distribute broadcast variables to the nodes and minimize communication cost. Once a broadcast variable is written to a single executor, that executor can send the broadcast variable to other executors. This concept reduces the load on the machine running the driver and allows the executors (aka the peers in the P2P model) to share the burden of broadcasting the data.

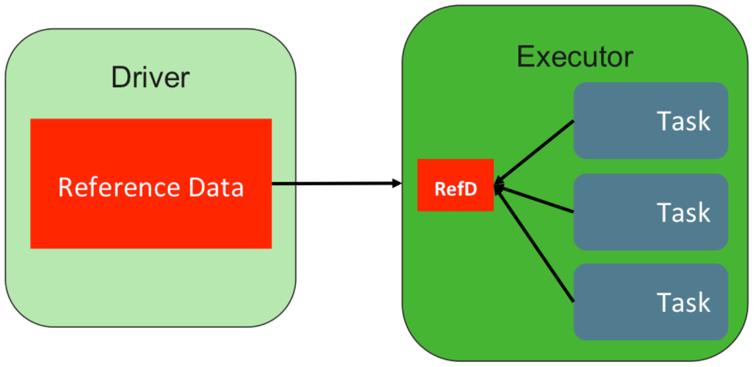
Broadcast variables are lazy and will not receive the broadcast data until needed. The first time a broadcast variable is read, the node will retrieve and store the data in case it is needed again. Thus, broadcast variables get sent to each node only once.

How do Broadcast Variables Work?



Without Broadcast Variables Reference Data is Sent to every Task

Without broadcast variables, reference data (such as lookup tables, lists, or other variables) is sent to every task on the executor, even though multiple tasks reuse the same variables. This what an application does normally.



Using Broadcast Variables, Spark sends Reference Data to the Node only Once

Using broadcast variables, Spark sends a copy to the node once and the data is stored in memory. Each task will reference the local copy of the data. These broadcast variables get stored in the executor memory overhead portion of the executor.

Why Use Broadcast Variables?

There are several reasons to use broadcast variables. The most obvious reason is to minimize network traffic by moving the dataset only once between executor worker processes instead of each time a task is run. As a general rule, always share/broadcast utilized local variables if they are at least 20K in size. Sizes smaller than this can be useful, but might require additional testing to determine how much performance increase can be expected.

An example of minimizing network traffic: Assume each task requires a small lookup table that is some number of megabytes in size. Spark can have thousands of tasks operating over hundreds of executors. Instead of sending a couple of megabytes over the network thousands of times, by broadcasting the small lookup table, the application will only have to send it 10-100 times, depending on the number of executors. Additionally, sending this data to the executors will be done in a P2P model as described earlier instead of all from the driver to each executor.

Spark's task scheduling system is optimized for running small tasks. The movement of data at the beginning of each task does not align well with the desired runtime behavior. It puts additional overhead on the task, which then takes more time to complete. Reducing task-launching overhead can speed up an application greatly.

It should be noted that broadcast variables are primarily used in RDD operations. Spark SQL has its own optimizations that provide some of the benefits being obtained in the more granular concept of broadcast variables.

Implementing Broadcast Variables

It is useful to see a code example of using broadcast variables, as well as how a particular problem could be solved with, and without, using them. We can then compare to verify that the same final results can be obtained.

For this example, the use case could be to break all of the words from a story into an RDD as shown by these two lines of code.

```
story = sc.parallelize(["I like a house", "I live in the house"])
words = story.flatMap(lambda line: line.split(" "))
```

This example is just creating a small two-line story and in practice the "story" RDD would be more likely created by using `sc.textFile()` to read a large HDFS dataset. The rest of the use case would be to strip out any words that were considered irrelevant (aka "noise") to further processing steps. In the code statements below, we are simply building a new RDD of elements that are not in the irrelevant word list.

```
noise = (["the","an","a","in"])
words.filter(lambda w: w not in noise).collect()
['I', 'like', 'house', 'I', 'live', 'house']
```

Next, we can elevate the irrelevant word list to a broadcast variable to distribute it to all workers and then simply reference it with the new variable name to obtain the same results.

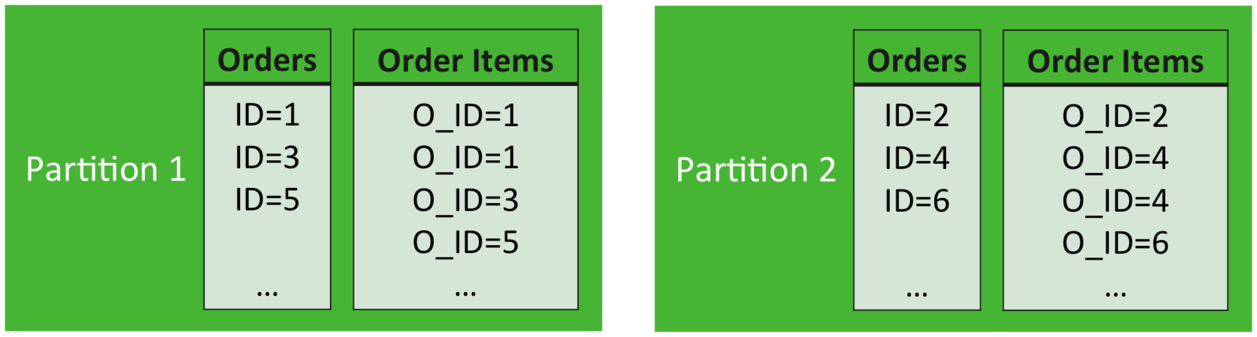
```
bcastNoise = sc.broadcast(noise)
words.filter(lambda w: w not in bcastNoise.value).collect()
['I', 'like', 'house', 'I', 'live', 'house']
```

Notice that the results are the same. While in this small illustrative example we would not show any performance gain, we would see improvements if the input "story" was of significant size and the irrelevant filter data itself was much bigger. Additionally, more savings would surface if the broadcast variable was used over again in subsequent stages.

Joining Strategies

While joins can happen on more than two datasets, this discussion will illustrate the use case of only two datasets which can be extrapolated upon when thinking of more than two datasets being joined. Additionally, the concepts discussed (unless otherwise called out) relate to RDD and DataFrame processing even though the illustrations will often reference RDD as the DataFrame will use the underlying RDD for these activities.

Spark Joins at the Partition Level

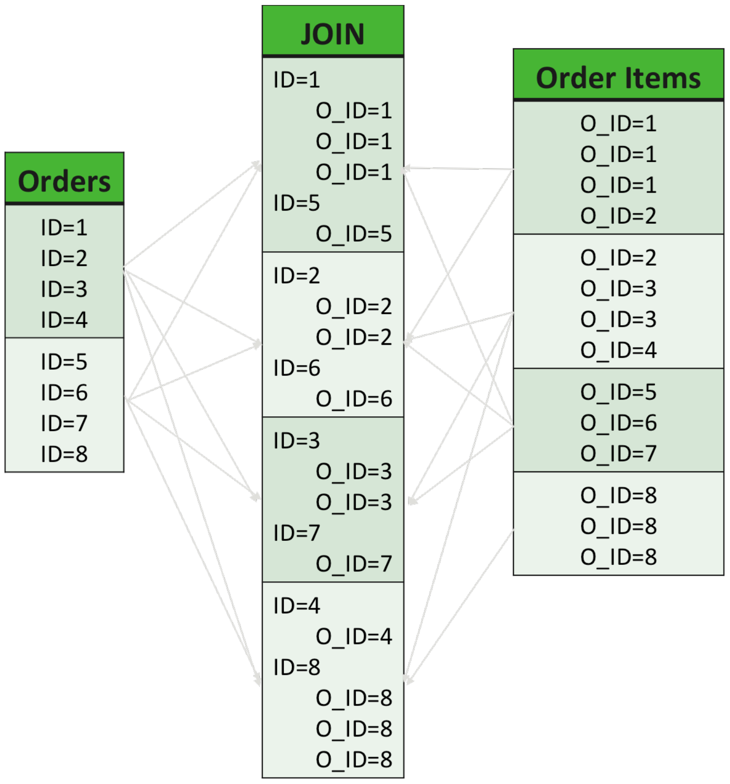


Joins are performed at the partition level

Spark performs joins at the partition level. It ensures that each partition from the datasets being joined are guaranteed to align with each other. That means that the join key will always be in the same numbered partition from each dataset being joined.

For that to happen, both datasets need to have the same number of partitions and have used the same hashing algorithm (described as "Hashed Partitions" earlier in this module) against the same join key before it can start the actual join processing.

Worst Case – No Common Hash Partitioning



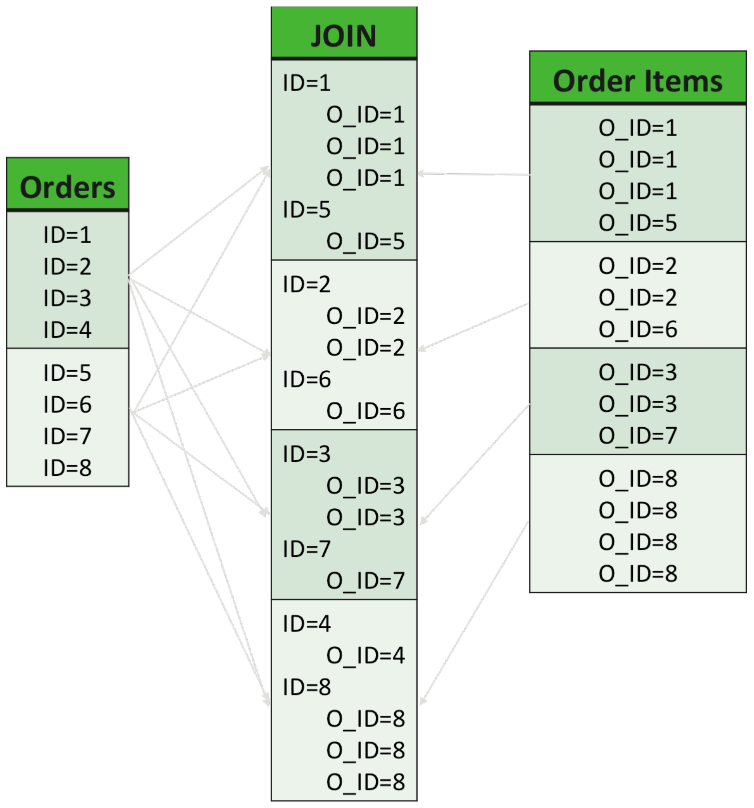
No Common Hash Partitioning

To help explain the intersection of joins and hash partitions, let's look at the worst case situation. We have two datasets that have different numbers of partitions (two on the left, four on the right) and which were not partitioned with the same key. Both of the datasets will require a shuffle to occur so that the equal number of hashed partitions will be created on the join key prior to the join operation being executed.

NOTE:

The number of hashed partitions created for the JOIN RDD was equal to the number of partitions from the dataset with the largest number of partitions.

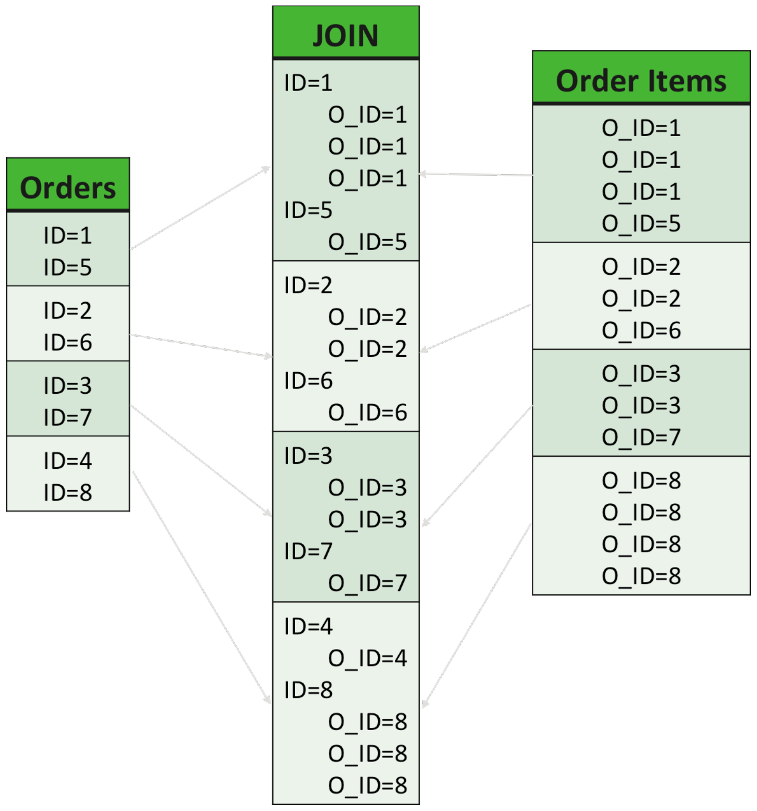
Better Case – One Dataset Hashed Partitioned



One Dataset Hashed Partitioned

A better scenario would occur if the larger dataset was already hashed partitioned on the join key. In this situation, only the second dataset would need to be shuffled. As before, more partitions would be created in the newly created dataset.

Best Case – Co-Partitioned



Co-Partitioned Datasets

The best situation would occur when the joining datasets already have the same number of hashed partitions using the same join key. In this situation, no additional shuffling would need to happen. This is called a ***co-partition join*** and is classified as a narrow operation.

Most likely, the aligned partitions will not be always on the same executor and thus one of them will need to be moved to the executor where its counterpart is located. While there is some expense to this, it is far less costly than requiring either of the joined datasets to perform a full shuffle operation.

Consideration and Guidelines

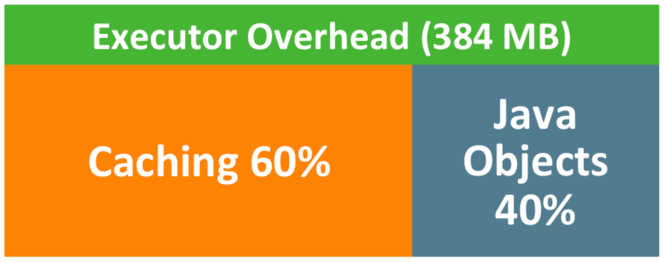
While the scenarios just described are representative of both RDD and DataFrame processing, core RDD programming needs to consider the order of operations more than with DataFrame programming since the Catalyst SQL optimizer, unlike the DAG optimizer, can reorder operations to improve performance.

When joining a dataset that will end up with a significant portion of its data removed from the final results, consider filtering the dataset first to be as close as possible a candidate set for the targeted join. An example would be if you join all flight data for many years to a dataset that contains data only from a single year from a single carrier, it would be beneficial to create a new dataset from the complete flight history that only contains the given year and carrier information.

And – If an explicitly built and referenced hash partitioned dataset will be used in subsequent joins, it is advisable to cache it. Otherwise, the transformation dataset will need to be recreated for each additional join, thus consuming additional processing time and resources.

Executor Optimization

Executors are highly configurable and are the first place to start when doing optimizations.



Executor Regions

Executors are broken into memory regions. The first is the overhead of the executor, which is almost always 384MB. The second region is reserved for creating Java objects, which makes up 40% of the executor. The third and final region is reserved for caching data, which makes up the other 60%. While these percentages are configurable, it is recommended that initial tuning occur in the overall configuration of an executor as well as the multiple of how many executors are needed.

When submitting an application, we tell the context how many and what size resources to request. To set these at runtime, we use the three following flags:

`--executor-memory`

This property defines how much memory will be allocated to a particular YARN container that will run a Spark executor.

`--executor-cores`

This property defines how many CPU cores will be allocated to a particular YARN container that will run a Spark executor.

`--num-executors`

This property defines how many YARN containers are being requested to run Spark executors within.

Configuring Executors

Deciding how many and how much resources for the executors can be difficult. Heres a good starting point.

- `executor-memory`
  - Should be between 8GB and 32GB.
  - 64GB would be a strong upper limit as executors with too much memory often are troubled with long JVM garbage collection processing times.
- `executor-cores`
  - At least two, but a max of four should be configured without performing tests to validate the additional cores are an overall advantage considering all other properties.
- `num-executors`
  - This is the most flexible as it is the multiple of the combination of memory and cores that make up an individual executor.
  - If caching data, it is desirable to have, at least, twice the dataset size as the total executor memory.

Many variables come into play including the size of the YARN cluster nodes that will be hosting the executors. A good starting point would be 16GB and two cores as almost all modern Hadoop cluster configurations would support YARN containers of this size.

If data set is 100GB, it would be ideal to have 100GB*2/(16GB) executors, which is 12.5. For this application, choosing 12 or 13 could be ideal.

This section presents the primary configuration switches available. Fine-tuning final answers will be best derived from direct testing results.

Knowledge Check

You can use the following question to assess your understanding of the concepts presented in this lesson.

Questions

1) By default, `parallelize()` creates a number of RDD partitions based on the number of __________________.

2) By default, `textFile()` creates a number of RDD partitions based on the number of _____________________.

3) Operations that require shuffles are also known as ___________ operations.

4) Which function should I use to reduce the number of partitions in an RDD without any data changes?

5) When all identical keys are shuffled to the same partition, this is called a _______________ partition.

6) True or False: DataFrames are structured objects, therefore a developer must work must work harder to optimize them than when working directly with RDDs.

Answers

1) By default, parallelize() creates a number of RDD partitions based on the number of __________________.

Answer: Executor CPU cores available

2) By default, textFile() creates a number of RDD partitions based on the number of ____________________.

Answer: HDFS blocks

3) Operations that require shuffles are also known as __________ operations.

Answer: Wide

4) Which function should I use to reduce the number of partitions in an RDD without any data changes?

Answer: coalesce()

5) When all identical keys are shuffled to the same partition, this is called a ______________ partition.

Answer: Hash (or hashed)

6) True or False: DataFrames are structured objects, therefore a developer must work must work harder to optimize them than when working directly with RDDs.

Answer: False. The Catalyst optimizer does this work for you when working with DataFrames.

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

Build and Submit Spark Applications

Lesson 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

Creating an Application to Submit to a Cluster

Zeppelin and REPLs are great tools for interactive analysis, as well as prototyping and testing programming logic before committing to production.

Spark Applications are the next stepping-stone. Spark applications run as standalone applications that can be integrated into workflows and scheduled with tools such as Falcon and Oozie. Fortunately, the differences between a Spark Application and running a job in Spark are minimal, which makes it easy to take code developed in Zeppelin and port it into an independent application with few to no modifications.

Writing an Application to Submit to YARN

When Zeppelin or a REPL is started, they do several things for the developer behind the scenes, including:

- Importing the `SparkContext` and `SparkConf` libraries,
- setting up a `main` program,
- creating an appropriate configuration object, and then
- creating and starting a `SparkContext` instance.

The developer has to do these things in a standalone application.

The difference between what a developer does in Zeppelin or a REPL and what a developer must do in a standalone application can be as simple as adding about five lines of code.

Importing Libraries

The first part a developer must do is import the `SparkContext` and `SparkConf` libraries. In addition, they will need to import all the other libraries they want to use in the application - for example, the `SQLContext` libraries. Doing this looks like any other application. Here is an example of importing some important libraries in Python.

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

To import other Spark libraries, it's the same as with any other application. Here is an example of importing more Spark libraries that are related to Dataframe processing:

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

Developers should be familiar with this concept. Applications are built with dependencies all the time, and Spark is no exception.

Creating a "main" Program

Zeppelin and the REPLs automatically set up the `main` program. Here is an example of how to set it up for a standalone application:

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

Again, this should look exactly the same as in any other application. There is some main part of the application that gets executed, and the same goes with Spark.

Creating a Spark Configuration

The next thing the developer must do is create the `SparkConf` configuration object. This configuration will tell the context some very important information about the application, like the resource manager, application name, amount of resources to request, etc. The developer can set configurations in multiple ways. Here is an example of creating the configuration.

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

The `conf.set('configuration','value')` lets the developer set any number of configurations. It is very common to have several of these in the application.

Creating the SparkContext

After creating the Spark Configuration, the developer must create the `SparkContext`. The `SparkContext` will communicate to the cluster, schedule tasks, and requests resources, among other things. Once the `SparkContext` is created, the application will begin interacting with other Hadoop resources such as YARN's Resource Manager. After creating the `SparkContext`, we now have an application that is ready to process distributed data in a parallel manner.

The `SparkContext` has some configurations that can be set as well. This example sets the log level to `ERROR`:

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

Check the API documentation for information about all the setters that can be used on the `SparkContext` after it has been created.

At the end of the application, the developer should stop the context. Failing to do so can leave some ghost processes running which will hold on to resources and may be very difficult to find and kill. Spark will not throw an error if the `stop()` method is not utilized, but the best practice is to use it.

Below is an example of creating the `SparkContext`, setting a configuration, and stopping the `SparkContext`. While not including the `sc.stop()` may have no impact on the developer, the overall cluster experience will diminish as resources will still be allocated and administrators will start having to manually kill these processes. This process is not trivial as identifying which processes are related to this problem amongst all YARN processing running is difficult. Developers should be cognizant of this multi-tenant nature of most Hadoop clusters and the fact that resources should be freed when no longer needed. This is easy to do by simply including the necessary call to `sc.stop()` in all applications.

```
sc = SparkContext(conf=conf)
sc.setLogLevel("ERROR")
sc.stop()
```

A Complete Python Application

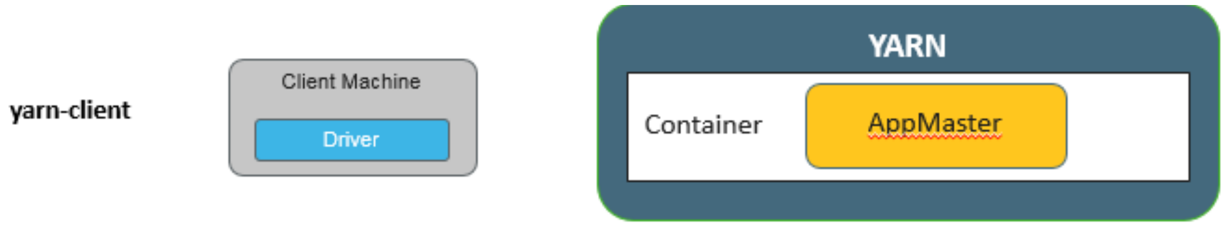
Here is an example of a complete application:

```
import os
import sys
from pyspark import SparkContext, SparkConf
if __name__ == "__main__":
conf = SparkConf() \
.setAppName("App Name") \
.setMaster("yarn-client")
sc = SparkContext(conf=conf)
sc.textFile("myfile.txt”) ## Do Stuff with this RDD
## Do more processing
sc.stop()
```

Once this scaffolding is in place, programmer-supplied code can be added from interactive learning and testing done in the REPL.

YARN Client vs. YARN Cluster

There are two modes to choose from when deploying Spark applications to YARN.

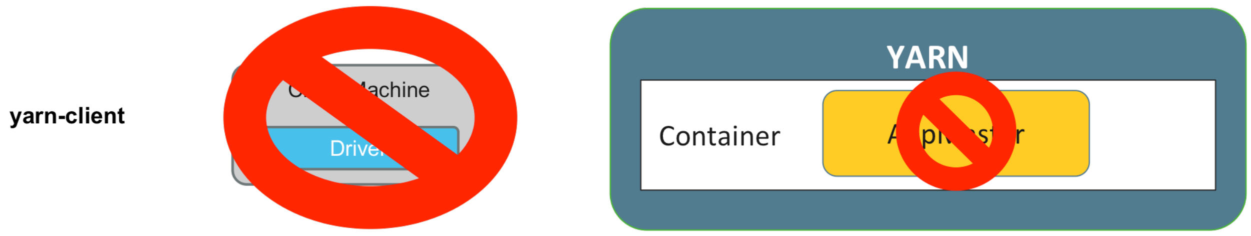


yarn-client Deployment Mode

The first, and probably the one that developers have the most experience with, is "**yarn-client**." In yarn-client mode, the driver program is a JVM started on the machine the application is submitted from. The SparkContext then lives in that JVM. This is the way the REPLs and Zeppelin start a Spark application. They provide an interactive way to use Spark, so the context must exist where the developer has access.

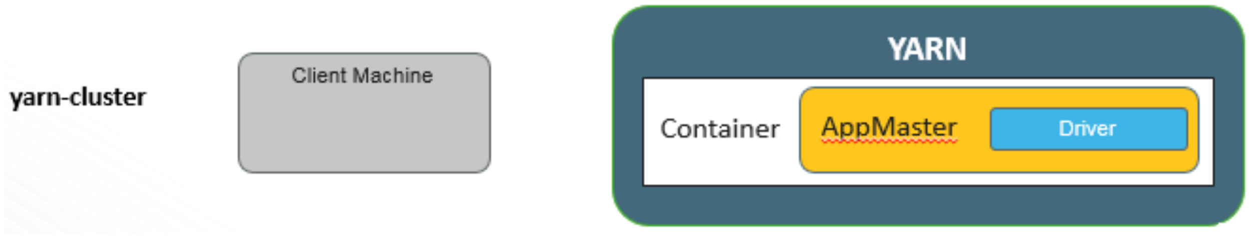
The other option, and the one that should be used for production applications, is "**yarn-cluster**." The biggest difference between the two is the location of the Spark Driver.

In client mode, the Spark driver exists on the client machine. If something should happen to that client machine, the application will fail.



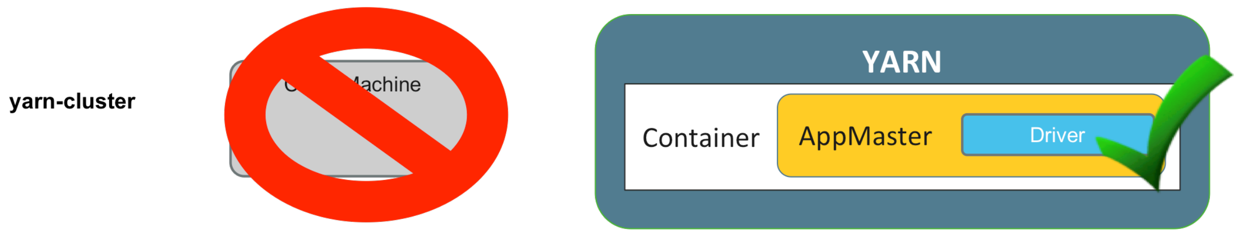
Client failure means application failure

In cluster mode, the application puts the Spark driver on the YARN ApplicationMaster process which is running on a worker node somewhere in the cluster.



yarn-cluster Deployment Mode

One big advantage of this is that even if the client machine that submitted the application to YARN fails, the Spark application will continue to run.



Client can fail and application will continue to run

YARN Application Submission

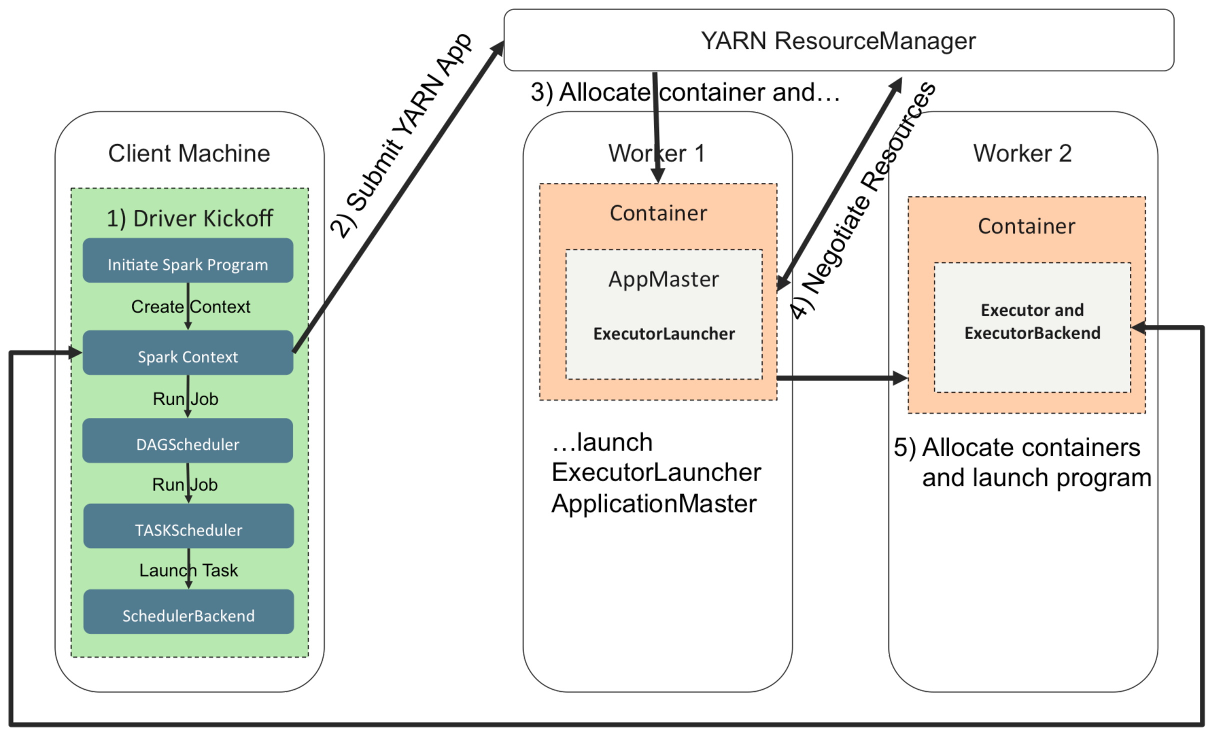
Use `yarn-client` when:

- Initial development of an application, especially if items are being printed to the screen
- Testing applications
- Used by default for REPLs and can be set for Zeppelin

Use `yarn-cluster` when:

- Scheduling a running production application

YARN Client Submission Process

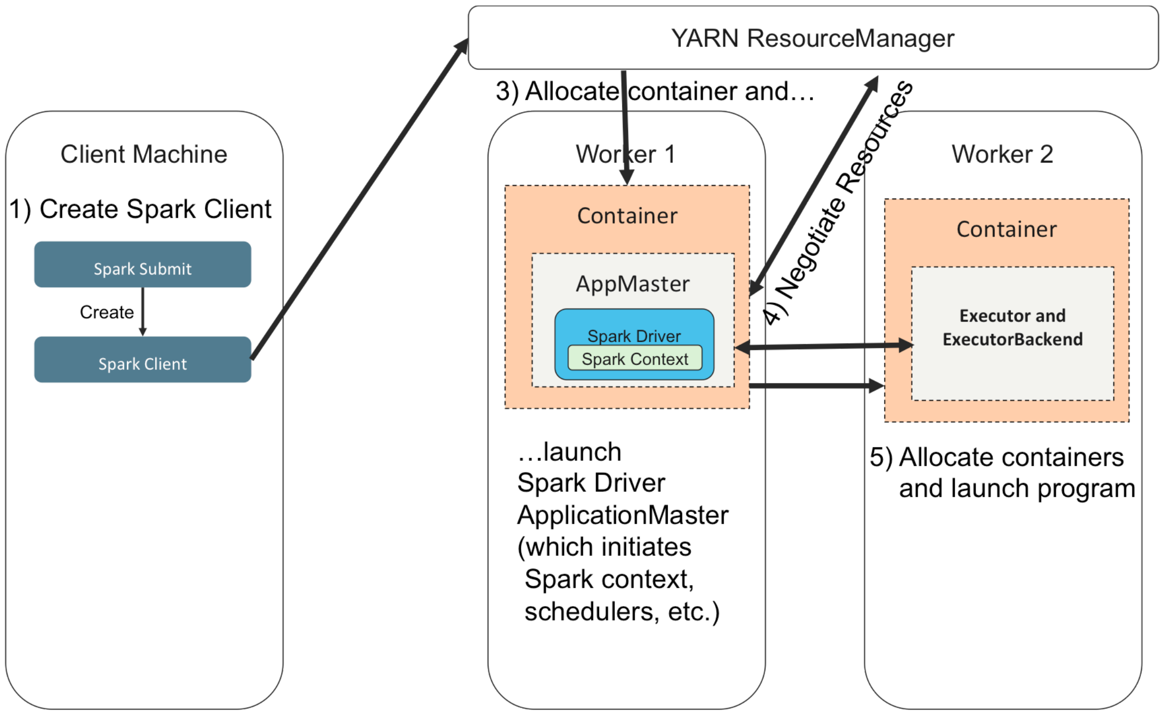


YARN Client Submission Process

In `yarn-client` mode, the driver and context are running on the client, as seen in the example.

When the application is submitted, the `SparkContext` reaches out to the resource manager to create an Application Master. The Application Master is then created, and asks the Resource Manager for the rest of the resources that were requested in the `SparkConf`, or from the runtime configurations. After the Application Master gets confirmation of resource availability, it contacts the Node Managers to launch the executors. The `SparkContext` will then start scheduling tasks for the executors to execute.

YARN Cluster Submission Process


YARN ResourceManager
3) Allocate container and…
Client Machine
1) Create Spark Client
Spark Submit
Create
Spark Client
Worker 1
Container
AppMaster
Spark Driver
Spark Context
4) Negotiate Resources
Worker 2
Container
Executor and ExecutorBackend
…launch Spark Driver ApplicationMaster (which initiates Spark context, schedulers, etc.)
5) Allocate containers and launch program

YARN Cluster Submission Process

In a yarn-cluster submission, the application starts similarly to yarn-client, except that a Spark client is created. The Spark client is a proxy that communicates with the Resource Manager to create the Application Master.

The Application Master then hosts the Spark driver and `SparkContext`. Once this handoff has occurred, the client machine can fail with no repercussions to the application. The only job the client had was to start the job and pass the binaries. Once the Application Master is started, it is the same internal process as during a yarn-client submission. The Application Master talks to the Node Managers to start the executors. Then the `SparkContext`, which resides in the Application Master can start assigning tasks to the executors.

This removes the single point of failure that exists with yarn-client job submissions. The Application Master needs to be functional, but in yarn-cluster, there is no need for the client after the application has launched.

In addition, many applications are often submitted from the same machine. The driver program, which holds the application, requires resources and a JVM. If there are too many applications running on the client, applications may have to wait until resources free up, which can create a bottleneck. A yarn-cluster submission moves that resource usage to the cluster.

Submitting and Application to YARN

After creating an application, the developer may submit it to the cluster. Spark uses the command `spark-submit` to submit a spark application from the command line. Here is an example of submitting an application named sparkDemo.py to the cluster:

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

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

Here are some of the runtime configurations that can be set, with the format to submit them:

```
--num-executors 2

--executor-memory 1g

--master yarn-cluster

--conf spark.executor.cores=2
```

NOTE:

The last configuration property in this list could also have been added as: `--executor-cores=2`.

Arguments can be added after the file name.

TIP:

Be careful when requesting resources. Spark will hold on to all resources it is allocated, even if they are not being used. While YARN can pre-empt containers, it is the developer's duty to make sure they are using a reasonable number of resources. Once allocated, Spark does not easily give up resources.

Using Different Versions of Python

A very common issue enterprises face when starting to use Spark with Python is the version of Python that is configured, and the location of their Python code. When submitting to YARN, there can be issues with the version of Python in the config file and what is available on the cluster. A safe way to circumvent the issue is to set the variable `PYSPARK_PYTHON` with the path to the version of Python the developer is using. This is a Python-specific issue, with different versions of Python such as Anaconda or Python 3.

A safe way to submit a pyspark application (called `sparkDemo.py`) using YARN is to use the following syntax:

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

Application Submission Example

Here is a sample of a full-fledged spark-submit example.

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

In the example above, the application is being submitted to YARN in the yarn-cluster mode. We're requesting four executors, each with 8GB of memory. The application being submitted is `/user/username/sparkDemo.py` which is being passed two arguments -- an input file and an output file.

NOTE:

the `PYSPARK_PYTHON` variable is not specified here for brevity, but it would be a good idea to include this in all submissions where there might be a conflict on Python versioning.

Important Configuration Settings

Spark applications present an opportunity to set several configurations more than once. Here is the priority in which they are taken from 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`

REFERENCE:

These can be seen in the documentation at `spark.apache.org`

Because of this, setting as few in the application is best practice, with the exception of some specific configurations. Pass the rest in at runtime or in a configuration file.

Important Runtime Configuration Settings

There are several very common, important runtime configurations that the developer will typically need to set, contained in the examples below:

Configurations with keywords

```
--num-executors 20

--executor-mem 8g

--executor-cores 2

--master yarn-client

--driver-memory 1g
```

Configurations set using --conf key=val

```
spark.shuffle.memoryFraction

spark.storage.memoryFraction

spark.default.parallelism

spark.speculation
```

Important In-Application Configuration Settings

An important configuration to set in the application is to specify the use of the Kryo serializer. Also, register any custom classes that are used if the application is not using Python. Python uses its own Pickle serializer for Python objects, but can use Kryo for JVM objects that will be created when using Spark SQL.

Setting speculative execution to **true** is also usually a good idea. This monitors tasks and compares them across their narrow operation counterparts. If one or more tasks are running slowly compared to other partitions in a stage, they will be re-launched.

Here is an example of setting both the Kryo serializer and speculative execution:

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

To register a class with Kryo in Scala use the following.

```
val kryo = Kryo()
kryo.register( classOf[ Array[ MyClass[_] ] ] )
```

Knowledge Check

You can use the following questions to assess your understanding of the concepts presented in this lesson.

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

Answers

1) What components does the developer need to recreate when creating a Spark Application as opposed to using Zeppelin or a REPL?

Answer: The developer must import the `SparkContext`, `SparkConf` libraries, create the main program, create a `SparkConf` and a `SparkContext`, and stop the `SparkContext` at the end of the application

2) What are the two YARN submission options the developer has?

Answer: `yarn-client` and `yarn-cluster` are the two yarn submission options

3) What is the difference between the two YARN submission options?

Answer: The difference between yarn-client and yarn-cluster is where the driver and `SparkContext` reside. The driver and context reside on the client in `yarn-client`, and in the application master in `yarn-cluster`.

4) When making a configuration setting, which location has the highest priority if the event of a conflict?

Answer: Settings configured inside the application

5) True or False: You should set your Python Spark SQL application to use Kryo serialization

Answer: True. It is used for JVM objects that will be created when using Spark SQL

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.

Introduction to Machine Learning with Spark

Lesson 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:

Machine learning is an expansive topic that could easily span multiple days of training without covering everything. Since this is a class for application developers and not specifically data scientists, an in-depth discussion on machine learning is out of scope.

However, Spark does come with a number of powerful machine learning tools and capabilities. Fully utilizing the packages and practices this lesson will discuss requires a fundamental understanding that goes well beyond what will be covered here. Even so, it is well worth a developer's time to be aware of machine learning algorithms and generally the kinds of things they can do.

Furthermore, the lab and suggested exercises that accompany this lesson will consist of pre-built scripts and sample applications that will demonstrate some of these topics in practice. A student interested in learning more is encouraged to take a look at additional and future Hortonworks University offerings that specifically focus on more advanced programming, data science, or both.

Machine Learning Basics

Machine learning attempts to take data and find actionable patterns within it. Machine learning algorithms create a model from previous data, and apply that model to new data in order to predict something.

There are two basic types of machine learning: **supervised** and **unsupervised**. Technically, there are others - such as semi-supervised and reinforcement learning - but discussions of those goes beyond the scope of this class.

Supervised Learning

Supervised learning is the most common type of machine learning. It occurs when a model is created using one or more variables to make a prediction, and then the accuracy of that prediction can be immediately tested.

There are two common types of predictions: **Classification** and **Regression.**

Classification attempts to answer a discrete question - is the answer yes or no? Will the application be approved or rejected? Is this email spam or safe to send to the user? "Will the flight depart on time?" It's either a yes or no answer - if we predict the flight departs early or on time, the answer is yes. If we predict it will be one minute late or more, the answer is no.

Regression attempts to determine what a value will be given specific information. What will the home sell for? What should their life insurance rate be? What time is the flight likely to depart? It's an answer where a specific value is being placed, rather than a simple yes or no is being applied. Therefore, we might say the flight will depart at 11:35 as our prediction.

Supervised learning starts by randomly breaking a dataset into two parts: **training data** and **testing data.**

Training data is what a machine-learning algorithm uses to create a model. It starts with a dataset, then performs statistical analysis of the effect one or more variables has on the final result. Since the answers (yes or no for classification, or the exact value - ex: flight departure time) are known, the training dataset can know with a high degree of certainty that the weight it applies to a variable is accurate within the training data.

Once a model that is accurate for the training dataset is built, that model is then applied to the testing dataset to see how accurate it is when the correct answers are not known ahead of time. The model will almost never be 100% accurate for testing data, but the better the model is, the better it will be at accurately predicting results where the answers are not known ahead of time.

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 are collected and Available Every Day

This is a simple example of what a supervised learning dataset might look like. We have many columns to choose from when selecting the variables we want to test. There would likely be thousands upon thousands of data points collected and available, with new information streaming in on a continuous basis, giving us massive historical data to work from.

Notice that this dataset could be used either for regression or classification. Classification would compare the Sched vs. Actual column and if the Actual value was less than or equal to Sched interpret it as a yes. If not, it would be a no. For regression, the actual departure time is known.

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

When creating a supervised learning application, there is a well-defined end-to-end flow.

- 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

The developer starts with a training dataset, labels and a target variable. The features must be extracted and then turned into a feature matrix. Given the labels and the feature matrix, a model can be trained. Once the model is created, as new data comes in, a feature vector must be extracted from the new data. Then the target variable can be predicted, using the model created.

Examining Supervised Learning Results

Once a model has been run against the testing dataset, its level of accuracy can be determined.

For classification results, the model either correctly or incorrectly predicts the result. For example: Will the flight depart on time? The model will accurately predict the answer to this question a certain percentage of the time, and will be inaccurate for all other observations in the testing dataset. Making that percentage incorrect as small as possible for the testing data is the overall goal of the model-building process.

For regression results, real-world predictions will often be inexact, but the better the model is, the closer it will come to predicting actual values for the test data.

The accuracy of a regression model is based on minimizing a value called "sum of mean squared error." The basic notion is that two different regression models may produce results that have the same average error, but a model with greater volatility will be penalized.

Sum of Mean Squared Error

To help understand the sum of mean squared error, take the following simple example:

Your training dataset contains four observations. Both Model A and Model B have an average value error of two. If we evaluated just the mean error, we would consider these two models to have equal predictive power.

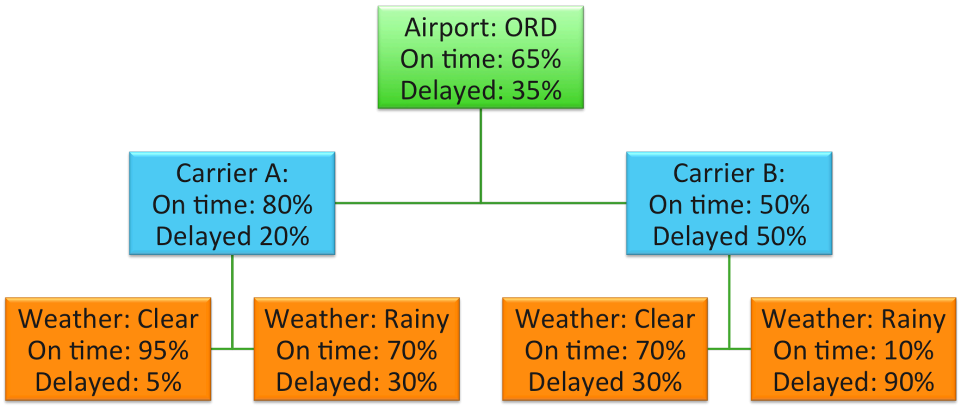
- Model A variance by observation: 0, 4, 0, 4
- Model B variance by observation: 1, 3, 2, 2

Let's take a closer look though and see why simply using the average value isn't the best approach.

In the case of Model A, in two observations, the model is exactly accurate, and in two it is off by 4. In the case of Model B, the model never predicts the value with 100% accuracy, but it tends to be closer, more often. Intuitively, Model B seems to fit the data better. This should make sense, but how do we quantify this?

The sum of mean squared error simply squares each of the variances of every observation and adds those values together. This adds an exponential penalty for observations as they get further away from the predicted value. Thus, the sum of mean squared error for Model A = 0 + 16 + 0 + 16, for a final value of 32. The sum of mean squared error for Model B = 1 + 9 + 4 + 4, for a final value of 18. Since the sum of mean squared error is lower for Model B, we can determine that it is the better model. Thus, we can both intuitively and mathematically determine that Model B is a better predictor than Model A.

Decision Tree Algorithm

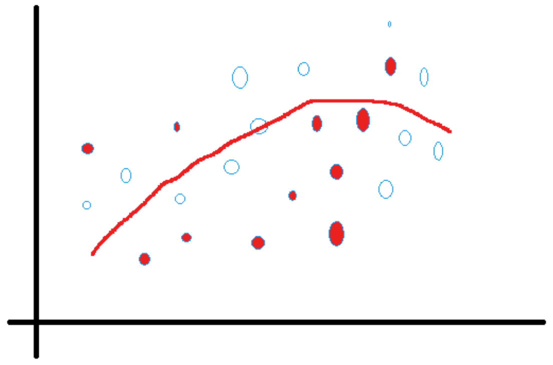


Decision Tree Algorithm Example

One commonly used classification algorithm is called the **Decision Tree algorithm.** In essence, a decision tree uses a selected variable to determine the probability of an outcome, and then - assuming that variable and probability are known - selects another variable and does the same thing. This continues through the dataset, with variables and their order of selection/evaluation determined by the data scientist. In the graphic, we see a small part of what would be a much larger decision tree, where an airport value of ORD has been evaluated, followed by carriers at ORD, followed by weather conditions.

There are often numerous ways in which decision trees might be constructed, and some paths will produce better predictions than others. The same target variables can be arranged into multiple decision tree paths, which can be combined into what is known as a Forest. In the end, the classification (prediction) that has the most "votes" is selected as the prediction.

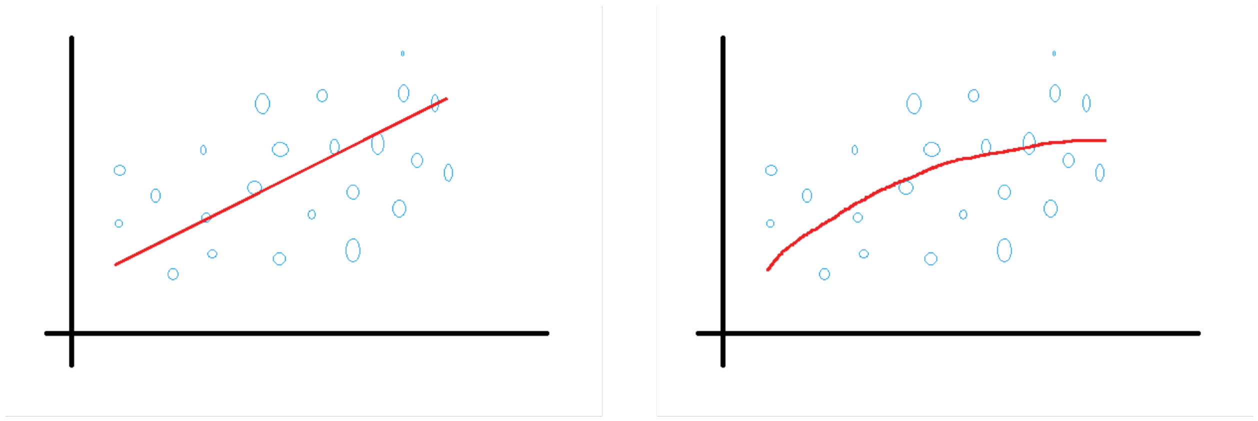
Classification Algorithms



Classification Algorithms

When creating a **classification** visualization, the model draws a line where it predicts the answers will be. This line can then be compared to the actual results in the test data. For example, in this simple visualization, the white-filled circles represent observations of target variables where actual departure time was less than or equal to the scheduled departure time. The red-filled circles represent observations where actual departure time was greater than scheduled departure time. The red line represents the predictions that the model made. Above the red line would be where the model predicted on-time departures, and below the red line would be were the model predicted delayed departures.

Linear Regression Algorithm



Linear Regression Algorithm

In the case of regression, the line drawn is predicting an actual value rather than a binary result. In the first diagram, we see a regression where only a single variable was selected and weighted - thus the result will be a straight line. As more variables are added, the regression curves, and in some cases, can curve wildly based on the variables and the weights determined by the model. The second diagram is what a model with two variables might look like. To determine which model was a better predictor, we would find out how far away each of the dots were from the prediction line and perform a sum of means squared error calculation.

Unsupervised Learning

Supervised learning is a powerful tool as long as you have clean, formatted data where every column has an accurate label. However, in some cases, what we start with is simply data, and appropriate labels may be unknown. For example, take product reviews that people leave on social media, blogs, and other web sites. Unlike reviews on retailers' pages, where the user explicitly gives a negative, neutral, or positive rating as part of creating their review (for example, a star rating), the social media and other reviews have no such rating or label applied. How then can we group them to determine whether any given review is positive, neutral, or negative, and determine whether the general consensus is positive or negative?

For a human evaluator, simply reading the review would be enough. However, if we are collecting thousands of reviews every day from various sources, employing a human to read and categorize each one would be highly inefficient. This is where unsupervised learning comes in. The goal of unsupervised learning is to define criteria by which a dataset will be evaluated, and then find patterns in the data that are made up of groupings with similar characteristics. The algorithm **does not** determine what those groupings mean - that is up to the data scientist to fill in. **All it determines is what should be grouped**, based on the supplied criteria.

For example, we might look at examples where certain phrases are compared, and the algorithm might determine that when a review contains phrase X, it quite usually also contains phrase Y. Therefore, a review that contains phrase X but not phrase Y would still be combined with the phrase Y group. After this processing is complete, the data scientist looks at a few of the phrase Y grouped reviews and determines that they are generally positive, and thus assigns them to the positive review category.

The most common type of unsupervised learning, and the one described in this example, is called clustering.

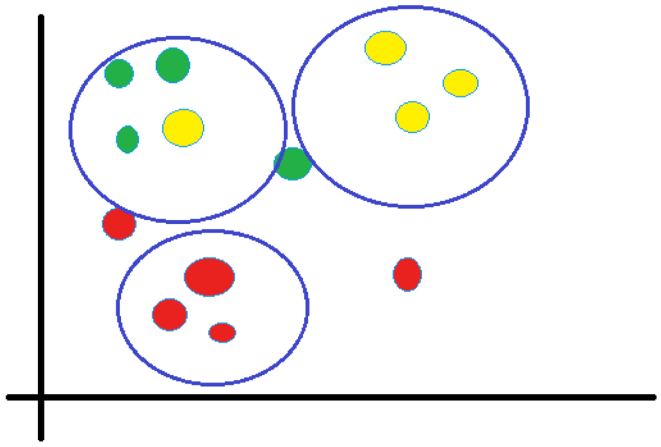
Unsupervised Learning Example Dataset

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

In this example, we have observations from which we have picked out phrases from a defined list we are looking for. The data has been cleaned of extraneous words and phrases, and then the remaining groups of phrases are evaluated to determine how frequently they are used within the same review. The algorithm searches for patterns so that reviews can be grouped, but has no idea whether any particular grouping represents positive, neutral, or negative reviews.

K-Means Algorithm



K-Means is Used to Identify Groupings that Likely Share the Same Label

Once the algorithm has grouped the results, the data scientist must determine the meaning. In the diagram, negative reviews are coded red, positive are coded green, and neutral are coded yellow. A clustering algorithm known as the K-Means algorithm was applied and groupings were created. Be aware that just as in supervised learning, not all reviews could be grouped closely with some others, and in some cases, reviews were grouped with the wrong category. The better the model is, the more accurate these groupings will be.

Other Popular Algorithms

In addition to the algorithms discussed in this section, there are several other popular algorithms and tools commonly employed by data scientists and available in Spark. These include:

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

In addition, Spark also offers a range of Basic Statistics tools, such as summary statistics, correlations, and random data generation.

Spark Machine Learning Libraries

Spark's machine learning capabilities include implementations of common learning algorithms and utilities. Because Spark is natively a highly parallelized, cluster-level application, machine learning algorithms can be run at cluster scale utilizing resources across dozens, hundreds, or thousands of machines at one time.

There are two packages available for Spark machine learning: `spark.mllib`, which operates on RDDs; and `spark.ml`, which operates on DataFrames.

Both packages contain modules with various functions and sub-functions, which provide powerful machine learning capabilities.

mllib Modules

This is a list of the modules available in Spark's `mllib` package:

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

ml Modules

This is a list of the modules available in Spark's `ml` package:

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

Spark Machine Learning Advantages

Machine learning on Spark provides several advantages over available alternatives. First, most machine learning libraries are designed to be run on a single machine, meaning the size of a usable dataset is determined in part by what can be fit into local memory on that machine. In addition, processing of the data occurs only on the CPU resources available to that one machine, whereas Spark can run parallel operations across dozens, hundreds, or thousands of machines at the same time.

There are other Hadoop machine learning libraries available as well - for example Mahout - which are based on older, less efficient platforms like MapReduce. Machine learning on Spark leverages Spark's in-memory processing capabilities, which can result in significantly faster processing times.

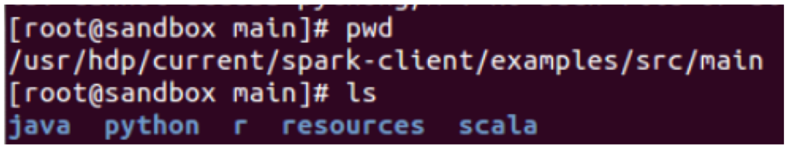
Between the two Spark packages, `ml` has several advantages over `mllib`. Because `ml` operates on DataFrames, it provides greater flexibility as well as automatic performance enhancements due to the underlying Catalyst optimization engine.

Machine Learning Pipelines

The `ml` package also provides the ability to create machine learning pipelines. Pipelines extend Spark RDD and DataFrame programming concepts like transformations and actions into the machine learning world, allowing for the construction of reusable sets of transformations (via components named Transformers) that become input for machine learning models (components named Estimators). As of Spark 1.6, entire pipelines can be persistent, meaning saved and loaded at a later time or across various clusters.

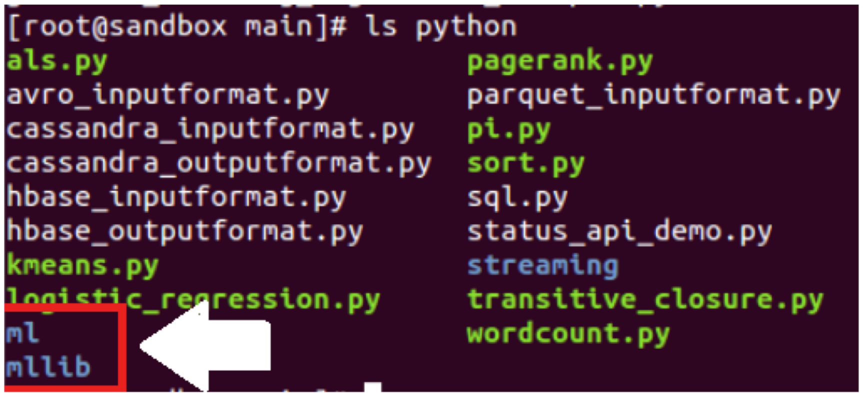
Sample Machine Learning Applications

Spark automatically installs two directories of application samples where machine learning algorithms can be seen in action. They are located at `/usr/hdp/current/spark-client/examples/src/main/<language>/<mllib | ml>/`.



```
[root@sandbox main]# pwd
/usr/hdp/current/spark-client/examples/src/main
[root@sandbox main]# ls
java  python  r  resources  scala
```

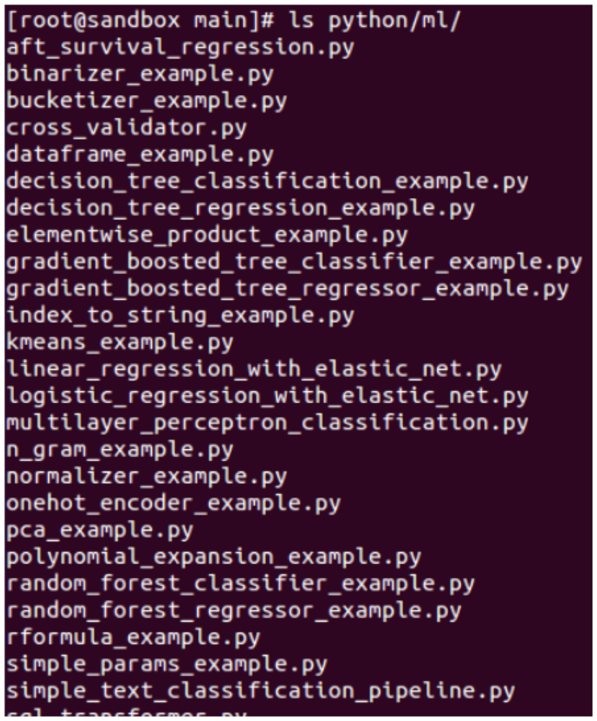
For example, if you wanted to view ml samples available for Python, you would browse to `/usr/hdp/current/spark-client/examples/src/main/Python/ml/`.



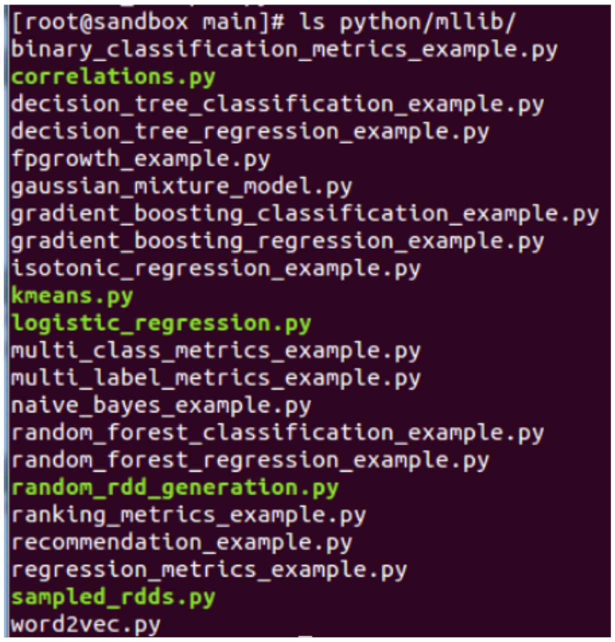
```
[root@sandbox main]# ls python
als.py                       pagerank.py
avro_inputformat.py          parquet_inputformat.py
cassandra_inputformat.py     pi.py
cassandra_outputformat.py    sort.py
hbase_inputformat.py         sql.py
hbase_outputformat.py        status_api_demo.py
kmeans.py                    streaming
logistic_regression.py       transitive_closure.py
ml                           wordcount.py
mllib
```

Sample Machine Learning Application Files

The application files are named according to what they demonstrate. In some cases, there is overlap between the two. For example, both the `ml` and `mllib` directories for Python contain a file named `decision_tree_classification_example.py`.

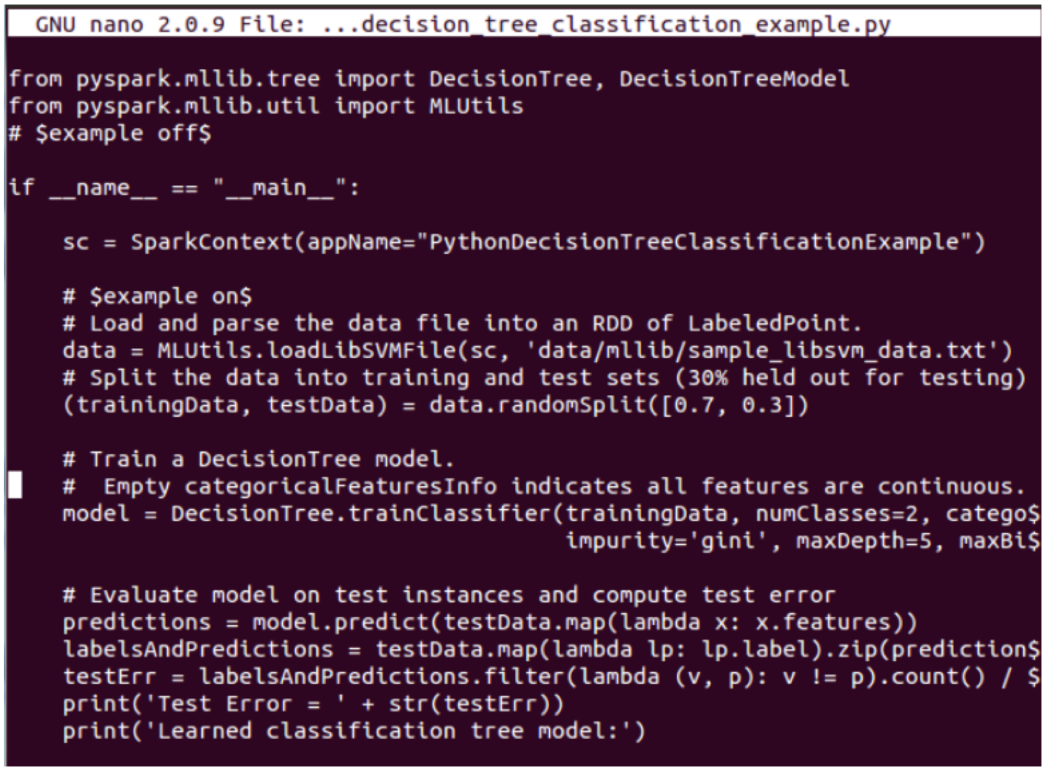


```
[root@sandbox main]# ls python/ml/
aft_survival_regression.py
binarizer_example.py
bucketizer_example.py
cross_validator.py
dataframe_example.py
decision_tree_classification_example.py
decision_tree_regression_example.py
elementwise_product_example.py
gradient_boosted_tree_classifier_example.py
gradient_boosted_tree_regressor_example.py
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
```

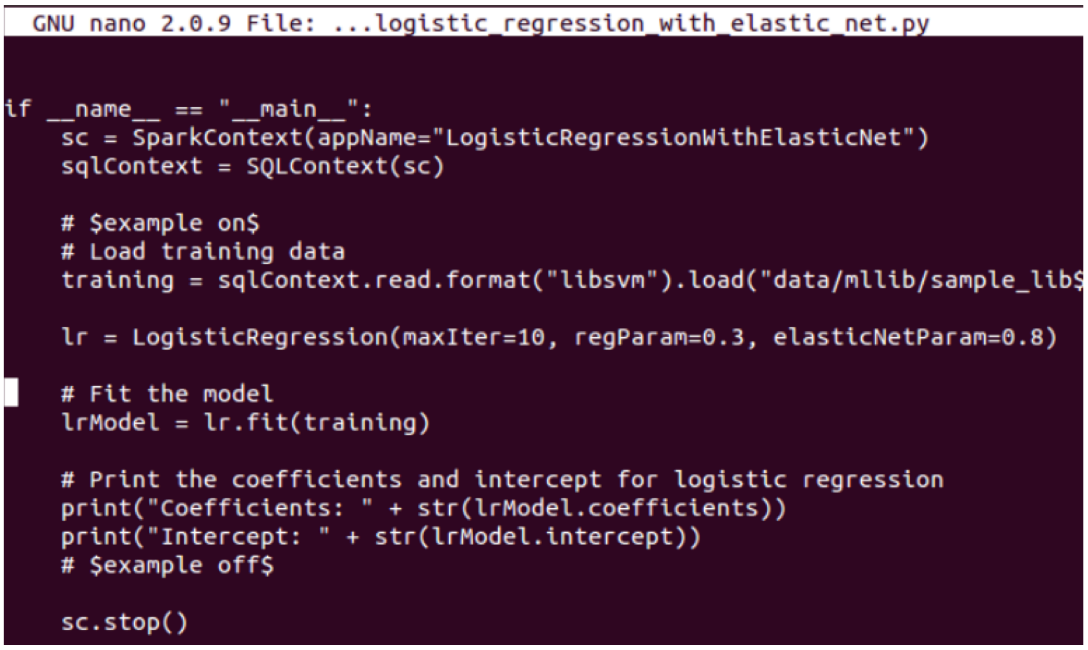
`mllib` Decision Tree Classification Example



```
  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:')
```

Using a text editor, you can open and examine the contents of each application. The examples are well commented. They can actually be used as teaching tools to help you learn how to employ Spark's machine learning capabilities for your own needs. In this example, we have opened the decision tree classification program in the Python `mllib` directory.

`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()
```

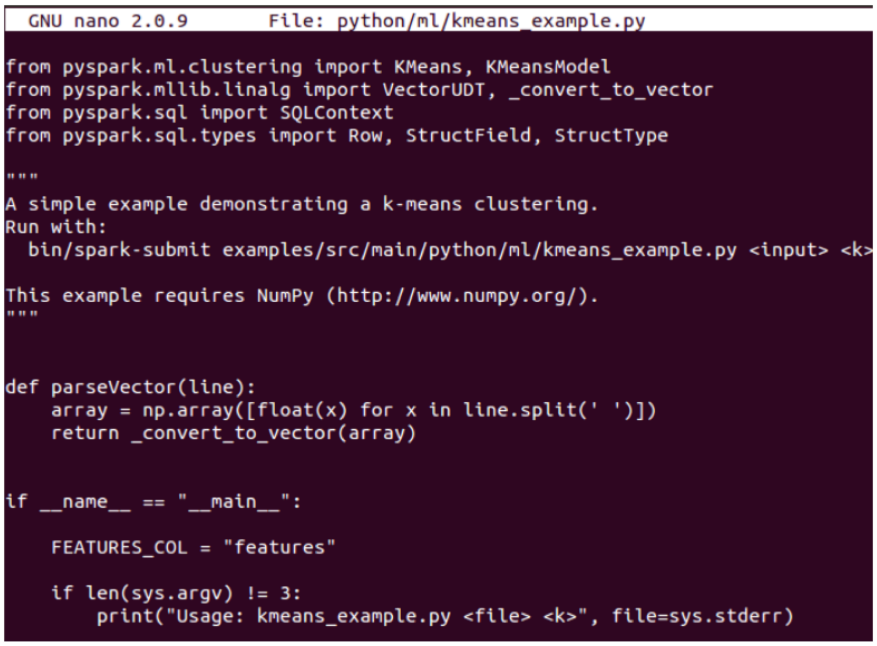
Here is another example, a logistic regression (which, as you will recall, is actually a classification algorithm) from the Python `ml` directory.

K-Means Clustering Examples

It can also be useful to compare similar programs across the two packages to see how they are programmed differently, and perhaps even to run them both and see the differences.

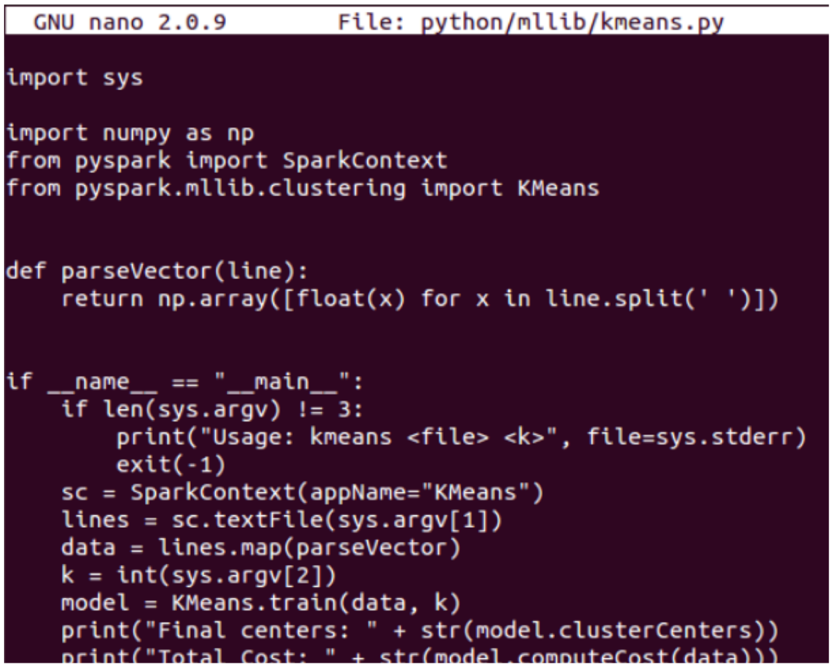
In this example, we took the k-means clustering examples from both the Python `ml` and `mllib` directories so we could compare them side by side.

Python ml K-Means Example



```
  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)
```

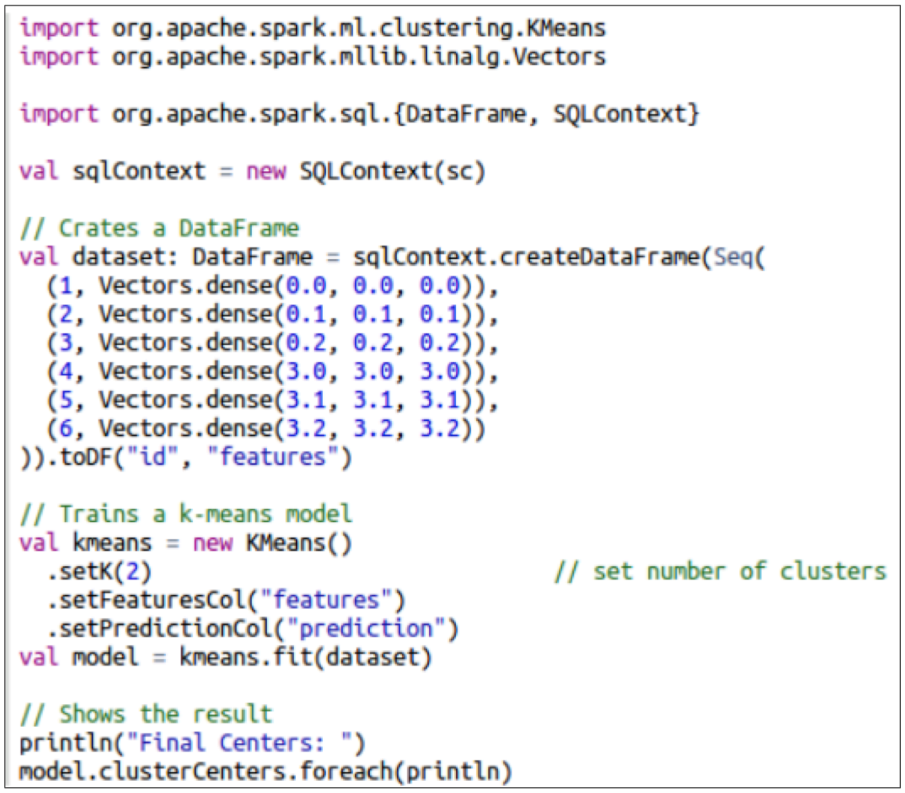
Python mllib K-Means Example



```
  GNU nano 2.0.9        File: python/mllib/kmeans.py

import sys

import numpy as np
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans


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


if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: kmeans <file> <k>", file=sys.stderr)
        exit(-1)
    sc = SparkContext(appName="KMeans")
    lines = sc.textFile(sys.argv[1])
    data = lines.map(parseVector)
    k = int(sys.argv[2])
    model = KMeans.train(data, k)
    print("Final centers: " + str(model.clusterCenters))
    print("Total Cost: " + str(model.computeCost(data)))
```

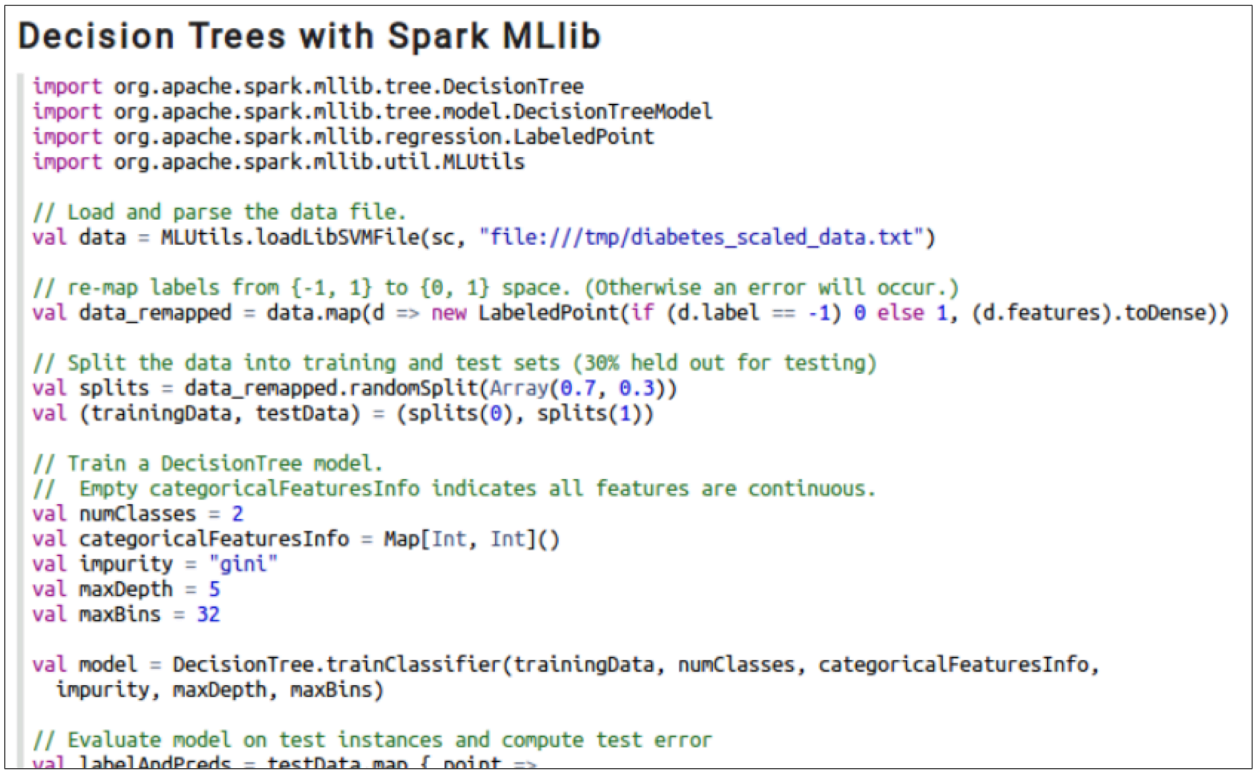
Zeppelin Machine Learning Lab Note

Zeppelin has a large number of notes available that can be imported into your Zeppelin notebook. The lab that accompanies this lesson will have you import a machine learning note, which also contains code that you can run.



```
import org.apache.spark.ml.clustering.KMeans
import org.apache.spark.mllib.linalg.Vectors

import org.apache.spark.sql.{DataFrame, SQLContext}

val sqlContext = new SQLContext(sc)

// Crates a DataFrame
val dataset: DataFrame = sqlContext.createDataFrame(Seq(
  (1, Vectors.dense(0.0, 0.0, 0.0)),
  (2, Vectors.dense(0.1, 0.1, 0.1)),
  (3, Vectors.dense(0.2, 0.2, 0.2)),
  (4, Vectors.dense(3.0, 3.0, 3.0)),
  (5, Vectors.dense(3.1, 3.1, 3.1)),
  (6, Vectors.dense(3.2, 3.2, 3.2))
)).toDF("id", "features")

// Trains a k-means model
val kmeans = new KMeans()
  .setK(2)                              // set number of clusters
  .setFeaturesCol("features")
  .setPredictionCol("prediction")
val model = kmeans.fit(dataset)

// Shows the result
println("Final Centers: ")
model.clusterCenters.foreach(println)
```

Sample code from a machine learning note imported into Zeppelin



Decision Trees with Spark MLlib

```
import org.apache.spark.mllib.tree.DecisionTree
import org.apache.spark.mllib.tree.model.DecisionTreeModel
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.util.MLUtils

// Load and parse the data file.
val data = MLUtils.loadLibSVMFile(sc, "file:///tmp/diabetes_scaled_data.txt")

// re-map labels from {-1, 1} to {0, 1} space. (Otherwise an error will occur.)
val data_remapped = data.map(d => new LabeledPoint(if (d.label == -1) 0 else 1, (d.features).toDense))

// Split the data into training and test sets (30% held out for testing)
val splits = data_remapped.randomSplit(Array(0.7, 0.3))
val (trainingData, testData) = (splits(0), splits(1))

// Train a DecisionTree model.
//  Empty categoricalFeaturesInfo indicates all features are continuous.
val numClasses = 2
val categoricalFeaturesInfo = Map[Int, Int]()
val impurity = "gini"
val maxDepth = 5
val maxBins = 32

val model = DecisionTree.trainClassifier(trainingData, numClasses, categoricalFeaturesInfo,
  impurity, maxDepth, maxBins)

// Evaluate model on test instances and compute test error
val labelAndPreds = testData.map { point =>
```

More sample code from the imported machine learning note in Zeppelin

Knowledge Check

You can use the following questions to assess your understanding of the concepts presented in this lesson.

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

Answers

1) What are two types of machine learning?

Answer: Supervised and unsupervised

2) What are two types of supervised learning?

Answer: Classification and regression

3) What do you call columns that are selected as variables to build a machine learning model?

Answer: Target variables

4) What is a row of data called in machine learning?

Answer: An observation

5) What is the goal of unsupervised learning?

Answer: The goal of unsupervised learning is to find groupings in unlabeled data

6) Name the two Spark machine learning packages.

Answer: mllib and ml

7) Which machine learning package is designed to take advantage of flexibility and performance benefits of DataFrames?

Answer: ml

8) Name two reasons to prefer Spark machine learning over other alternatives

Answer: Cluster-level resource availability, parallel processing, in-memory processing (vs. older Hadoop machine learning libraries)

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

HORTONWORKS® UNIVERSITY

Learn from the company focused solely on Hadoop.

What Makes Us Different?

1. Our courses are designed by the **leaders and committers** of Hadoop
2. We provide an **immersive** experience in **real-world** scenarios
3. We prepare you to **be an expert** with highly valued, **fresh skills**
4. Our courses are available **near you**, or accessible **online**

Hortonworks University courses are designed by the leaders and committers of Apache Hadoop. We provide immersive, real-world experience in scenario-based training. Courses offer unmatched depth and expertise available in both the classroom or online from anywhere in the world. We prepare you to be an expert with highly valued skills and for Certification.