



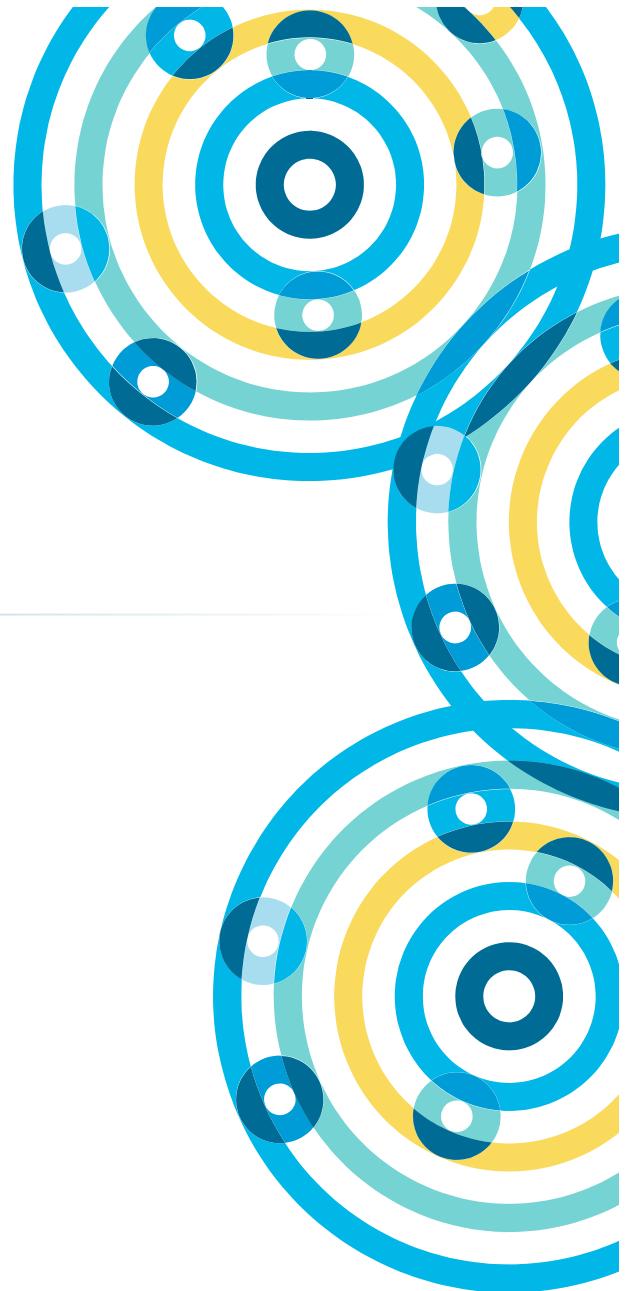
Developer Training for Spark and Hadoop I





Introduction

Chapter 1



Course Chapters

■ Introduction

- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS

- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data Partitioning

- Capturing Data with Apache Flume

■ Spark Basics

- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames

■ Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

Distributed Data Processing with
Spark

Course Conclusion

Chapter Topics

Introduction

Course Introduction

■ About This Course

- About Cloudera
- Course Logistics
- Introductions

Course Objectives

During this course, you will learn

- **How the Hadoop Ecosystem fits in with the data processing lifecycle**
- **How data is distributed, stored and processed in a Hadoop cluster**
- **How to use Sqoop and Flume to ingest data**
- **How to process distributed data with Spark**
- **Best practices for data storage**
- **How to model structured data as tables in Impala and Hive**
- **How to choose a data storage format for your data usage patterns**

Chapter Topics

Introduction

Course Introduction

- About This Course
- **About Cloudera**
- Course Logistics
- Introductions

About Cloudera (1)



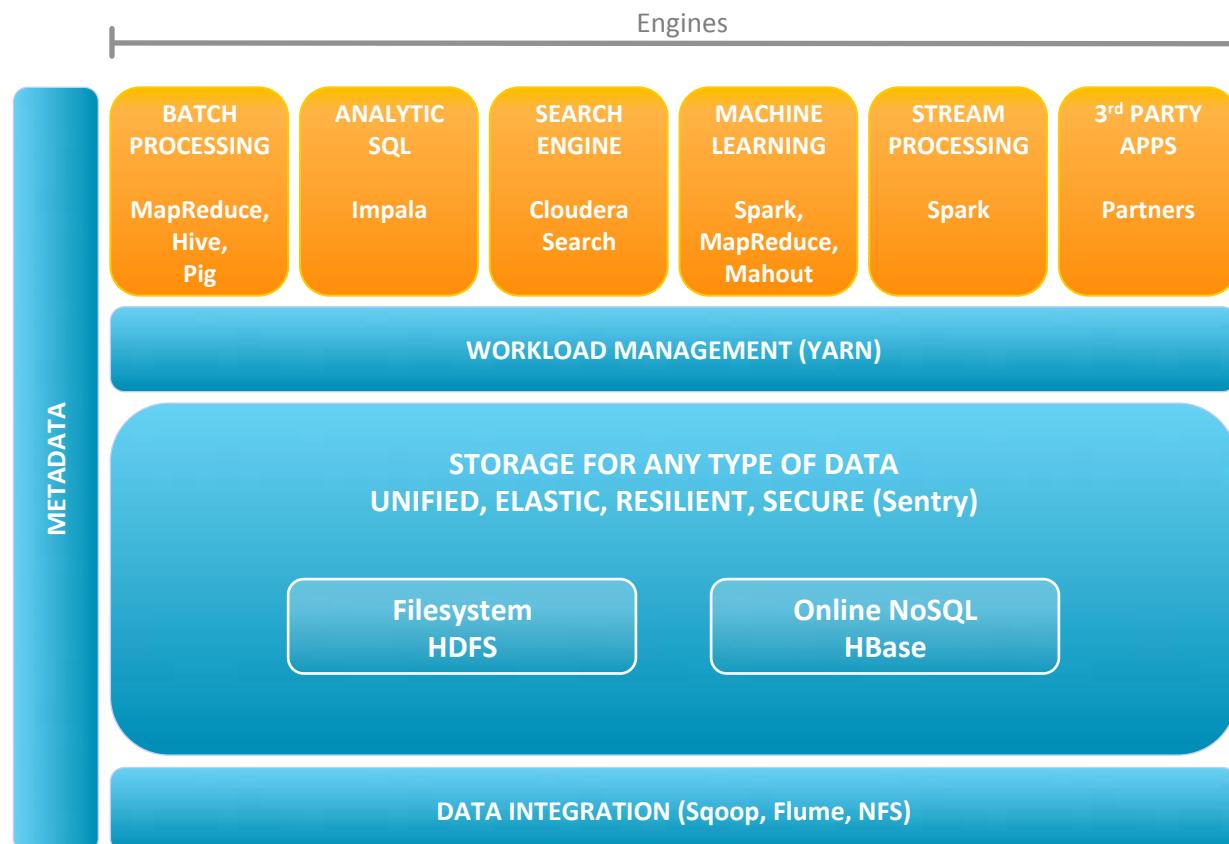
- The leader in Apache Hadoop-based software and services
- Founded by leading experts on Hadoop from Facebook, Yahoo, Google, and Oracle
- Provides support, consulting, training, and certification for Hadoop users
- Staff includes committers to virtually all Hadoop projects
- Many authors of industry standard books on Apache Hadoop projects
 - Tom White, Lars George, Kathleen Ting, etc.

About Cloudera (2)

- **Customers include many key users of Hadoop**
 - Allstate, AOL Advertising, Box, CBS Interactive, eBay, Experian, Groupon, National Cancer Institute, Orbitz, Social Security Administration, Trend Micro, Trulia, US Army, ...
- **Cloudera public training including**
 - Cloudera Developer Training for Apache Spark
 - Developer Training for Spark and Hadoop II: Advanced Techniques
 - Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop
 - Cloudera Training for Apache HBase
 - Introduction to Data Science: Building Recommender Systems
 - Cloudera Essentials for Apache Hadoop
- **Onsite and custom training is also available**

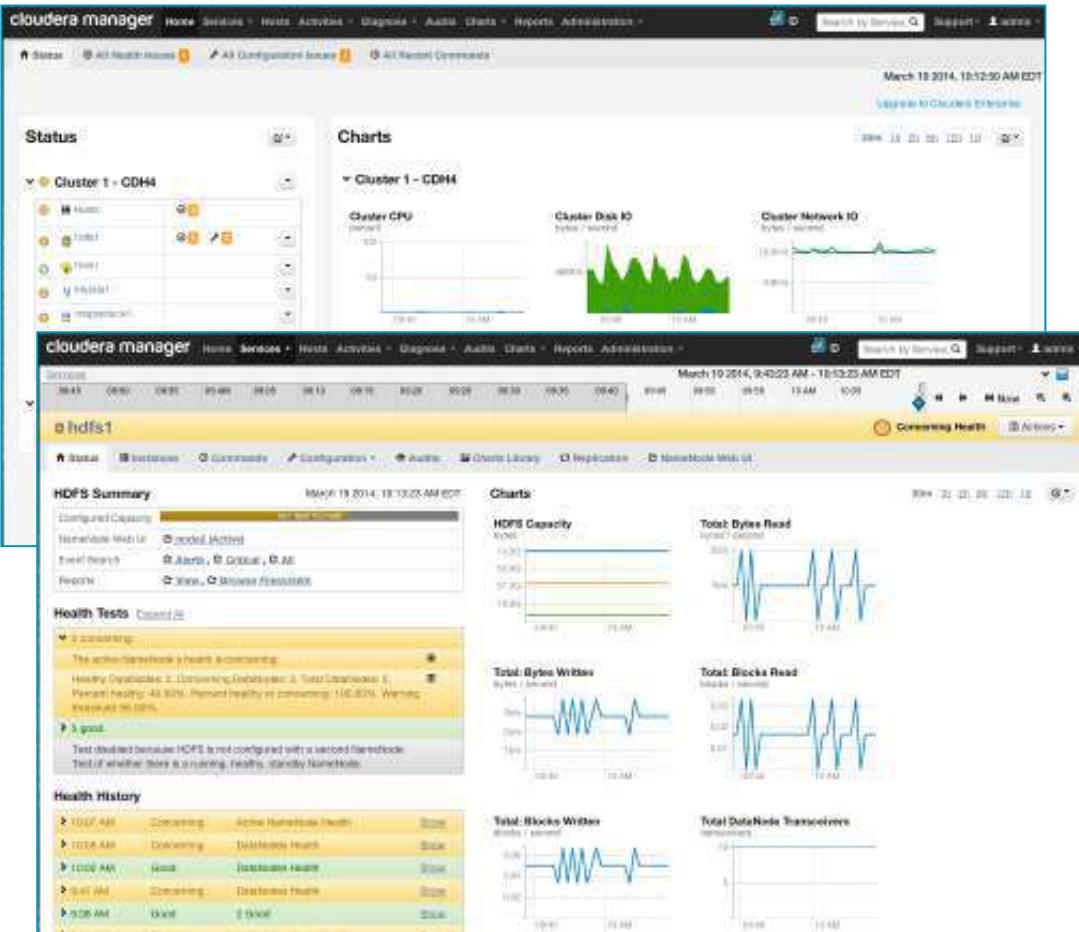
CDH (Cloudera's Distribution including Apache Hadoop)

- **100% open source, enterprise-ready distribution of Hadoop and related projects**
- **The most complete, tested, and widely-deployed distribution of Hadoop**
- **Integrates all the key Hadoop ecosystem projects**
- **Available as RPMs and Ubuntu, Debian, or SuSE packages, or as a tarball**



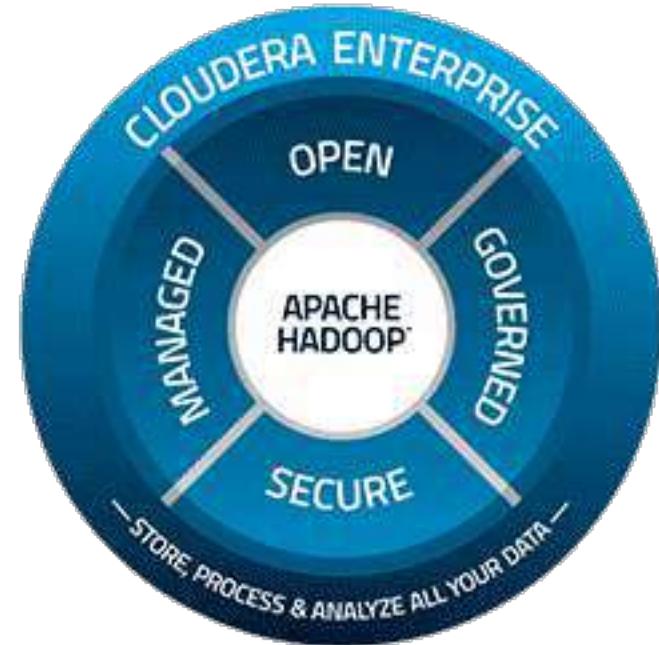
Cloudera Express

- **Cloudera Express**
 - Completely free to download and use
- **The best way to get started with Hadoop**
- **Includes CDH**
- **Includes Cloudera Manager**
 - End-to-end administration for Hadoop
 - Deploy, manage, and monitor your cluster



Cloudera Enterprise

- **Cloudera Enterprise**
 - Subscription product including CDH and Cloudera Manager
- **Includes support**
- **Includes extra Cloudera Manager features**
 - Configuration history and rollbacks
 - Rolling updates
 - LDAP integration
 - SNMP support
 - Automated disaster recovery
- **Extend capabilities with Cloudera Navigator subscription**
 - Event auditing, metadata tagging capabilities, lineage exploration
 - Available in both the Cloudera Enterprise Flex and Data Hub editions



Chapter Topics

Introduction

Course Introduction

- About This Course

- About Cloudera

- **Course Logistics**

- Introductions

Logistics

- Course start and end times
- Lunch
- Breaks
- Restrooms

Chapter Topics

Introduction

Course Introduction

- About This Course
- About Cloudera
- Course Logistics
- **Introductions**

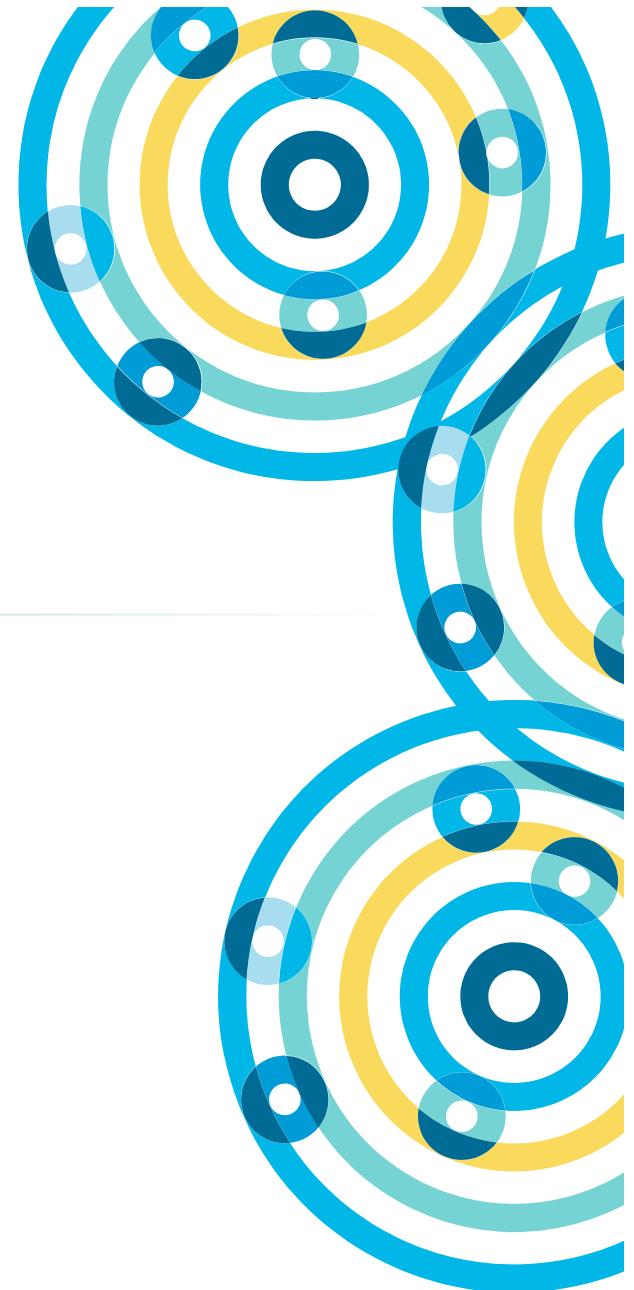
Introductions

- **About your instructor**
- **About you**
 - Company and role?
 - Experience with Hadoop and Spark?
 - Language preference: Python or Scala?
 - Expectations from the course?



Introduction to Hadoop and the Hadoop Ecosystem

Chapter 2



Course Chapters

- Introduction
- **Introduction to Hadoop and the Hadoop Ecosystem**
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

Distributed Data Processing with
Spark

Course Conclusion

Introduction to Hadoop and the Hadoop Ecosystem

In this chapter you will learn

- **What Hadoop is and how it addresses big data challenges**
- **The guiding principles behind Hadoop**
- **The major components of the Hadoop Ecosystem**
- **What tools you will be using in the Hands-On Exercises in this course**

Chapter Topics

Introduction to Hadoop and the Hadoop Ecosystem

Introduction to Hadoop

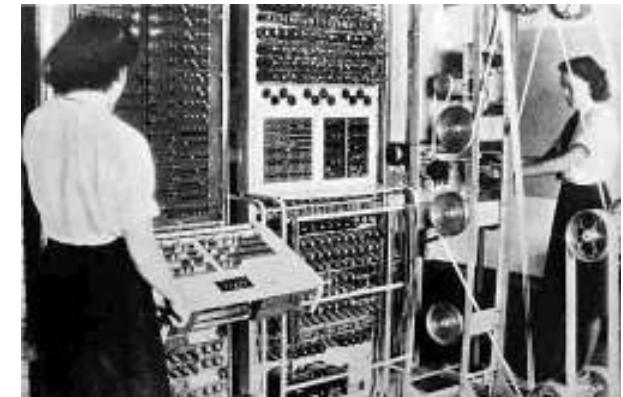
■ Problems with Traditional Large-scale Systems

- Hadoop!
- Data Storage and Ingest
- Data Processing
- Data Analysis and Exploration
- Other Ecosystem Tools
- Introduction to the Hands-On Exercises
- Conclusion

Traditional Large-Scale Computation

- Traditionally, computation has been processor-bound

- Relatively small amounts of data
 - Lots of complex processing



- The early solution: bigger computers

- Faster processor, more memory
 - But even this couldn't keep up

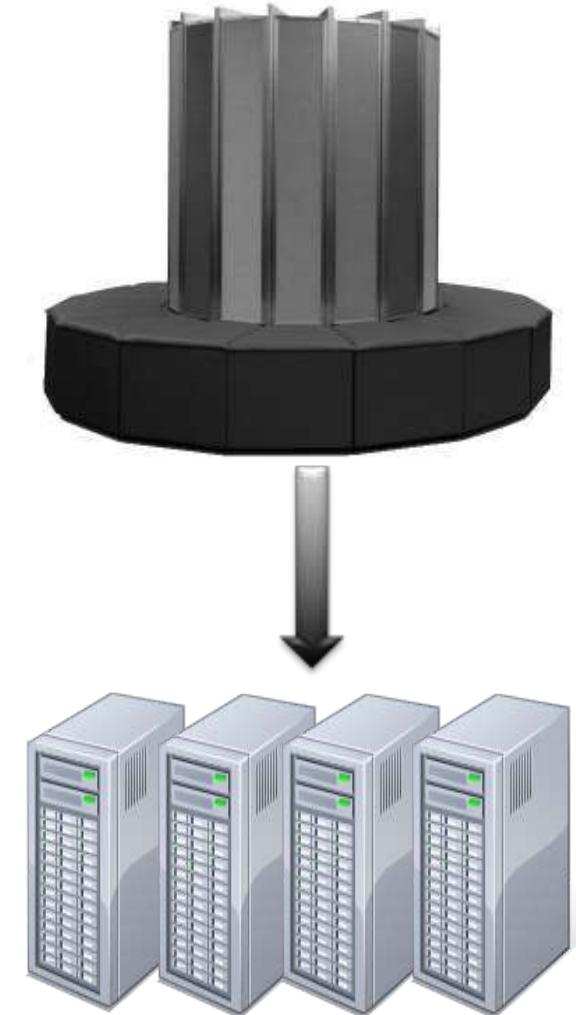


Distributed Systems

- **The better solution: more computers**
 - Distributed systems – use multiple machines for a single job

“In pioneer days they used oxen for heavy pulling, and when one ox couldn’t budge a log, we didn’t try to grow a larger ox. We shouldn’t be trying for bigger computers, but for *more systems* of computers.”

– Grace Hopper



Challenges with Distributed Systems

- **Challenges with distributed systems**
 - Programming complexity
 - Keeping data and processes in sync
 - Finite bandwidth
 - Partial failures

- **The solution?**
 - Hadoop!

Chapter Topics

Introduction to Hadoop and the Hadoop Ecosystem

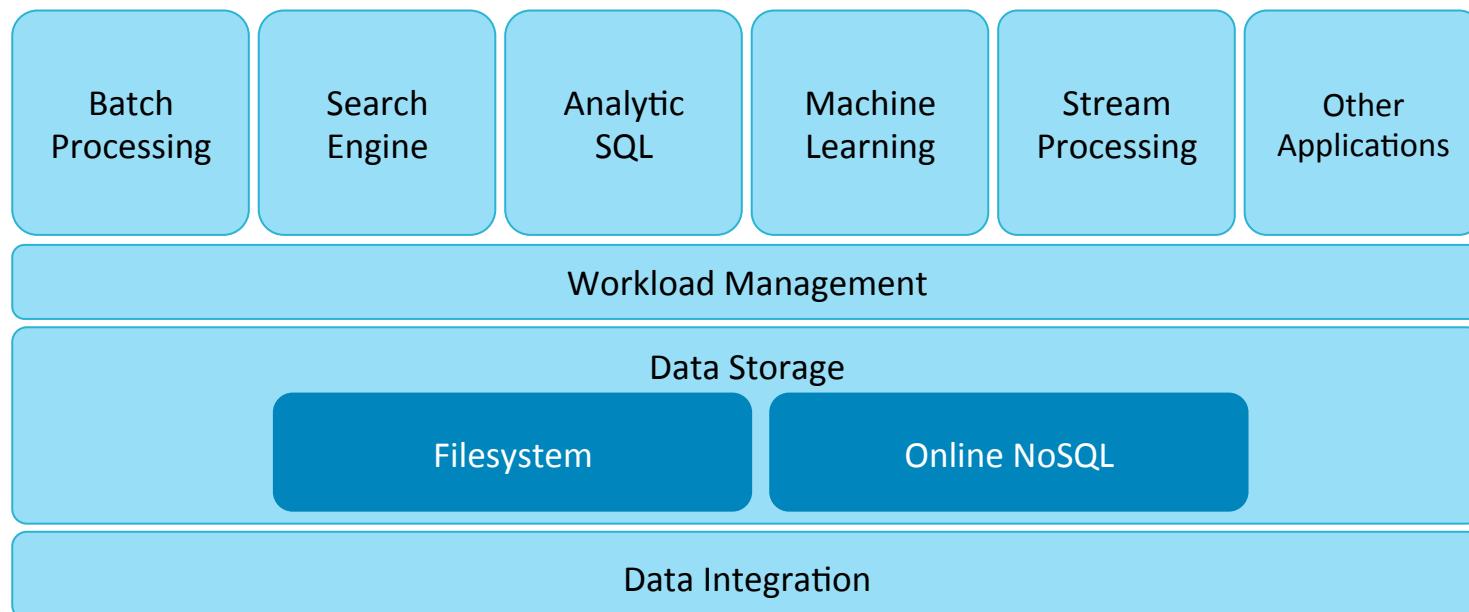
Introduction to Hadoop

- Problems with Traditional Large-scale Systems
- **Hadoop!**
- Data Storage and Ingest
- Data Processing
- Data Analysis and Exploration
- Other Ecosystem Tools
- Introduction to the Hands-On Exercises
- Conclusion

What is Apache Hadoop?



- **Scalable and economical data storage, processing and analysis**
 - Distributed and fault-tolerant
 - Harnesses the power of industry standard hardware
- **Heavily inspired by technical documents published by Google**

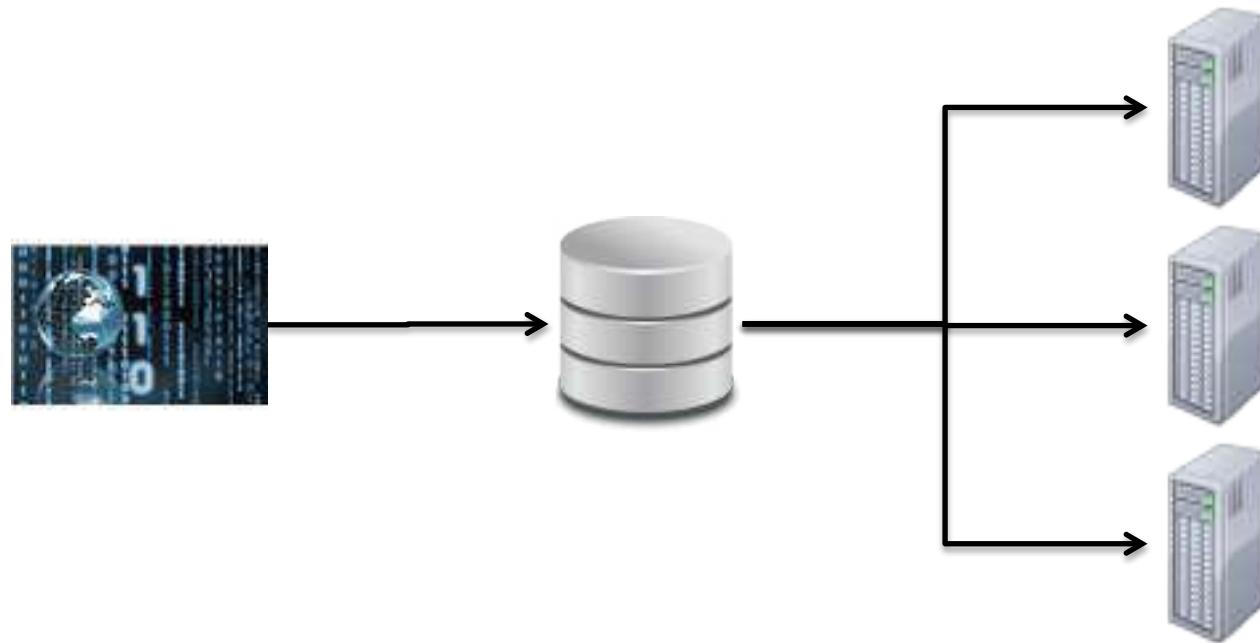


Common Hadoop Use Cases

- Extract/Transform/Load (ETL)
 - Text mining
 - Index building
 - Graph creation and analysis
 - Pattern recognition
 - Collaborative filtering
 - Prediction models
 - Sentiment analysis
 - Risk assessment
-
- What do these workloads have in common? Nature of the data...
 - Volume
 - Velocity
 - Variety

Distributed Systems: The Data Bottleneck (1)

- Traditionally, data is stored in a central location
- Data is copied to processors at runtime
- Fine for limited amounts of data

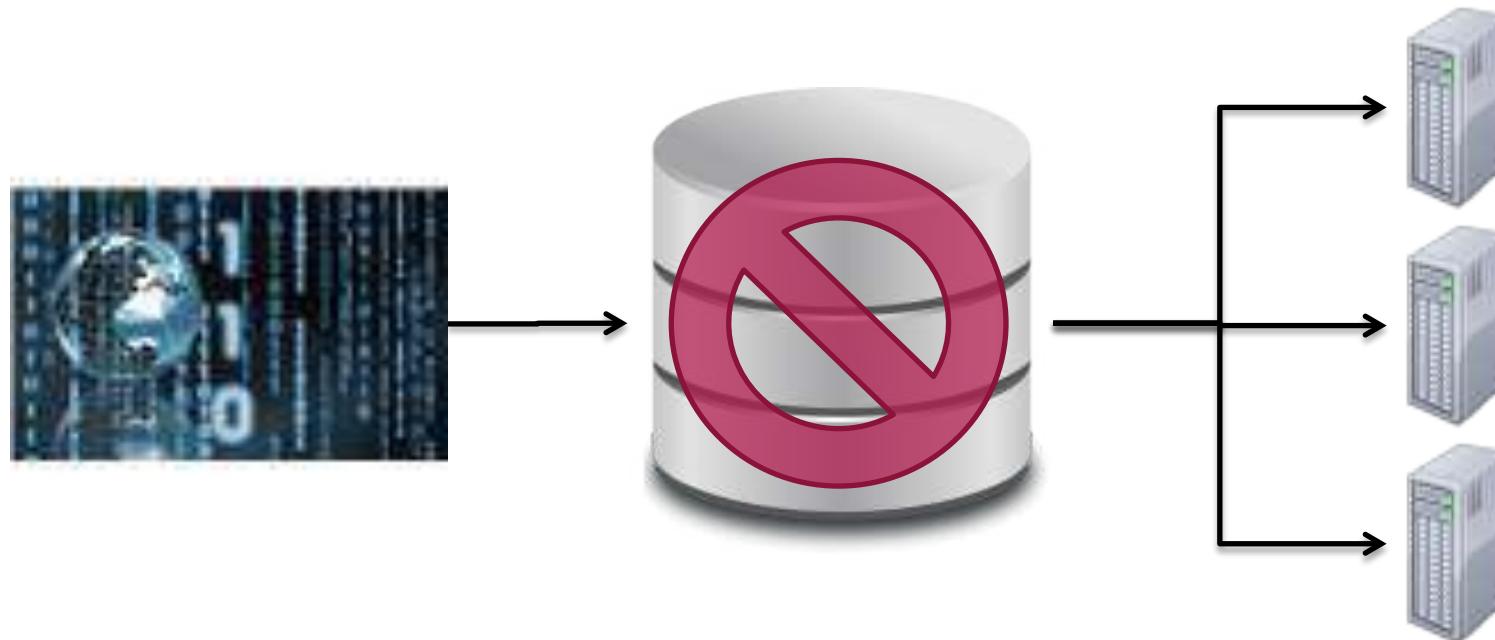


Distributed Systems: The Data Bottleneck (2)

- Modern systems have much more data

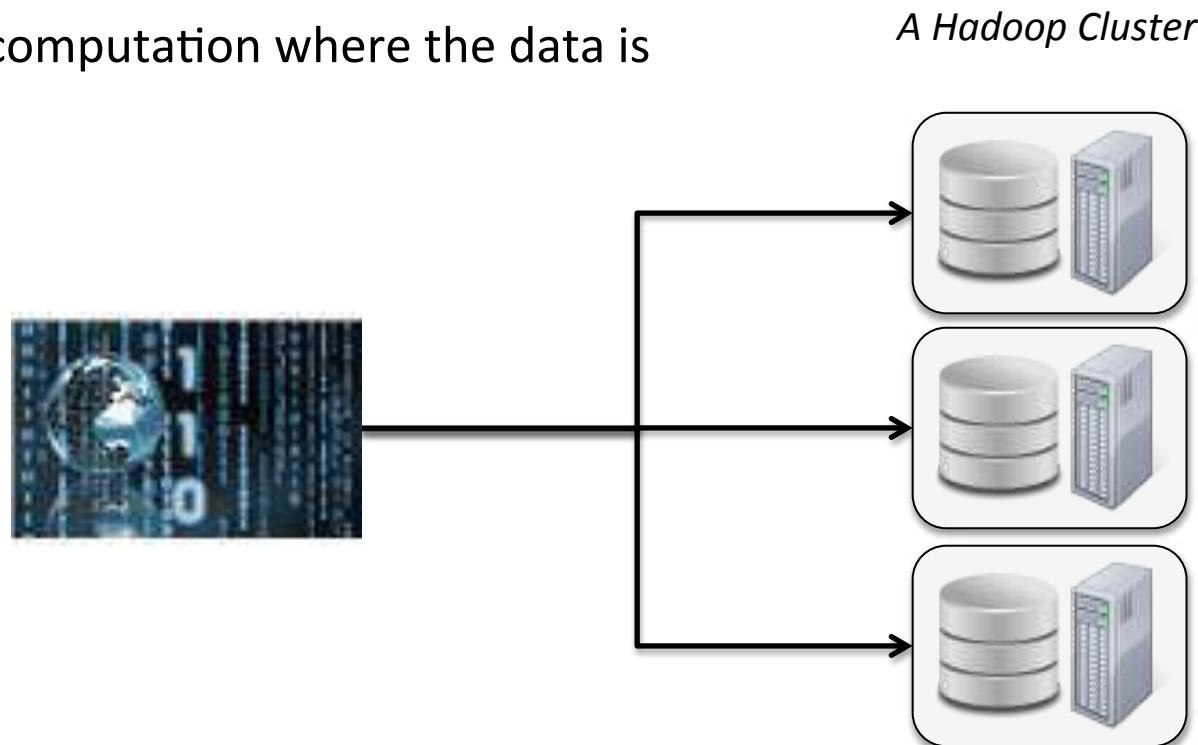
- terabytes+ a day
 - petabytes+ total

- We need a new approach...

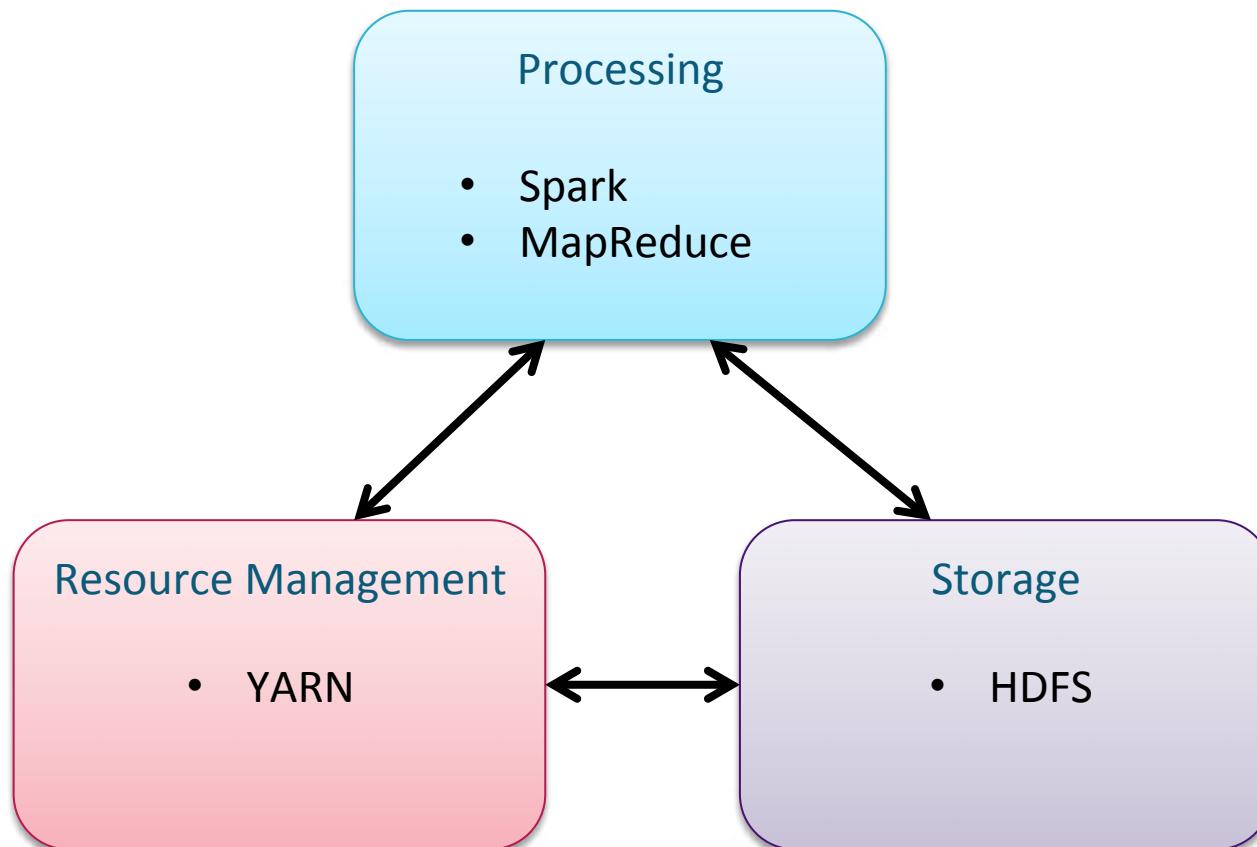


Big Data Processing with Hadoop

- **Hadoop introduced a radical new approach:**
 - Bring the program to the data rather than the data to the program
- **Based on two key concepts**
 - Distribute data when the data is stored
 - Run computation where the data is



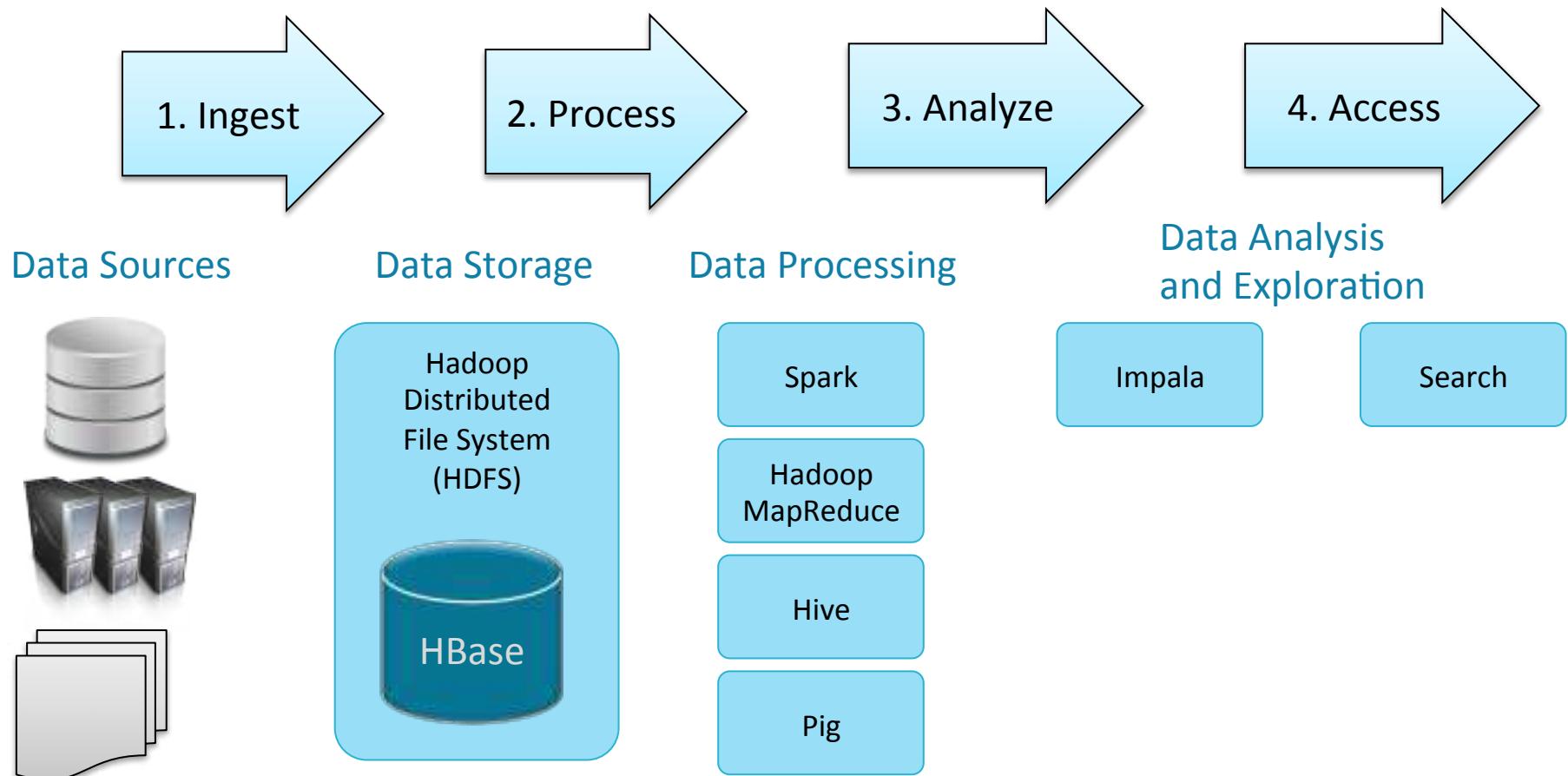
Core Hadoop



A Hadoop Cluster



Big Data Processing



Chapter Topics

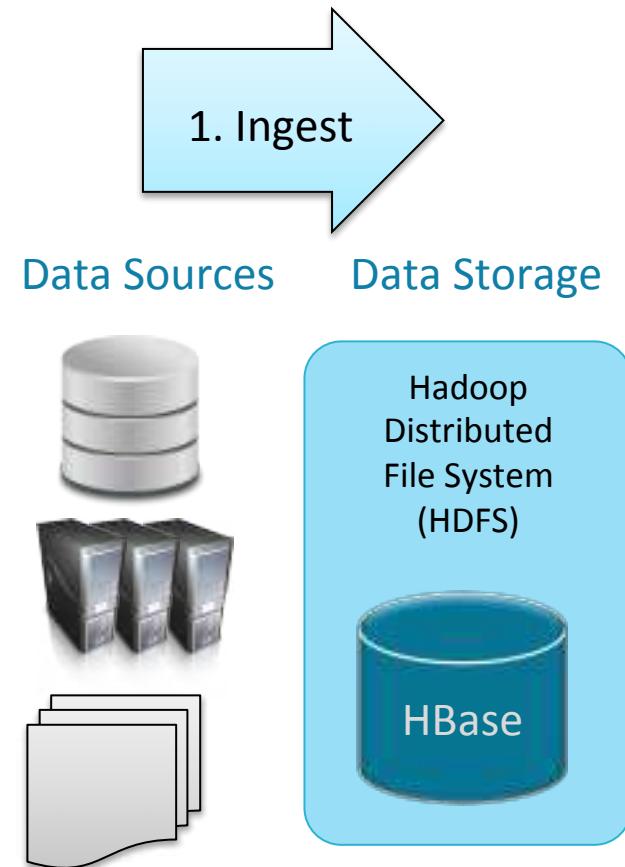
Introduction to Hadoop and the Hadoop Ecosystem

Introduction to Hadoop

- Problems with Traditional Large-scale Systems
- Hadoop!
- **Data Storage and Ingest**
- Data Processing
- Data Analysis and Exploration
- Other Ecosystem Tools
- Introduction to the Hands-On Exercises
- Conclusion

Data Ingest and Storage

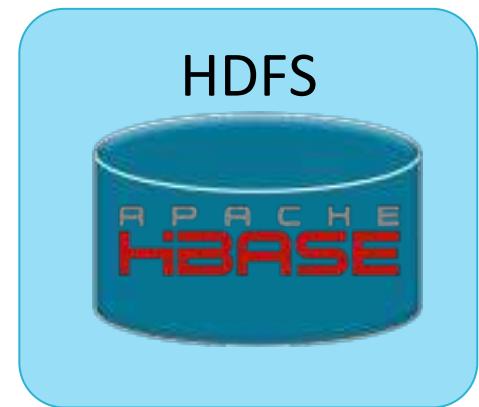
- Hadoop typically ingests data from many sources and in many formats
 - Traditional data management systems, e.g. databases
 - Logs and other machine generated data (event data)
 - Imported files



Data Storage

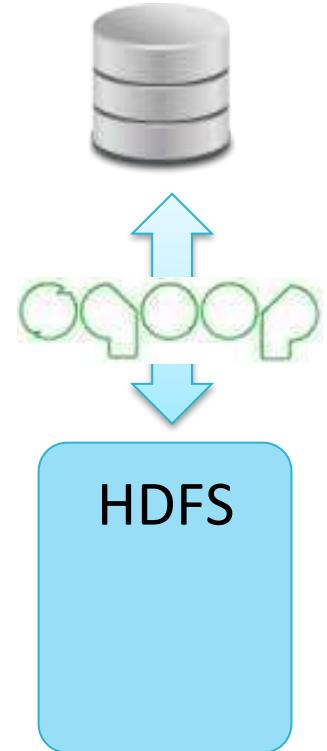
- **Hadoop Distributed File System (HDFS)**
 - HDFS is the storage layer for Hadoop
 - Provides inexpensive reliable storage for massive amounts of data on industry-standard hardware
 - Data is distributed when stored
 - **Covered later in this course**

- **Apache HBase: The Hadoop Database**
 - A NoSQL distributed database built on HDFS
 - Scales to support very large amounts of data and high throughput
 - A table can have thousands of columns
 - **Covered in depth in *Cloudera Training for Apache HBase***



Data Ingest Tools (1)

- **HDFS**
 - Direct file transfer
- **Apache Sqoop**
 - High speed import to HDFS from Relationship Database (and vice versa)
 - Supports many data storage systems
 - e.g. Netezza, Mongo, MySQL, Teradata, Oracle
 - **Covered later in this course**



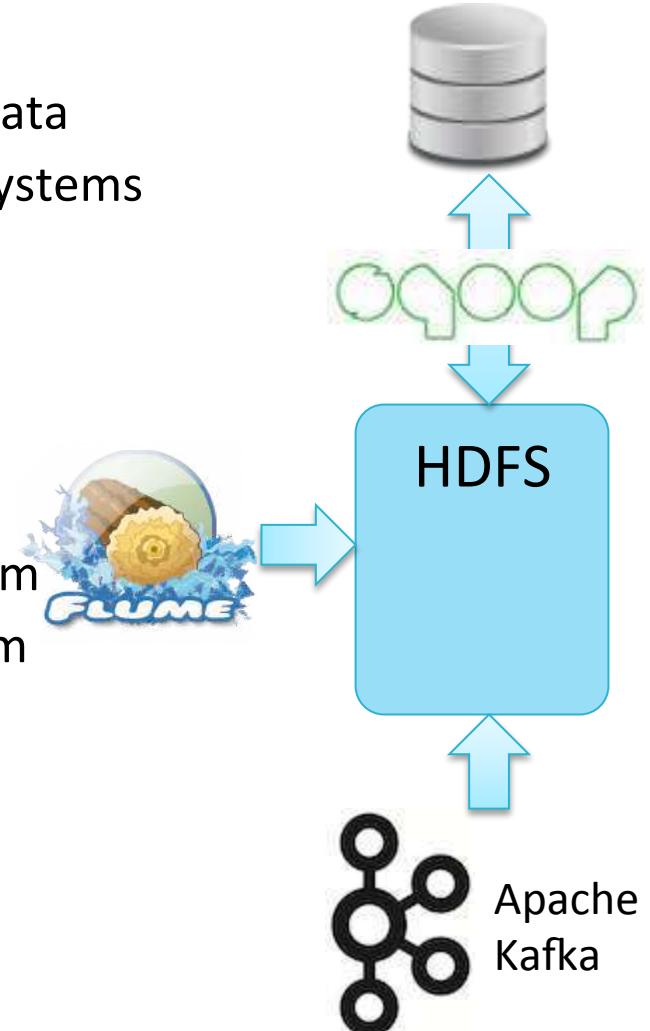
Data Ingest Tools (2)

- **Apache Flume**

- Distributed service for ingesting streaming data
- Ideally suited for event data from multiple systems
 - For example, log files
- **Covered later in this course**

- **Kafka**

- A high throughput, scalable messaging system
- Distributed, reliable publish-subscribe system
- Integrates with Flume and Spark Streaming
- **Covered in *Developer Training for Spark and Hadoop II: Advanced Techniques***



Chapter Topics

Introduction to Hadoop and the Hadoop Ecosystem

Introduction to Hadoop

- Problems with Traditional Large-scale Systems
- Hadoop!
- Data Storage and Ingest
- **Data Processing**
- Data Analysis and Exploration
- Other Ecosystem Tools
- Introduction to the Hands-On Exercises
- Conclusion

Apache Spark: An Engine For Large-scale Data Processing

- **Spark is large-scale data processing engine**
 - General purpose
 - Runs on Hadoop clusters and data in HDFS
- **Supports a wide range of workloads**
 - Machine learning
 - Business intelligence
 - Streaming
 - Batch Processing
- **This course uses Spark for data processing**



Hadoop MapReduce: The Original Hadoop Processing Engine

- **Hadoop MapReduce is the original Hadoop framework**
 - Primarily Java based
- **Based on the MapReduce programming model**
- **The core Hadoop processing engine before Spark was introduced**
- **Still the dominant technology**
 - But losing ground to Spark fast
- **Many existing tools are still built using MapReduce code**
- **Has extensive and mature fault tolerance built into the framework**



Apache Pig: Scripting for MapReduce

- Apache Pig builds on Hadoop to offer high-level data processing
 - This is an alternative to writing low-level MapReduce code
 - Pig is especially good at joining and transforming data
- The Pig interpreter runs on the client machine
 - Turns Pig Latin scripts into MapReduce or Spark jobs
 - Submits those jobs to a Hadoop cluster
 - Covered in Cloudera *Data Analyst Training*



```
people = LOAD '/user/training/customers' AS (cust_id, name);  
orders = LOAD '/user/training/orders' AS (ord_id, cust_id, cost);  
groups = GROUP orders BY cust_id;  
totals = FOREACH groups GENERATE group, SUM(orders.cost) AS t;  
result = JOIN totals BY group, people BY cust_id;  
DUMP result;
```

Chapter Topics

Introduction to Hadoop and the Hadoop Ecosystem

Introduction to Hadoop

- Problems with Traditional Large-scale Systems
- Hadoop!
- Data Storage and Ingest
- Data Processing
- **Data Analysis and Exploration**
- Other Ecosystem Tools
- Introduction to the Hands-On Exercises
- Conclusion

Cloudera Impala: High Performance SQL

- **Impala is a high-performance SQL engine**
 - Runs on Hadoop clusters
 - Data stored in HDFS files
 - Inspired by Google's Dremel project
 - Very low latency – measured in milliseconds
 - Ideal for interactive analysis
- **Impala supports a dialect of SQL (Impala SQL)**
 - Data in HDFS modeled as database tables
- **Impala was developed by Cloudera**
 - 100% open source, released under the Apache software license
- **Impala is used for data analysis in this course**



Apache Hive: SQL on MapReduce

- **Hive is an abstraction layer on top of Hadoop**
 - Hive uses a SQL-like language called HiveQL
 - Similar to Impala SQL
 - Useful for data processing and ETL
 - Impala is preferred for ad hoc analytics
- **Hive executes queries using MapReduce**
 - Hive on Spark is available for early adopters; not yet recommended for production
- **Hive can optionally be used for data analysis in this course**
 - Covered in more depth in *Cloudera Data Analyst Training*



Cloudera Search: A Platform For Data Exploration

- Interactive full-text search for data in a Hadoop cluster
- Allows non-technical users to access your data
 - Nearly everyone can use a search engine
- Cloudera Search enhances Apache Solr
 - Integrates Solr with HDFS, MapReduce, HBase, and Flume
 - Supports file formats widely used with Hadoop
 - Dynamic Web-based dashboard interface with Hue
 - Apache Sentry based security
- Cloudera Search is 100% open source
- Covered in depth in *Cloudera Search Training*



Chapter Topics

Introduction to Hadoop and the Hadoop Ecosystem

Introduction to Hadoop

- Problems with Traditional Large-scale Systems
- Hadoop!
- Data Storage and Ingest
- Data Processing
- Data Analysis and Exploration
- **Other Ecosystem Tools**
- Introduction to the Hands-On Exercises
- Conclusion

Hue: The UI for Hadoop

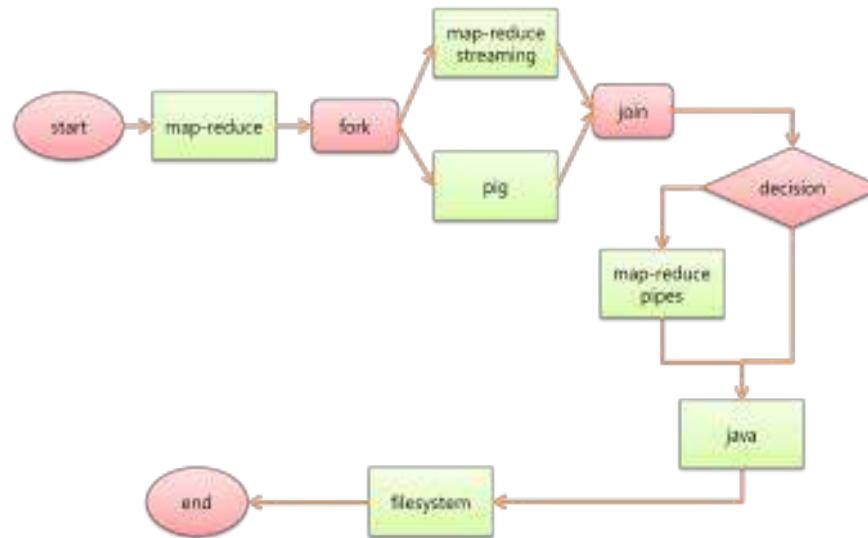
- **Hue = Hadoop User Experience**
- **Hue provides a Web front-end to a Hadoop**
 - Upload and browse data
 - Query tables in Impala and Hive
 - Run Spark and Pig jobs and workflows
 - Search
 - And much more
- **Makes Hadoop easier to use**
- **Hue is 100% open-source**
- **Created by Cloudera**
 - Open source, released under Apache license
- **Hue is used throughout this course**



Apache Oozie: Workflow Management



- **Oozie**
 - Workflow engine for Hadoop jobs
 - Defines dependencies between jobs
- **The Oozie server submits the jobs to the server in the correct sequence**



Apache Sentry: Hadoop Security

- **Sentry provides fine-grained access control (authorization) to various Hadoop ecosystem components**
 - Impala
 - Hive
 - Cloudera Search
 - HDFS
- **In conjunction with Kerberos authentication, Sentry authorization provides a complete cluster security solution**
- **Created by Cloudera**
 - Now an open-source Apache project



Chapter Topics

Introduction to Hadoop and the Hadoop Ecosystem

Introduction to Hadoop

- Problems with Traditional Large-scale Systems
- Hadoop!
- Data Storage and Ingest
- Data Processing
- Data Analysis and Exploration
- Other Ecosystem Tools
- **Introduction to the Hands-On Exercises**
- Conclusion

Introduction to the Hands-On Exercises

- **The best way to learn is to *do!***
- **Most topics in this course have Hands-On Exercises to practice the skills you have learned in the course**

Scenario Explanation (1)

- **The exercises are based on a hypothetical scenario**
 - However, the concepts apply to nearly any organization
- **Loudacre Mobile is a (fictional) fast-growing wireless carrier**
 - Provides mobile service to customers throughout western USA



Scenario Explanation (2)

- **Loudacre needs to migrate their existing infrastructure to Hadoop**
 - The size and velocity and their data has exceeded their ability to process and analyze their data
- **Loudacre data sources**
 - MySQL database – customer account data (name, address, phone numbers, devices)
 - Apache web server logs from Customer Service site
 - HTML files – Knowledge base articles
 - XML files – Device activation records
 - Real-time device status logs
 - Base stations – cell tower locations

Introduction to Exercises: Getting Started

- **Instructions are in the Hands-On Exercise Manual**
- **Start with**
 - General Notes
 - Setting Up
 - Run setup script for the course

Introduction to Exercises: Classroom Virtual Machine

- Your virtual machine
 - Log in as user **training** (password **training**)
 - Pre-installed and configured with
 - Spark and CDH (Cloudera's Distribution, including Apache Hadoop)
 - Various tools including Firefox, gedit, Emacs, Eclipse, and Maven
- Training materials: **~/training_materials/dev1** folder on the VM
 - **examples** – all the example code in this course
 - **exercises** – starter files, scripts and solutions for the Hands-On Exercises
 - **scripts** – course setup scripts
- Course data: **~/training_materials/data**

Chapter Topics

Introduction to Hadoop and the Hadoop Ecosystem

Introduction to Hadoop

- Problems with Traditional Large-scale Systems
- Hadoop!
- Data Storage and Ingest
- Data Processing
- Data Analysis and Exploration
- Other Ecosystem Tools
- Introduction to the Hands-On Exercises
- **Conclusion**

Essential Points

- **Hadoop is a framework for distributed storage and processing**
- **Core Hadoop includes HDFS for storage and YARN for cluster resource management**
- **The Hadoop ecosystem includes many components for**
 - Ingesting data (Flume, Sqoop, Kafka)
 - Storing data (HDFS, HBase)
 - Processing data (Spark, Hadoop MapReduce, Pig)
 - Modeling data as tables for SQL access (Impala, Hive)
 - Exploring data (Hue, Search)
 - Protecting Data (Sentry)
- **This course introduces most of the key Hadoop infrastructure**
- **Hands-On Exercises let you practice and refine your Hadoop skills!**

Bibliography

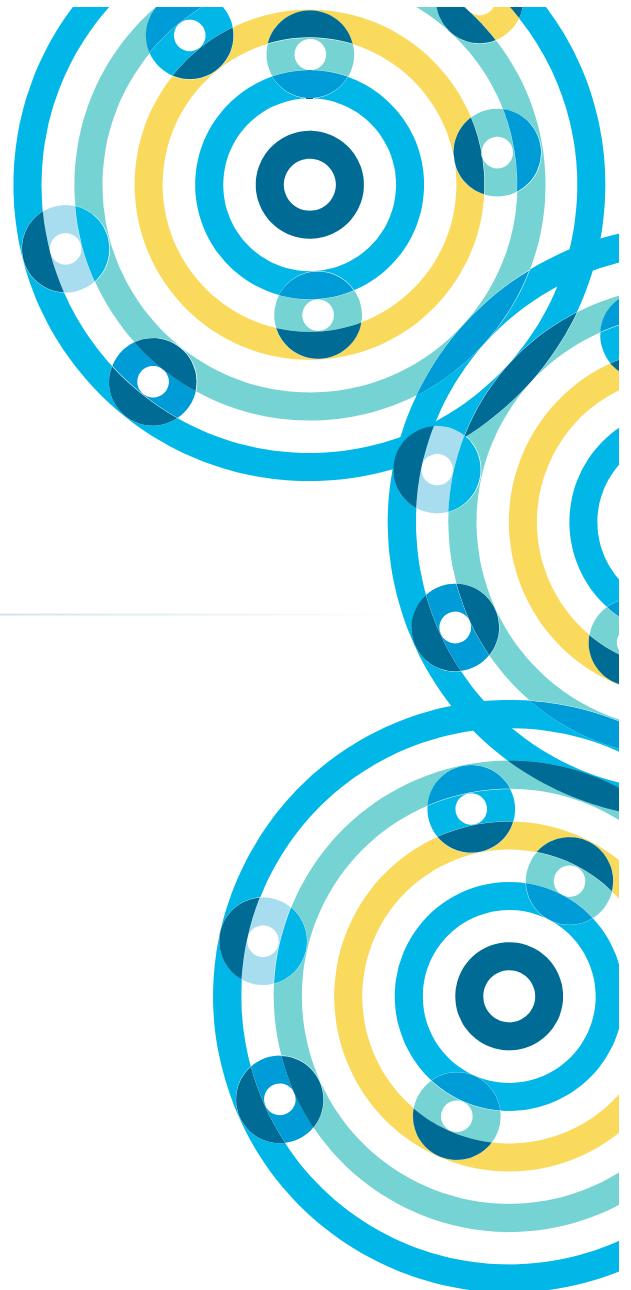
The following offer more information on topics discussed in this chapter

- ***Hadoop: The Definitive Guide* (published by O'Reilly)**
 - <http://tiny.cloudera.com/hadooptdg>
- ***Cloudera Essentials for Apache Hadoop – free online training***
 - <http://tiny.cloudera.com/esscourse>



Hadoop Architecture and HDFS

Chapter 3



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- **Hadoop Architecture and HDFS**
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

Distributed Data Processing with
Spark

Course Conclusion

Hadoop Architecture and HDFS

In this chapter you will learn

- **How Hadoop Distributed File System stores data across a cluster**
- **How to use HDFS using the Hue File Browser or the `hdfs` command**
- **How Hadoop YARN provides cluster resource management for distributed data processing**
- **How to use Hue, the YARN Web UI or the `yarn` command to monitor your cluster**

Chapter Topics

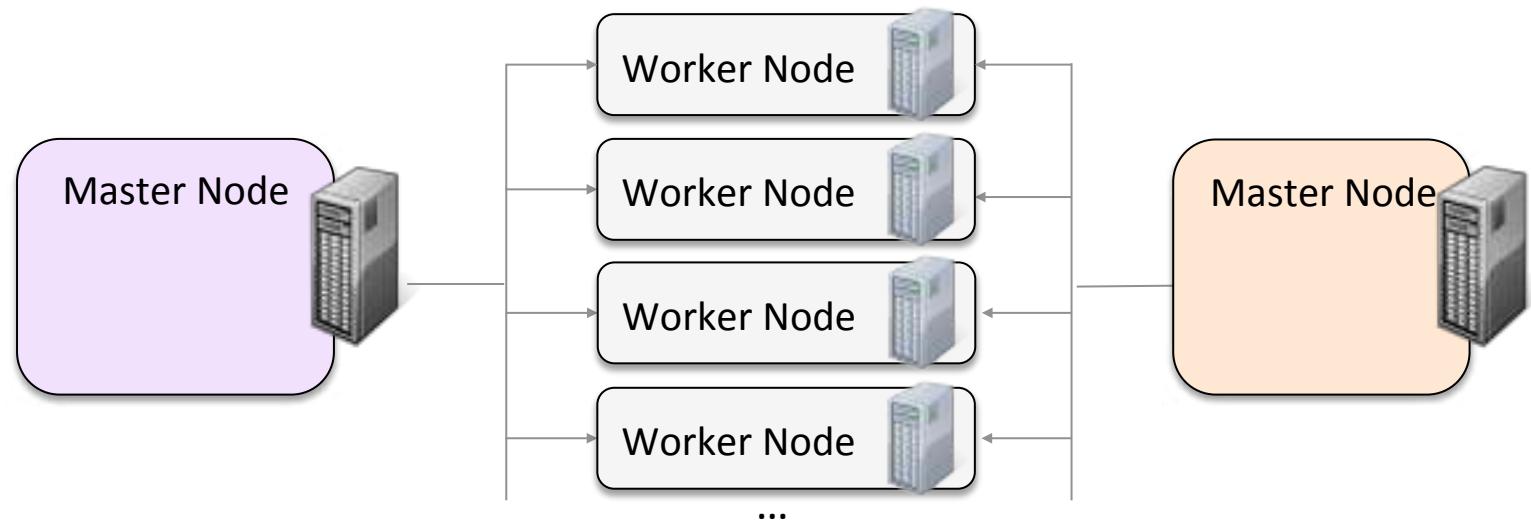
Hadoop Architecture and HDFS

Introduction to Hadoop

- **Distributed Processing on a Cluster**
 - Storage: HDFS Architecture
 - Storage: Using HDFS
 - Hands-on Exercises: Access HDFS with Command Line and Hue
 - Resource Management: YARN Architecture
 - Resource Management: Working with YARN
 - Conclusion
 - Hands-On Exercises: Run a YARN Job

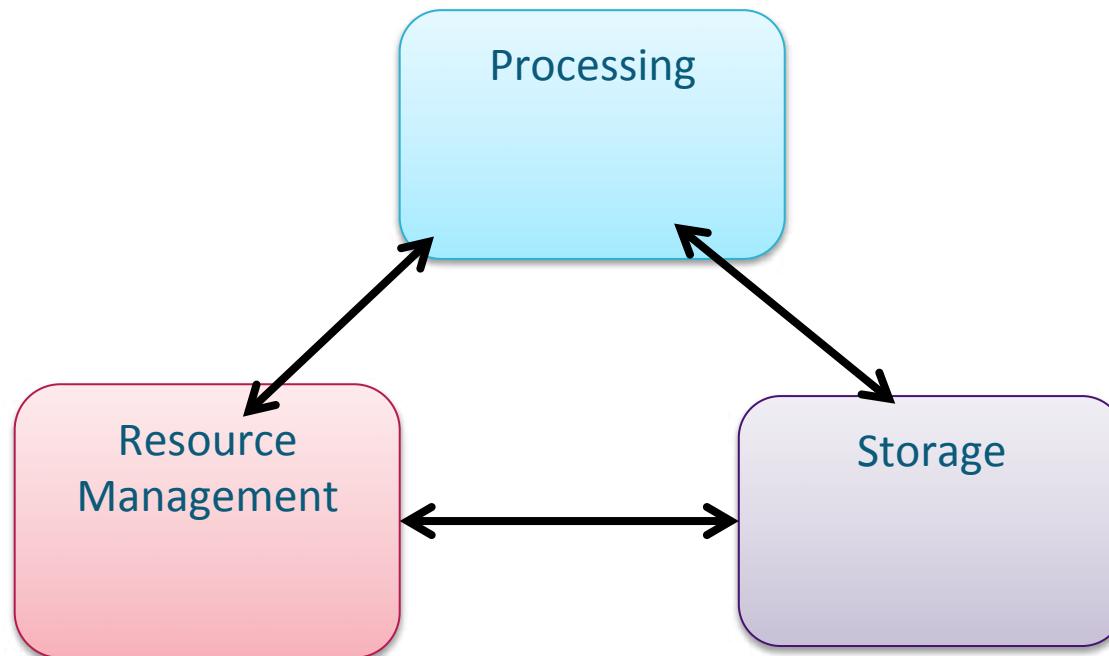
Hadoop Cluster Terminology

- A **cluster** is a group of computers working together
 - Provides data storage, data processing, and resource management
- A **node** is an individual computer in the cluster
 - Master nodes manage distribution of work and data to *worker* or *slave* nodes
- A **daemon** is a program running on a node
 - Each performs different functions in the cluster



Cluster Components

- Three main components of a cluster
- Work together to provide distributed data processing
- We will start with the Storage component
 - HDFS



Chapter Topics

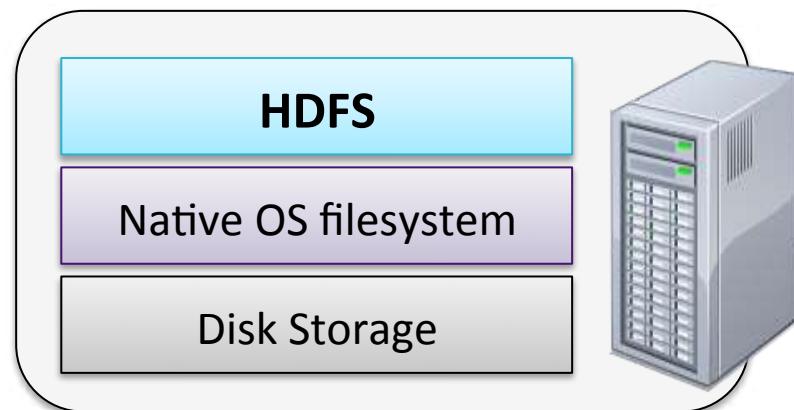
Hadoop Architecture and HDFS

Introduction to Hadoop

- Distributed Processing on a Cluster
- **Storage: HDFS Architecture**
- Storage: Using HDFS
- Hands-on Exercises: Access HDFS with Command Line and Hue
- Resource Management: YARN Architecture
- Resource Management: Working with YARN
- Conclusion
- Hands-On Exercises: Run a YARN Job

HDFS Basic Concepts (1)

- **HDFS is a filesystem written in Java**
 - Based on Google's GFS
- **Sits on top of a native filesystem**
 - Such as ext3, ext4, or xfs
- **Provides redundant storage for massive amounts of data**
 - Using readily-available, industry-standard computers

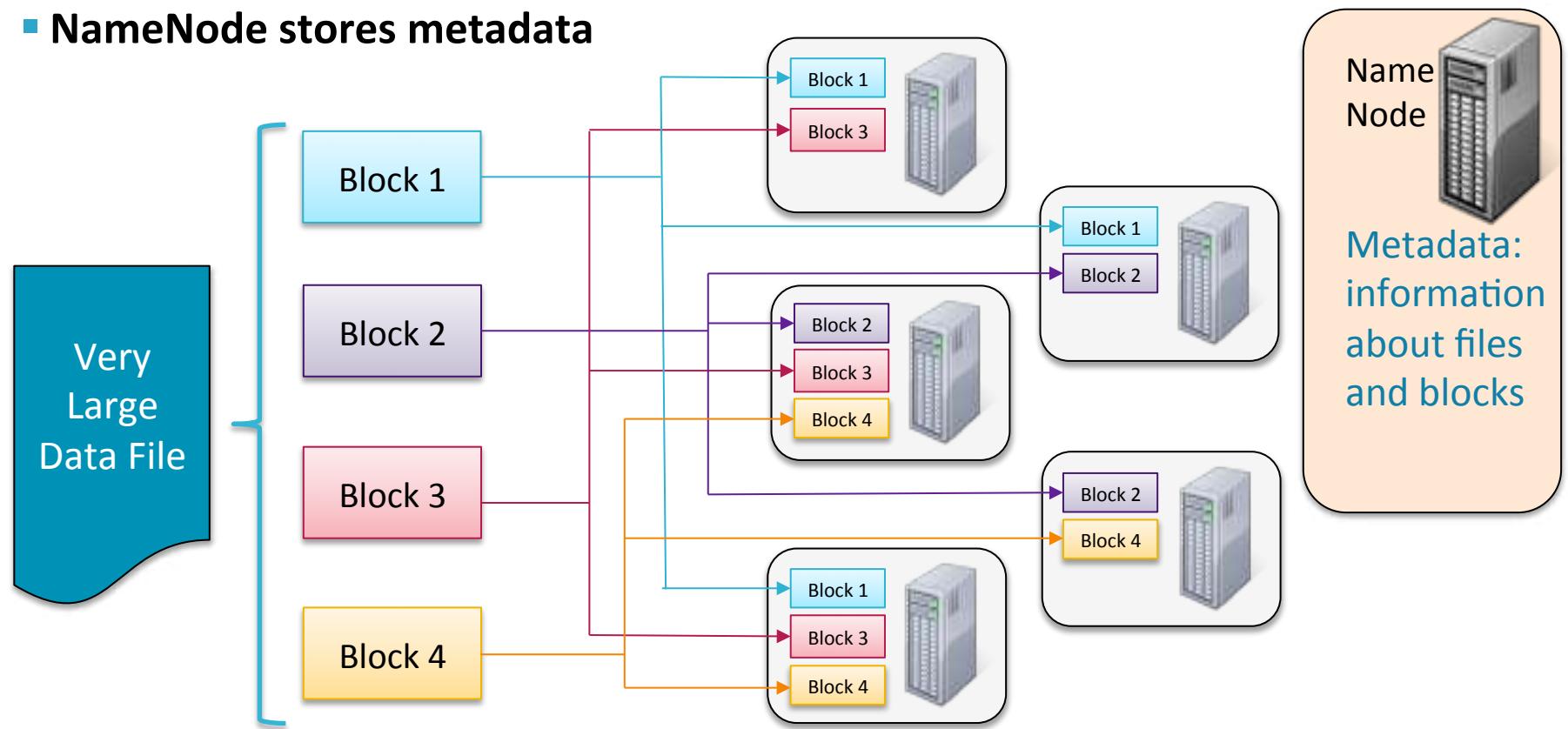


HDFS Basic Concepts (2)

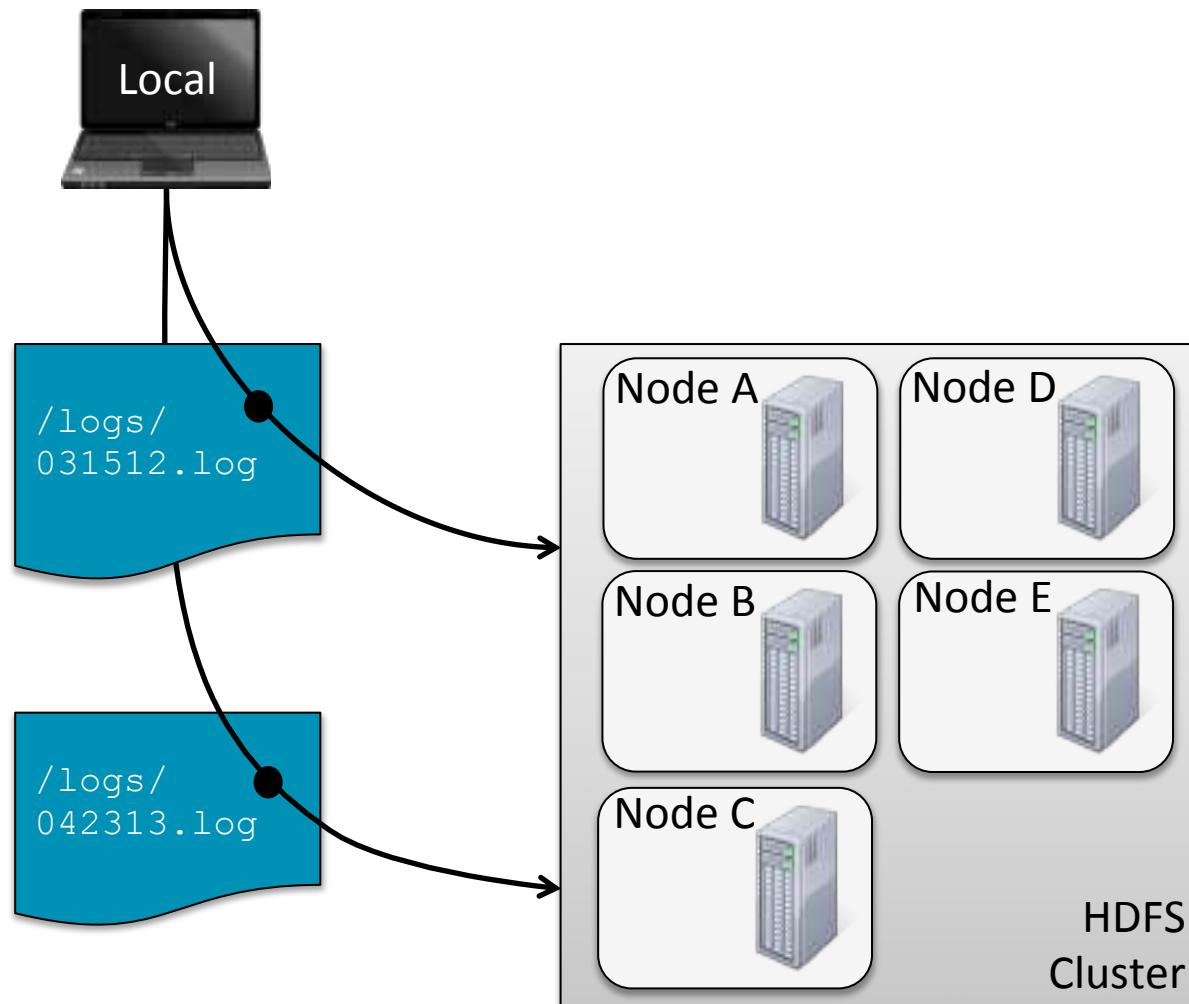
- **HDFS performs best with a ‘modest’ number of large files**
 - Millions, rather than billions, of files
 - Each file typically 100MB or more
- **Files in HDFS are ‘write once’**
 - No random writes to files are allowed
- **HDFS is optimized for large, streaming reads of files**
 - Rather than random reads

How Files Are Stored

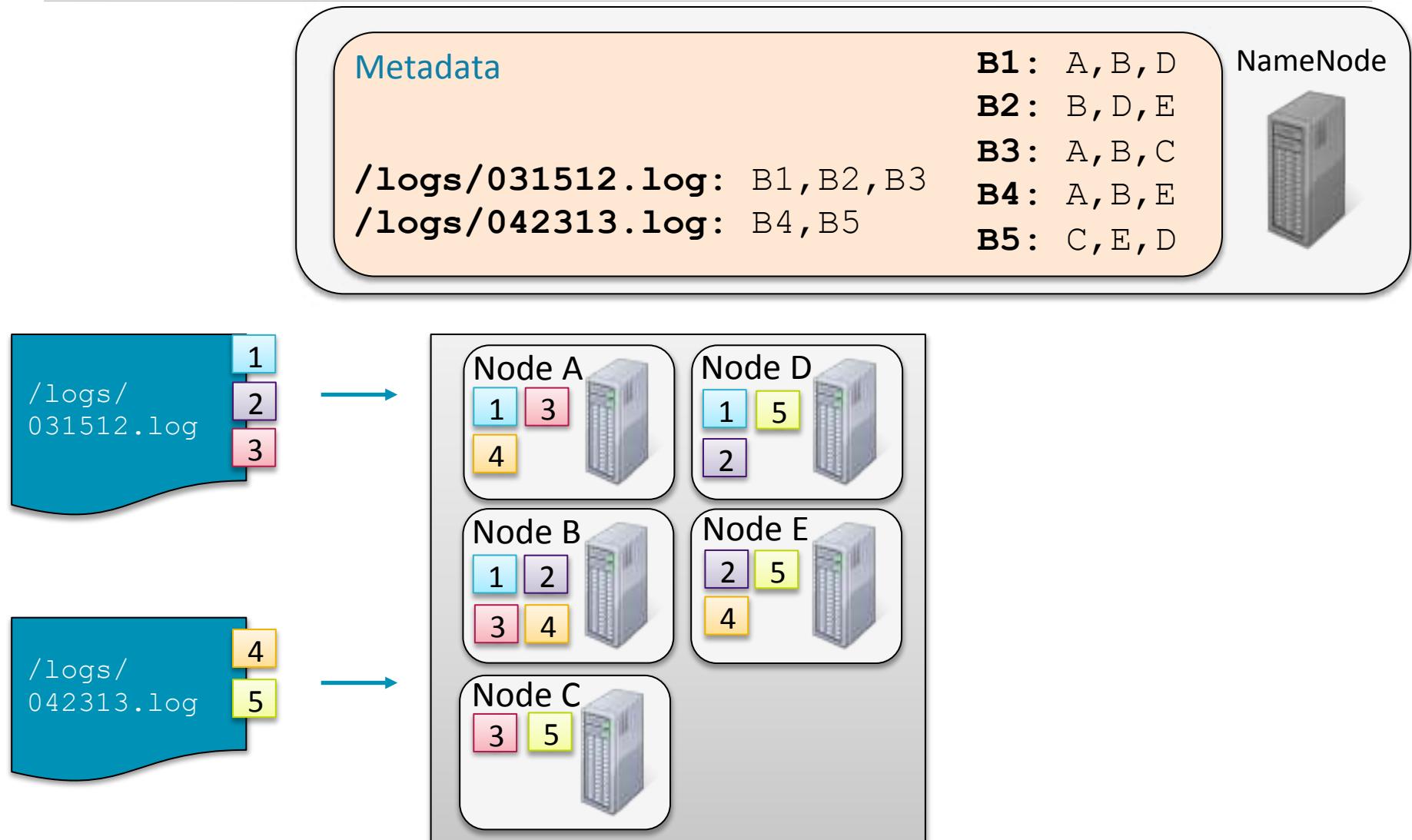
- Data files are split into 128MB blocks which are distributed at load time
- Each block is replicated on multiple data nodes (default 3x)
- NameNode stores metadata



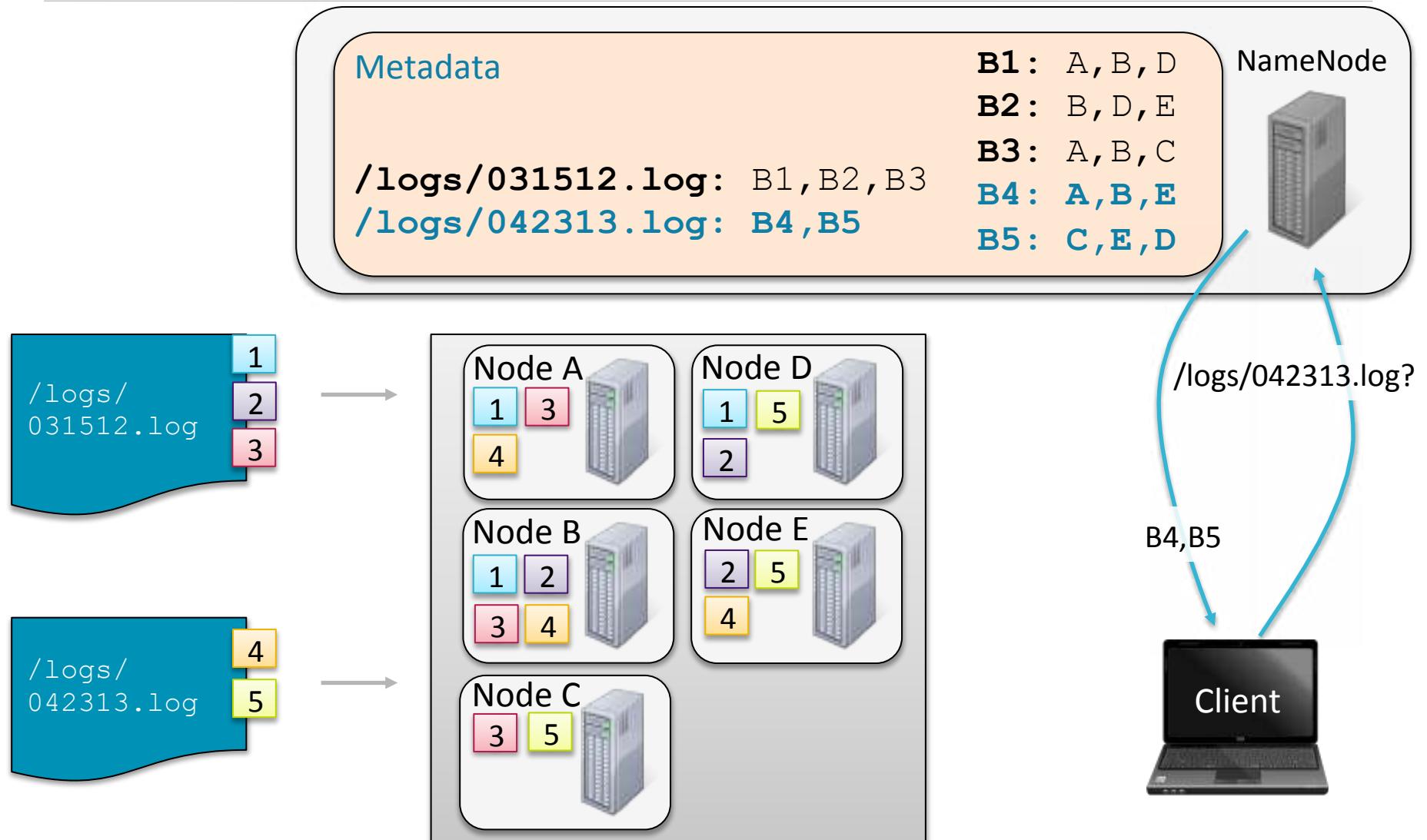
Example: Storing and Retrieving Files (1)



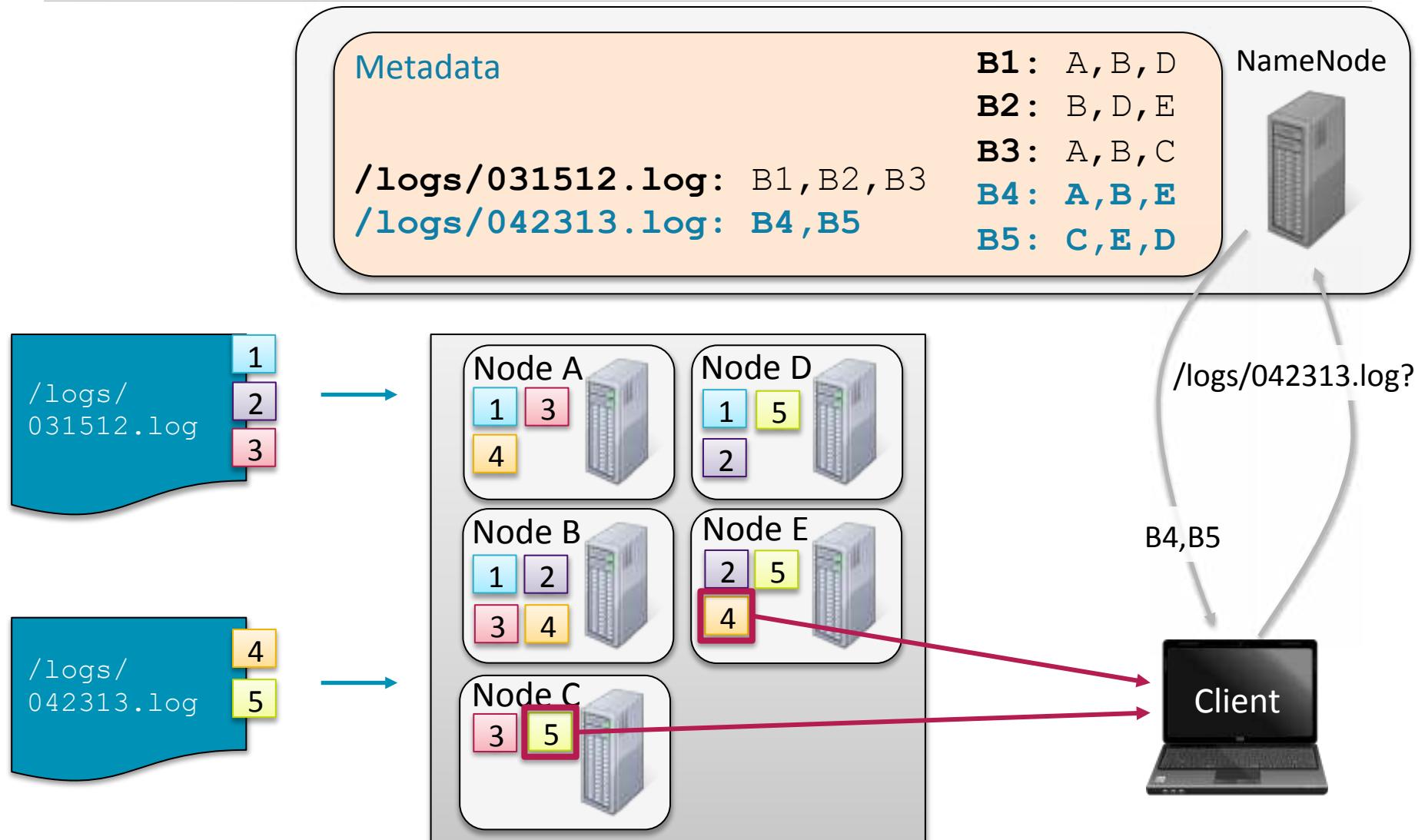
Example: Storing and Retrieving Files (2)



Example: Storing and Retrieving Files (3)



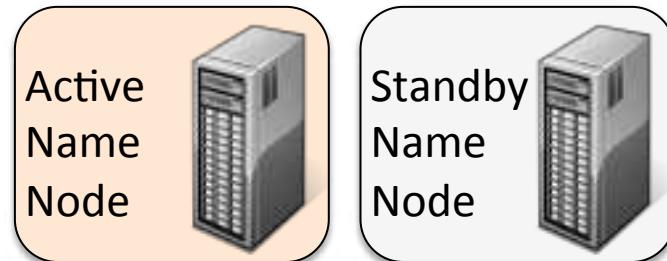
Example: Storing and Retrieving Files (4)



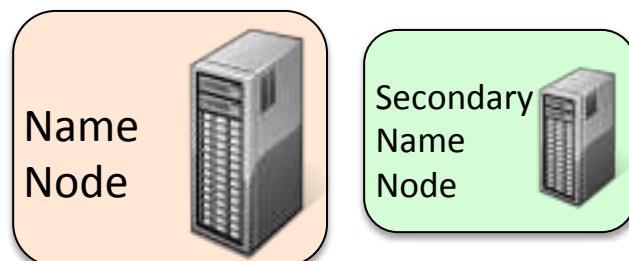
HDFS NameNode Availability

- **The NameNode daemon must be running at all times**
 - If the NameNode stops, the cluster becomes inaccessible

- **HDFS is typically set up for High Availability**
 - Two NameNodes: Active and Standby



- **Small clusters may use ‘Classic mode’**
 - One NameNode
 - One “helper” node called the Secondary NameNode
 - Bookkeeping, not backup



Chapter Topics

Hadoop Architecture and HDFS

Introduction to Hadoop

- Distributed Processing on a Cluster
- Storage: HDFS Architecture
- **Storage: Using HDFS**
- Hands-on Exercises: Access HDFS with Command Line and Hue
- Resource Management: YARN Architecture
- Resource Management: Working with YARN
- Conclusion
- Hands-On Exercises: Run a YARN Job

Options for Accessing HDFS

- From the command line

- FsShell:

- \$ **hdfs dfs**

- In Spark

- By URI, e.g.

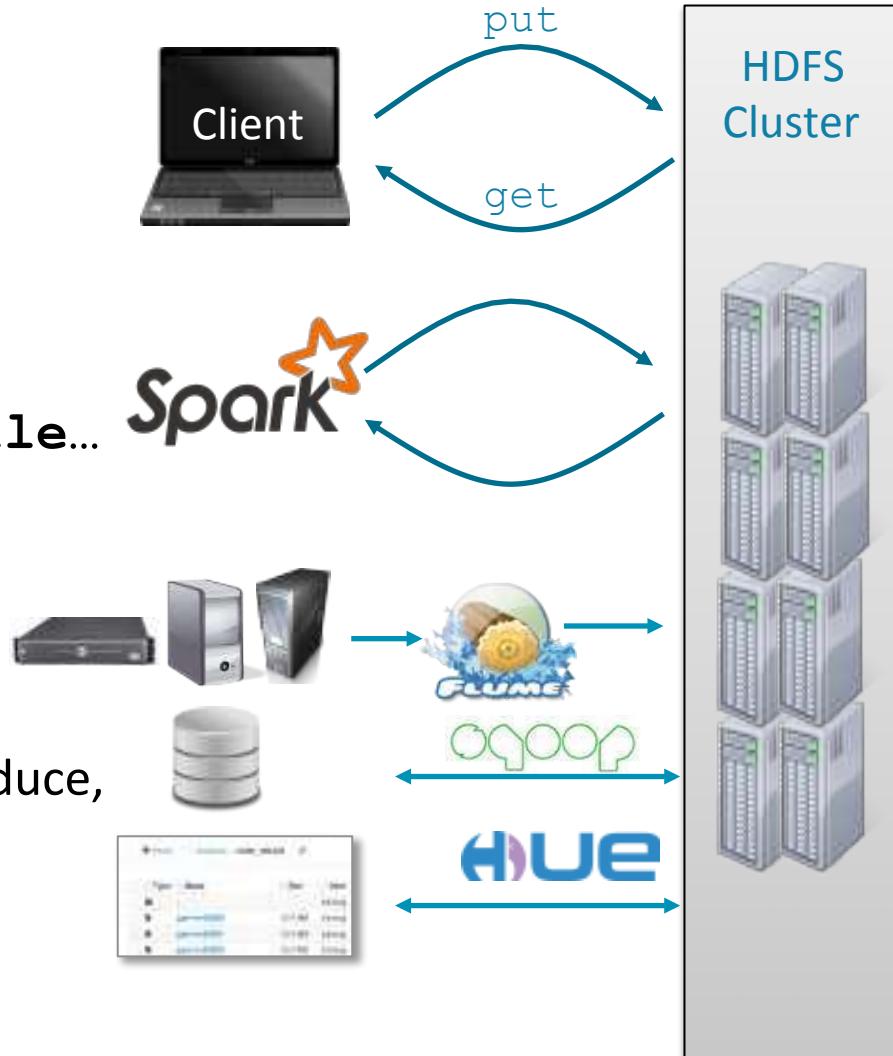
- hdfs://nnhost:port/file...**

- Other programs

- Java API

- Used by Hadoop MapReduce,
Impala, Hue, Sqoop,
Flume, etc.

- RESTful interface



HDFS Command Line Examples (1)

- Copy file `foo.txt` from local disk to the user's directory in HDFS

```
$ hdfs dfs -put foo.txt foo.txt
```

– This will copy the file to `/user/username/foo.txt`

- Get a directory listing of the user's home directory in HDFS

```
$ hdfs dfs -ls
```

- Get a directory listing of the HDFS root directory

```
$ hdfs dfs -ls /
```

HDFS Command Line Examples (2)

- Display the contents of the HDFS file `/user/fred/bar.txt`

```
$ hdfs dfs -cat /user/fred/bar.txt
```

- Copy that file to the local disk, named as `baz.txt`

```
$ hdfs dfs -get /user/fred/bar.txt baz.txt
```

- Create a directory called `input` under the user's home directory

```
$ hdfs dfs -mkdir input
```

Note: `copyFromLocal` is a synonym for `put`; `copyToLocal` is a synonym for `get`

HDFS Command Line Examples (3)

- Delete the directory `input_old` and all its contents

```
$ hdfs dfs -rm -r input_old
```

The Hue HDFS File Browser

- **The File Browser in Hue lets you view and manage your HDFS directories and files**
 - Create, move, rename, modify, upload, download and delete directories and files
 - View file contents

The screenshot shows the Hue HDFS File Browser interface. At the top, there is a navigation bar with links for 'Query Editors', 'Data Browsers', 'Workflows', 'Search', 'File Browser' (which is highlighted with a red box), 'Job Browser', and user information. Below the navigation bar, the title 'File Browser' is displayed with a blue icon. The main area shows a search bar, an 'Actions' dropdown, and a 'Move to trash' button. The path 'Home / user / training' is shown, along with a edit icon. On the right, there are 'History' and 'Trash' links. The main content area displays a table of files in the 'training' directory:

Name	Size	User	Group	Permissions	Date
..		hdfs	supergroup	drwxrwxrwx	March 03, 2015 11:44 AM
.		training	supergroup	drwxrwxrwx	February 25, 2015 01:48 PM

At the bottom, there are buttons for 'Show 45' items, a page number '1 of 1', and navigation icons.

HDFS Recommendations

- **HDFS is a repository for all your data**
 - Structure and organize carefully!
- **Best practices include**
 - Define a standard directory structure
 - Include separate locations for staging data
- **Example organization**
 - **/user** / ... – data and configuration belonging only to a single user
 - **/etl** – Work in progress in Extract/Transform/Load stage
 - **/tmp** – Temporary generated data shared between users
 - **/data** – Data sets that are processed and available across the organization for analysis
 - **/app** – Non-data files such as configuration, JAR files, SQL files, etc.

Chapter Topics

Hadoop Architecture and HDFS

Introduction to Hadoop

- Distributed Processing on a Cluster
- Storage: HDFS Architecture
- Storage: Using HDFS
- **Hands-on Exercises: Access HDFS with Command Line and Hue**
- Resource Management: YARN Architecture
- Resource Management: Working With YARN
- Conclusion
- Hands-On Exercises: Run a YARN Job

Hands-On Exercise: Access HDFS with Command Line and Hue

- **In this exercise you will**
 - Create a `/loudacre` base directory for course exercises
 - Practice uploading and viewing data files
- **Please refer to the Hands-On Exercise Manual**

Chapter Topics

Hadoop Architecture and HDFS

Introduction to Hadoop

- Distributed Processing on a Cluster
- Storage: HDFS Architecture
- Storage: Using HDFS
- Hands-on Exercises: Access HDFS with Command Line and Hue
- **Resource Management: YARN Architecture**
- Resource Management: Working With YARN
- Conclusion
- Hands-On Exercises: Run a YARN Job

What is YARN?

- **YARN = Yet Another Resource Negotiator**
- **YARN is the Hadoop processing layer that contains**
 - A resource manager
 - A job scheduler
- **YARN allows multiple data processing engines to run on a single Hadoop cluster**
 - Batch programs (e.g. Spark, MapReduce)
 - Interactive SQL (e.g. Impala)
 - Advanced analytics (e.g. Spark, Impala)
 - Streaming (e.g. Spark Streaming)

YARN Daemons

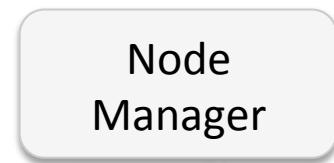
- **Resource Manager (RM)**

- Runs on master node
- Global resource scheduler
- Arbitrates system resources between competing applications
- Has a pluggable scheduler to support different algorithms (capacity, fair scheduler, etc.)



- **Node Manager (NM)**

- Runs on slave nodes
- Communicates with RM



Running an Application in YARN

- **Containers**

- Created by the RM upon request
- Allocate a certain amount of resources (memory, CPU) on a slave node
- Applications run in one or more containers



Container

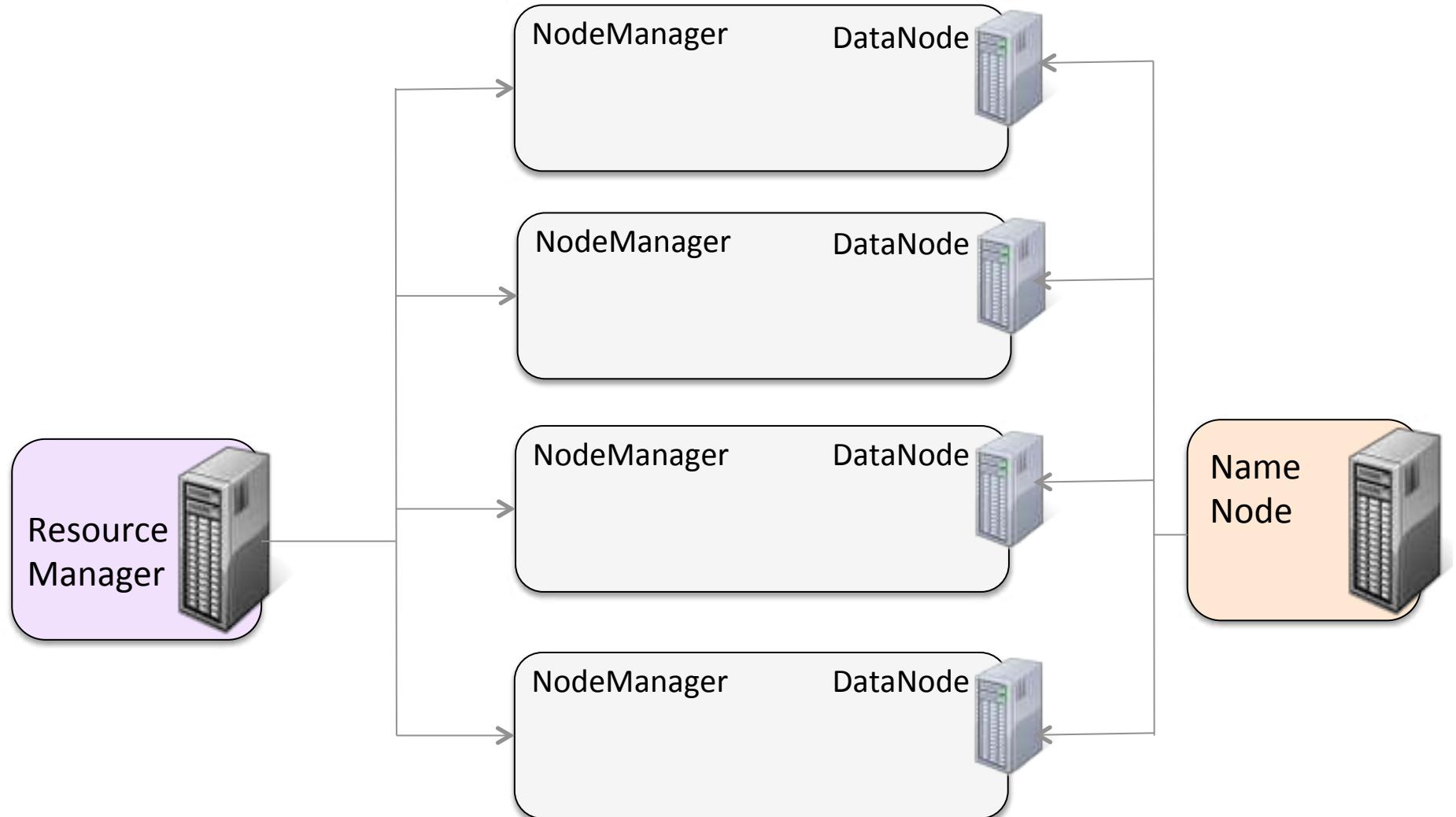
- **Application Master (AM)**

- One per application
- Framework/application specific
- Runs in a container
- Requests more containers to run application tasks

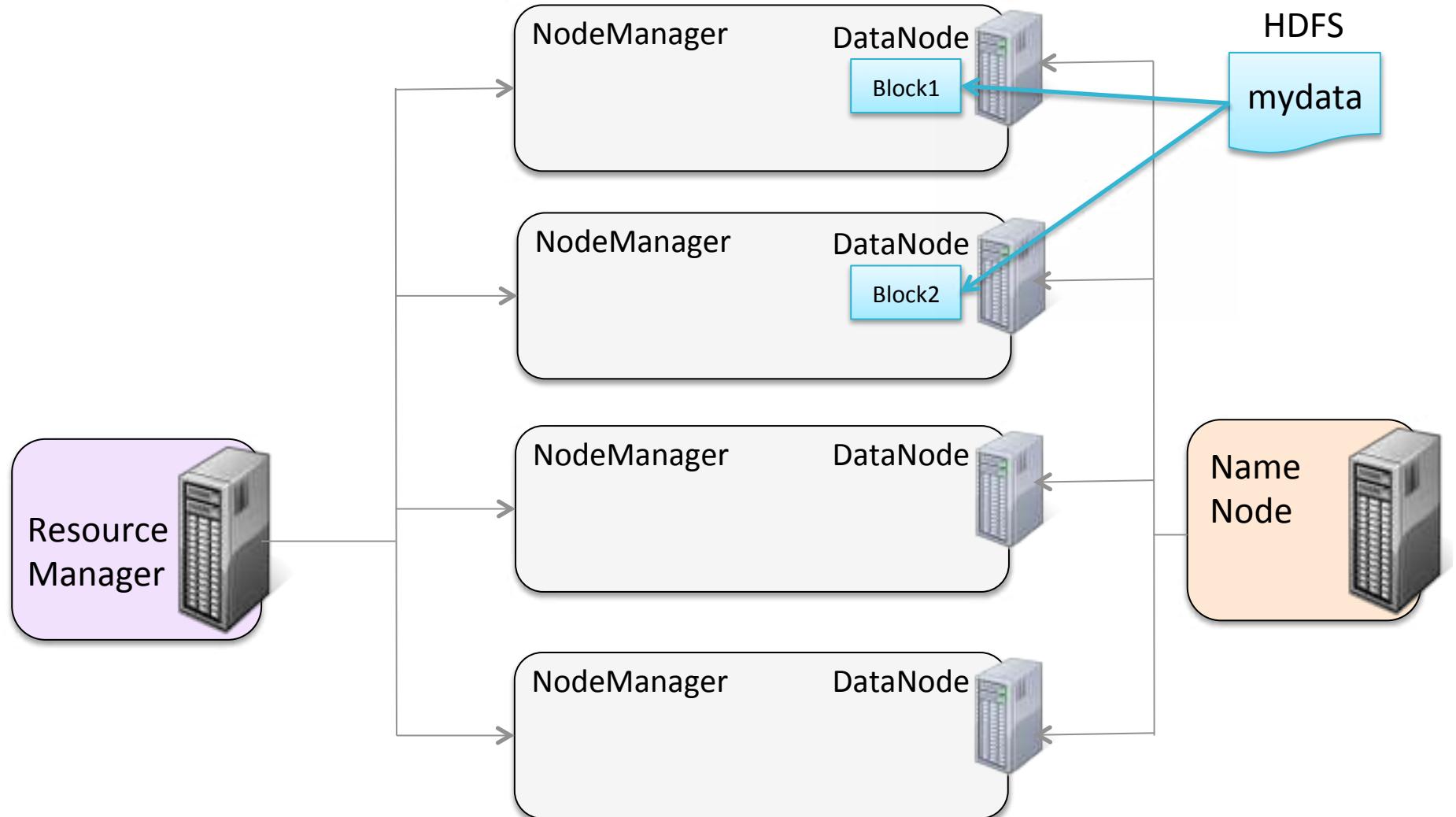


Application
Master

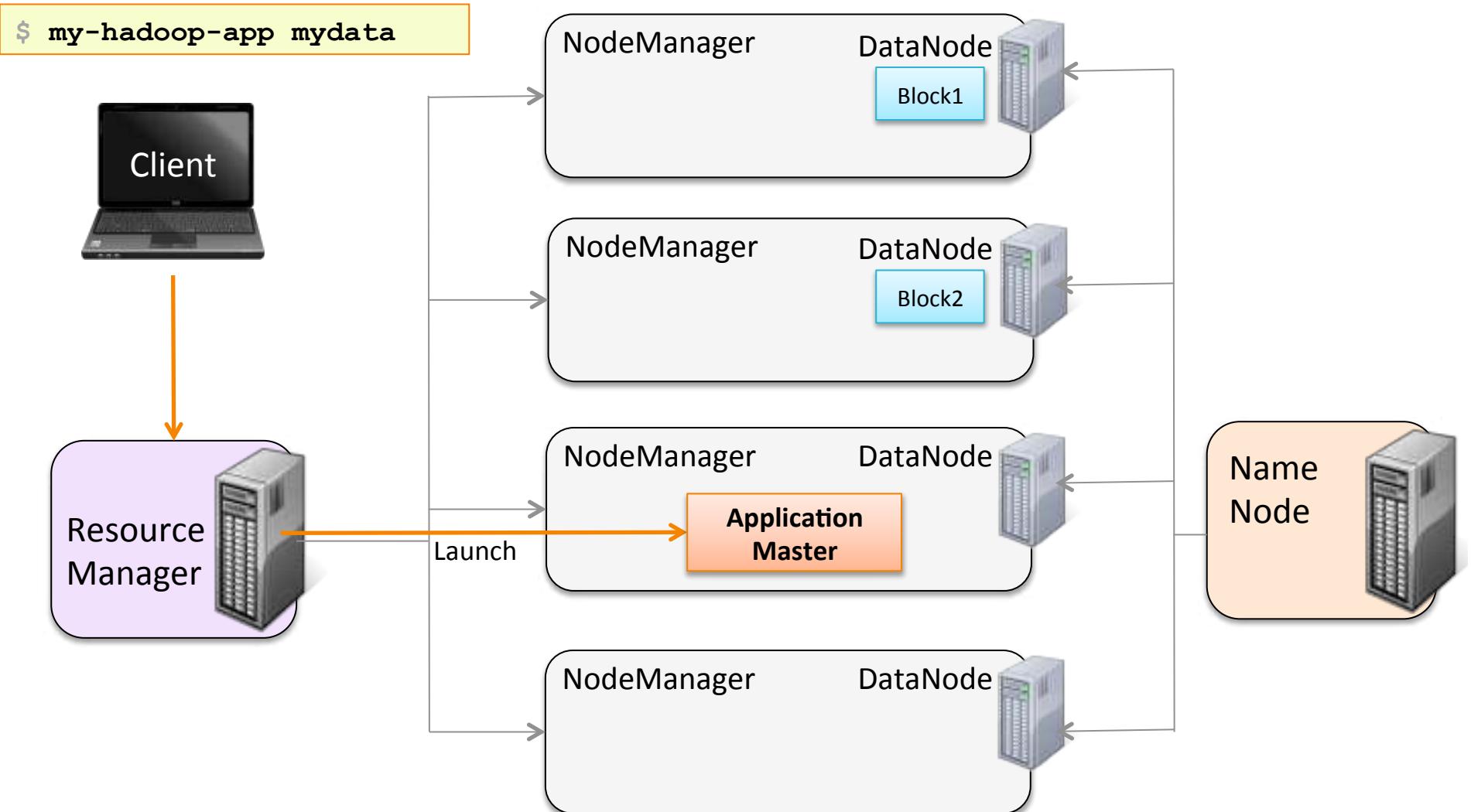
Running an Application on YARN (1)



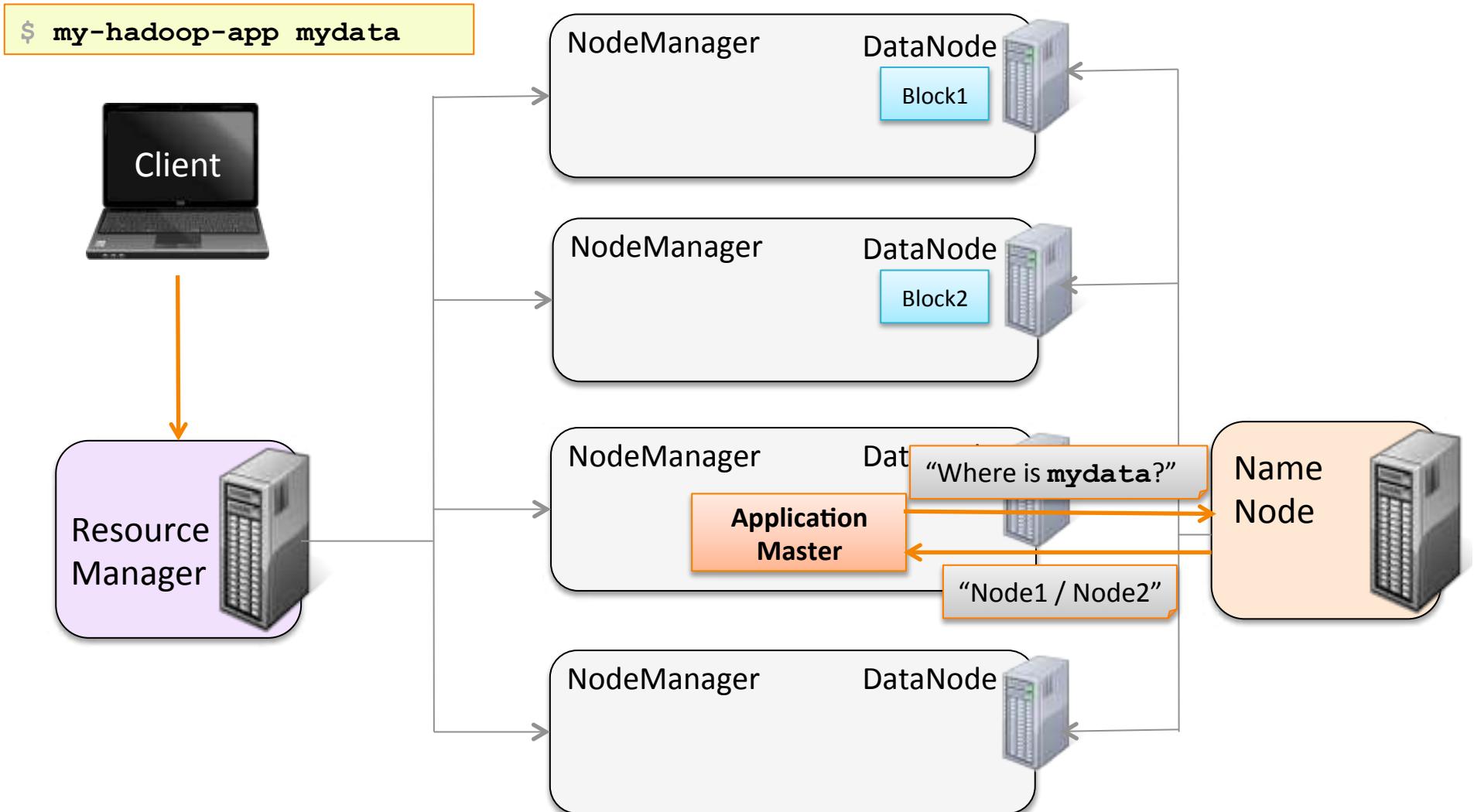
Running an Application on YARN (2)



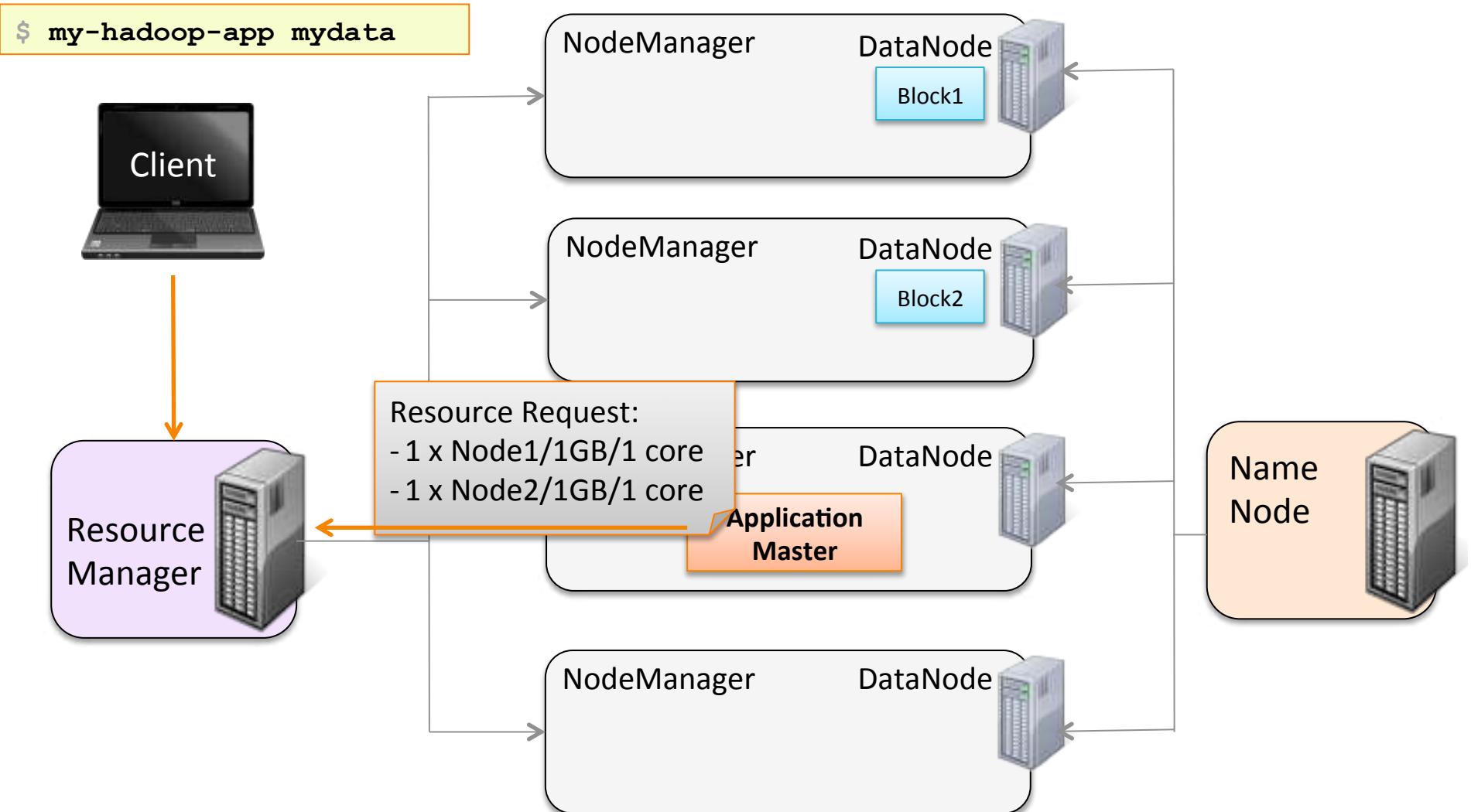
Running an Application on YARN (3)



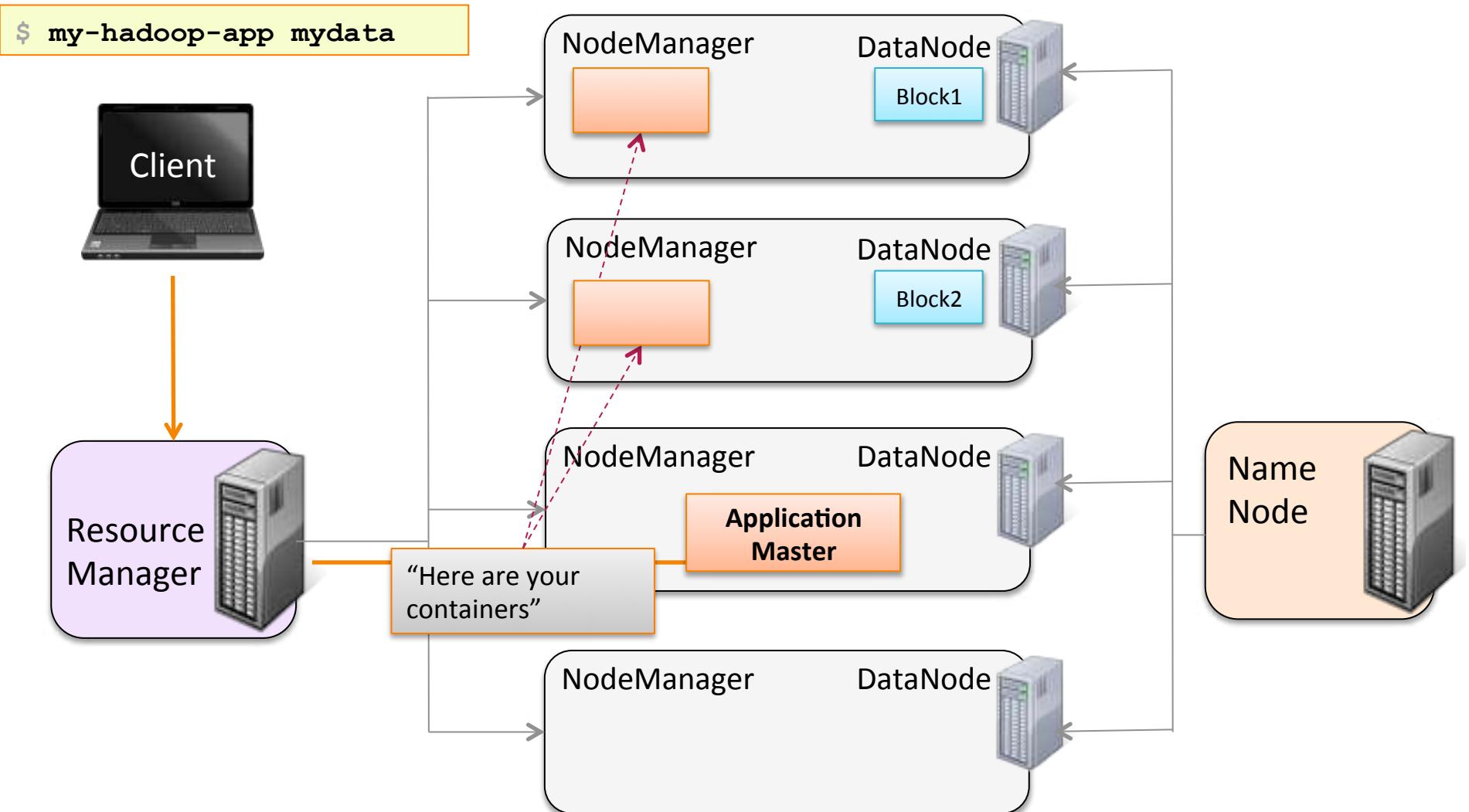
Running an Application on YARN (4)



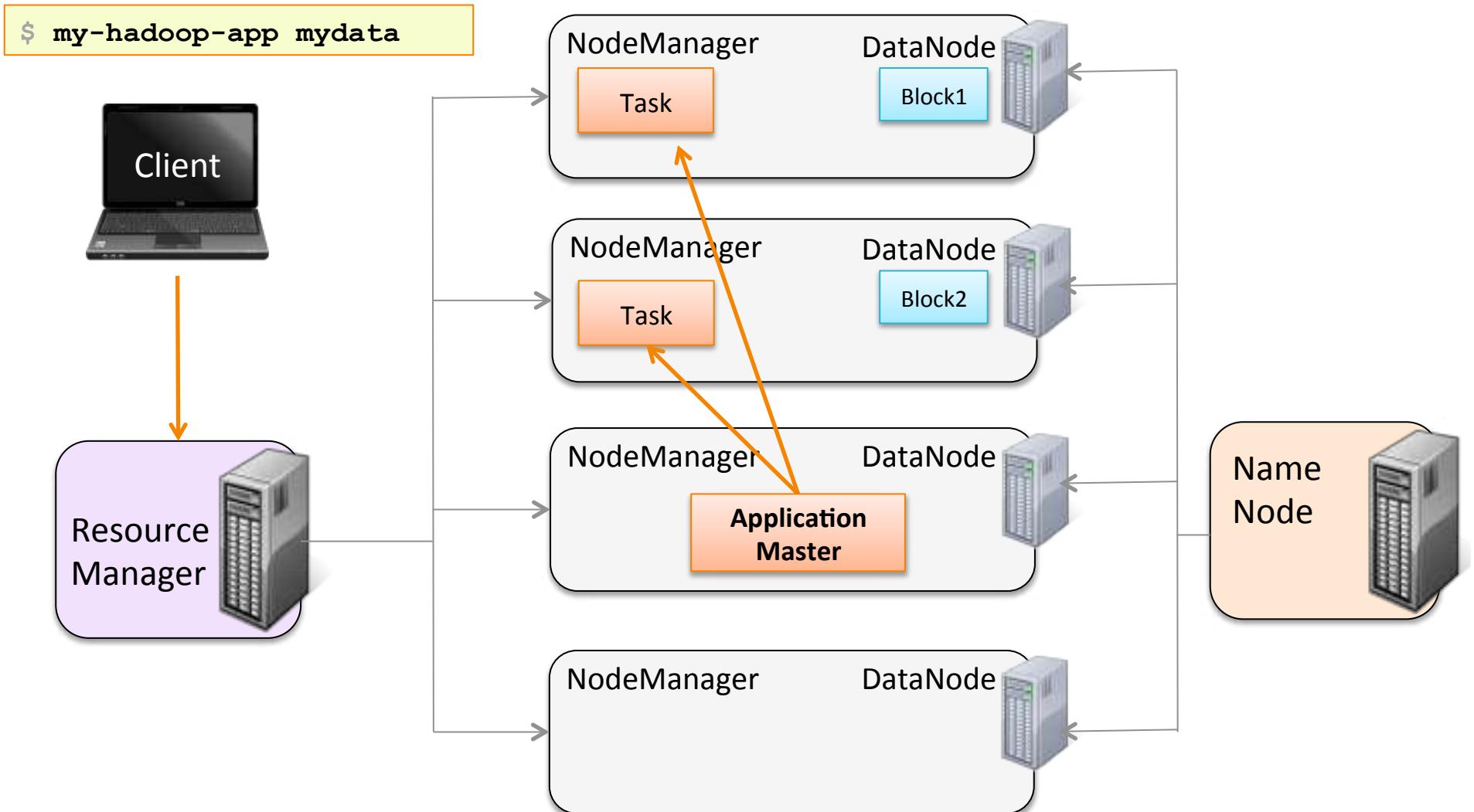
Running an Application on YARN (5)



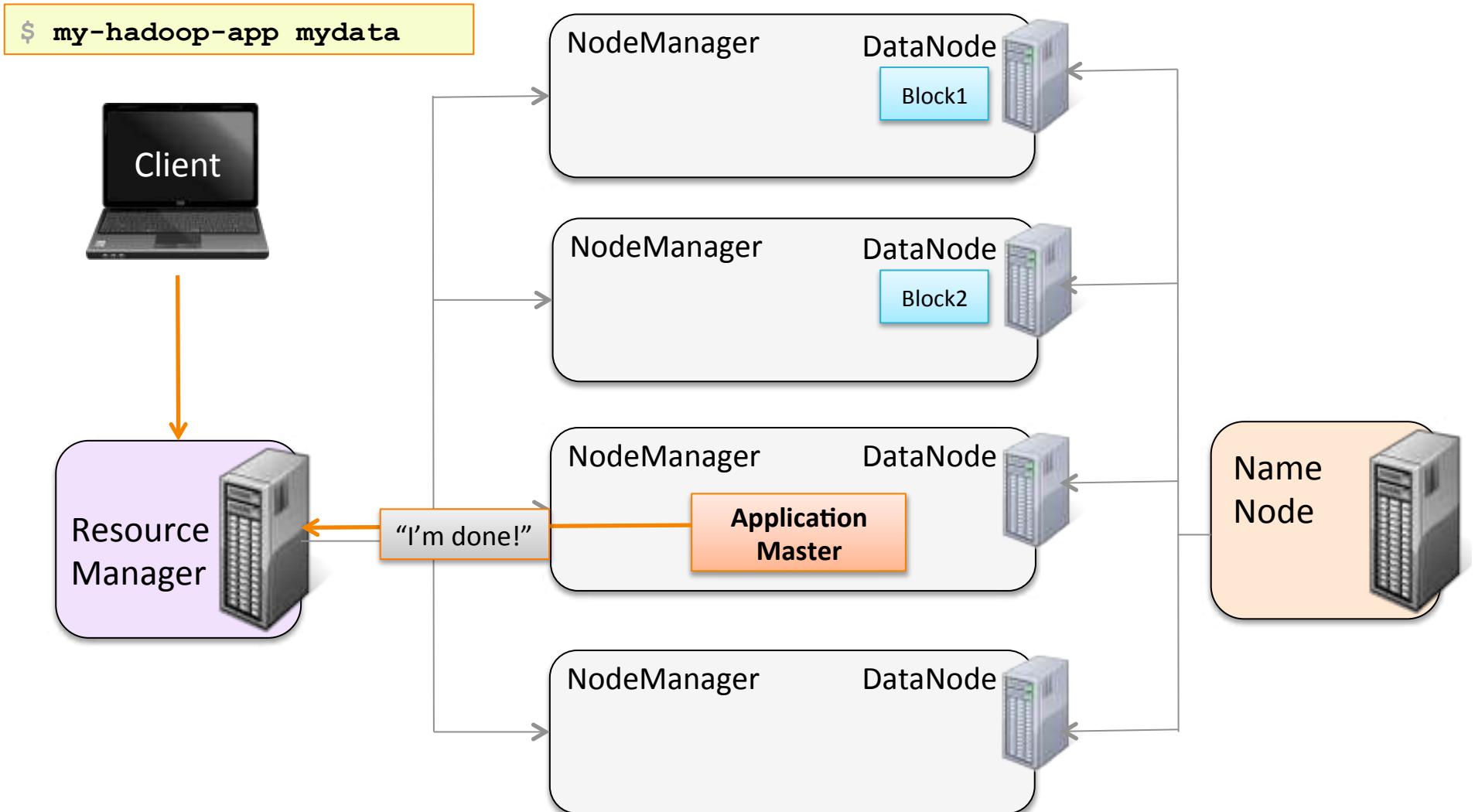
Running an Application on YARN (6)



Running an Application on YARN (7)



Running an Application on YARN (8)



Chapter Topics

Hadoop Architecture and HDFS

Introduction to Hadoop

- Distributed Processing on a Cluster
- Storage: HDFS Architecture
- Storage: Using HDFS
- Hands-on Exercises: Access HDFS with Command Line and Hue
- Resource Management: YARN Architecture
- Resource Management: Working With YARN**
- Conclusion
- Hands-On Exercises: Run a YARN Job

Working With YARN

- **Developers need to be able to**
 - Submit jobs (applications) to run on the YARN cluster
 - Monitor and manage jobs
- **Hadoop includes three major YARN tools for developers**
 - The Hue Job Browser
 - The YARN Web UI
 - The YARN command line
- **YARN administrators can use Cloudera Manager**
 - May also be helpful for developers
 - Included in Cloudera Express and Cloudera Enterprise
 - Not covered in this course

The Hue Job Browser

- The Hue Job Browser allows you to
 - Monitor the status of a job
 - View the logs
 - Kill a running job

Logs	ID	Name	Status	User	Maps	Reduces	Queue	Priority	Duration	Submitted	
1424901249645_0002	webpage.jar	RUNNING	training	5%	5%	root.training	N/A	18s	03/06/15 11:00:17	Kill	
1424901249645_0001	accounts.jar	SUCCEEDED	training	100%	100%	root.training	N/A	1m:21s	03/06/15 10:57:48		

The YARN Web UI

- **Resource Manager UI is the main entry point**
 - Runs on the RM host on port 8080 by default
- **Provides more detailed view than Hue**
- **Does not provide any control or configuration**

Resource Manager UI: Nodes

Logged in as: dr.who

Nodes of the cluster

Cluster Overview

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
4	0	1	3	8	8 GB	8 GB	2 GB	2	0	0	0	0

User Metrics for dr.who

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved
0	0	1	3	0	0	0	0 B	0 B	0 B

Show 20 entries Search:

Rack	Node State	Node Address	Node HTTP Address	Last health-update	Health-report	Containers	Mem Used	Mem Avail
/default-rack	RUNNING	qsslave1:8041	qsslave1:8042	21-Nov-2013 13:07:26		4	4 GB	0 B
/default-rack	RUNNING	qsmaster:8041	qsmaster:8042	21-Nov-2013 13:07:17		4	4 GB	0 B

Showing 1 to 2 of 2 entries First Previous 1 Next Last

link to Node Manager UI

List of each node in cluster

Resource Manager UI: Applications

The screenshot shows the Hadoop Resource Manager UI with the title "All Applications". The interface includes a sidebar with cluster metrics and a main area for user metrics. The main area displays a table of running and recent applications.

Cluster Metrics:

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
8	0	1	7	5	8 GB	8 GB	0 B	1	0	0	0	0

User Metrics for dr.who:

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved
0	0	1	7	0	0	0	0 B	0 B	0 B

Application Table:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1384200217415_0009	training	Process Logs	MAPREDUCE	root.training	Tue, 12 Nov 2013 18:54:38 GMT	N/A	RUNNING	UNDEFINED	<div style="width: 0%;"></div>	ApplicationMaster
application_1384200217415_0008	training	Average Word Length	MAPREDUCE	root.training	Mon, 11 Nov 2013 21:55:21 GMT	Mon, 11 Nov 2013 21:57:30 GMT	FINISHED	SUCCEEDED	<div style="width: 100%;"></div>	History
application_1384200217415_0007	training	Process Logs	MAPREDUCE	root.training	Mon, 11 Nov 2013 21:36:39 GMT	Mon, 11 Nov 2013 21:44:19 GMT	FINISHED	SUCCEEDED	<div style="width: 100%;"></div>	History
application_1384200217415_0006	training	Process Logs	MAPREDUCE	root.training	Mon, 11 Nov 2013	Mon, 11 Nov 2013	FINISHED	SUCCEEDED	<div style="width: 100%;"></div>	History

Callouts:

- Link to Application Details... (next slide)
- List of running and recent applications

Resource Manager UI: Application Detail

The screenshot shows the Hadoop Resource Manager UI. At the top left is the Hadoop logo. At the top right, it says "Logged in as: dr.who". On the left, there's a sidebar with "Cluster" expanded, showing "About", "Nodes", "Applications" (with sub-options: NEW, NEW SAVING, SUBMITTED, ACCEPTED, RUNNING, REMOVING, FINISHING, FINISHED, FAILED, KILLED), and "Scheduler". Below that is a "Tools" section. The main area is titled "Application Overview". It displays details for a running application: User: training, Name: Process Logs, Application Type: MAPREDUCE, State: RUNNING, FinalStatus: UNDEFINED, Started: 12-Nov-2013 13:54:38, Elapsed: 38sec, Tracking URL: ApplicationMaster, and Diagnostics. A dashed blue circle highlights the tracking URL. Below this is a table for the ApplicationMaster, with one row showing Attempt Number 1, Start Time 12-Nov-2013 13:54:38, Node localhost.localdomain:8042, and Logs logs. Two purple callout boxes point to the tracking URL and the logs link. The tracking URL box contains the text: "Link to Application Master (UI depends on specific framework)". The logs link box contains the text: "View aggregated log files (optional)".

Logged in as: dr.who

Application Overview

User: training
Name: Process Logs
Application Type: MAPREDUCE
State: RUNNING
FinalStatus: UNDEFINED
Started: 12-Nov-2013 13:54:38
Elapsed: 38sec
Tracking URL: [ApplicationMaster](#)
Diagnostics:

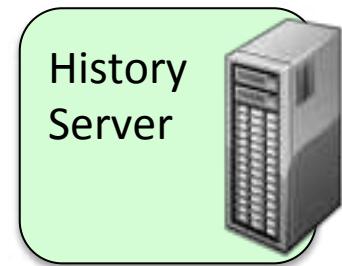
ApplicationMaster	Attempt Number	Start Time	Node	Logs
	1	12-Nov-2013 13:54:38	localhost.localdomain:8042	logs

Link to Application Master (UI depends on specific framework)

View aggregated log files (optional)

Job History Server

- YARN does not keep track of job history
- Spark and MapReduce each provide a Job History Server
 - Archives job's metrics and metadata
 - Can be accessed through Job History UI or Hue



Logged in as: clwhd

hadoop

Retired Jobs

Show 20 entries

Search:

Start Time	Finish Time	job ID	Name	User	Queue	State	Maps Total	Maps Completed	Reduces Total	Reduces Completed
2013.11.21 13:07:36 PST	2013.11.21 13:08:27 PST	job_1385066116114_0004	Process Logs	cloudera	default	SUCCEEDED	4	4	12	12
2013.11.21 13:03:53 PST	2013.11.21 13:04:42 PST	job_1385066116114_0003	Process Logs	cloudera	default	SUCCEEDED	4	4	12	12
2013.11.21 13:01:35 PST	2013.11.21 13:02:28 PST	job_1385066116114_0002	Process Logs	cloudera	default	SUCCEEDED	4	4	12	12
2013.11.21 12:48:00 PST	2013.11.21 12:50:43 PST	job_1385066116114_0001	Word Count	cloudera	default	SUCCEEDED	4	4	1	1
2013.11.21 09:24:45 PST	2013.11.21 09:28:19 PST	job_1385049040288_0003	Word Count	cloudera	default	SUCCEEDED	4	4	1	1

YARN Command Line

- Command to configure and view information about the YARN cluster
 - **yarn <command>**
- Most YARN command line tools are for administrators rather than developers
- Some helpful commands for developers
 - **yarn application**
 - Use **-list** to see running applications
 - Use **-kill** to kill a running application
 - **yarn logs -applicationId <app-id>**
 - View the logs of the specified application

Cloudera Manager

- Cloudera Manager provides a greater ability to monitor and configure a cluster from a single location
 - Covered in *Cloudera Administrator Training for Apache Hadoop*

The screenshot shows two main windows of the Cloudera Manager interface.

Left Window: Resource Management Configuration

Category	Property	Value	Description
Default	Application Master Memory yarn.app.mapreduce.am.resource.mb	256 MB	The physical memory requirement, in MB, for the Application Master.
Resource Management	Application Master Virtual CPU Cores yarn.app.mapreduce.am.resource.cpu-vcores	1	default value
ResourceManagement Base Group	Application Master Java Maximum Heap Size mapreduce.map.java.opts	2048	Reset to the default value
Monitoring	Map Task Memory mapreduce.map.memory.mb	256	Reset to the default value
	Map Task CPU Virtual Cores mapreduce.map.cpu.vcores	1	default value

Right Window: Resource Pools Status

Status					November 21 2013, 9:39:02 AM PST
YARN is using 0 vcores and 0 B of memory.					
Pools Status					
Pool Name	Allocated Memory	Allocated Vcores	Allocated Containers	Pending Containers	Actions
clou...	1.1 GiB	1.1	1.1	0.2	<input type="button" value="Edit"/>
default	0 B	0	0	0	<input type="button" value="Edit"/>

Chapter Topics

Hadoop Architecture and HDFS

Introduction to Hadoop

- Distributed Processing on a Cluster
- Storage: HDFS Architecture
- Storage: Using HDFS
- Hands-on Exercises: Access HDFS with Command Line and Hue
- Resource Management: YARN Architecture
- Resource Management: Working With YARN
- **Conclusion**
- Hands-On Exercises: Run a YARN Job

Essential Points

- **HDFS is the storage layer for Hadoop**
- **Chunks data into blocks and distributes them across the cluster when data is stored**
- **Slave nodes run DataNode daemons, managed by a single NameNode on a master node**
- **Access HDFS using Hue, the `hdfs` command or via the HDFS API**
- **YARN manages resources in a Hadoop cluster and schedules jobs**
- **YARN works with HDFS to run tasks where the data is stored**
- **Slave nodes run NodeManager daemons, managed by a ResourceManager on a master node**
- **Monitor jobs using Hue, the YARN Web UI or the `yarn` command**

Bibliography

The following offer more information on topics discussed in this chapter

- ***Hadoop Application Architectures: Designing Real-World Big Data Applications* (published by O'Reilly)**
 - <http://tiny.cloudera.com/archbook>
- **HDFS User Guide**
 - <http://tiny.cloudera.com/hdfsuser>
- **YARN documentation**
 - <http://tiny.cloudera.com/yarndocs>
- **Cloudera Engineering Blog YARN articles**
 - <http://tiny.cloudera.com/yarnblog>

Chapter Topics

Hadoop Architecture and HDFS

Introduction to Hadoop

- Distributed Processing on a Cluster
- Storage: HDFS Architecture
- Storage: Using HDFS
- Hands-on Exercises: Access HDFS with Command Line and Hue
- Resource Management: YARN Architecture
- Resource Management: Working With YARN
- Conclusion
- **Hands-On Exercises: Run a YARN Job**

Hands-On Exercise: Run a YARN Job

- **In this exercise, you will**
 - Use the YARN Web UI to view your YARN cluster “at rest”
 - Submit an application to run on the cluster
 - Monitor the job using both the YARN UI and Hue
- **Please refer to the Hands-On Exercise Manual**



Importing Relational Data with Sqoop

Chapter 4

Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop**
 - Introduction to Impala and Hive
 - Modeling and Managing Data with Impala and Hive
 - Data Formats
 - Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured Data

Ingesting Streaming Data

Distributed Data Processing with Spark

Course Conclusion

Importing Relational Data with Apache Sqoop

In this chapter you will learn

- **How to import tables from an RDBMS into your Hadoop cluster**
- **How to change the delimiter and file format of imported tables**
- **How to control which columns and rows are imported**
- **What techniques you can use to improve Sqoop's performance**
- **How the next-generation version of Sqoop compares to the original**

Chapter Topics

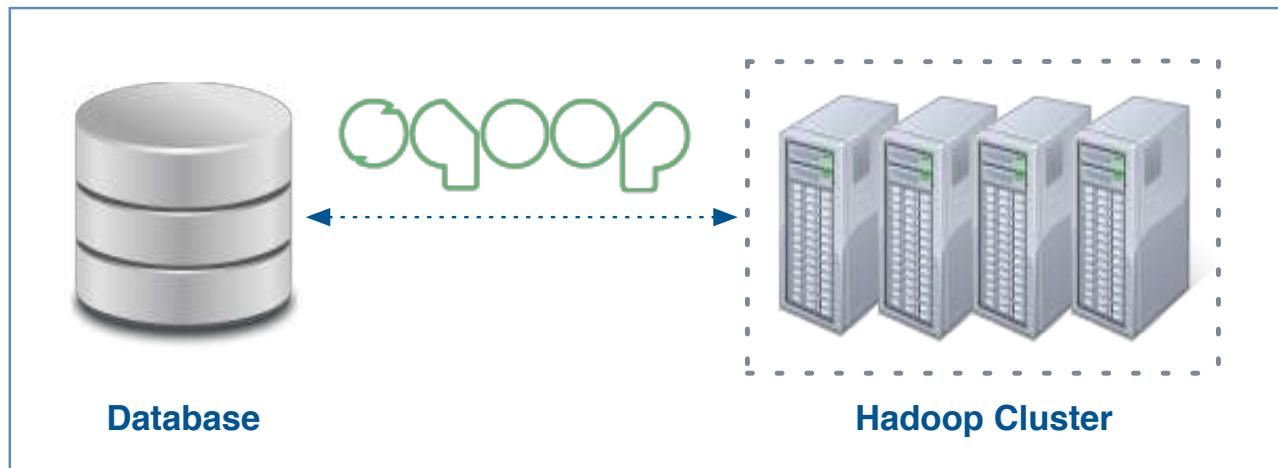
Importing Relational Data with Apache Sqoop

Importing and Modeling Structured Data

- **Sqoop Overview**
- Basic Imports and Exports
- Limiting Results
- Improving Sqoop's Performance
- Sqoop 2
- Conclusion
- Hands-On Exercise: Import Data from MySQL Using Sqoop

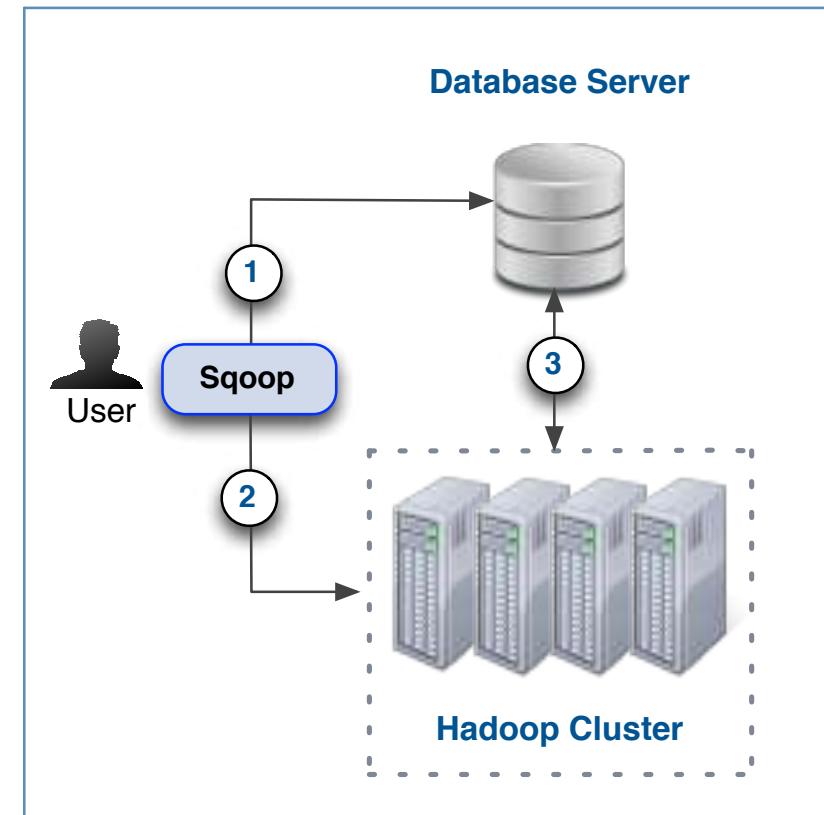
What is Apache Sqoop?

- **Open source Apache project originally developed by Cloudera**
 - The name is a contraction of “SQL-to-Hadoop”
- **Sqoop exchanges data between a database and HDFS**
 - Can import all tables, a single table, or a partial table into HDFS
 - Data can be imported a variety of formats
 - Sqoop can also export data from HDFS to a database



How Does Sqoop Work?

- Sqoop is a client-side application that imports data using Hadoop MapReduce
- A basic import involves three steps orchestrated by Sqoop
 1. Examine table details
 2. Create and submit job to cluster
 3. Fetch records from table and write this data to HDFS



Basic Syntax

- **Sqoop is a command-line utility with several subcommands, called *tools***
 - There are tools for import, export, listing database contents, and more
 - Run **sqoop help** to see a list of all tools
 - Run **sqoop help tool-name** for help on using a specific tool
- **Basic syntax of a Sqoop invocation**

```
$ sqoop tool-name [tool-options]
```

- **This command will list all tables in the `loudacre` database in MySQL**

```
$ sqoop list-tables \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser \
  --password pw
```

Chapter Topics

Importing Relational Data with Apache Sqoop

Importing and Modeling Structured Data

- Sqoop Overview
- **Basic Imports and Exports**
- Limiting Results
- Improving Sqoop's Performance
- Sqoop 2
- Conclusion
- Hands-On Exercise: Import Data from MySQL Using Sqoop

Overview of the Import Process

- **Imports are performed using Hadoop MapReduce jobs**
- **Sqoop begins by examining the table to be imported**
 - Determines the primary key, if possible
 - Runs a *boundary query* to see how many records will be imported
 - Divides result of boundary query by the number of tasks (mappers)
 - Uses this to configure tasks so that they will have equal loads
- **Sqoop also generates a Java source file for each table being imported**
 - It compiles and uses this during the import process
 - The file remains after import, but can be safely deleted

Importing an Entire Database with Sqoop

- The **import-all-tables** tool imports an entire database
 - Stored as comma-delimited files
 - Default base location is your HDFS home directory
 - Data will be in subdirectories corresponding to name of each table

```
$ sqoop import-all-tables \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw
```

- Use the **--warehouse-dir** option to specify a different base directory

```
$ sqoop import-all-tables \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --warehouse-dir /loudacre
```

Importing a Single Table with Sqoop

- The `import` tool imports a single table
- This example imports the `accounts` table
 - It stores the data in HDFS as comma-delimited fields

```
$ sqoop import --table accounts \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw
```

- This variation writes tab-delimited fields instead

```
$ sqoop import --table accounts \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --fields-terminated-by "\t"
```

Incremental Imports (1)

- What if records have changed since last import?
 - Could re-import all records, but this is inefficient
- Sqoop's **incremental lastmodified** mode imports new and modified records
 - Based on a timestamp in a specified column
 - You must ensure timestamps are updated when records are added or changed in the database

```
$ sqoop import --table invoices \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --incremental lastmodified \
  --check-column mod_dt \
  --last-value '2015-09-30 16:00:00'
```

Incremental Imports (2)

- Or use Sqoop's **incremental append** mode to import only *new* records
 - Based on value of last record in specified column

```
$ sqoop import --table invoices \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --incremental append \
  --check-column id \
  --last-value 9478306
```

Exporting Data from Hadoop to RDBMS with Sqoop

- Sqoop's **import** tool pulls records from an RDBMS into HDFS
- It is sometimes necessary to **push** data in HDFS back to an RDBMS
 - Good solution when you must do batch processing on large data sets
 - Export results to a relational database for access by other systems
- Sqoop supports this via the **export** tool
 - The RDBMS table must already exist prior to export

```
$ sqoop export \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --export-dir /loudacre/recommender_output \
  --update-mode allowinsert \
  --table product_recommendations
```

Chapter Topics

Importing Relational Data with Apache Sqoop

Importing and Modeling Structured Data

- Sqoop Overview
- Basic Imports and Exports
- **Limiting Results**
- Improving Sqoop's Performance
- Sqoop 2
- Conclusion
- Hands-On Exercise: Import Data from MySQL Using Sqoop

Importing Partial Tables with Sqoop

- Import only specified columns from accounts table

```
$ sqoop import --table accounts \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --columns "id,first_name,last_name,state"
```

- Import only matching rows from accounts table

```
$ sqoop import --table accounts \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --where "state='CA'"
```

Using a Free-Form Query

- You can also import the results of a query, rather than a single table
- Supply a complete SQL query using the **--query** option
 - You must add the *literal* WHERE \$CONDITIONS token
 - Use **--split-by** to identify field used to divide work among mappers
 - The **--target-dir** option is required for free-form queries

```
$ sqoop import \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --target-dir /data/loudacre/payable \
  --split-by accounts.id \
  --query 'SELECT accounts.id, first_name,
last_name, bill_amount FROM accounts JOIN invoices ON
(accounts.id = invoices.cust_id) WHERE $CONDITIONS'
```

Using a Free-Form Query with WHERE Criteria

- The **--where** option is ignored in a free-form query
 - You must specify your criteria using AND following the WHERE clause

```
$ sqoop import \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  --target-dir /data/loudacre/payable \
  --split-by accounts.id \
  --query 'SELECT accounts.id, first_name,
last_name, bill_amount FROM accounts JOIN invoices ON
(accounts.id = invoices.cust_id) WHERE $CONDITIONS AND
bill_amount >= 40'
```

Chapter Topics

Importing Relational Data with Apache Sqoop

Importing and Modeling Structured Data

- Sqoop Overview
- Basic Imports and Exports
- Limiting Results
- **Improving Sqoop's Performance**
- Sqoop 2
- Conclusion
- Hands-On Exercise: Import Data from MySQL Using Sqoop

Options for Database Connectivity

- **Generic (JDBC)**

- Compatible with nearly any database
 - Overhead imposed by JDBC can limit performance

- **Direct Mode**

- Can improve performance through use of database-specific utilities
 - Currently supports MySQL and Postgres (use --direct option)
 - Not all Sqoop features are available in direct mode

- **Cloudera and partners offer high-performance Sqoop connectors**

- These use native database protocols rather than JDBC
 - Connectors available for Netezza, Teradata, and Oracle
 - Download these from Cloudera's Web site
 - Not open source due to licensing issues, but free to use

Controlling Parallelism

- By default, Sqoop typically imports data using four parallel tasks (called mappers)
 - Increasing the number of tasks might improve import speed
 - Caution: Each task adds load to your database server
- You can *influence* the number of tasks using the **-m** option
 - Sqoop views this only as a hint and might not honor it

```
$ sqoop import --table accounts \
  --connect jdbc:mysql://dbhost/loudacre \
  --username dbuser --password pw \
  -m 8
```

- Sqoop assumes all tables have an evenly-distributed numeric primary key
 - Sqoop uses this column to divide work among the tasks
 - You can use a different column with the **--split-by** option

Chapter Topics

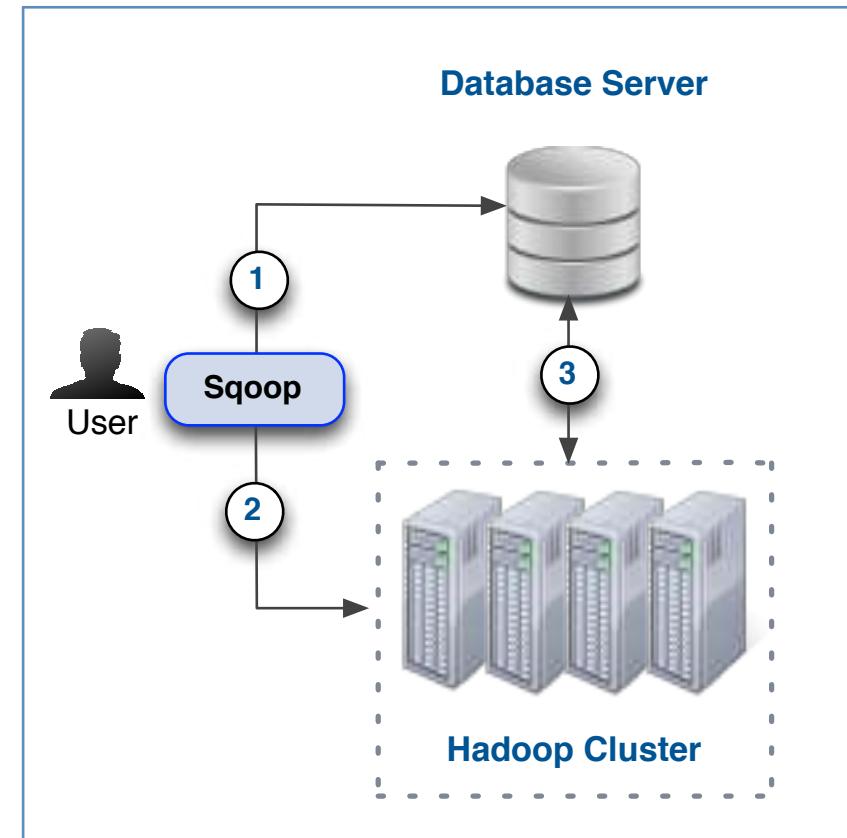
Importing Relational Data with Apache Sqoop

Importing and Modeling Structured Data

- Sqoop Overview
- Basic Imports and Exports
- Limiting Results
- Improving Sqoop's Performance
- **Sqoop 2**
- Conclusion
- Hands-On Exercise: Import Data from MySQL Using Sqoop

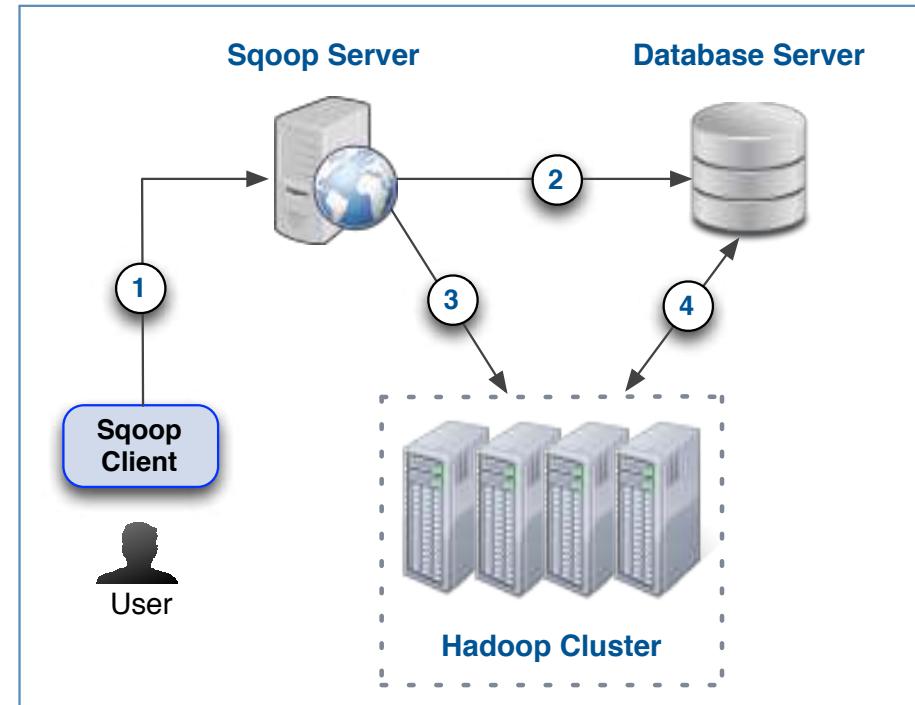
Limitations of Sqoop

- **Sqoop is stable and has been used successfully in production for years**
- **However, its client-side architecture does impose some limitations**
 - Requires connectivity to RDBMS from the client (client must have JDBC drivers installed)
 - Requires connectivity to cluster from the client
 - Requires user to specify RDBMS username and password
 - Difficult to integrate a CLI within external applications
- **Also tightly coupled to JDBC semantics**
 - A problem for NoSQL databases



Sqoop 2 Architecture

- **Sqoop 2 is the next-generation version of Sqoop**
 - Client-server design addresses limitations described earlier
 - API changes also simplify development of other Sqoop connectors
- **Client requires connectivity only to the Sqoop server**
 - DB connections are configured on the server by a system administrator
 - End users no longer need to possess database credentials
 - Centralized audit trail
 - Better resource management
 - Sqoop server is accessible via CLI, REST API, and Web UI



Sqoop 2 Status

- **Sqoop 2 is being actively developed**
 - It began shipping (alongside Sqoop) starting in CDH 4.2
- **Sqoop 2 is not yet at feature parity with Sqoop**
 - Implemented features are regarded as stable
 - Consider using Sqoop 2 unless you require a feature it lacks
- **We use Sqoop, rather than Sqoop 2, in this class**
 - Primarily due to memory constraints in the VM

Chapter Topics

Importing Relational Data with Apache Sqoop

Importing and Modeling Structured Data

- Sqoop Overview
- Basic Imports and Exports
- Limiting Results
- Improving Sqoop's Performance
- Sqoop 2
- **Conclusion**
- Hands-On Exercise: Import Data from MySQL Using Sqoop

Essential Points

- **Sqoop exchanges data between a database and the Hadoop cluster**
 - Provides subcommands (*tools*) for importing, exporting, and more
- **Tables are imported using MapReduce jobs**
 - These are written as comma-delimited text by default
 - You can specify alternate delimiters or file formats
 - Uncompressed by default, but you can specify a codec to use
- **Sqoop provides many options to control imports**
 - You can select only certain columns or limit rows
 - Supports using joins in free-form queries
- **Sqoop 2 is the next-generation version of Sqoop**
 - Client-server design improves administration and resource management

Bibliography

The following offer more information on topics discussed in this chapter

- **Sqoop User Guide**
 - <http://tiny.cloudera.com/sqoopuserguide>
- **Apache Sqoop Cookbook (published by O'Reilly)**
 - <http://tiny.cloudera.com/sqoopcookbook>
- **A New Generation of Data Transfer Tools for Hadoop: Sqoop 2**
 - <http://tiny.cloudera.com/adcc05c>

Chapter Topics

Importing Relational Data with Apache Sqoop

Importing and Modeling Structured Data

- Sqoop Overview
- Basic Imports and Exports
- Limiting Results
- Improving Sqoop's Performance
- Sqoop 2
- Conclusion
- **Hands-On Exercise: Import Data from MySQL Using Sqoop**

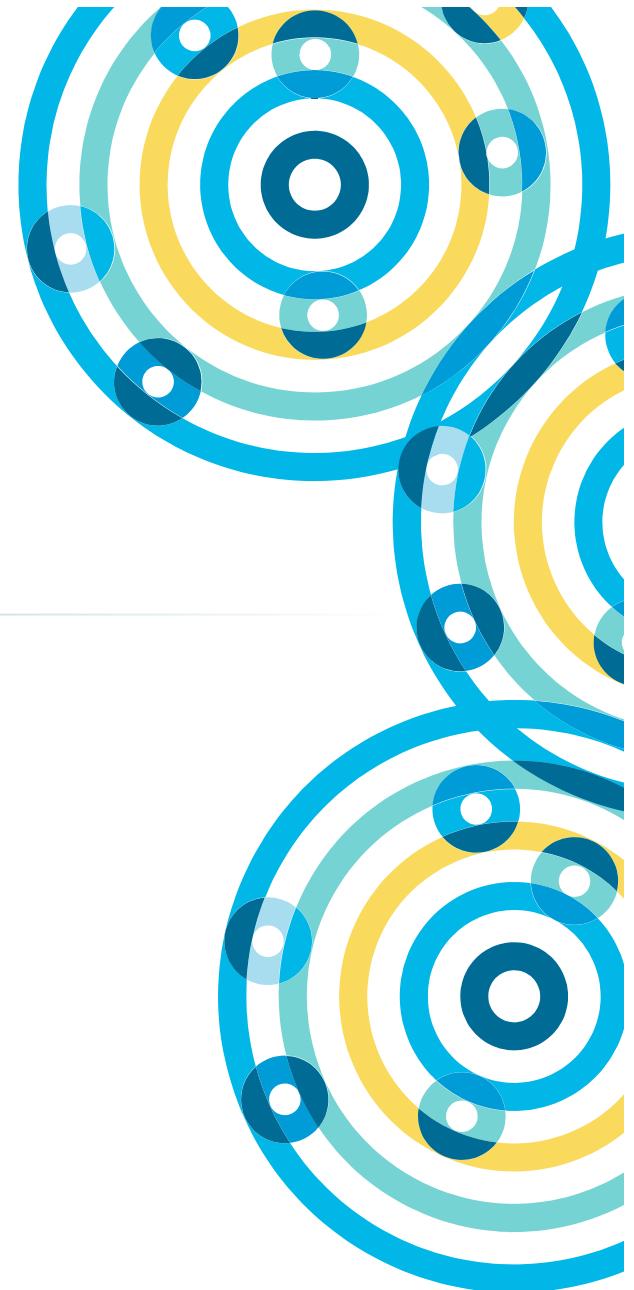
Hands-on Exercise: Import Data from MySQL Using Sqoop

- **In this exercise, you will**
 - Use Sqoop to import web page and customer account data from an RDBMS to HDFS
 - Perform incremental imports of new and updated account data
- **Please refer to the Hands-On Exercise Manual for instructions**



Introduction to Impala and Hive

Chapter 5



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- **Introduction to Impala and Hive**
- Working with Tables in Impala
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured Data

Ingesting Streaming Data

Distributed Data Processing with Spark

Course Conclusion

Introduction to Impala and Hive

In this chapter you will learn

- **What Hive is**
- **What Impala is**
- **How Impala and Hive Compare**
- **How to query data using Impala and Hive**
- **How Hive and Impala differ from a relational database**
- **Ways in which organizations use Hive and Impala**

Chapter Topics

Introduction to Impala and Hive

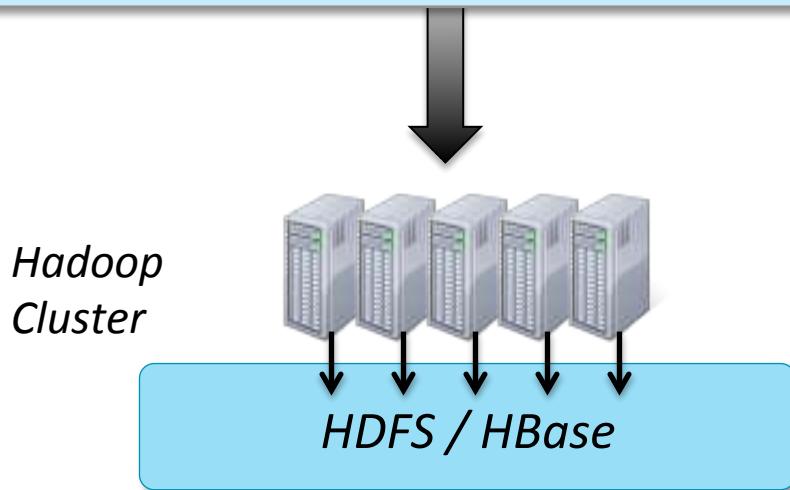
Importing and Modeling Structured Data

- **Introduction to Impala and Hive**
- Why Use Impala and Hive?
- Querying Data With Impala and Hive
- Comparing Hive and Impala to Traditional Databases

Introduction to Impala and Hive (1)

- Impala and Hive are both tools that provide SQL querying of data stored in HDFS / HBase

```
SELECT zipcode, SUM(cost) AS total  
FROM customers  
JOIN orders  
ON (customers.cust_id = orders.cust_id)  
WHERE zipcode LIKE '63%'  
GROUP BY zipcode  
ORDER BY total DESC;
```



Introduction to Impala and Hive (2)

- **Apache Hive is a high-level abstraction on top of MapReduce**
 - Uses HiveQL
 - Generates MapReduce or Spark* jobs that run on the Hadoop cluster
 - Originally developed at Facebook around 2007
 - Now an open-source Apache project
- **Cloudera Impala is a high-performance dedicated SQL engine**
 - Uses Impala SQL
 - Inspired by Google's Dremel project
 - Query latency measured in milliseconds
 - Developed at Cloudera in 2012
 - Open-source with an Apache license



* Hive-on-Spark is currently in beta testing

What's the Difference?

- **Hive has more features**

- E.g. Complex data types (arrays, maps) and full support for windowing analytics
 - Highly extensible
 - Commonly used for batch processing

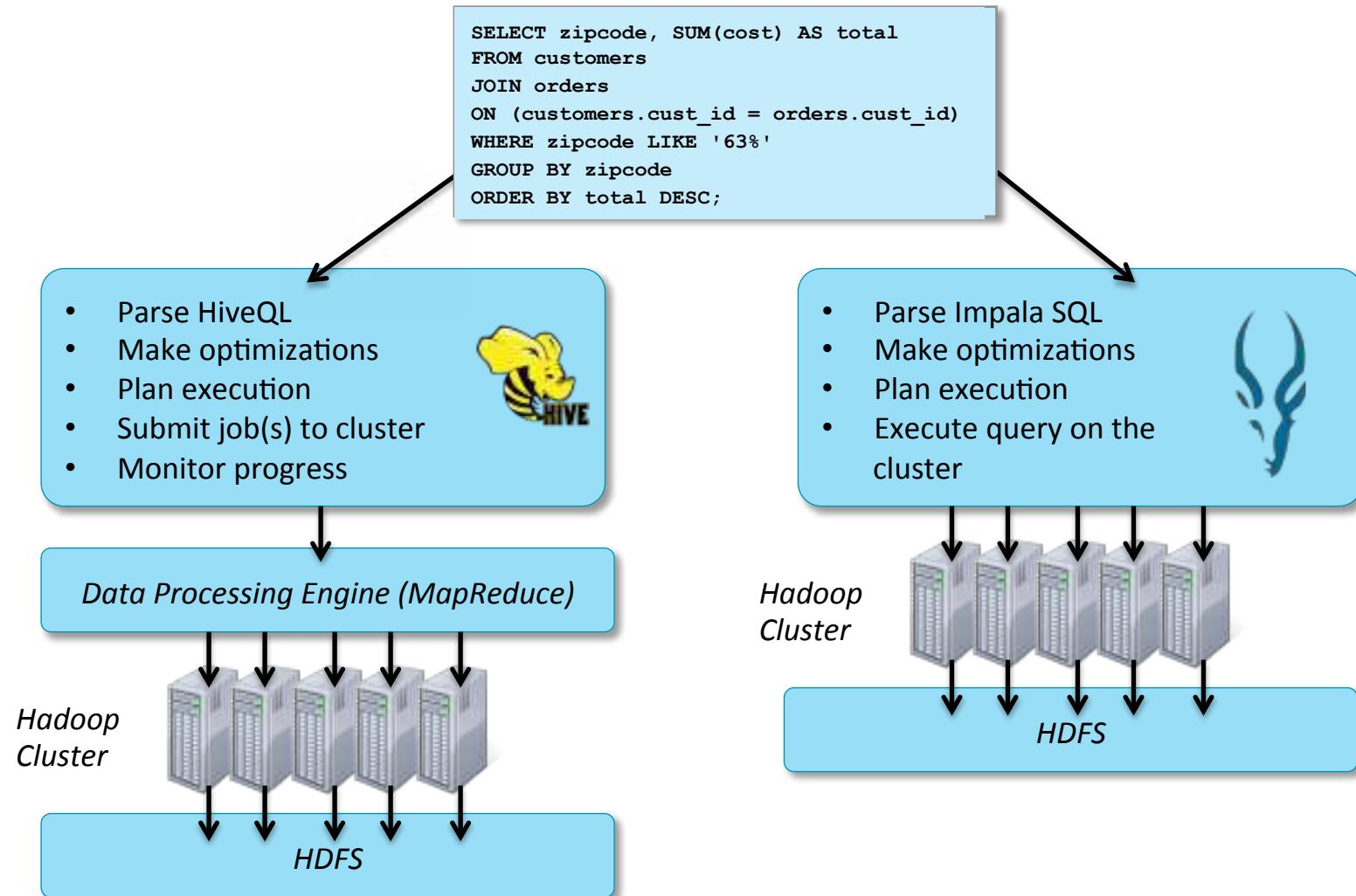


- **Impala is much faster**

- Specialized SQL engine offers 5x to 50x better performance
 - Ideal for interactive queries and data analysis
 - More features being added over time



High-Level Overview



Chapter Topics

Introduction to Impala and Hive

Importing and Modeling Structured Data

- Introduction to Impala and Hive
- **Why Use Impala and Hive?**
- Querying Data With Impala and Hive
- Comparing Hive to Traditional Databases
- Conclusion

Why Use Hive and Impala?

- **Brings large-scale data analysis to a broader audience**
 - No software development experience required
 - Leverage existing knowledge of SQL
- **More productive than writing MapReduce or Spark directly**
 - Five lines of HiveQL/Impala SQL might be equivalent to 200 lines or more of Java
- **Offers interoperability with other systems**
 - Extensible through Java and external scripts
 - Many business intelligence (BI) tools support Hive and/or Impala

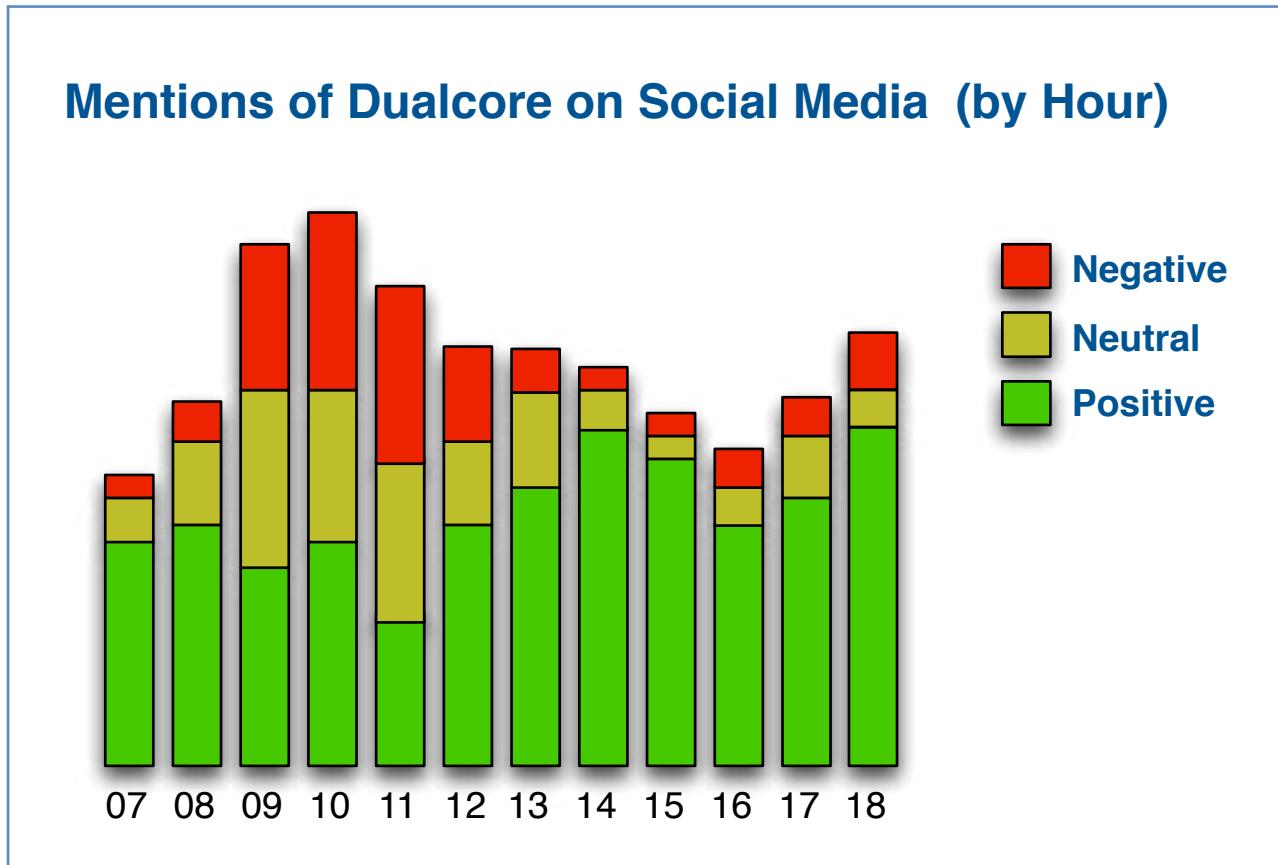
Use Case: Log File Analytics

- Server log files are an important source of data
- Hive and Impala allow you to treat a directory of log files like a table
 - Allows SQL-like queries against raw data

Dualcore Inc. Public Web Site (June 1 - 8)					
Product	Unique Visitors	Page Views	Average Time on Page	Bounce Rate	Conversion Rate
Tablet	5,278	5,894	17 seconds	23%	65%
Notebook	4,139	4,375	23 seconds	47%	31%
Stereo	2,873	2,981	42 seconds	61%	12%
Monitor	1,749	1,862	26 seconds	74%	19%
Router	987	1,139	37 seconds	56%	17%
Server	314	504	53 seconds	48%	28%
Printer	86	97	34 seconds	27%	64%

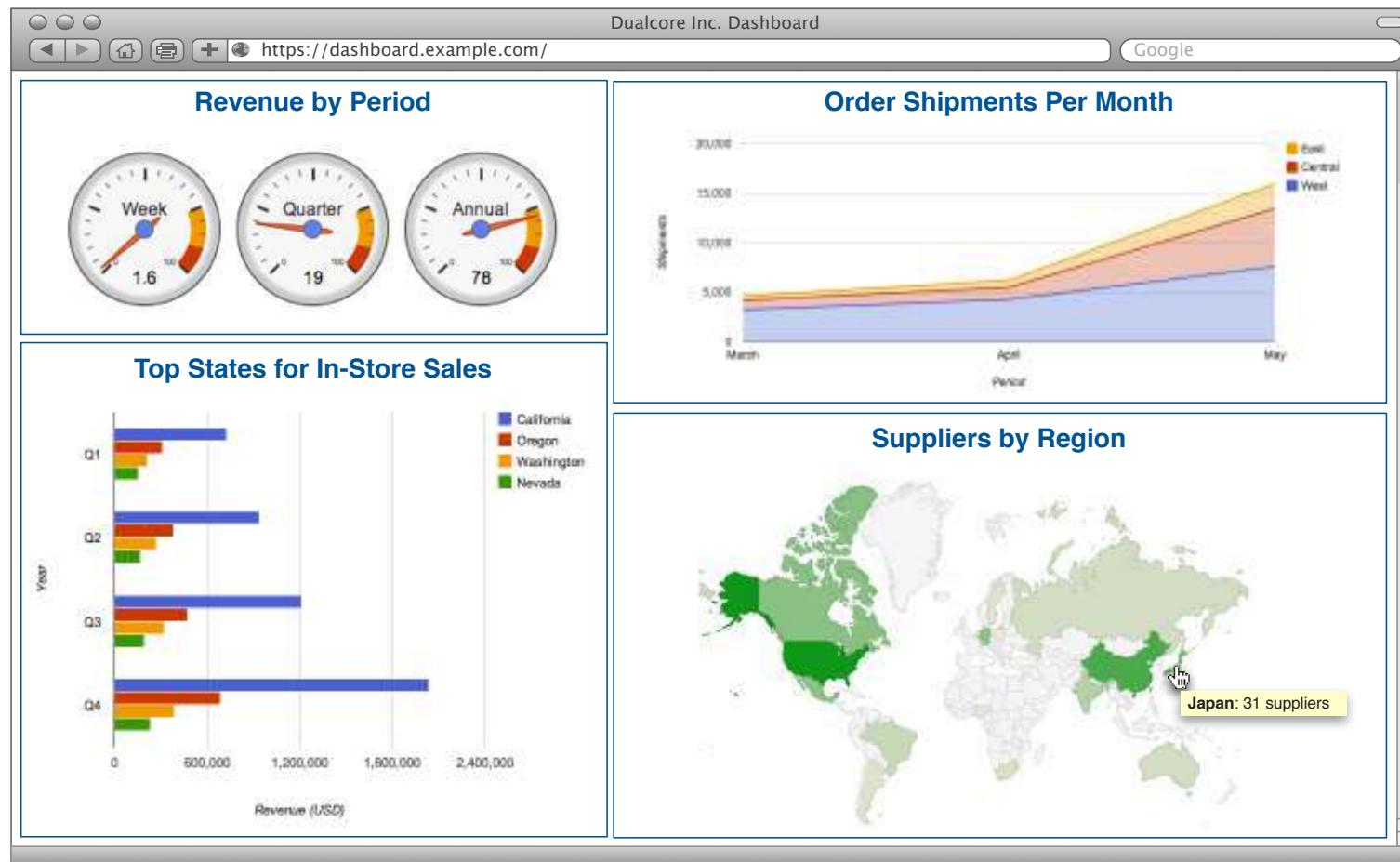
Use Case: Sentiment Analysis

- Many organizations use Hive or Impala to analyze social media coverage



Use Case: Business Intelligence

- Many leading business intelligence tools support Hive and Impala



Chapter Topics

Introduction to Impala and Hive

Importing and Modeling Structured Data

- Introduction to Impala and Hive
- Why Use Impala and Hive?
- **Querying Data With Hive and Impala**
- Comparing Hive to Traditional Databases
- Conclusion

Interacting with Hive and Impala

- **Hive and Impala offer many interfaces for running queries**
 - Command-line shell
 - Impala: Impala shell
 - Hive: Beeline
 - Hue Web UI
 - Hive Query Editor
 - Impala Query Editor
 - Metastore Manager
 - ODBC / JDBC

Starting the Impala Shell

- You can execute statements in the Impala shell
 - This interactive tool is similar to the shell in MySQL
- Execute the `impala-shell` command to start the shell
 - Some log messages truncated to better fit the slide

```
$ impala-shell
Connected to localhost.localdomain:21000
Server version: impalad version 2.1.0-cdh5 (...)

Welcome to the Impala shell.
[localhost.localdomain:21000] >
```

- Use `-i hostname:port` option to connect to a different server

```
$ impala-shell -i myserver.example.com:21000
[myserver.example.com:21000] >
```

Using the Impala Shell

- **Enter semicolon-terminated statements at the prompt**
 - Hit [Enter] to execute a query or command
 - Use the `quit` command to exit the shell
- **Use `impala-shell --help` for a full list of options**

Executing Queries in the Impala Shell

```
> SELECT lname, fname FROM customers WHERE state = 'CA'  
limit 50;  
  
Query: select lname, fname FROM customers WHERE state =  
'CA' limit 50  
+-----+-----+  
| lname      | fname       |  
+-----+-----+  
| Ham        | Marilyn    |  
| Franks     | Gerard     |  
| Preston    | Mason      |  
| Cortez     | Pamela     |  
| ...         |            |  
| Falgoust   | Jennifer   |  
+-----+-----+  
Returned 50 row(s) in 0.17s  
  
>
```

Note: shell prompt abbreviated as >

Interacting with the Operating System

- Use **shell** to execute system commands from within Impala shell

```
> shell date;  
Mon May 20 16:44:35 PDT 2013
```

- No direct support for HDFS commands

- But could run hdfs dfs using shell

```
> shell hdfs dfs -mkdir /reports/sales/2013;
```

Running Impala Queries from the Command Line

- You can execute a file containing queries using the **-f** option

```
$ impala-shell -f myquery.sql
```

- Run queries directly from the command line with the **-q** option

```
$ impala-shell -q 'SELECT * FROM users'
```

- Use **-o** to capture output to file

- Optionally specify delimiter

```
$ impala-shell -f myquery.sql \
-o results.txt \
--delimited \
--output_delimiter=','
```

Starting Beeline (Hive's Shell)

- You can execute HiveQL statements in the Beeline shell
 - Interactive shell based on the SQLLine utility
 - Similar to the Impala shell
- Start Beeline by specifying the URL for a Hive2 server
 - Plus username and password if required

```
$ beeline -u jdbc:hive2://host:10000 \
-n username -p password

0: jdbc:hive2://localhost:10000>
```

Executing Queries in Beeline

- SQL commands are terminated with semi-colon (;)
- Similar to Impala shell
 - Results formatting is slightly different

```
1: url> SELECT lname, fname FROM customers
. . . > WHERE state = 'CA' LIMIT 50;

+-----+-----+
| lname | fname |
+-----+-----+
| Ham   | Marilyn |
| Franks | Gerard |
| Preston | Mason |
...
| Falgoust | Jennifer |
+-----+-----+
50 rows selected (15.829 seconds)

1: url>
```

Using Beeline

- Execute Beeline commands with ‘!’
 - No terminator character
- Some commands
 - **!connect url** – connect to a different Hive2 server
 - **!exit** – exit the shell
 - **!help** – show the full list of commands
 - **!verbose** – show added details of queries

```
0: jdbc:hive2://localhost:10000> !exit
```

Executing Hive Queries from the Command Line

- You can also execute a file containing HiveQL code using the **-f** option

```
$ beeline -u ... -f myquery.hql
```

- Or use HiveQL directly from the command line using the **-e** option

```
$ beeline -u ... -e 'SELECT * FROM users'
```

- Use the **--silent** option to suppress informational messages
 - Can also be used with **-e** or **-f** options

```
$ beeline -u ... --silent
```

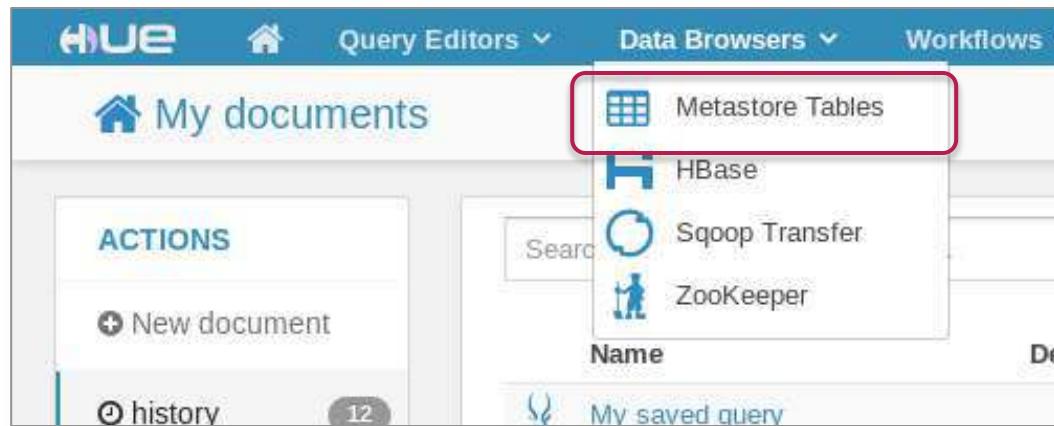
Using Hue with Hive and Impala

You can use Hue to...

Query data with
Hive or Impala



View and manage
the Metastore



The Hue Query Editor

- The Impala and Hive Query editors are nearly identical

The screenshot shows the Hue Query Editor interface. On the left, there's a sidebar with 'Assist' and 'Settings' tabs, a 'DATABASE...' dropdown set to 'default', and sections for 'Table name...', 'customers' (with columns: cust_id, fname, lname, address, city, state, zipcode), 'order_details', 'orders', and 'products'. A blue box labeled 'Choose a database' points to the 'DATABASE...' dropdown. Another blue box labeled 'Explore schema and sample data' points to the 'customers' schema section. The main area has a 'Query Editor' tab selected, showing a code editor with the query: `1 SELECT * FROM customers WHERE state = 'CA';`. A blue box labeled 'Enter, edit, save and execute queries' points to the code editor. Below the editor are buttons for 'Execute', 'Save as...', 'Explain', and 'or create a New query'. The bottom half of the screen shows a results table with data from the 'customers' table, where rows 0 through 3 are displayed. A blue box labeled 'View results, logs, reports, etc.' points to the results table. The results table has columns: cust_id, fname, lname, address, city, state, and zipcode.

	cust_id	fname	lname	address	city	state	zipcode
0	1000002	Marilyn	Ham	25831 North 25th Street	Concord	CA	94522
1	1000006	Gerard	Franks	356 Turner Street	Pioneer	CA	95666
2	1000010	Mason	Preston	2656 West 13th Street	Redwood Valley	CA	95470
3	1000012	Pamela	Cortez	2279 North Mulberry Avenue	San Francis		

Chapter Topics

Introduction to Impala and Hive

Importing and Modeling Structured Data

- Introduction to Impala and Hive
- Why Use Impala and Hive?
- Querying Data With Impala and Hive
- **Comparing Hive and Impala to Traditional Databases**
- Conclusion

Your Cluster is Not a Database Server

- **Client-server database management systems have many strengths**
 - Very fast response time
 - Support for transactions
 - Allow modification of existing records
 - Can serve thousands of simultaneous clients
- **Your Hadoop cluster is not an RDBMS**
 - Hive generates processing engine jobs (MapReduce) from HiveQL queries
 - Limitations of HDFS and MapReduce still apply
 - Impala is faster but not intended for the throughput speed required for an OLTP database
 - No transaction support

Comparing Hive and Impala To A Relational Database

	Relational Database	Hive	Impala
Query language	SQL (full)	SQL (subset)	SQL (subset)
Update individual records	Yes	No	No
Delete individual records	Yes	No	No
Transactions	Yes	No	No
Index support	Extensive	Limited	No
Latency	Very low	High	Low
Data size	Terabytes	Petabytes	Petabytes

Chapter Topics

Introduction to Impala and Hive

Importing and Modeling Structured Data

- Introduction to Impala and Hive
- Why Use Impala and Hive?
- Querying Data With Impala and Hive
- Comparing Hive and Impala to Traditional Databases
- **Conclusion**

Essential Points

- **Impala and Hive are tools for performing SQL queries on data in HDFS**
- **HiveQL and Impala SQL are very similar to SQL-92**
 - Easy to learn for those with relational database experience
 - However, does *not* replace your RDBMS
- **Hive generates jobs that run on the Hadoop cluster data processing engine**
 - Runs MapReduce jobs on Hadoop based on HiveQL statements
- **Impala execute queries directly on the Hadoop cluster**
 - Uses a very fast specialized SQL engine, not MapReduce

Bibliography

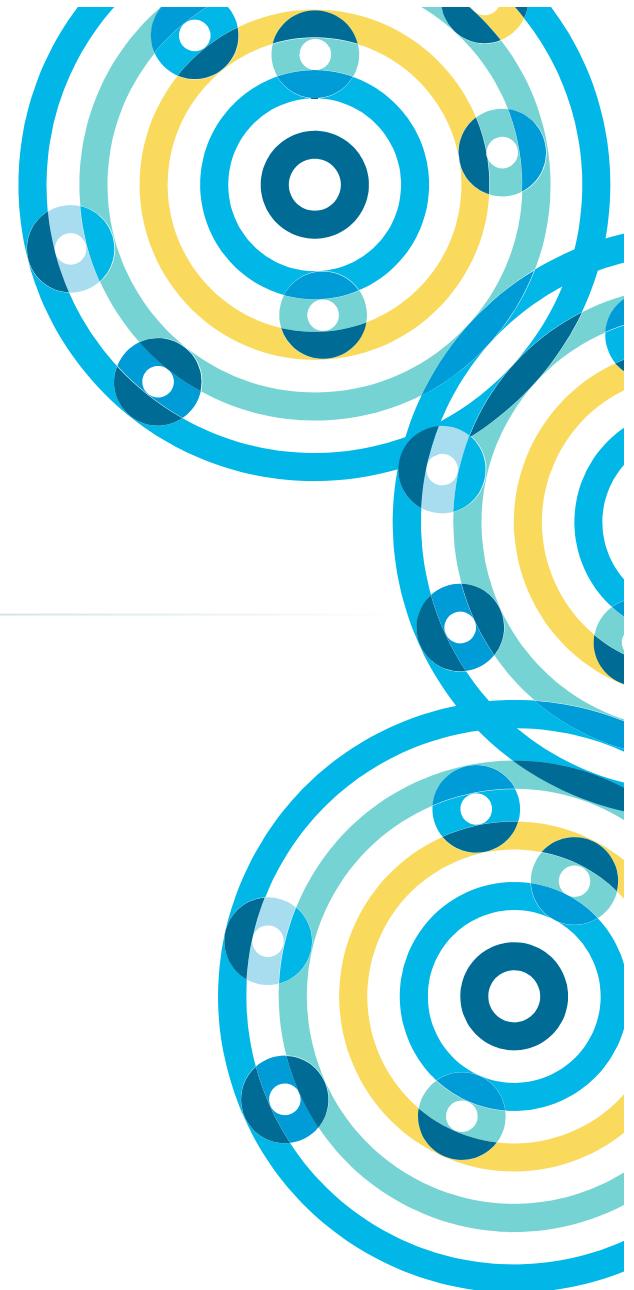
The following offer more information on topics discussed in this chapter

- ***Cloudera Impala (free O'Reilly book)***
 - <http://tiny.cloudera.com/impalabook>
- ***Programming Hive (O'Reilly book)***
 - <http://tiny.cloudera.com/programminghive>
- **Data Analysis with Hadoop and Hive at Orbitz**
 - <http://tiny.cloudera.com/dac09b>
- **Sentiment Analysis Using Apache Hive**
 - <http://tiny.cloudera.com/dac09c>
- ***Wired Article on Impala***
 - <http://tiny.cloudera.com/wiredimpala>



Modeling and Managing Data with Impala and Hive

Chapter 6



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- **Modeling and Managing Data with Impala and Hive**
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured Data

Ingesting Streaming Data

Distributed Data Processing with Spark

Course Conclusion

Modeling and Managing Data in Impala and Hive

In this chapter you will learn

- **How Impala and Hive use the Metastore**
- **How to use Impala SQL and HiveQL DDL to create tables**
- **How to create and manage tables using Hue or HCatalog**
- **How to load data into tables using Impala, Hive, or Sqoop**

Chapter Topics

Modeling and Managing Data With Impala and Hive

Importing and Modeling Structured Data

- **Data Storage Overview**

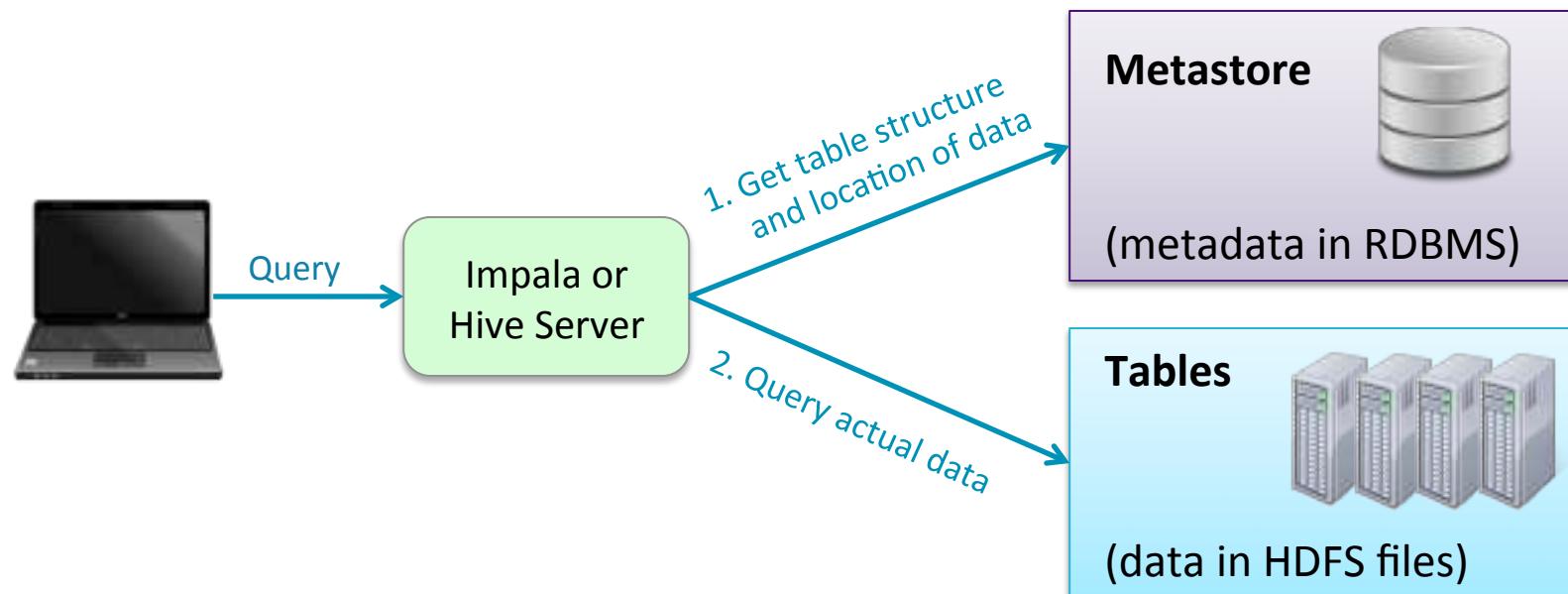
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

How Hive and Impala Load and Store Data (1)

- **Queries operate on tables, just like in an RDBMS**
 - A table is simply an HDFS directory containing one or more files
 - Default path: `/user/hive/warehouse/<table_name>`
 - Supports many formats for data storage and retrieval
- **What is the structure and location of tables?**
 - These are specified when tables are created
 - This metadata is stored in the *Metastore*
 - Contained in an RDBMS such as MySQL
- **Hive and Impala work with the same data**
 - Tables in HDFS, metadata in the Metastore

How Hive and Impala Load and Store Data (2)

- **Hive and Impala use the Metastore to determine data format and location**
 - The query itself operates on data stored in HDFS



Data and Metadata

- **Data refers to the information you store and process**
 - Billing records, sensor readings, and server logs are examples of data
- **Metadata describes important aspects of that data**
 - Field name and order are examples of metadata

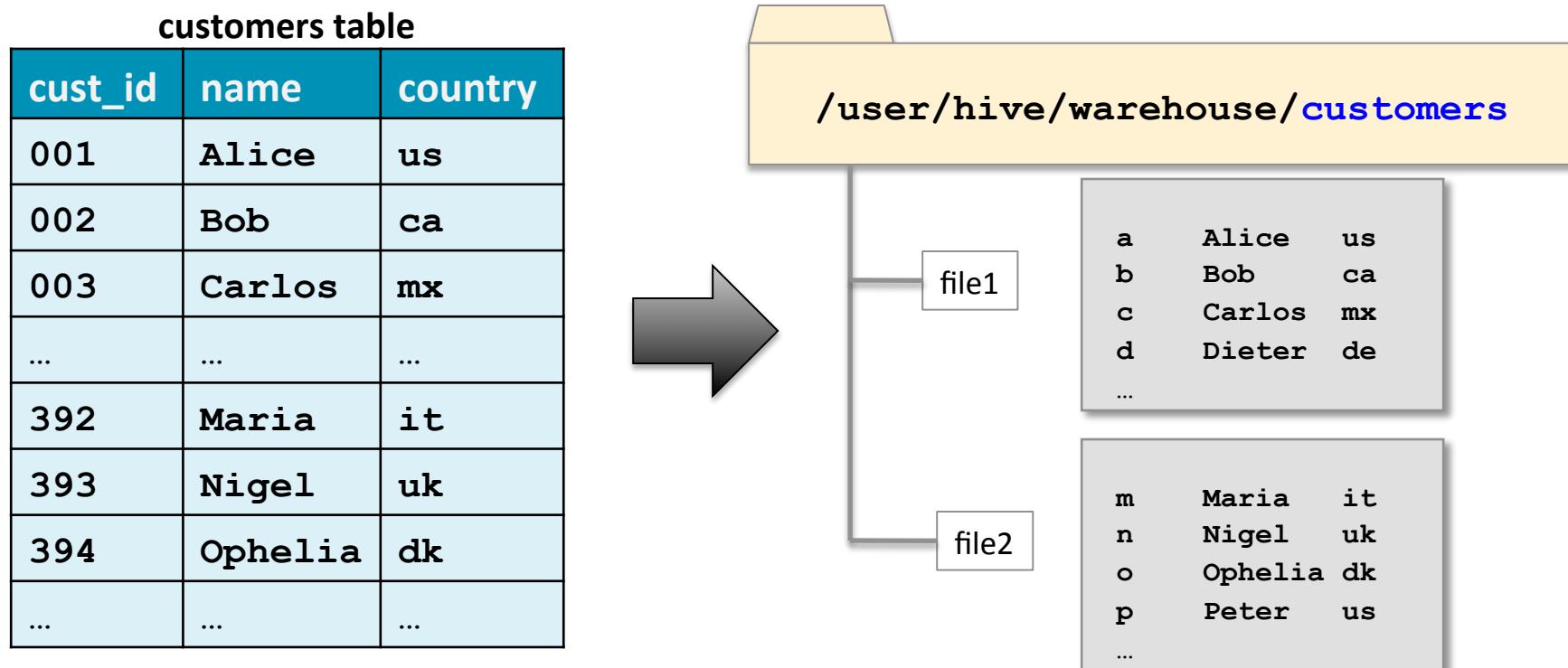


The diagram illustrates the relationship between metadata and data. A vertical bracket on the left side groups the labels "Metadata" and "Data" with the table to their right. The table consists of a header row and eight data rows, each containing three columns: "cust_id", "name", and "country".

cust_id	name	country
001	Alice	us
002	Bob	ca
003	Carlos	mx
...
392	Maria	it
393	Nigel	uk
394	Ophelia	dk
...

The Data Warehouse Directory

- By default, data is stored in the HDFS directory `/user/hive/warehouse`
- Each table is a subdirectory containing any number of files



Chapter Topics

Modeling and Managing Data With Impala and Hive

Importing and Modeling Structured Data

- Data Storage Overview
- **Creating Databases and Tables**
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

Defining Databases and Tables

- **Databases and tables are created and managed using the DDL (Data Definition Language) of HiveQL or Impala SQL**
 - Very similar to standard SQL DDL
 - Some minor differences between Hive and Impala DDL will be noted

Creating a Database

- **Hive and Impala databases are simply namespaces**
 - Helps to organize your tables
- **To create a new database**

```
CREATE DATABASE loudacre;
```

1. Adds the database definition to the metastore
2. Creates a storage directory in HDFS
e.g./user/hive/warehouse/loudacre.db

- **To conditionally create a new database**
 - Avoids error in case database already exists (useful for scripting)

```
CREATE DATABASE IF NOT EXISTS loudacre;
```

Removing a Database

- Removing a database is similar to creating it
 - Just replace **CREATE** with **DROP**

```
DROP DATABASE loudacre;
```

```
DROP DATABASE IF EXISTS loudacre;
```

- These commands will fail if the database contains tables
 - In Hive: Add the **CASCADE** keyword to force removal
 - Caution: this command might remove data in HDFS!



```
DROP DATABASE loudacre CASCADE;
```

Data Types

- **Each column is assigned a specific data type**
 - These are specified when the table is created
 - NULL values are returned for non-conforming data in HDFS
- **Here are some common data types**

Name	Description	Example Value
STRING	Character data (of any length)	Alice
BOOLEAN	TRUE or FALSE	TRUE
TIMESTAMP	Instant in time	2014-03-14 17:01:29
INT	Range: same as Java int	84127213
BIGINT	Range: same as Java long	7613292936514215317
FLOAT	Range: same as Java float	3.14159
DOUBLE	Range: same as Java double	3.1415926535897932385



Hive (not Impala) also supports a few complex types such as maps and arrays

Creating a Table (1)

- Basic syntax for creating a table:

```
CREATE TABLE tablename (colname DATATYPE, ...)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY char  
STORED AS {TEXTFILE|SEQUENCEFILE|...}
```

- Creates a subdirectory in the database's **warehouse** directory in HDFS
 - Default database:
/user/hive/warehouse/*tablename*
 - Named database:
/user/hive/warehouse/*dbname*.db/*tablename*

Creating a Table (2)

```
CREATE TABLE tablename (colname DATATYPE, ...)
```

```
ROW FORMAT DELIMITED
```

```
FIELDS
```

```
STORED
```

Specify a name for the table, and list the column names and datatypes (see later)

Creating a Table (3)

```
CREATE TABLE tablename (colname DATATYPE, ...)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY char  
STORED AS TEXTFILE
```

This line states that fields in each file in the table's directory are delimited by some character. The default delimiter is Control-A, but you may specify an alternate delimiter...

Creating a Table (4)

```
CREATE TABLE tablename (colname DATATYPE, ...)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY char  
STORED AS {TEXTFILE|SEQUENCEFILE|...}
```

...for example, tab-delimited data would require that you specify **FIELDS TERMINATED BY '\t'**

Creating a Table (5)

```
CREATE TABLE tablename (colname DATATYPE, ...)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY char  
STORED AS {TEXTFILE|SEQUENCEFILE|...}
```

Finally, you may declare the file format. **STORED AS TEXTFILE** is the default and does not need to be specified.
Other formats will be discussed later in the course.

Example Table Definition

- The following example creates a new table named **jobs**
 - Data stored as text with four comma-separated fields per line

```
CREATE TABLE jobs (
    id INT,
    title STRING,
    salary INT,
    posted TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

- Example of corresponding record for the table above

```
1,Data Analyst,100000,2013-06-21 15:52:03
```

Creating Tables Based On Existing Schema

- Use **LIKE** to create a new table based on an existing table definition

```
CREATE TABLE jobs_archived LIKE jobs;
```

- Column definitions and names are derived from the existing table
 - New table will contain no data

Creating Tables Based On Existing Data

- **Create a table based on a SELECT statement**
 - Often known as ‘Create Table As Select’ (CTAS)

```
CREATE TABLE ny_customers AS
    SELECT cust_id, fname, lname
        FROM customers
    WHERE state = 'NY';
```

- **Column definitions are derived from the existing table**
- **Column names are inherited from the existing names**
 - Use aliases in the SELECT statement to specify new names
- **New table will contain the selected data**

Controlling Table Data Location

- By default, table data is stored in the warehouse directory
- This is not always ideal
 - Data might be shared by several users
- Use LOCATION to specify the directory where table data resides

```
CREATE TABLE jobs (
    id INT,
    title STRING,
    salary INT,
    posted TIMESTAMP
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/loudacre/jobs';
```

Externally Managed Tables

- **CAUTION: Dropping a table removes its data in HDFS**
 - Tables are “managed” or “internal” by default
- **Using EXTERNAL when creating the table avoids this behavior**
 - Dropping an *external* table removes only its *metadata*

```
CREATE EXTERNAL TABLE adclicks
( campaign_id STRING,
  click_time TIMESTAMP,
  keyword STRING,
  site STRING,
  placement STRING,
  was_clicked BOOLEAN,
  cost SMALLINT)
LOCATION '/loudacre/ad_data' ;
```

Exploring Tables (1)

- The **SHOW TABLES** command lists all tables in the current database

```
SHOW TABLES;
+-----+
| tab_name |
+-----+
| accounts |
| employees |
| job       |
| vendors   |
+-----+
```

- The **DESCRIBE** command lists the fields in the specified table

```
DESCRIBE jobs;
+-----+-----+-----+
| name | type  | comment |
+-----+-----+-----+
| id   | int   |          |
| title| string|          |
| salary| int   |          |
| posted| timestamp|          |
+-----+-----+-----+
```

Exploring Tables (2)

- **DESCRIBE FORMATTED** also shows table properties

```
DESCRIBE FORMATTED jobs;
+-----+-----+-----+
| name      | type       | comment |
+-----+-----+-----+
| # col_name | data_type  | comment |
| id         | int        | NULL    |
| title      | string     | NULL    |
| salary     | int        | NULL    |
| posted     | timestamp  | NULL    |
|             | NULL       | NULL    |
| # Detailed Table | NULL     | NULL    |
| Information   |           |          |
| Database:     | default    | NULL    |
| Owner:        | training   | NULL    |
| CreateTime:   | Wed Jun 17 09:41:23 PDT 2015 | NULL    |
| LastAccessTime: | UNKNOWN | NULL    |
| Protect Mode: | None     | NULL    |
| Retention:    | 0        | NULL    |
| Location:     | hdfs://localhost:8020/loudacre/jobs | NULL    |
| Table Type:   | MANAGED_TABLE | NULL    |
...
...
```

Exploring Tables (3)

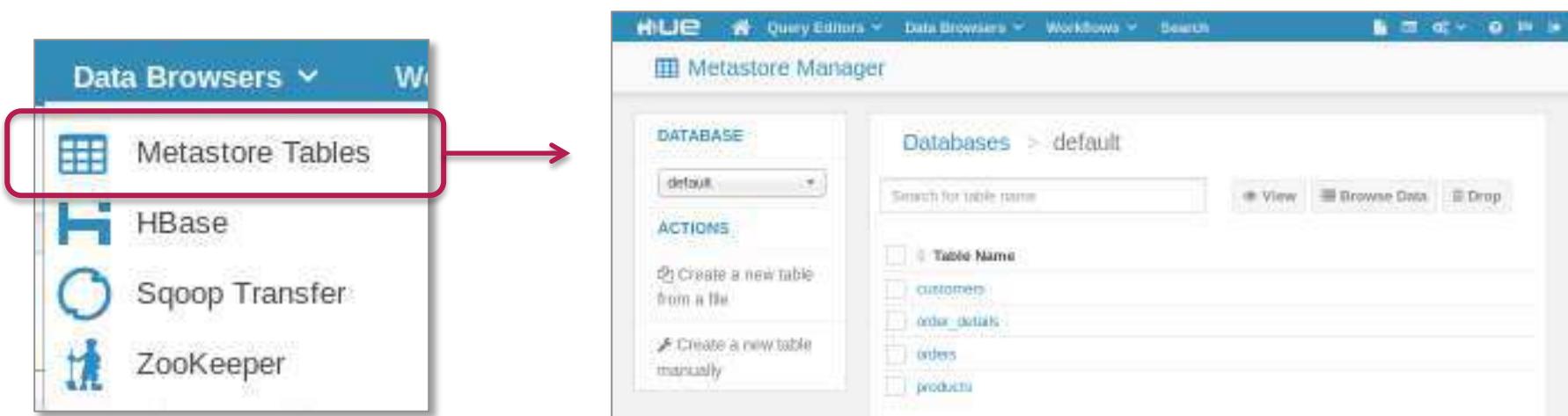
- **SHOW CREATE TABLE** displays the SQL command to create the table

```
SHOW CREATE TABLE jobs;
+-----+
| CREATE TABLE default.jobs
|   id INT,
|   title STRING,
|   salary INT,
|   posted TIMESTAMP
| )
| ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' '
| ...
|
```

Using the Hue Metastore Manager

■ The Hue Metastore Manager

- An alternative to using SQL commands to manage metadata
- Allows you to create, load, preview, and delete databases and tables
 - Not all features are supported yet



Chapter Topics

Modeling and Managing Data With Impala and Hive

Importing and Modeling Structured Data

- Data Storage Overview
- Creating Databases and Tables
- **Loading Data into Tables**
- HCatalog
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

Data Validation

- **Impala and Hive are ‘schema on read’**
 - Unlike an RDBMS, they do not validate data on insert
 - Files are simply moved into place
 - Loading data into tables is therefore very fast
 - Errors in file format will be discovered when queries are performed
- **Missing or invalid data will be represented as NULL**

Loading Data From HDFS Files

- To load data, simply add files to the table's directory in HDFS
 - Can be done directly using the `hdfs dfs` commands
 - This example loads data from HDFS into the `sales` table

```
$ hdfs dfs -mv \
  /tmp/sales.txt /user/hive/warehouse/sales/
```

- Alternatively, use the `LOAD DATA INPATH` command
 - Done from within Hive or Impala
 - This *moves* data within HDFS, just like the command above
 - Source can be either a file or directory

```
LOAD DATA INPATH '/tmp/sales.txt'
  INTO TABLE sales;
```

Overwriting Data From Files

- Add the **OVERWRITE** keyword to delete all records before import
 - Removes all files within the table's directory
 - Then moves the new files into that directory

```
LOAD DATA INPATH '/tmp/sales.txt'  
OVERWRITE INTO TABLE sales;
```

Appending Selected Records to a Table

- Another way to populate a table is through a query
 - Use **INSERT INTO** to add results to an *existing* Hive table

```
INSERT INTO TABLE accounts_copy
SELECT * FROM accounts;
```

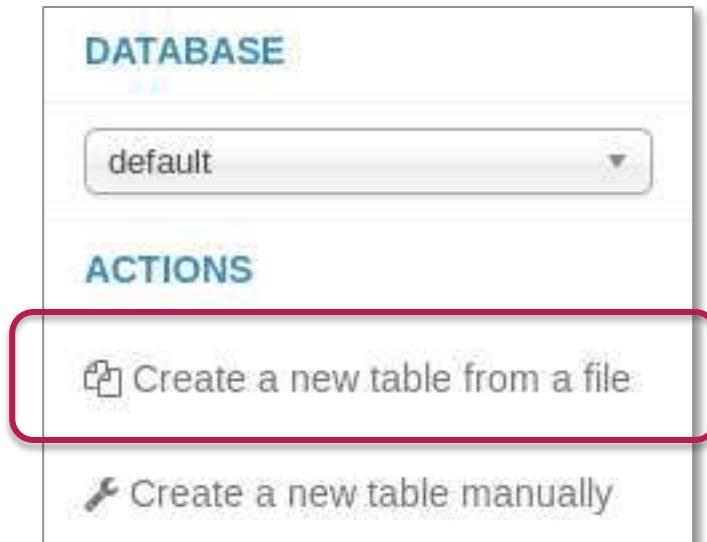
- Specify a **WHERE** clause to control which records are appended

```
INSERT INTO TABLE loyal_customers
SELECT * FROM accounts
WHERE YEAR(acct_create_dt) = 2008
AND acct_close_dt IS NULL;
```

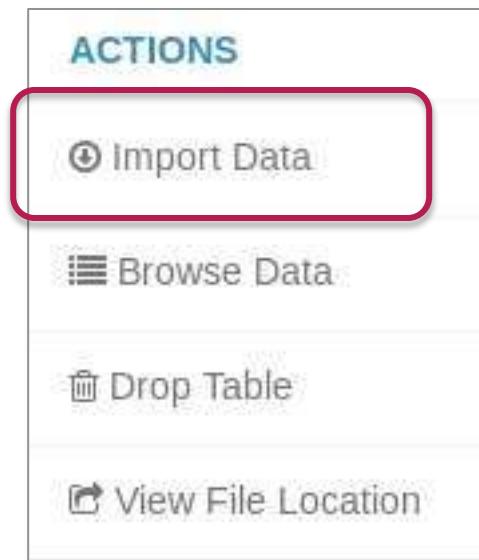
Loading Data Using the Metastore Manager

- The Metastore Manager provides two ways to load data into a table

Table creation wizard



Import data wizard



Loading Data From a Relational Database

- Sqoop has built-in support for importing data into Hive and Impala
- Add the **--hive-import** option to your Sqoop command
 - Creates the table in the Hive metastore
 - Imports data from the RDBMS to the table's directory in HDFS

```
$ sqoop import \
  --connect jdbc:mysql://localhost/loudacre \
  --username training \
  --password training \
  --fields-terminated-by '\t' \
  --table employees \
  --hive-import
```

- Note that **--hive-import** creates a table accessible in both Hive and Impala

Chapter Topics

Modeling and Managing Data With Impala and Hive

Importing and Modeling Structured Data

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- **HCatalog**
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

The Metastore and HCatalog

- **HCatalog is a Hive sub-project that provides access to the Metastore**
 - Accessible via command line and REST API
 - Allows you to define tables using HiveQL DDL syntax
 - Access those tables from Hive, Impala, MapReduce, Pig, and other tools
 - Included with CDH 4.2 and later

Creating Tables in HCatalog

- HCatalog uses Hive's DDL (data definition language) syntax
 - You can specify a single command using the `-e` option

```
$ hcat -e "CREATE TABLE vendors \
(id INT, company STRING, email STRING) \
ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' \
LOCATION '/dualcore/vendors'"
```

- Tip: save longer commands to a text file and use the `-f` option
 - If the file has more than one command, separate each with a semicolon

```
$ hcat -f createtable.txt
```

Displaying Metadata in HCatalog

- The SHOW TABLES command also shows tables created directly in Hive

```
$ hcat -e 'SHOW TABLES'  
employees  
vendors
```

- The DESCRIBE command lists the fields in a specified table
 - Use DESCRIBE FORMATTED instead to see detailed information

```
$ hcat -e 'DESCRIBE vendors'  
id      int  
company string  
email   string
```

Removing a Table in HCatalog

- The **DROP TABLE** command has the same behavior as it does in Hive and Impala
 - Caution: this will remove the data as well as the metadata (unless table is **EXTERNAL**)!

```
$ hcat -e 'DROP TABLE vendors'
```

Chapter Topics

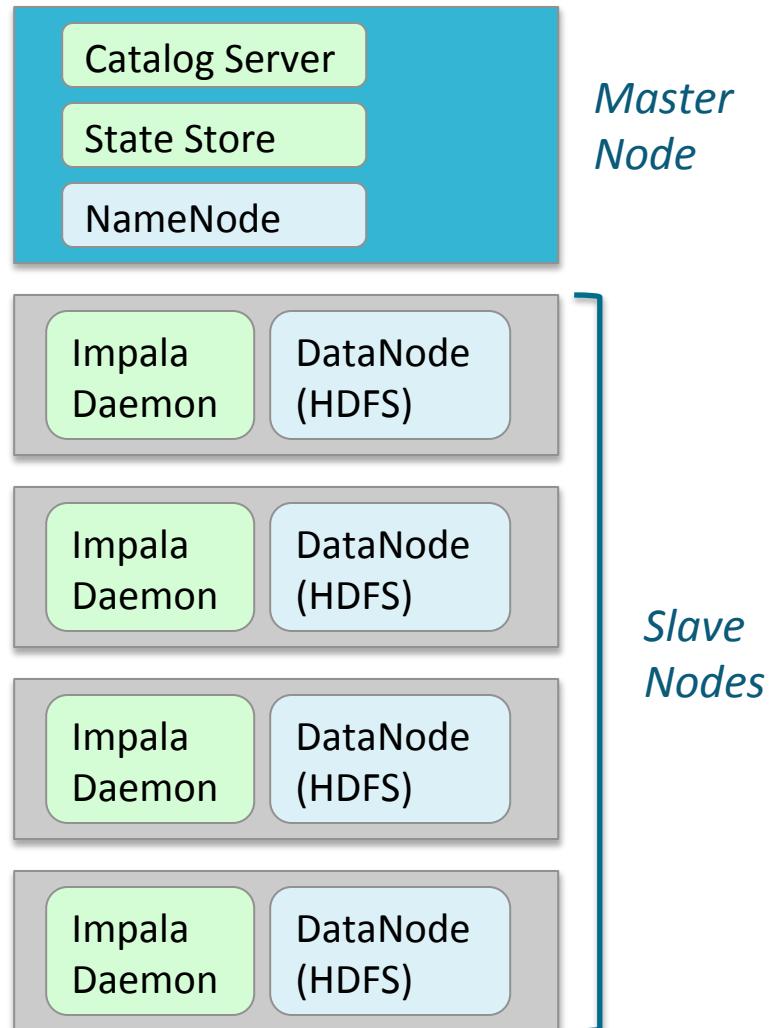
Modeling and Managing Data With Impala and Hive

Importing and Modeling Structured Data

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- **Impala Metadata Caching**
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

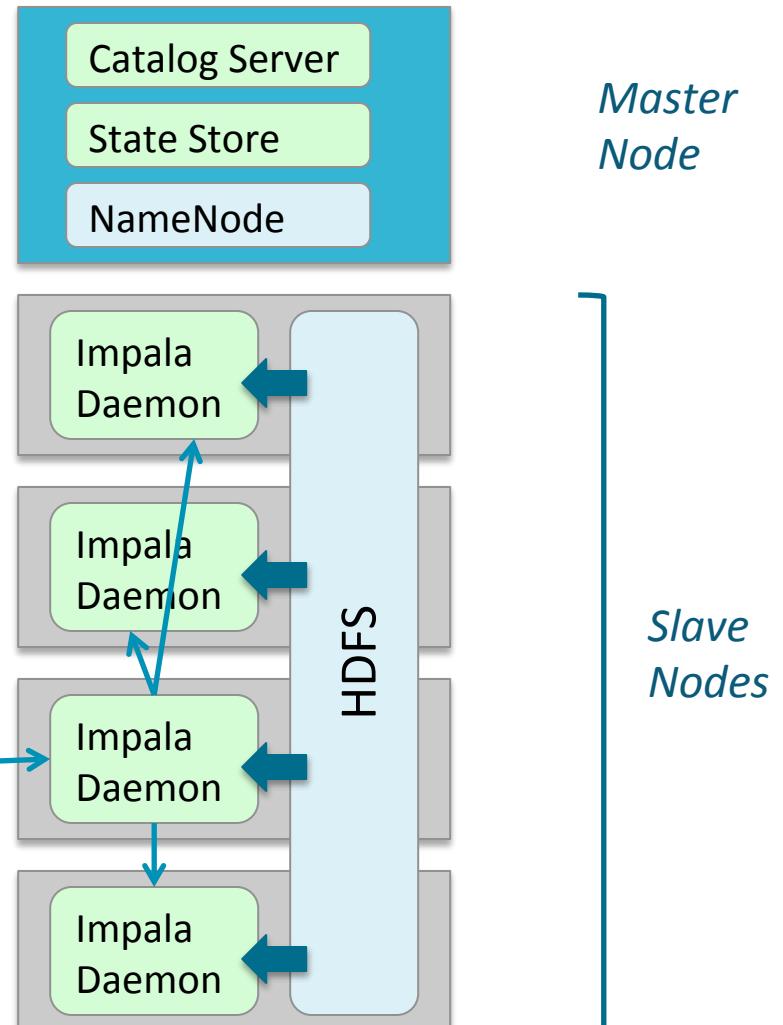
Impala in the Cluster

- **Each slave node in the cluster runs an Impala daemon**
 - Co-located with the HDFS slave daemon (DataNode)
- **Two other daemons running on master nodes support query execution**
 - The **State Store** daemon
 - Provides lookup service for Impala daemons
 - Periodically checks status of Impala daemons
 - The **Catalog** daemon
 - Relays metadata changes to all the Impala daemons in a cluster



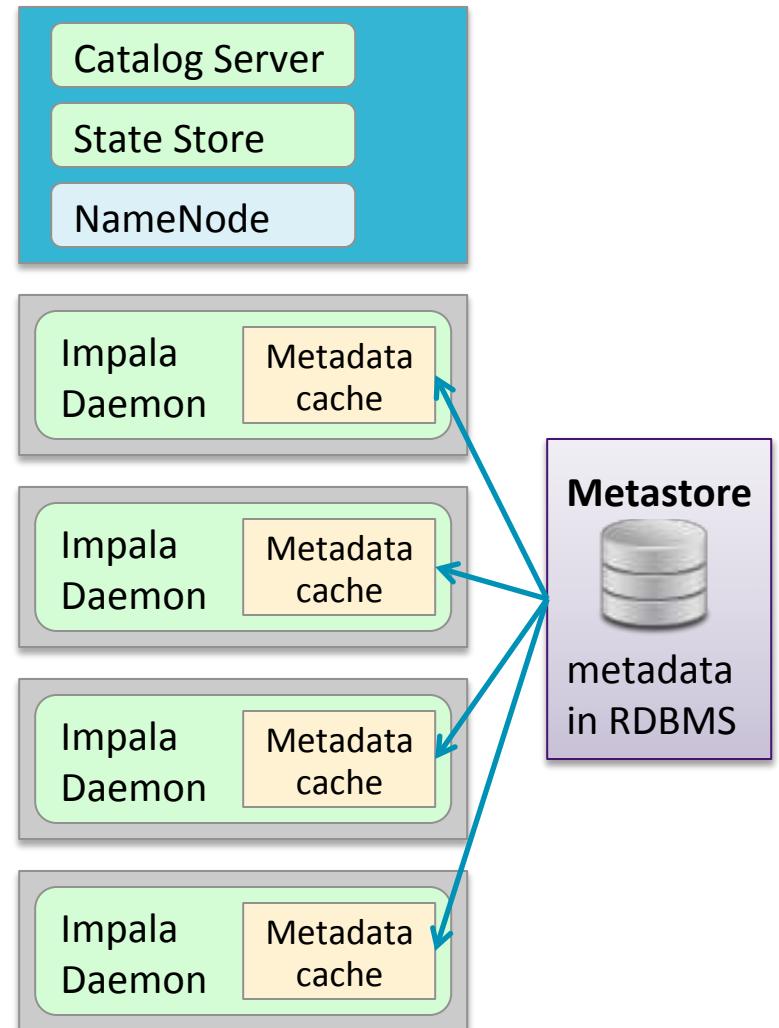
How Impala Executes a Query

- **Impala daemon plans the query**
 - Client (impala-shell or Hue) connects to a local impala daemon
 - This is the *coordinator*
 - Coordinator requests a list of other Impala daemons in the cluster from the State Store
 - Coordinator distributes the query across other Impala daemons
 - Streams results to client



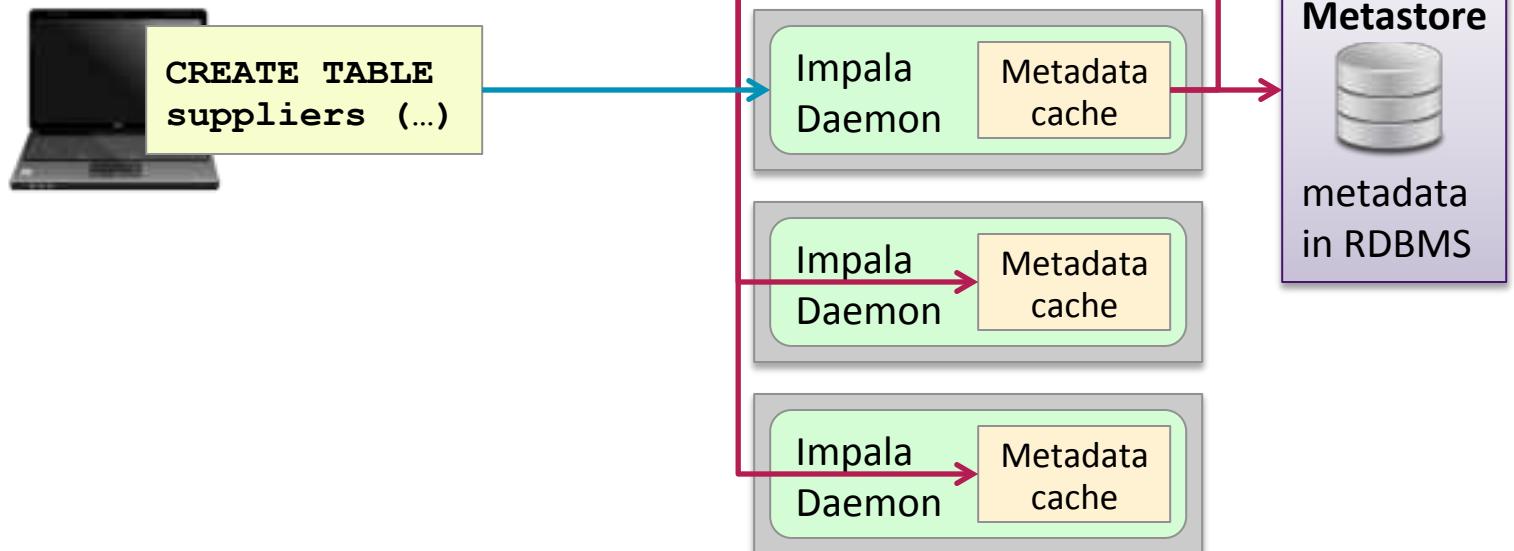
Metadata Caching (1)

- **Impala daemons cache metadata**
 - The tables' schema definitions
 - The locations of tables' HDFS blocks
- **Metadata is cached from the Metastore at startup**



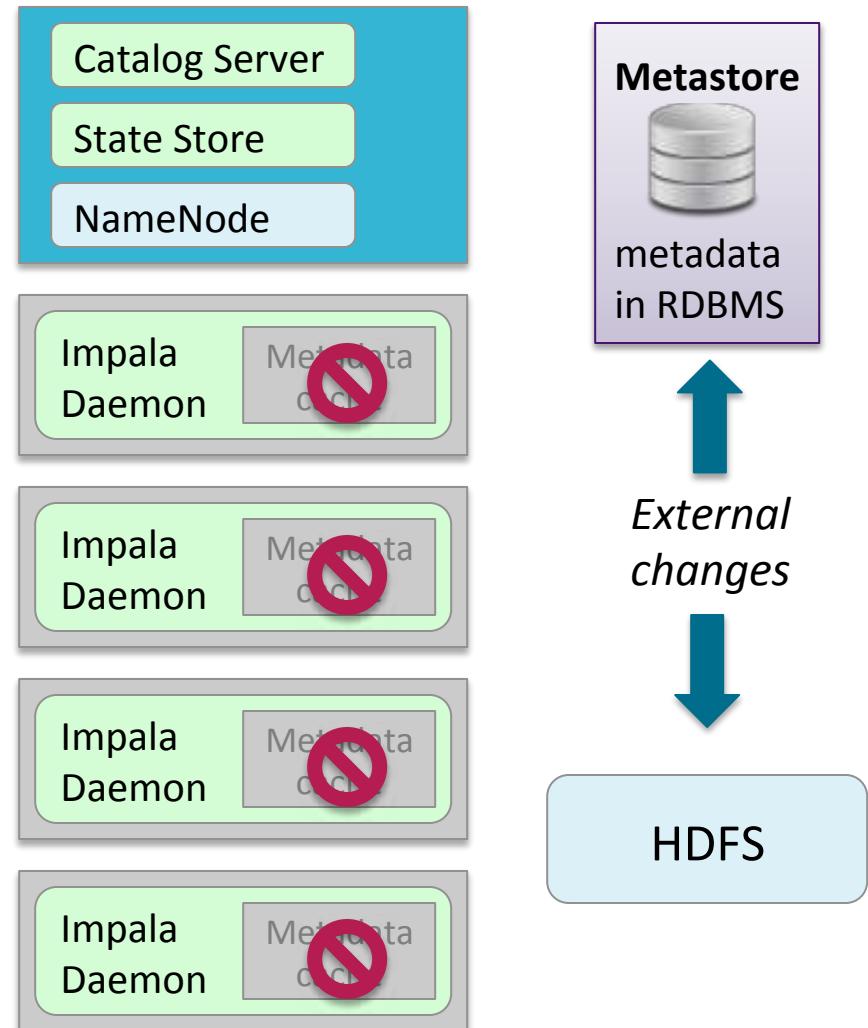
Metadata Caching (2)

- When one Impala daemon changes the metastore, it notifies the catalog service
- The catalog service notifies all Impala daemons to update their cache



External Changes and Metadata Caching

- **Metadata updates made *from outside of Impala* are not known to Impala, e.g.**
 - Changes via Hive, HCatalog or Hue Metadata Manager
 - Data added directly to directory in HDFS
- **Therefore the Impala metadata caches will be invalid**
- **You must manually refresh or invalidate Impala's metadata cache**



Updating the Impala Metadata Cache

External Metadata Change	Required Action	Effect on Local Caches
New table added	INVALIDATE METADATA (with no table name)	Marks the entire cache as stale; metadata cache is reloaded as needed.
Table schema modified <i>or</i> New data added to a table	REFRESH <table>	Reloads the metadata for one table <i>immediately</i> . Reloads HDFS block locations for new data files only.
Data in a table extensively altered, such as by HDFS balancing	INVALIDATE METADATA <table>	Marks the metadata for a single table as stale. When the metadata is needed, all HDFS block locations are retrieved.

Chapter Topics

Modeling and Managing Data With Impala and Hive

Importing and Modeling Structured Data

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- **Conclusion**
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

Essential Points

- **Each table maps to a directory in HDFS**
 - Table data is stored as one or more files
 - Default format: plain text with delimited fields
- **The Metastore stores data *about* the data in an RDBMS**
 - E.g. Location, column names and types
- **Tables are created and managed using the Impala SQL or HiveQL Data Definition Language**
- **Impala caches metadata from the Metastore**
 - Invalidate or refresh the cache if tables are modified outside Impala
- **HCatalog provides access to the Metastore from tools outside Hive or Impala (e.g. Pig, MapReduce)**

Bibliography

The following offer more information on topics discussed in this chapter

- **Impala Concepts and Architecture**
 - <http://tiny.cloudera.com/adcc12a>
- **Impala SQL Language Reference**
 - <http://tiny.cloudera.com/impalasql>
- **Impala-related Articles on Cloudera's Blog**
 - <http://tiny.cloudera.com/adcc12e>
- **Apache Hive Web Site**
 - <http://hive.apache.org/>
- **HiveQL Language Manual**
 - <http://tiny.cloudera.com/adcc10b>

Chapter Topics

Modeling and Managing Data With Impala and Hive

Importing and Modeling Structured Data

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- Conclusion
- **Hands-On Exercise: Create and Populate Tables in Impala or Hive**

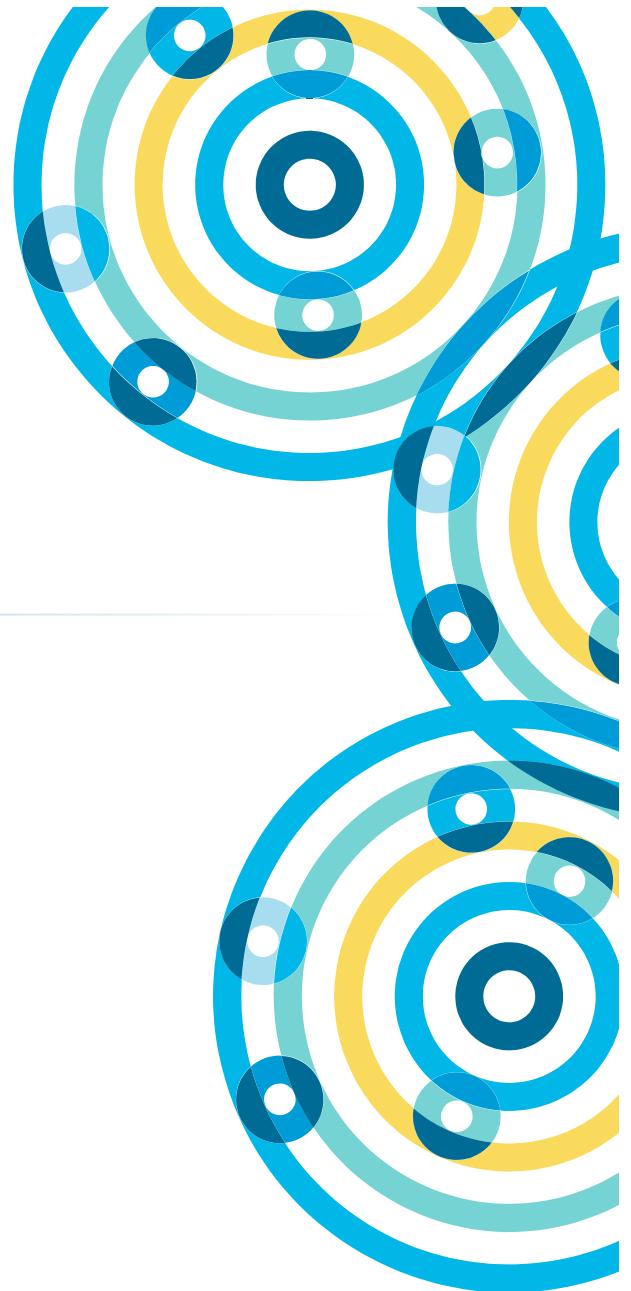
Hands-On Exercise: Create and Populate Tables in Impala

- **In this exercise you will**
 - Create a table in Impala to model and view existing data
 - Use Sqoop to create a new table automatically from data imported from MySQL
- **Please refer to the Hands-On Exercise Manual for instructions**



Data Formats

Chapter 7



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- **Data Formats**
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured Data

Ingesting Streaming Data

Distributed Data Processing with Spark

Course Conclusion

Data Formats

In this chapter you will learn

- **How to select the best data format for your needs**
- **How various Hadoop tools support different data formats**
- **How to define Avro schemas**
- **How Avro schemas can evolve to accommodate changing requirements**
- **How to extract data and metadata from an Avro data file**

Chapter Topics

Data Formats

Importing and Modeling Structured Data

■ File Formats

- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- Using Parquet with Impala, Hive and Sqoop
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

File Storage Formats

- In previous chapters you saw that alternate file formats were available
 - E.g., in Hive and Impala

```
CREATE TABLE tablename (colname DATATYPE, . . .)
  ROW FORMAT DELIMITED
    FIELDS TERMINATED BY char
    STORED AS format
```

- E.g., in Sqoop
 - *--as-format*
- What formats are available?
- Which should you choose and why?

Hadoop File Formats: Text Files

- **Text files are the most basic file type in Hadoop**
 - Can be read or written from virtually any programming language
 - Comma- and tab-delimited files are compatible with many applications
- **Text files are human readable, since everything is a string**
 - Useful when debugging
- **At scale, this format is inefficient**
 - Representing numeric values as strings wastes storage space
 - Difficult to represent binary data such as images
 - Often resort to techniques such as Base64 encoding
 - Conversion to/from native types adds performance penalty
- **Verdict: Good interoperability, but poor performance**

Hadoop File Formats: Sequence Files

- **SequenceFiles store key-value pairs in a binary container format**
 - Less verbose and more efficient than text files
 - Capable of storing binary data such as images
 - Format is Java-specific and tightly coupled to Hadoop
- **Verdict: Good performance, but poor interoperability**

Hadoop File Formats: Avro Data Files

- Efficient storage due to optimized binary encoding
- Widely supported throughout the Hadoop ecosystem
 - Can also be used outside of Hadoop
- Ideal for long-term storage of important data
 - Can read and write from many languages
 - Embeds schema in the file, so will always be readable
 - Schema evolution can accommodate changes
- Verdict: Excellent interoperability and performance
 - Best choice for general-purpose storage in Hadoop
- *More detail in coming slides*



Columnar Formats

- Hadoop also supports **columnar format**
 - These organize data storage by column, rather than by row
 - Very efficient when selecting only a small subset of a table's columns

id	name	city	occupation	income	phone
1	Alice	Palo Alto	Accountant	85000	650-555-9748
2	Bob	Sunnyvale	Accountant	81500	650-555-8865
3	Bob	Palo Alto	Dentist	196000	650-555-7185
4	Bob	Palo Alto	Manager	87000	650-555-2518
5	Carol	Palo Alto	Manager	79000	650-555-3951
6	David	Sunnyvale	Mechanic	62000	650-555-4754

Organization of data in traditional row-based formats

id	name	city	occupation	income	phone
1	Alice	Palo Alto	Accountant	85000	650-555-9748
2	Bob	Sunnyvale	Accountant	81500	650-555-8865
3	Bob	Palo Alto	Dentist	196000	650-555-7185
4	Bob	Palo Alto	Manager	87000	650-555-2518
5	Carol	Palo Alto	Manager	79000	650-555-3951
6	David	Sunnyvale	Mechanic	62000	650-555-4754

Organization of data in columnar formats

Hadoop File Formats: Parquet Files

- **Parquet is a columnar format developed by Cloudera and Twitter**
 - Supported in Spark, MapReduce, Hive, Pig, Impala, Crunch, and others
 - Schema metadata is embedded in the file (like Avro)
- **Uses advanced optimizations described in Google's Dremel paper**
 - Reduces storage space
 - Increases performance
- **Most efficient when adding many records at once**
 - Some optimizations rely on identifying repeated patterns
- **Verdict: Excellent interoperability and performance**
 - Best choice for column-based access patterns



Chapter Topics

Data Formats

Importing and Modeling Structured Data

- Selecting a File Format
- Hadoop Tool Support for File Formats
- **Avro Schemas**
- Using Avro with Hive and Sqoop
- Avro Schema Evolution
- Compression
- Conclusion
- Hands-On Exercise: Working with Avro Schemas

Data Serialization

- **To understand Avro, you must first understand *serialization***
 - A way of representing data in memory as a series of bytes
 - Allows you to save data to disk or send it across the network
 - *Deserialization* allows you to read that data back into memory
- **For example, how do you serialize the number 108125150?**
 - 4 bytes when stored as a Java `int`
 - 9 bytes when stored as a Java `String`
- **Many programming languages and libraries support serialization**
 - Such as `Serializable` in Java or `pickle` in Python
- **Backwards compatibility and cross-language support can be challenging**
 - Avro was developed to address these challenges

What is Apache Avro?

- **Avro Data File Format is just one part of the Avro project**
 - But it is the part this course focuses on
- **Avro is an efficient data serialization framework**
 - Top-level Apache project created by Doug Cutting (creator of Hadoop)
 - Widely supported throughout Hadoop and its ecosystem
- **Offers compatibility without sacrificing performance**
 - Data is serialized according to a *schema* you define
 - Read/write data in Java, C, C++, C#, Python, PHP, and other languages
 - Serializes data using a highly-optimized binary encoding
 - Specifies rules for *evolving* your schema over time
- **Avro also supports Remote Procedure Calls (RPC)**
 - Can be used for building custom network protocols
 - Flume uses this for internal communication

Supported Types in Avro Schemas (Simple)

- A simple type holds exactly one value

Name	Description	Java Equivalent
null	An absence of a value	null
boolean	A binary value	boolean
int	32-bit signed integer	int
long	64-bit signed integer	long
float	Single-precision floating point value	float
double	Double-precision floating point value	double
bytes	Sequence of 8-bit unsigned bytes	java.nio.ByteBuffer
string	Sequence of Unicode characters	java.lang.CharSequence

Supported Types in Avro Schemas (Complex)

- Avro also supports complex types

Name	Description
record	A user-defined type composed of one or more named fields
enum	A specified set of values
array	Zero or more values of the same type
map	Set of key-value pairs; key is string while value is of specified type
union	Exactly one value matching a specified set of types
fixed	A fixed number of 8-bit unsigned bytes

- The **record** type is the most important
 - Main use of other types is to define a record's fields

Basic Schema Example

- Excerpt from a SQL CREATE TABLE statement

```
CREATE TABLE employees
  (id INT, name STRING, title STRING, bonus INT)
```

- Equivalent Avro schema

```
{"namespace": "com.loudacre.data",
"type": "record",
"name": "Employee",
"fields": [
    {"name": "id", "type": "int"},
    {"name": "name", "type": "string"},
    {"name": "title", "type": "string"},
    {"name": "bonus", "type": "int"}]
```

Specifying Default Values in the Schema

- Avro also supports setting a default value in the schema
 - Used when no value was explicitly set for a field
 - Similar to SQL

```
{"namespace": "com.loudacre.data",
"type": "record",
"name": "Invoice",
"fields": [
    {"name": "id", "type": "int"},
    {"name": "taxcode", "type": "int", "default": "39"},
    {"name": "lang", "type": "string", "default": "EN_US"}]
```

The `taxcode` and `lang` fields have default values

Avro Schemas and Null Values

- Avro checks for null values when serializing the data
- Null values are only allowed *when explicitly specified* in the schema

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "Employee",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name", "type": "string"},
     {"name": "title", "type": ["null", "string"]},
     {"name": "bonus", "type": ["null", "int"]}]
}
```

The **title** and **bonus** fields allow null values

Schema Example with Complex Types

- The following example shows a record with an enum and a string array

```
{ "namespace" : "com.loudacre.data",
  "type" : "record",
  "name" : "CustomerServiceTicket",
  "fields" : [
    { "name" : "id", "type" : "int" },
    { "name" : "agent", "type" : "string" },
    { "name" : "category", "type" : {
        "name" : "CSCategory", "type" : "enum",
        "symbols" : [ "Order", "Shipping", "Device" ] }
    },
    { "name" : "tags", "type" : {
        "type" : "array", "items" : "string" }
    }
}
```

The **category** field has three enumerated possible values

tags is an array of strings

Documenting Your Schema

- It's a good practice to document any ambiguities in a schema
 - All types (including record) support an optional doc attribute

```
{ "namespace": "com.loudacre.data",
  "type": "record",
  "name": "WebProduct",
  "doc": "Item currently sold in Loudacre's online store",
  "fields": [
    {"name": "id", "type": "int", "doc": "Product SKU"},
    {"name": "shipwt", "type": "int",
     "doc": "Shipping weight, in pounds"},
    {"name": "price", "type": "int",
     "doc": "Retail price, in cents (US)"}]
}
```

Avro Container Format

- **Avro also defines a container file format for storing Avro records**
 - Also known as “Avro data file format”
 - Similar to Hadoop SequenceFile format
 - Cross-language support for reading and writing data
- **Supports compressing *blocks* (groups) of records**
 - It is “splittable” for efficient processing in Hadoop
- **This format is self-describing**
 - Each file contains a copy of the schema used to write its data
 - All records in a file must use the same schema

Inspecting Avro Data Files with Avro Tools

- **Avro data files are an efficient way to store data**
 - However, the binary format makes debugging difficult
- **Each Avro release contains an Avro Tools JAR file**
 - Allows you to read the schema or data for an Avro file
 - Included with CDH 5 and later
 - Available for download from the Avro Web site or Maven repository

```
$ java -jar avro-tools*.jar tojson mydatafile.avro
{"name": "Alice", "salary": 56500, "city": "Anaheim"}
 {"name": "Bob", "salary": 51400, "city": "Bellevue"}

$ java -jar avro-tools*.jar getschema mydatafile.avro
{
  "type" : "record",
  "name" : "DeviceData",
  "namespace" : "com.loudacre.data", ...rest of schema follows
```

Chapter Topics

Data Formats

Importing and Modeling Structured Data

- Selecting a File Format
- Hadoop Tool Support for File Formats
- Avro Schemas
- Using Avro with Impala, Hive and Sqoop
- **Avro Schema Evolution**
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Schema Evolution

- **The structure of your data will change over time**
 - Fields may be added, removed, changed, or renamed
 - In SQL, these are handled with **ALTER TABLE** statements
- **These changes can break compatibility with many formats**
 - Objects serialized in **SequenceFiles** become unreadable
- **Data written to Avro data files is always readable**
 - The schema used to write the data is embedded in the file itself
 - However, an application reading data might expect the *new* structure
- **Avro has a unique approach to maintaining forward compatibility**
 - A reader can use a different schema than the writer

Schema Evolution: A Practical Example (1)

- Imagine that we have written millions of records with this schema

```
{ "namespace": "com.loudacre.data",
  "type": "record",
  "name": "CustomerContact",
  "fields": [
    { "name": "id", "type": "int" },
    { "name": "name", "type": "string" },
    { "name": "faxNumber", "type": "string" }
  ]
}
```

Schema Evolution: A Practical Example (2)

- We would like to modernize this based on the schema below
 - Rename id field to customerId and change type from int to long
 - Remove faxNumber field
 - Add prefLang field
 - Add email field

```
{ "namespace": "com.loudacre.data",
  "type": "record",
  "name": "CustomerContact",
  "fields": [
    {"name": "customerId", "type": "long"},
    {"name": "name", "type": "string"},
    {"name": "prefLang", "type": "string"},
    {"name": "email", "type": "string"}
  ]
}
```

Schema Evolution: A Practical Example (3)

- **We could use the new schema to write new data**
 - Applications that use the new schema could read the new data
- **Unfortunately, new applications wouldn't be able to read the *old* data**
 - We must make a few schema changes to improve compatibility

Schema Evolution: A Practical Example (4)

- If you rename a field, you must specify an alias for the old name(s)
 - Here, we map the old `id` field to the new `customerId` field

```
{ "namespace": "com.loudacre.data",
  "type": "record",
  "name": "CustomerContact",
  "fields": [
    { "name": "customerId", "type": "long",
      "aliases": ["id"] },
    { "name": "name", "type": "string" },
    { "name": "prefLang", "type": "string" },
    { "name": "email", "type": "string" }
  ]
}
```

Schema Evolution: A Practical Example (5)

- Newly-added fields will lack values for records previously written
 - You must specify a default value

```
{ "namespace": "com.loudacre.data",
  "type": "record",
  "name": "CustomerContact",
  "fields": [
    { "name": "customerId", "type": "long",
      "aliases": ["id"] },
    { "name": "name", "type": "string" },
    { "name": "prefLang", "type": "string",
      "default": "en_US" },
    { "name": "email",
      "type": ["null", "string"], "default": null }
  ]
}
```

Default value for
prefLang is
`en_US`

`email` is nullable
so `null` can be the
default

Schema Evolution: Compatible Changes

- **The following changes will not affect existing readers**

- Adding, changing, or removing a `doc` attribute
- Changing a field's default value
- Adding a new field with a default value
- Removing a field that specified a default value
- Promoting a field to a wider type (e.g., `int` to `long`)
- Adding aliases for a field

Schema Evolution: Incompatible Changes

- **The following are some changes that might break compatibility**
 - Changing the record's name or namespace attributes
 - Adding a new field without a default value
 - Removing a symbol from an enum
 - Removing a type from a union
 - Modifying a field's type to one that could result in truncation
- **To handle these incompatibilities**
 1. Read your old data (using the original schema)
 2. Modify data as needed in your application
 3. Write the new data (using the new schema)
- **Existing readers/writers may need to be updated to use new schema**

Chapter Topics

Data Formats

Importing and Modeling Structured Data

- Selecting a File Format
- Hadoop Tool Support for File Formats
- Avro Schemas
- Avro Schema Evolution
- **Using Avro with Impala, Hive and Sqoop**
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Using Avro with Sqoop

- Sqoop supports importing data as Avro, or exporting data from existing Avro data files
- `sqoop import` saves the schema JSON file in local directory

```
$ sqoop import \
--connect jdbc:mysql://localhost/loudacre \
--username training --password training \
--table accounts \
--target-dir /loudacre/accounts_avro \
--as-avrodatafile
```

Using Avro with Impala and Hive (1)

- **Hive and Impala support Avro**
 - Hive supports all Avro types
 - Impala does not support complex types
 - **enum, array**, etc.

Using Avro with Impala and Hive (2)

- Table creation can include schema inline or in a separate file

```
CREATE TABLE order_details_avro
  STORED AS AVRO
  TBLPROPERTIES ('avro.schema.url'=
    'hdfs://localhost/loudacre/accounts_schema.json');
```

```
CREATE TABLE order_details_avro
  STORED AS AVRO
  TBLPROPERTIES ('avro.schema.literal'=
    ' { "name": "order",
      "type": "record",
      "fields": [
        { "name": "order_id", "type": "int" },
        { "name": "cust_id", "type": "int" },
        { "name": "order_date", "type": "string" }
      ] }');
```

Chapter Topics

Data Formats

Importing and Modeling Structured Data

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- **Using Parquet with Impala, Hive and Sqoop**
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Using Parquet with Swoop

- Swoop supports importing data as Parquet, or exporting data from existing Parquet data files
 - Use the **--as-parquetfile** option

```
$ swoop import \
--connect jdbc:mysql://localhost/loudacre \
--username training --password training \
--table accounts \
--target-dir /loudacre/accounts_parquet \
--as-parquetfile
```

Using Parquet with Hive and Impala (1)

- Create a new table stored in Parquet format

```
CREATE TABLE order_details_parquet (
    order_id INT,
    prod_id INT)
STORED AS PARQUET;
```

* **STORED AS PARQUET** supported in Impala, and in Hive 0.13 and later

Using Parquet with Hive and Impala (2)

- In Impala, use **LIKE PARQUET** to use column metadata from an existing Parquet data file
- Example: Create a new table to access existing Parquet format data



```
CREATE EXTERNAL TABLE ad_data
  LIKE PARQUET '/loudacre/ad_data/datafile1.parquet'
  STORED AS PARQUET
  LOCATION '/loudacre/ad_data/' ;
```

Chapter Topics

Data Formats

Importing and Modeling Structured Data

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- Using Parquet with Impala, Hive and Sqoop
- **Compression**
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Performance Considerations

- You have just learned how different file formats affect performance
- Another factor can significantly affect performance: data compression

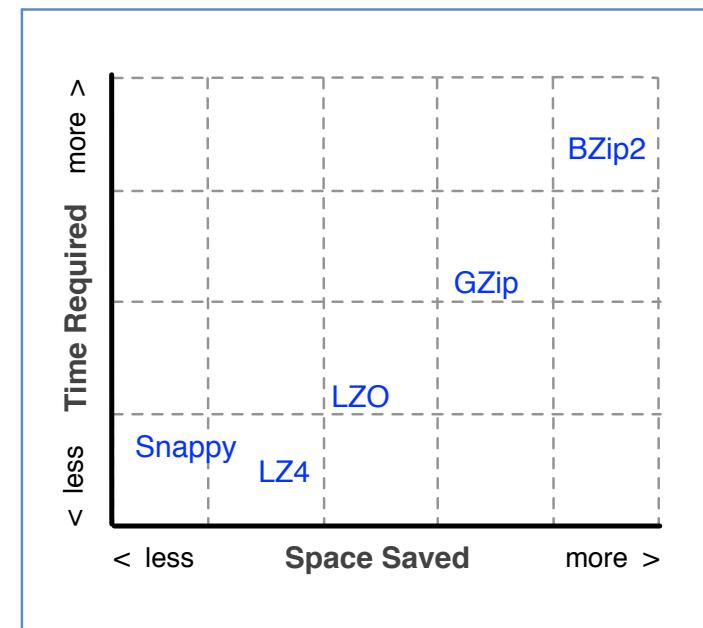
Data Compression

- **Each file format may also support compression**
 - This reduces amount of disk space required to store data
- **Compression is a tradeoff between CPU time and bandwidth/storage space**
 - Aggressive algorithms take a long time, but save more space
 - Less aggressive algorithms save less space but are much faster
- **Can significantly improve performance**
 - Many Hadoop jobs are I/O-bound
 - Using compression allows you to handle more data per I/O operation
 - Compression can also improve the performance of network transfers



Compression Codecs

- The implementation of a compression algorithm is known as a *codec*
 - Short for compressor/decompressor
- Many codecs are commonly used with Hadoop
 - Each has different performance characteristics
 - Not all Hadoop tools are compatible with all codecs
- Overall, BZip2 saves the most space
 - But LZ4 and Snappy are much faster
 - Impala supports Snappy but not LZ4
- For "hot" data, speed matters most
 - Better to compress by 40% in one second than by 80% in 10 seconds



Using Compression With Sqoop

- Sqoop – use **--compression-codec** flag

- Example

```
--compression-codec  
org.apache.hadoop.io.compress.SnappyCodec
```

Using Compression With Impala and Hive

- Not all file format/compression combinations are supported
 - Properties and syntax varies
 - See the documentation for a full list of supported formats and codecs for Impala and Hive
 - Caution: Impala queries data in memory – both compressed and uncompressed data are stored in memory
- Impala example

```
> CREATE TABLE mytable_parquet LIKE mytable_text  
      STORED AS PARQUET;  
> set PARQUET_COMPRESSION_CODEC=snappy;  
> INSERT INTO mytable_parquet  
      SELECT * FROM mytable_text;
```

Chapter Topics

Data Formats

Importing and Modeling Structured Data

- Selecting a File Format
- Hadoop Tool Support for File Formats
- Avro Schemas
- Using Avro with Hive and Sqoop
- Avro Schema Evolution
- Compression
- **Conclusion**
- Hands-On Exercise: Select a Format for a Data File

Essential Points (1)

- **Hadoop and its ecosystem support many file formats**
 - May ingest data in one format, but convert to another as needed
- **Selecting the format for your data set involves several considerations**
 - Ingest pattern
 - Tool compatibility
 - Expected lifetime
 - Storage and performance requirements

Essential Points (2)

- **Choose from the three main Hadoop file format options**
 - Text – Good for testing and interoperability
 - Avro – Best for general purpose performance and evolving schemas
 - Parquet – Best performance for column-oriented access patterns
- **Avro is a serialization framework that includes a data file format**
 - Compact binary encodings provide good performance
 - Supports schema evolution for long-term storage
- **Compression saves disk storage space and IO times at the cost of CPU time**
 - Hadoop tools support a number of different codecs

Bibliography

The following offer more information on topics discussed in this chapter

- **Avro Getting Started Guide (Java)**
 - <http://tiny.cloudera.com/adcc03a>
- **Avro Specification**
 - <http://tiny.cloudera.com/adcc03b>
- **Parquet**
 - <https://parquet.apache.org>
- **Announcing Parquet 1.0: Columnar Storage for Hadoop**
 - <http://tiny.cloudera.com/adcc03c>

Chapter Topics

Data Formats

Importing and Modeling Structured Data

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- Using Parquet with Impala, Hive and Sqoop
- Compression
- Conclusion
- **Hands-On Exercise: Select a Format for a Data File**

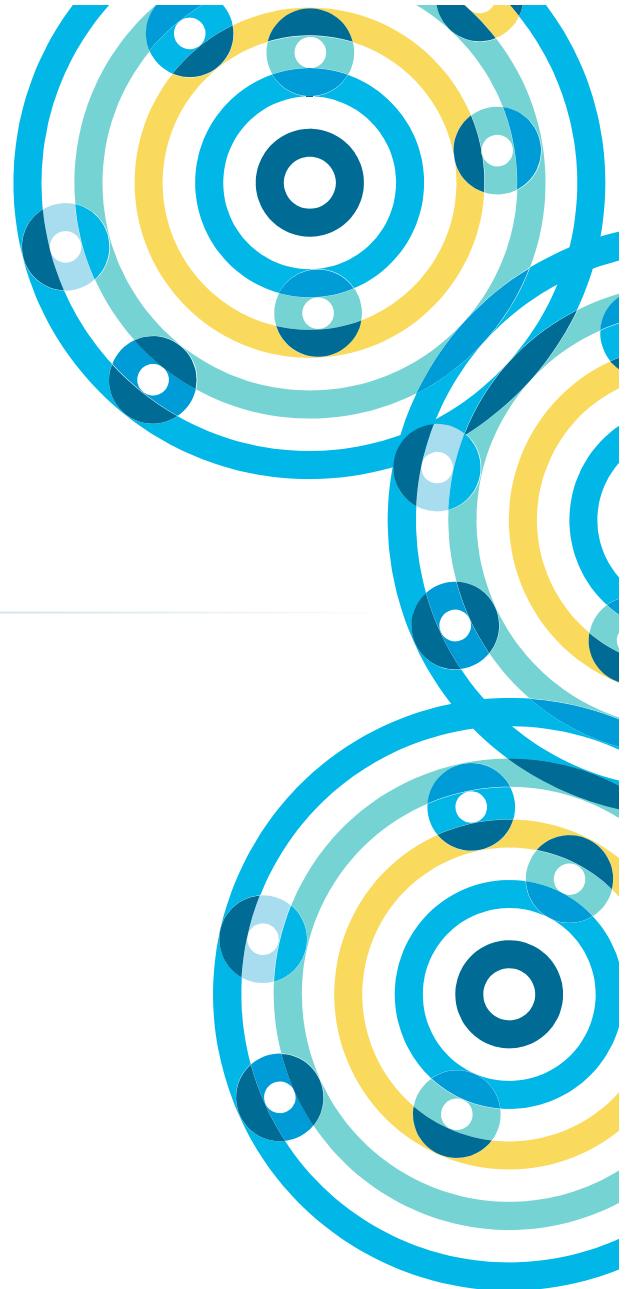
Hands-On Exercise: Select a Format for a Data File

- **In this exercise you will**
 - Use Sqoop to import the accounts table in Avro format
 - Define an Impala table to access the Avro accounts data
 - Bonus: Save an existing plain text Impala table as Parquet
- **Please refer to the Hands-On Exercise Manual for instructions**



Data File Partitioning

Chapter 8



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- **Data File Partitioning**
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured Data

Ingesting Streaming Data

Distributed Data Processing with Spark

Course Conclusion

Data File Partitioning

In this chapter you will learn

- **How to improve query performance with data file partitioning**
- **How to create and populate partitioned tables in Impala and Hive**

Chapter Topics

Data File Partitioning

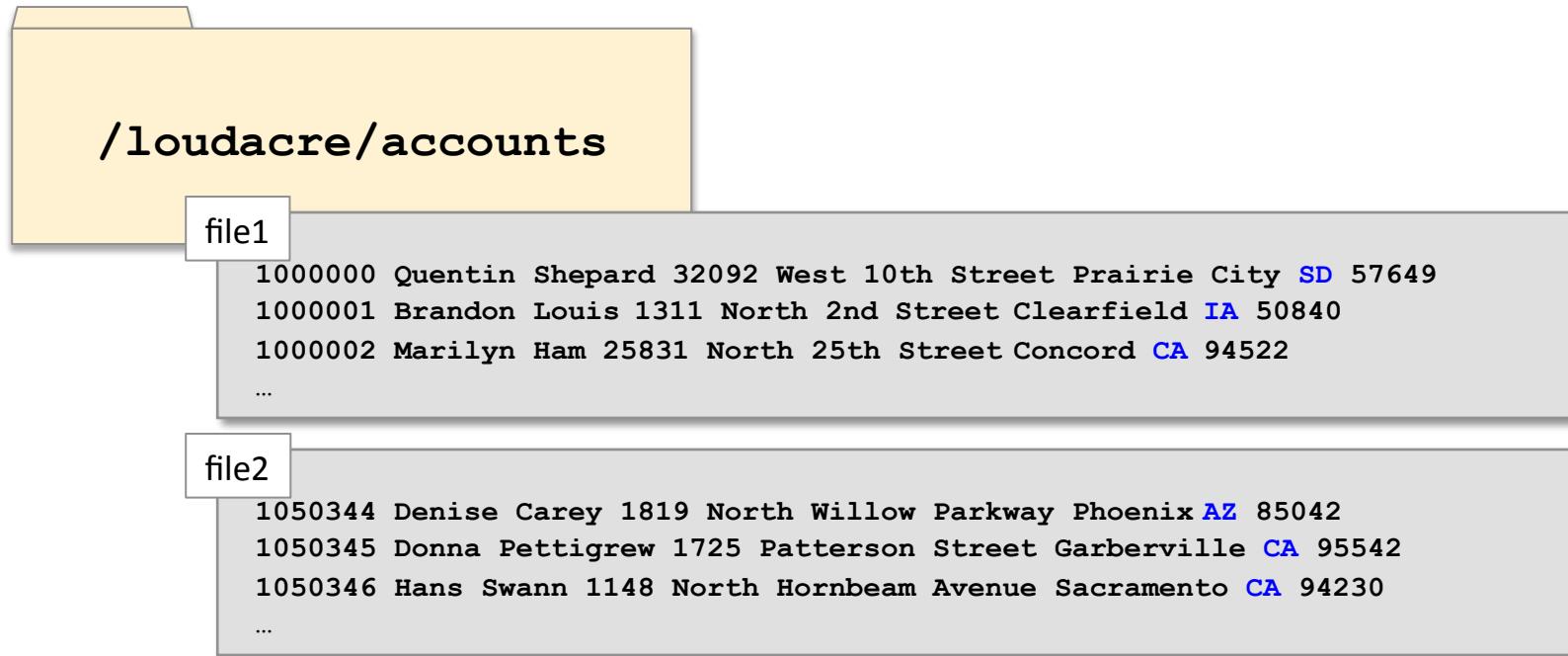
Importing and Modeling Structured Data

■ Partitioning Overview

- Partitioning in Impala and Hive
- Conclusion
- Hands-On Exercise: Partition Data in Impala or Hive

Data Storage Partitioning (1)

- By default, all files in a data set are stored in a single HDFS directory
 - All files in the directory are read during analysis or processing
 - “Full table scan”

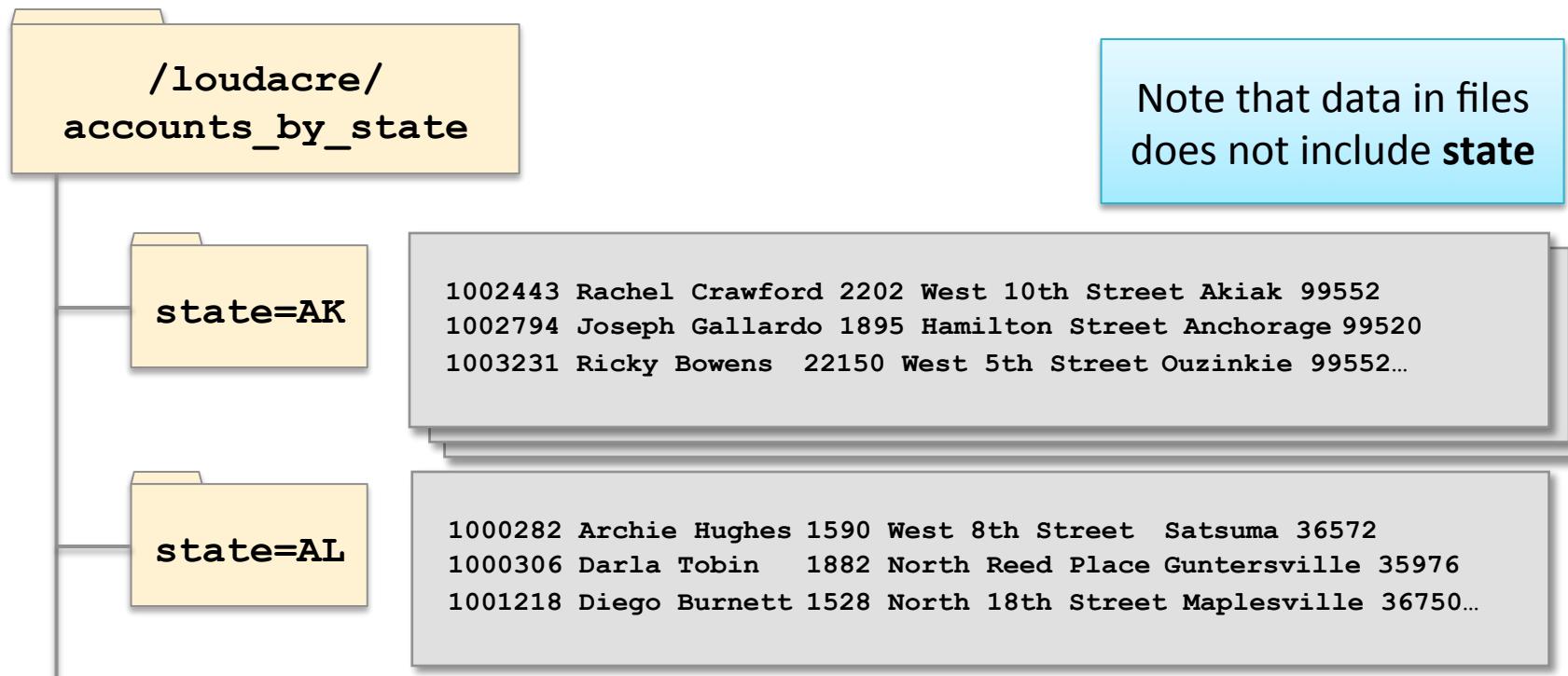


Data Storage Partitioning (2)

- **Partitioning** subdivides the data

- Analysis can be done on only the relevant subset of data
 - Potentially much faster!

- Hadoop partitions using subdirectories



Hadoop Partitioning

- **Partitioning is involved at two phases**
 - Storage – putting the data into correct partition (subdirectory)
 - Retrieval – getting the data out of the correct partition based on the query or analysis being done
- **Hadoop with built-in support for partitioning**
 - Hive and Impala (covered in next section)
 - Sqoop – When using the `--hive-import` option you can specify flags `--hive-partition-key` and `--hive-partition-value`
- **Other tools can be used to store partitioned data**
 - Spark and MapReduce
 - Flume (at ingestion)

Chapter Topics

Data File Partitioning

Importing and Modeling Structured Data

- Partitioning Overview
- **Partitioning in Impala and Hive**
- Conclusion
- Hands-On Exercise: Partition Data in Impala or Hive

Example: Impala/Hive Partitioning Accounts By State (1)

- Example: accounts is a non-partitioned table

```
CREATE EXTERNAL TABLE accounts(
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    state STRING,
    zipcode STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/loudacre/accounts';
```

Example: Impala/Hive Partitioning Accounts By State (2)

- What if most of Loudacre's analysis on the customer table was done by state? For example:

```
SELECT fname, lname  
      FROM accounts  
     WHERE state='NY';
```

- By default, all queries have to scan *all* files in the directory
- Use partitioning to store data in separate files by state
 - State-based queries scan only the relevant files

Example: Impala/Hive Partitioning Accounts By State (3)

- Create a partitioned table using PARTITIONED BY

```
CREATE EXTERNAL TABLE accounts_by_state(
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    state STRING,
    zipcode STRING)
PARTITIONED BY (state STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LOCATION '/loudacre/accounts_by_state';
```

Partition Columns

- The partition column is displayed if you DESCRIBE the table

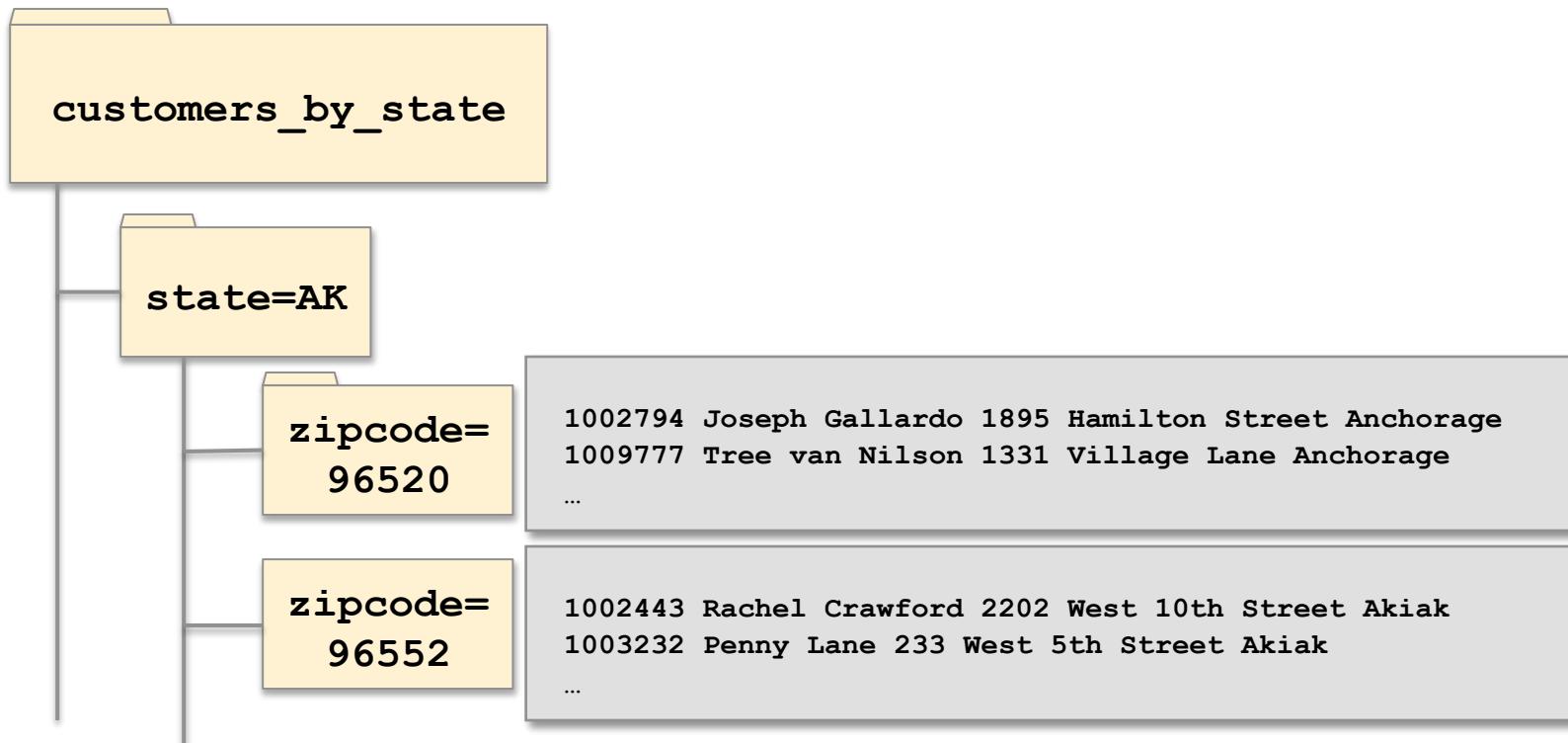
```
DESCRIBE accounts_by_state;
+-----+-----+-----+
| name      | type      | comment |
+-----+-----+-----+
| cust_id   | int       |          |
| fname     | string    |          |
| lname     | string    |          |
| address   | string    |          |
| city      | string    |          |
| zipcode   | string    |          |
| state     | string    |          |
+-----+-----+-----+
```

A partition column is a “virtual column”; data is not stored in the file

Nested Partitions

- You can also create nested partitions

```
... PARTITIONED BY (state STRING, zipcode STRING)
```



Loading Data Into a Partitioned Table

- **Dynamic partitioning**

- Impala/Hive add new partitions automatically as needed at load time
 - Data is stored into the correct partition (subdirectory) based on column value

- **Static partitioning**

- You define new partitions using ADD PARTITION
 - When loading data, you specify which partition to store data in

Dynamic Partitioning

- We can create new partitions dynamically from existing data

```
INSERT OVERWRITE TABLE accounts_by_state
PARTITION(state)
SELECT cust_id, fname, lname, address,
city, zipcode, state FROM accounts;
```

- Partitions are automatically created based on the value of the *last* column
 - If the partition does not already exist, it will be created
 - If the partition does exist, it will be overwritten

Static Partitioning Example: Partition Calls by Day (1)

- Loudacre's customer service phone system generates logs detailing calls received
 - Analysts use this data to summarize previous days' calls
 - For example:

```
SELECT event_type, COUNT(event_type)
  FROM call_log
 WHERE call_date = '2014-10-01'
 GROUP BY event_type;
```

Static Partitioning Example: Partition Calls by Day (2)

- Logs are generated daily, e.g.

call-20141001.log

```
19:45:19,312-555-7834,CALL_RECEIVED  
19:45:23,312-555-7834,OPTION_SELECTED,Shipping  
19:46:23,312-555-7834,ON_HOLD  
19:47:51,312-555-7834,AGENT_ANSWER,Agent ID N7501  
19:48:37,312-555-7834,COMPLAINT,Item not received  
19:48:41,312-555-7834,CALL_END,Duration: 3:22
```

...

call-20141002.log

```
03:45:01,505-555-2345,CALL_RECEIVED  
03:45:09,505-555-2345,OPTION_SELECTED,Billing  
03:56:21,505-555-2345,AGENT_ANSWER,Agent ID A1503  
03:57:01,505-555-2345,QUESTION
```

...

Static Partitioning Example: Partition Calls by Day (3)

- In the previous example, existing data was partitioned dynamically based on a column value
- This time we use static partitioning
 - Because the data files do not include the partitioning data

Static Partitioning Example: Partition Calls by Day (4)

- The partitioned table is defined the same way

```
CREATE TABLE call_logs (
    call_time STRING,
    phone STRING,
    event_type STRING,
    details STRING)
PARTITIONED BY (call_date STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ',';
```

Loading Data Into Static Partitions (1)

- With static partitioning, you create new partitions as needed
- e.g. For each new day of call log data, add a partition:

```
ALTER TABLE call_logs
ADD PARTITION (call_date='2014-10-02');
```

- This command
 1. Adds the partition to the table's metadata
 2. Creates subdirectory
/user/hive/warehouse/call_logs/
call_date=2014-10-02

Loading Data Into Static Partitions (2)

- Then load the day's data into the correct partition

```
LOAD DATA INPATH '/mystaging/call-20141002.log'  
    INTO TABLE call_logs  
    PARTITION(call_date='2014-10-02');
```

- This command moves the HDFS file `call-20141002.log` to the partition subdirectory
- To overwrite all data in a partition

```
LOAD DATA INPATH '/mystaging/call-20141002.log'  
    INTO TABLE call_logs OVERWRITE  
    PARTITION(call_date='2014-10-02');
```

Hive Only: Shortcut for Loading Data Into Partitions

- Hive will create a new partition if the one specified doesn't exist



```
LOAD DATA INPATH '/mystaging/call-20141002.log'  
    INTO TABLE call_logs  
    PARTITION(call_date='2014-10-02');
```

- This command

1. Adds the partition to the table's metadata if it doesn't exist
2. Creates subdirectory
`/user/hive/warehouse/call_logs/call_date=2014-10-02` if it doesn't exist
3. Moves the HDFS file `call-20141002.log` to the partition subdirectory

Viewing, Adding, and Removing Partitions

- To view the current partitions in a table

```
SHOW PARTITIONS call_logs;
```

- Use ALTER TABLE to add or drop partitions

```
ALTER TABLE call_logs
  ADD PARTITION (call_date='2013-06-05')
  LOCATION '/loudacre/call_logs/call_date=2013-06-05' ;
```

```
ALTER TABLE call_logs
  DROP PARTITION (call_date='2013-06-06') ;
```

Creating Partitions from Existing Partition Directories in HDFS

- Partition directories in HDFS can be created and populated outside Hive or Impala
 - For example, by a Spark or MapReduce application
- In Hive, use the `MSCK REPAIR TABLE` command to create (or recreate) partitions for an existing table



```
MSCK REPAIR TABLE call_logs;
```

When To Use Partitioning

- **Use partitioning for tables when**

- Reading the entire data set takes too long
 - Queries almost always filter on the partition columns
 - There are a reasonable number of different values for partition columns
 - The data generation or ETL process segments data by file or directory names
 - Partition column values are not in the data itself

When Not To Use Partitioning

- **Avoid partitioning data into numerous small data files**
 - Don't partition on columns with too many unique values
- **Caution: This can happen easily when using dynamic partitioning!**
 - For example, partitioning customers by first name could produce thousands of partitions



Partitioning in Hive (1)

- In older versions of Hive, dynamic partitioning is not enabled by default
 - Enable it by setting these two properties

```
SET hive.exec.dynamic.partition=true;  
SET hive.exec.dynamic.partition.mode=nonstrict;
```

- Note: Hive variables set in Beeline are for the current session only
 - Your system administrator can configure settings permanently



Partitioning in Hive (2)

- Caution: if the partition column has many unique values, many partitions will be created
- Three Hive configuration properties exist to limit this
 - **hive.exec.max.dynamic.partitions.pernode**
 - Maximum number of dynamic partitions that can be created by any given node involved in a query
 - Default 100
 - **hive.exec.max.dynamic.partitions**
 - Total number of dynamic partitions that can be created by one HiveQL statement
 - Default 1000
 - **hive.exec.max.created.files**
 - Maximum total files (on all nodes) created by a query
 - Default 100000

Chapter Topics

Data File Partitioning

Importing and Modeling Structured Data

- Partitioning Overview
- Partitioning in Impala and Hive
- **Conclusion**
- Hands-On Exercise: Partition Data in Impala or Hive

Essential Points

- Partitioning splits table storage by column values for improved query performance
- Partitions are HDFS directories
 - Names follow the format *column=value*
- Partitions can be defined and loaded dynamically or statically
- Only partition on columns with a reasonable number of possible values

Bibliography

The following offer more information on topics discussed in this chapter

- **Impala documentation on partitioning**
 - <http://tiny.cloudera.com/impalapart>
- **Improving Query Performance Using Partitioning in Apache Hive (Cloudera Engineering Blog)**
 - <http://tiny.cloudera.com/partblog>

Chapter Topics

Data File Partitioning

Importing and Modeling Structured Data

- Partitioning Overview
- Partitioning in Impala and Hive
- Conclusion
- **Hands-On Exercise: Partition Data in Impala or Hive**

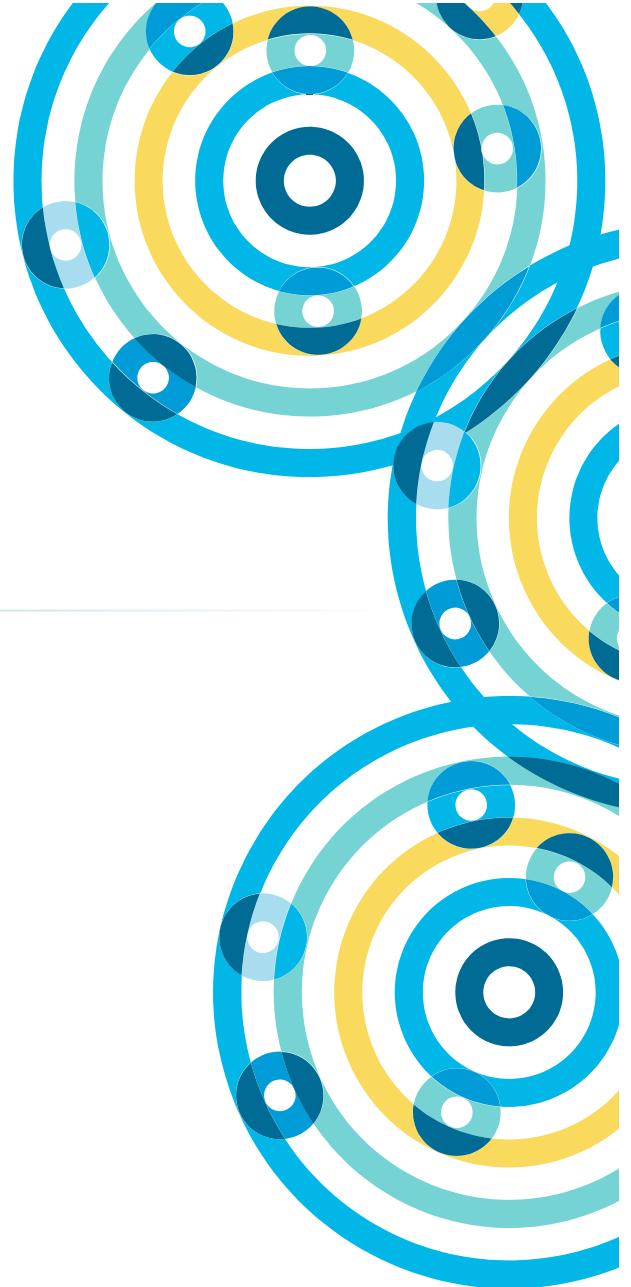
Hands-On Exercise: Partition Data in Impala or Hive

- **In this exercise you will**
 - Create a table for accounts that is partitioned by area code
- **Please refer to the Hands-On Exercise Manual for instructions**



Capturing Data with Apache Flume

Chapter 9



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume**
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured Data

Ingesting Streaming Data

Distributed Data Processing with Spark

Course Conclusion

Capturing Data with Apache Flume

In this chapter you will learn

- **What are the main architectural components of Flume**
- **How these components are configured**
- **How to launch a Flume agent**
- **How to configure a standard Java application to log data using Flume**

Chapter Topics

Capturing Data with Apache Flume

Introduction to Flume

- **What is Apache Flume?**
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

What Is Apache Flume?

- **Apache Flume is a high-performance system for data collection**
 - Name derives from original use case of near-real time log data ingestion
 - Now widely used for collection of any streaming event data
 - Supports aggregating data from many sources into HDFS
- **Originally developed by Cloudera**
 - Donated to Apache Software Foundation in 2011
 - Became a top-level Apache project in 2012
 - Flume OG gave way to Flume NG (Next Generation)
- **Benefits of Flume**
 - Horizontally-scalable
 - Extensible
 - Reliable



Flume's Design Goals: Reliability

- **Channels provide Flume's reliability**
- **Memory Channel**
 - Data will be lost if power is lost
- **Disk-based Channel**
 - Disk-based queue guarantees durability of data in face of a power loss
- **Data transfer between Agents and Channels is transactional**
 - A failed data transfer to a downstream agent rolls back and retries
- **Can configure multiple Agents with the same task**
 - For example, 2 Agents doing the job of 1 ‘collector’ – if one agent fails then upstream agents would fail over

Flume's Design Goals: Scalability

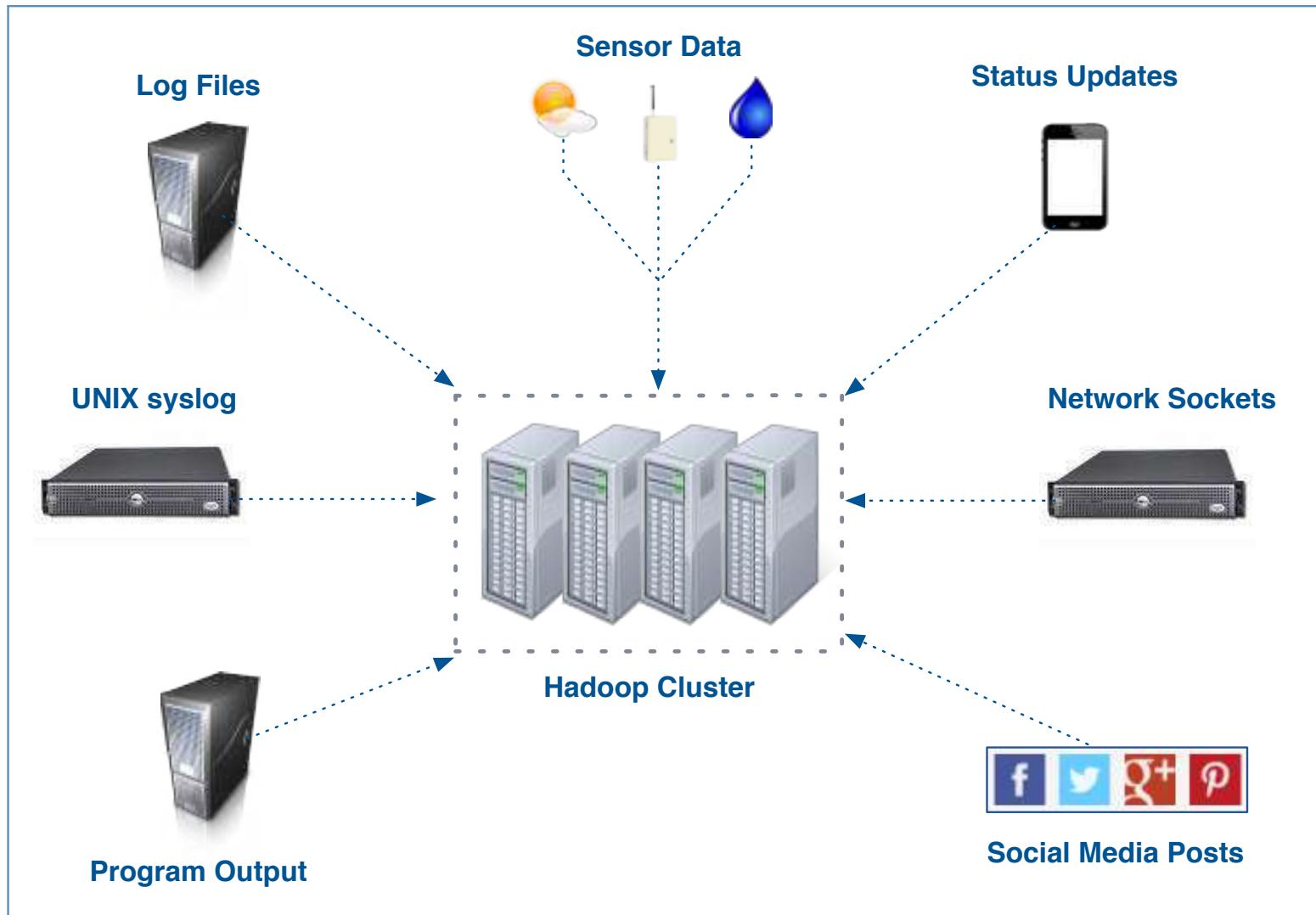
- **Scalability**

- The ability to increase system performance linearly – or better – by adding more resources to the system
 - Flume scales horizontally
 - As load increases, more machines can be added to the configuration

Flume's Design Goals: Extensibility

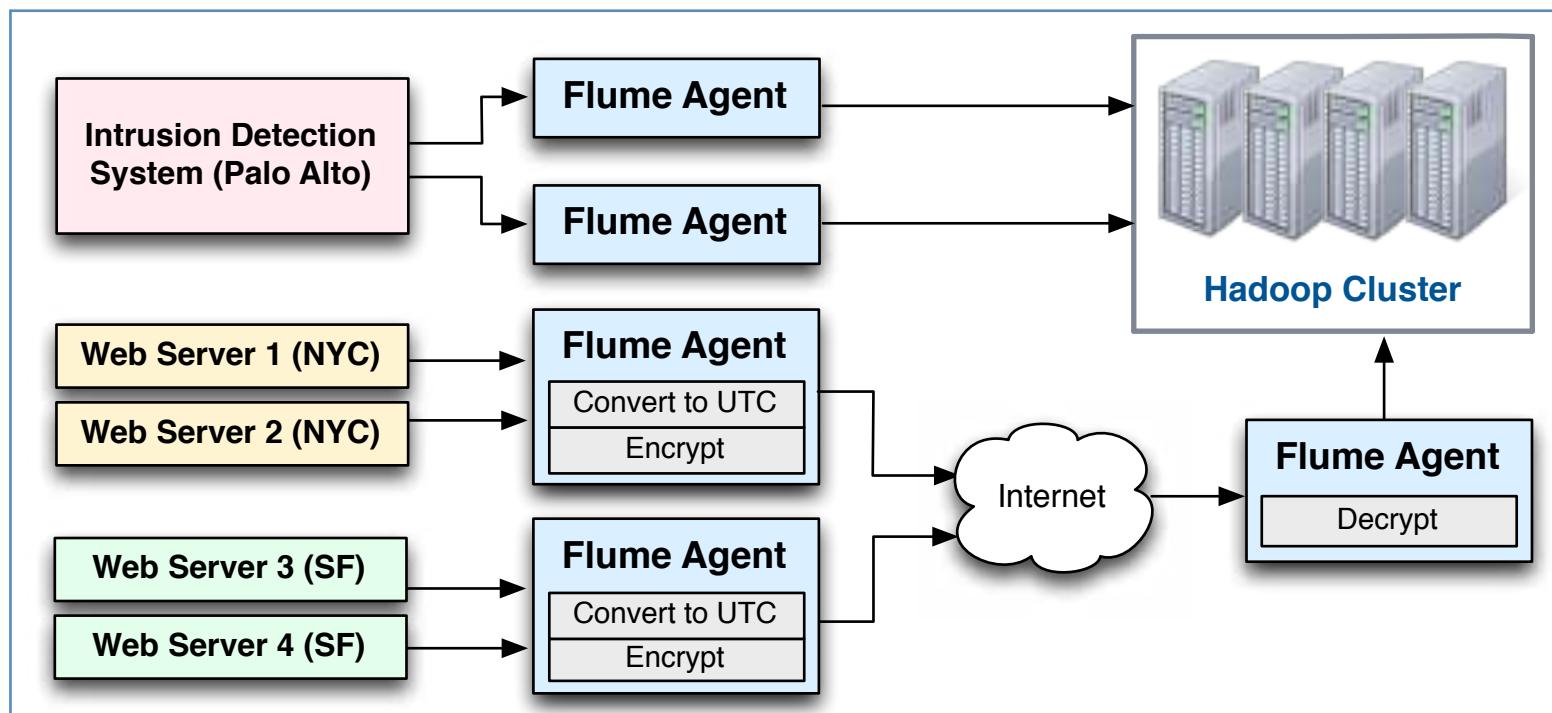
- **Extensibility**
 - The ability to add new functionality to a system
- **Flume can be extended by adding Sources and Sinks to existing storage layers or data platforms**
 - General Sources include data from files, syslog, and standard output from any Linux process
 - General Sinks include files on the local filesystem or HDFS
 - Developers can write their own Sources or Sinks

Common Flume Data Sources



Large-Scale Deployment Example

- Flume collects data using configurable “agents”
 - Agents can receive data from many sources, including other agents
 - Large-scale deployments use multiple tiers for scalability and reliability
 - Flume supports inspection and modification of in-flight data



Chapter Topics

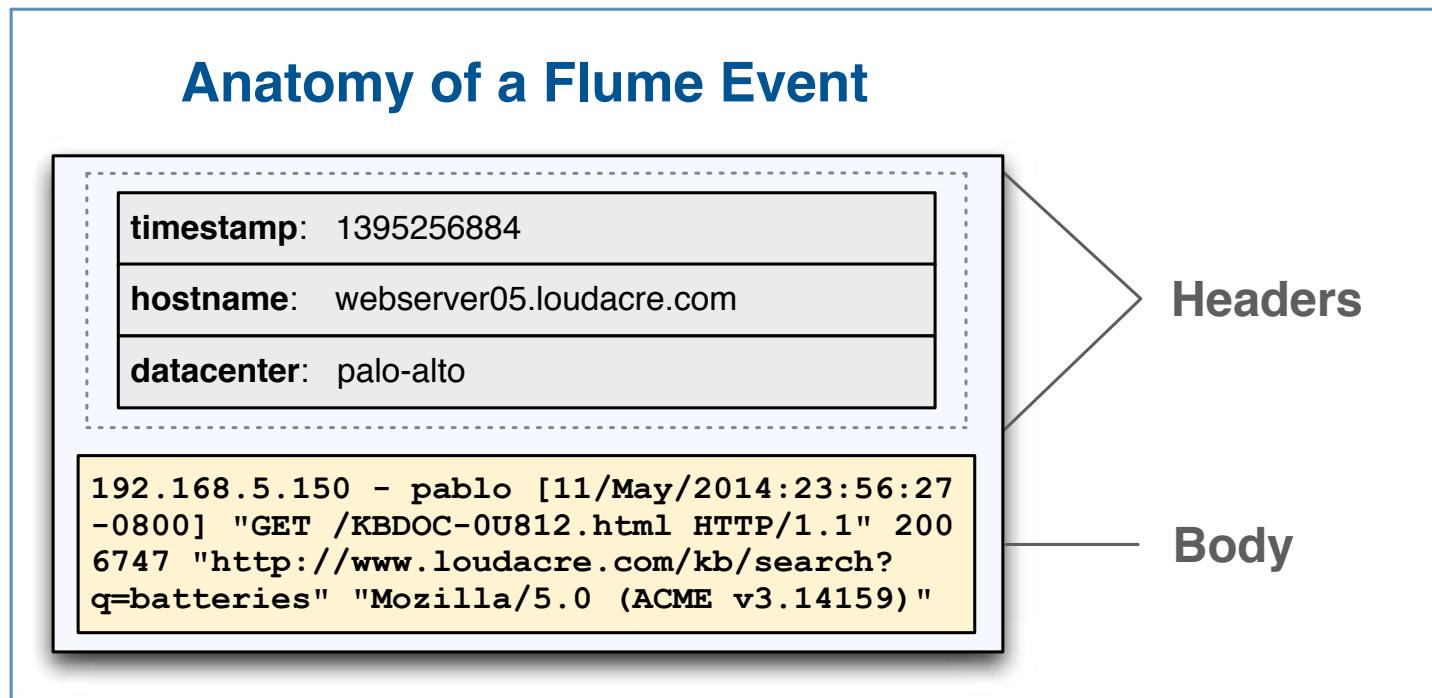
Capturing Data with Apache Flume

Introduction to Flume

- What is Apache Flume?
- **Basic Flume Architecture**
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

Flume Events

- An **event** is the fundamental unit of data in Flume
 - Consists of a body (payload) and a collection of headers (metadata)
- Headers consist of name-value pairs
 - Headers are mainly used for directing output



Components in Flume's Architecture

- **Source**

- Receives events from the external actor that generates them

- **Sink**

- Sends an event to its destination

- **Channel**

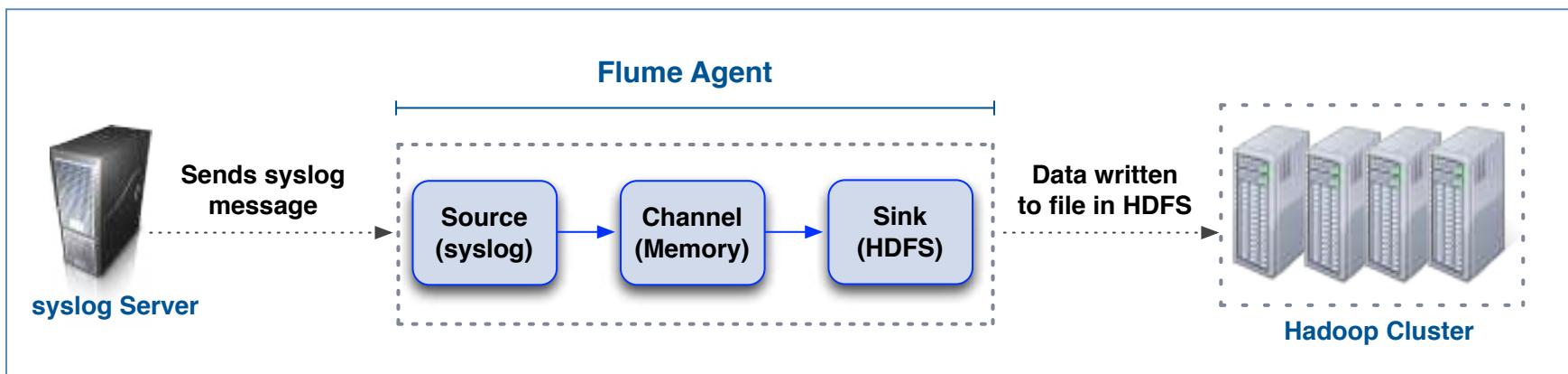
- Buffers events from the source until they are drained by the sink

- **Agent**

- Java process that configures and hosts the source, channel, and sink

Flume Data Flow

- This diagram illustrates how syslog data might be captured to HDFS
 1. Message is logged on a server running a syslog daemon
 2. Flume agent configured with syslog source receives event
 3. Source pushes event to the channel, where it is buffered in memory
 4. Sink pulls data from the channel and writes it to HDFS



Chapter Topics

Capturing Data with Apache Flume

Introduction to Flume

- What is Apache Flume?
- Basic Flume Architecture
- **Flume Sources**
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

Notable Built-in Flume Sources

- **Syslog**

- Captures messages from UNIX syslog daemon over the network

- **Netcat**

- Captures any data written to a socket on an arbitrary TCP port

- **Exec**

- Executes a UNIX program and reads events from standard output *

- **Spooldir**

- Extracts events from files appearing in a specified (local) directory

- **HTTP Source**

- Receives events from HTTP requests

* Asynchronous sources do not guarantee that events will be delivered

Chapter Topics

Capturing Data with Apache Flume

Introduction to Flume

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- **Flume Sinks**
- Flume Channels
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collecting Web Server Logs with Flume

Interesting Built-in Flume Sinks

- **Null**
 - Discards all events (Flume equivalent of `/dev/null`)
- **Logger**
 - Logs event to INFO level using SLF4J
- **IRC**
 - Sends event to a specified Internet Relay Chat channel
- **HDFS**
 - Writes event to a file in the specified directory in HDFS
- **HBaseSink**
 - Stores event in HBase

SLF4J: Simple Logging Façade for Java

Chapter Topics

Capturing Data with Apache Flume

Introduction to Flume

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- **Flume Channels**
- Flume Configuration
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

Built-In Flume Channels

- **Memory**

- Stores events in the machine's RAM
 - Extremely fast, but not reliable (memory is volatile)

- **File**

- Stores events on the machine's local disk
 - Slower than RAM, but more reliable (data is written to disk)

- **JDBC**

- Stores events in a database table using JDBC
 - Slower than file channel

Chapter Topics

Capturing Data with Apache Flume

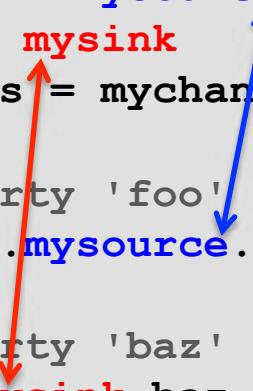
Introduction to Flume

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- **Flume Configuration**
- Conclusion
- Hands-On Exercise: Collect Web Server Logs with Flume

Flume Agent Configuration File

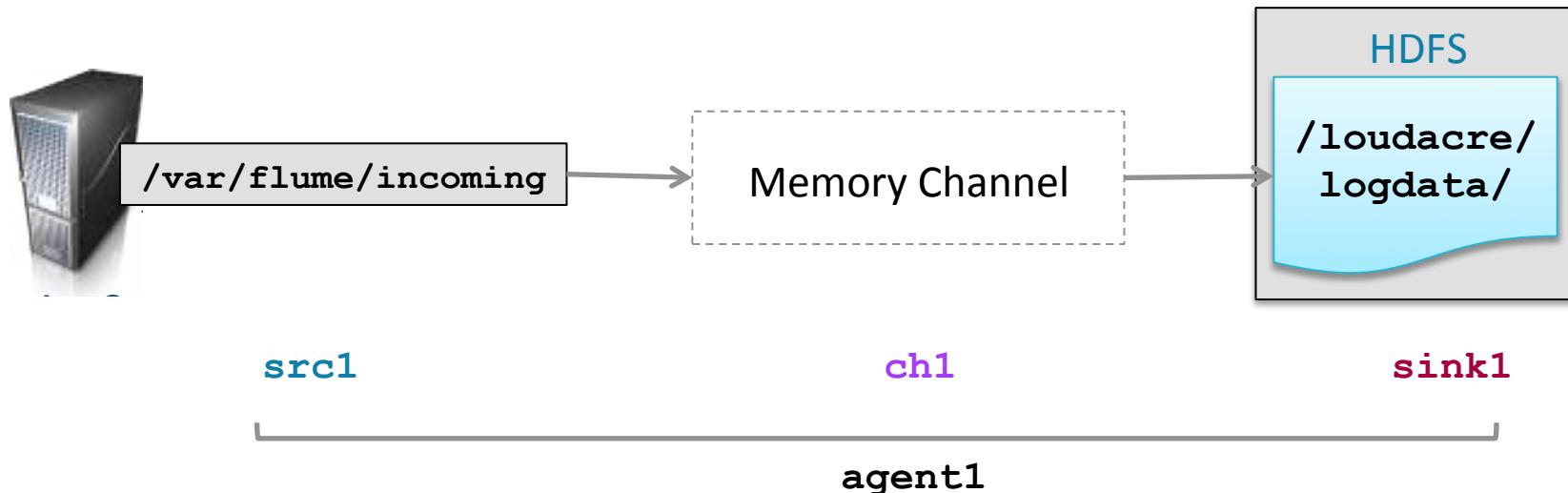
- Flume agent is configured through a Java properties file
 - Multiple agents can be configured in a single file
- The configuration file uses hierarchical references
 - Each component is assigned a user-defined ID
 - That ID is used in the names of additional properties

```
# Define sources, sinks, and channel for agent named 'agent1'  
agent1.sources = mysource  
agent1.sinks = mysink  
agent1.channels = mychannel  
  
# Sets a property 'foo' for the source associated with agent1  
agent1.sources.mysource.foo = bar  
  
# Sets a property 'baz' for the sink associated with agent1  
agent1.sinks.mysink.baz = bat
```



Example: Configuring Flume Components (1)

- Example: Configure a Flume Agent to collect data from remote spool directories and save to HDFS



Example: Configuring Flume Components (2)

```
agent1.sources = src1
agent1.sinks = sink1
agent1.channels = ch1

agent1.channels.ch1.type = memory

agent1.sources.src1.type = spooldir
agent1.sources.src1.spoolDir = /var/flume/incoming
agent1.sources.src1.channels = ch1

agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /loudacre/logdata
agent1.sinks.sink1.channel = ch1
```

Connects **source** and channel

Connects **sink** and channel

- Properties vary by component type (source, channel, and sink)
 - Properties also vary by subtype (e.g., netcat source vs. syslog source)
 - See the Flume user guide for full details on configuration

Aside: HDFS Sink Configuration

- Path may contain patterns based on event headers, such as timestamp
- The HDFS sink writes uncompressed SequenceFiles by default
 - Specifying a codec will enable compression

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /loudacre/logdata/%y-%m-%d
agent1.sinks.sink1.hdfs.codec = snappy
agent1.sinks.sink1.channel = ch1
```

- Setting fileType parameter to DataStream writes *raw* data
 - Can also specify a file extension, if desired

```
agent1.sinks.sink1.type = hdfs
agent1.sinks.sink1.hdfs.path = /loudacre/logdata/%y-%m-%d
agent1.sinks.sink1.hdfs.fileType = DataStream
agent1.sinks.sink1.hdfs.fileSuffix = .txt
agent1.sinks.sink1.channel = ch1
```

Starting a Flume Agent

- **Typical command line invocation**

- The **--name** argument must match the agent's name in the configuration file
 - Setting root logger as shown will display log messages in the terminal

```
$ flume-ng agent \
  --conf /etc/flume-ng/conf \
  --conf-file /path/to/flume.conf \
  --name agent1 \
  -Dflume.root.logger=INFO,console
```

* ng = Next Generation (prior version now referred to as og)

Chapter Topics

Capturing Data with Apache Flume

Introduction to Flume

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- **Conclusion**
- Hands-On Exercise: Collect Web Server Logs with Flume

Essential Points

- **Apache Flume is a high-performance system for data collection**
 - Scalable, extensible, and reliable
- **A Flume agent manages the source, channels, and sink**
 - Source receives event data from its origin
 - Sink sends the event to its destination
 - Channel buffers events between the source and sink
- **The Flume agent is configured using a properties file**
 - Each component is given a user-defined ID
 - This ID is used to define properties of that component

Bibliography

The following offer more information on topics discussed in this chapter

- **Flume User Guide**

- <http://flume.apache.org/FlumeUserGuide.html>

Chapter Topics

Capturing Data with Apache Flume

Introduction to Flume

- What is Apache Flume?
- Basic Flume Architecture
- Flume Sources
- Flume Sinks
- Flume Channels
- Flume Configuration
- Conclusion
- **Hands-On Exercise: Collect Web Server Logs with Flume**

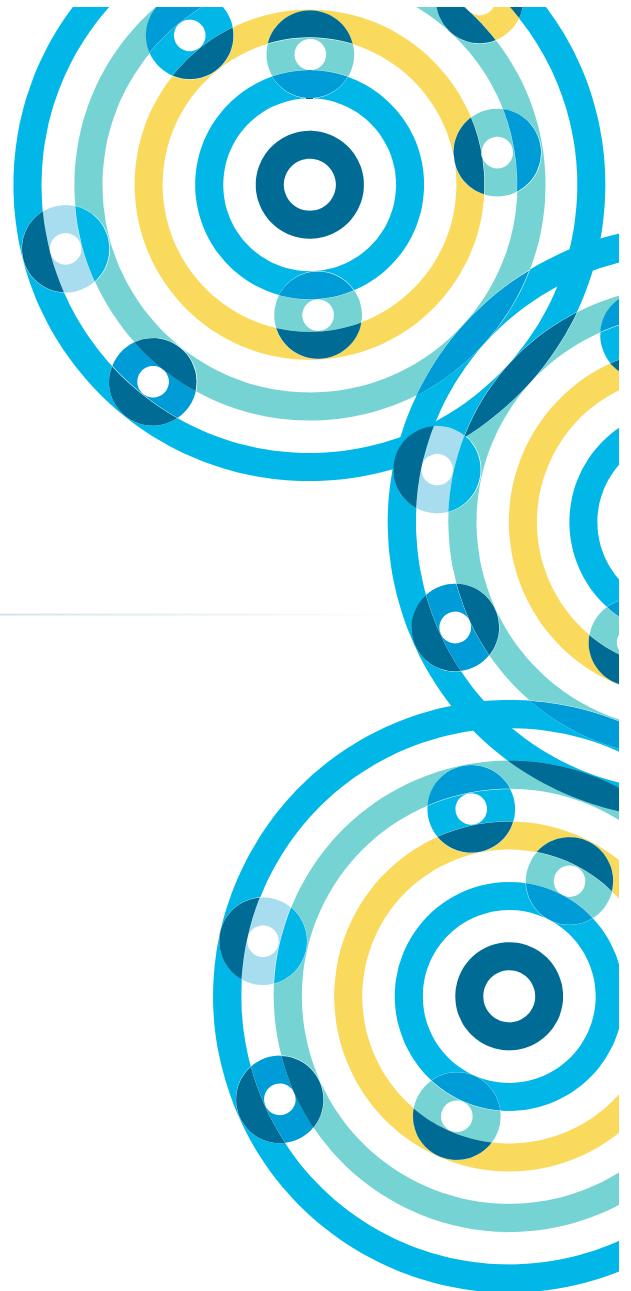
Hands-On Exercise: Collect Web Server Logs with Flume

- **In this exercise you will**
 - Configure Flume to ingest web server log data to HDFS
- **Please refer to your Hands-On Exercise Manual for instructions**



Spark Basics

Chapter 10



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- **Spark Basics**
 - Working with RDDs in Spark
 - Aggregating Data with Pair RDDs
 - Writing and Deploying Spark Applications
 - Parallel Processing in Spark
 - Spark RDD Persistence
 - Common Patterns in Spark
 - Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

Spark Basics

In this chapter you will learn

- **How to start the Spark Shell**
- **About the SparkContext**
- **Key Concepts of Resilient Distributed Datasets (RDDs)**
 - What are they?
 - How do you create them?
 - What operations can you perform with them?
- **How Spark uses the principles of functional programming**

Chapter Topics

Spark Basics

Distributed Data Processing with Spark

- **What is Apache Spark?**
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- Functional Programming in Spark
- Conclusion
- Hands-On Exercises

What is Apache Spark?

- **Apache Spark is a fast and general engine for large-scale data processing**
- **Written in Scala**
 - Functional programming language that runs in a JVM
- **Spark Shell**
 - Interactive – for learning or data exploration
 - Python or Scala
- **Spark Applications**
 - For large scale data processing
 - Python, Scala, or Java



Chapter Topics

Spark Basics

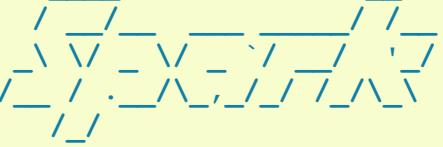
Distributed Data Processing with Spark

- What is Apache Spark?
- **Using the Spark Shell**
- RDDs (Resilient Distributed Datasets)
- Functional Programming in Spark
- Conclusion
- Hands-On Exercises

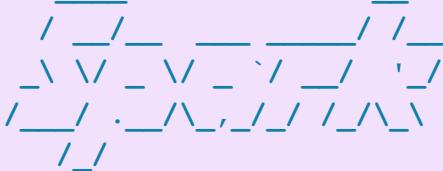
Spark Shell

- The Spark Shell provides interactive data exploration (REPL)
- Writing Spark applications without the shell will be covered later

Python Shell: **pyspark**

```
$ pyspark  
  
Welcome to  
 version 1.3.0  
  
Using Python version 2.7.8 (default, Aug 27  
2015 05:23:36)  
SparkContext available as sc, HiveContext  
available as sqlCtx.  
>>>
```

Scala Shell: **spark-shell**

```
$ spark-shell  
  
Welcome to  
 version 1.3.0  
  
Using Scala version 2.10.4 (Java HotSpot(TM)  
64-Bit Server VM, Java 1.7.0_67)  
Created spark context..  
Spark context available as sc.  
SQL context available as sqlContext.  
  
scala>
```

REPL: Read/Evaluate/Print Loop

Spark Context

- Every Spark application requires a Spark Context
 - The main entry point to the Spark API
- Spark Shell provides a preconfigured Spark Context called sc

Python

```
Using Python version 2.7.8 (default, Aug 27 2015 05:23:36)
SparkContext available as sc, HiveContext available as sqlCtx.

>>> sc.appName
u'PySparkShell'
```

Scala

```
...
Spark context available as sc.
SQL context available as sqlContext.

scala> sc.appName
res0: String = Spark shell
```

Chapter Topics

Spark Basics

Distributed Data Processing with Spark

- What is Apache Spark?
- Using the Spark Shell
- **RDDs (Resilient Distributed Datasets)**
- Functional Programming With Spark
- Conclusion
- Hands-On Exercise: Getting Started with RDDs

RDD (Resilient Distributed Dataset)

- **RDD (Resilient Distributed Dataset)**
 - Resilient – if data in memory is lost, it can be recreated
 - Distributed – processed across the cluster
 - Dataset – initial data can come from a file or be created programmatically
- **RDDs are the fundamental unit of data in Spark**
- **Most Spark programming consists of performing operations on RDDs**

Creating an RDD

- **Three ways to create an RDD**
 - From a file or set of files
 - From data in memory
 - From another RDD

Example: A File-based RDD

```
> val mydata = sc.textFile("purplecow.txt")
...
15/01/29 06:20:37 INFO storage.MemoryStore:
  Block broadcast_0 stored as values to
  memory (estimated size 151.4 KB, free 296.8
  MB)

> mydata.count()
...
15/01/29 06:27:37 INFO spark.SparkContext: Job
finished: take at <stdin>:1, took
0.160482078 s
```

4

File: purplecow.txt

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

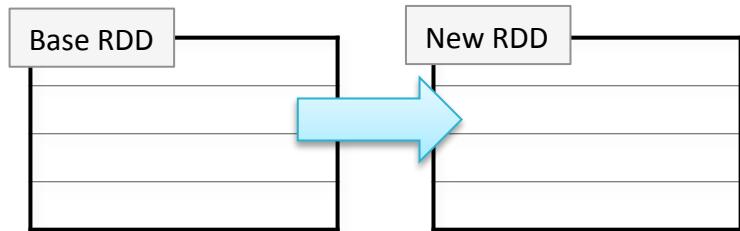
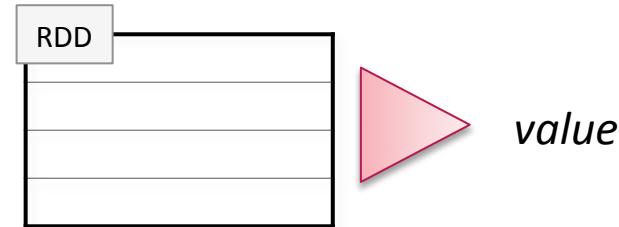
RDD: mydata

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

RDD Operations

- Two types of RDD operations

- Actions – return values
 - Transformations – define a new RDD based on the current one(s)

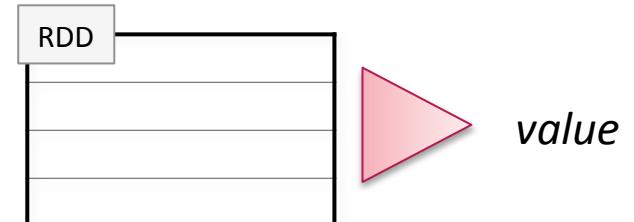


- Pop quiz:
 - Which type of operation is `count()`?

RDD Operations: Actions

- Some common actions

- **count()** – return the number of elements
- **take(n)** – return an array of the first n elements
- **collect()** – return an array of all elements
- **saveAsTextFile(file)** – save to text file(s)



```
> mydata =  
  sc.textFile("purplecow.txt")  
  
> mydata.count()  
4  
  
> for line in mydata.take(2):  
    print line  
I've never seen a purple cow.  
I never hope to see one;
```

```
> val mydata =  
  sc.textFile("purplecow.txt")  
  
> mydata.count()  
4  
  
> for (line <- mydata.take(2))  
    println(line)  
I've never seen a purple cow.  
I never hope to see one;
```

RDD Operations: Transformations

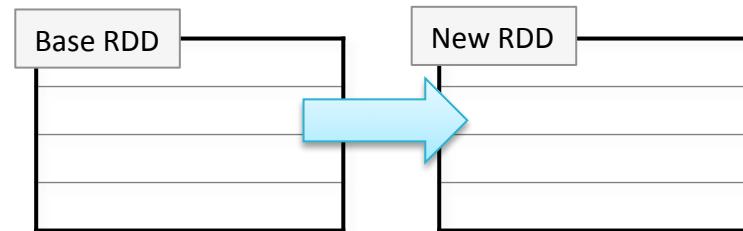
- **Transformations create a new RDD from an existing one**

- **RDDs are immutable**

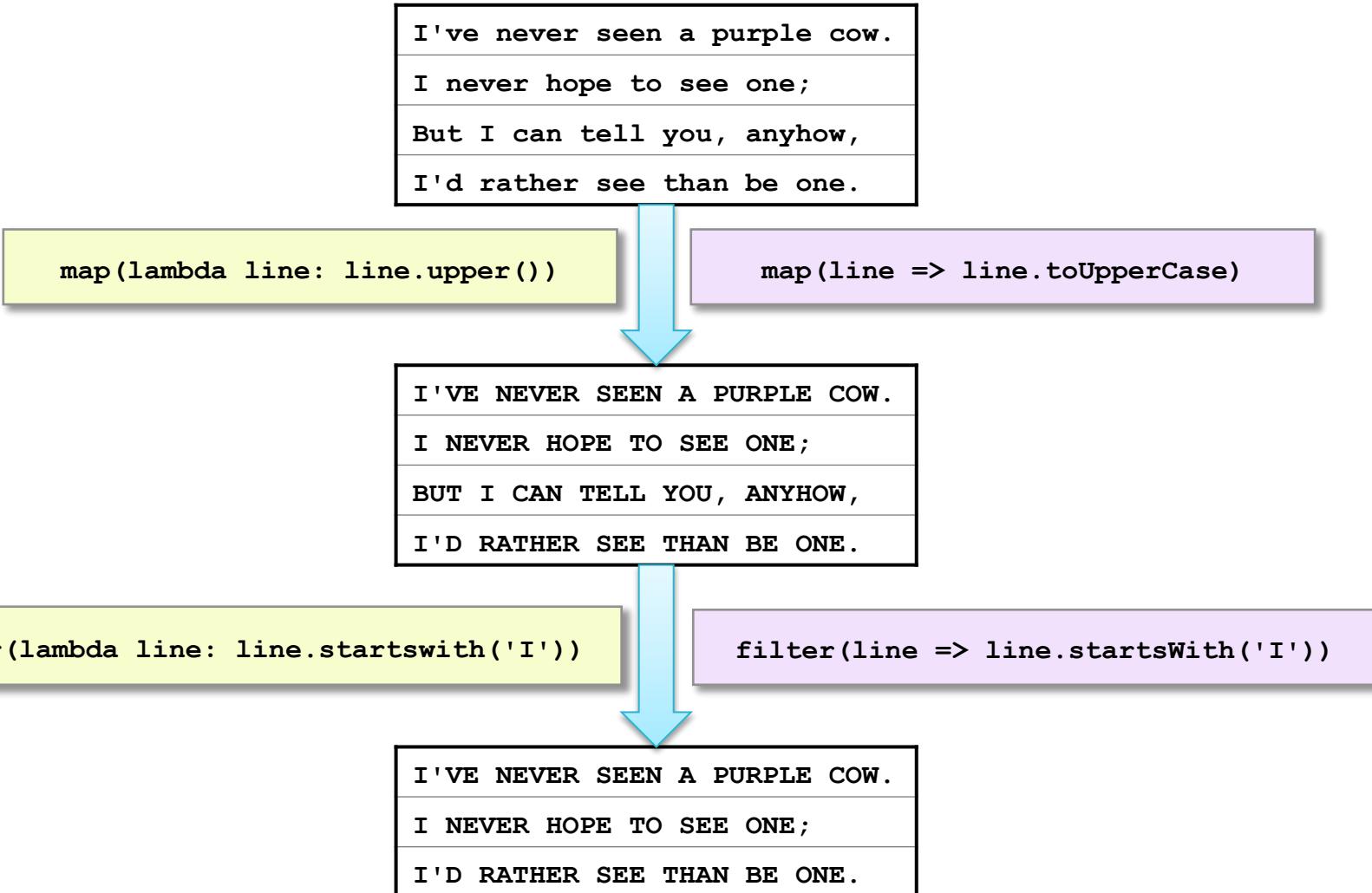
- Data in an RDD is never changed
 - Transform in sequence to modify the data as needed

- **Some common transformations**

- **map (*function*)** – creates a new RDD by performing a function on each record in the base RDD
 - **filter (*function*)** – creates a new RDD by including or excluding each record in the base RDD according to a boolean function



Example: map and filter Transformations



Lazy Execution (1)

- Data in RDDs is not processed until an *action* is performed

File: purplecow.txt

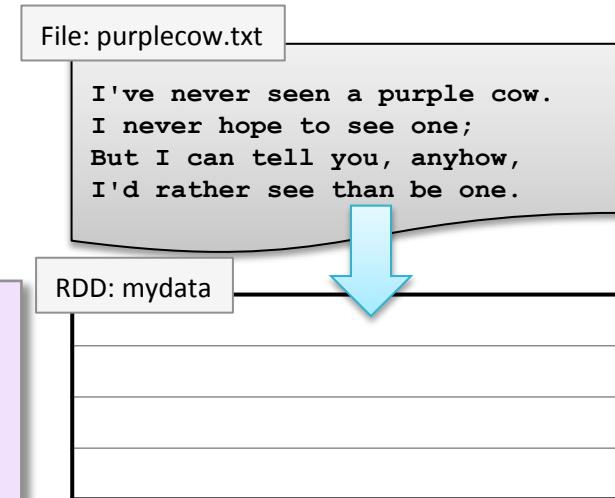
```
I've never seen a purple cow.  
I never hope to see one;  
But I can tell you, anyhow,  
I'd rather see than be one.
```

>

Lazy Execution (2)

- Data in RDDs is not processed until an *action* is performed

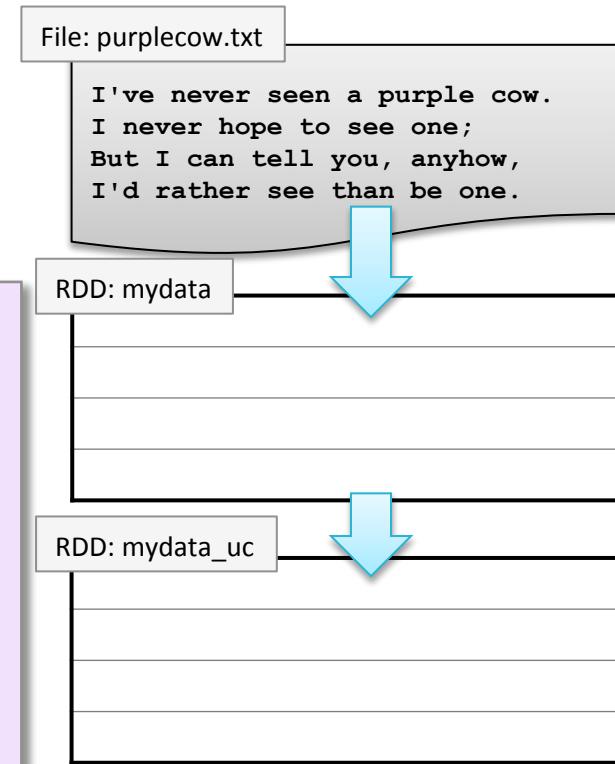
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy Execution (3)

- Data in RDDs is not processed until an *action* is performed

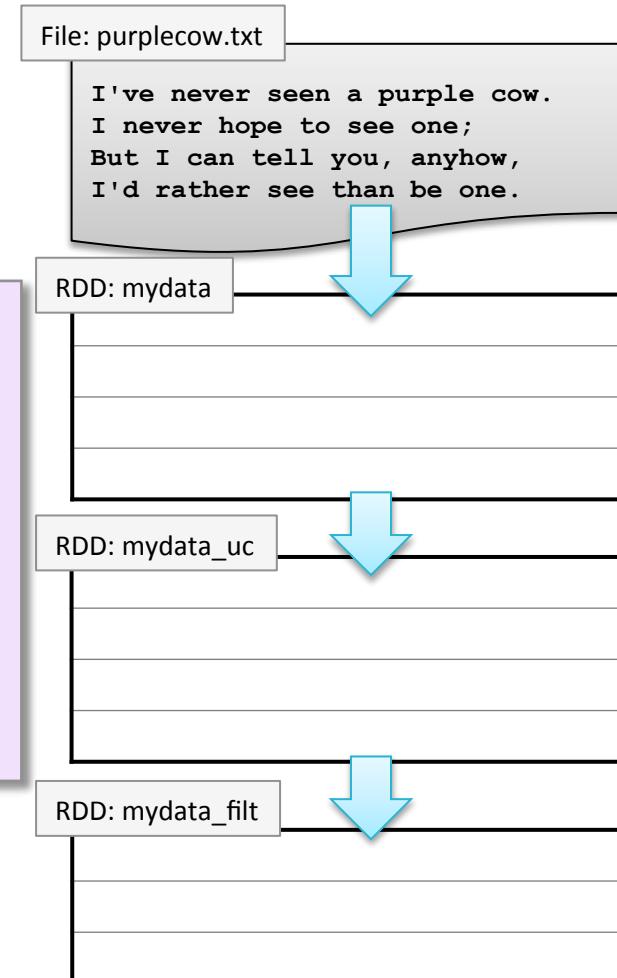
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy Execution (4)

- Data in RDDs is not processed until an *action* is performed

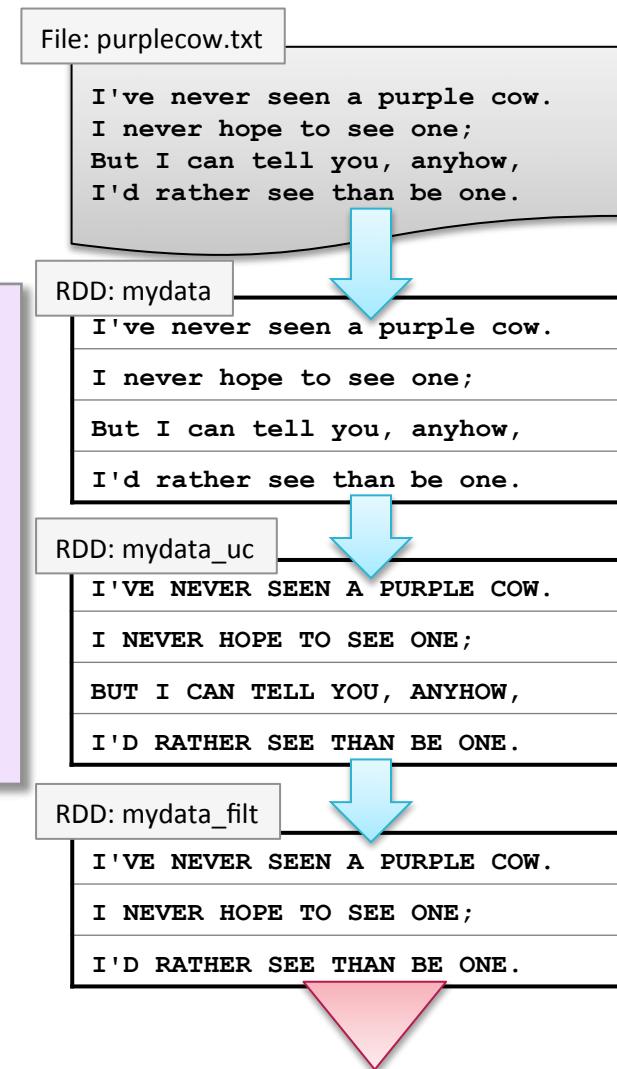
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution (5)

- Data in RDDs is not processed until an *action* is performed

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



Chaining Transformations (Scala)

- Transformations may be chained together

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line => line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line => line.startsWith("I"))
> mydata_filt.count()
3
```

is exactly equivalent to

```
> sc.textFile("purplecow.txt").map(line => line.toUpperCase()).
  filter(line => line.startsWith("I")).count()
```

3

Chaining Transformations (Python)

- Same example in Python

```
> mydata = sc.textFile("purplecow.txt")
> mydata_uc = mydata.map(lambda s: s.upper())
> mydata_filt = mydata_uc.filter(lambda s: s.startswith('I'))
> mydata_filt.count()
3
```

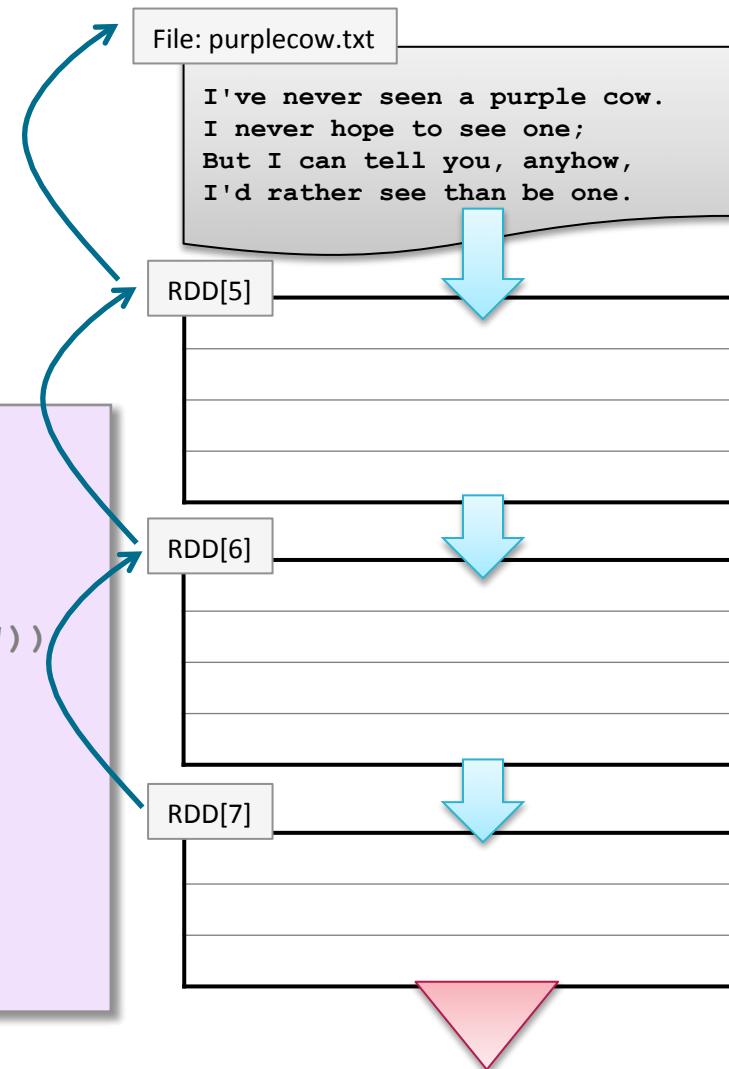
is exactly equivalent to

```
> sc.textFile("purplecow.txt").map(lambda line: line.upper()) \
  .filter(lambda line: line.startswith('I')).count()
3
```

RDD Lineage and `toDebugString` (Scala)

- Spark maintains each RDD's *lineage* – the previous RDDs on which it depends
- Use `toDebugString` to view the lineage of an RDD

```
> val mydata_filt =  
  sc.textFile("purplecow.txt") .  
    map(line => line.toUpperCase()) .  
    filter(line => line.startsWith("I"))  
> mydata_filt.toDebugString  
  
(2) FilteredRDD[7] at filter ...  
| MappedRDD[6] at map ...  
| purplecow.txt MappedRDD[5] ...  
| purplecow.txt HadoopRDD[4] ...
```



RDD Lineage and `toDebugString` (Python)

- `toDebugString` output is not displayed as nicely in Python

```
> mydata_filt.toDebugString()
(1) PythonRDD[8] at RDD at ...\\n | purplecow.txt MappedRDD[7] at textFile
at ...[]\\n | purplecow.txt HadoopRDD[6] at textFile at ...[]
```

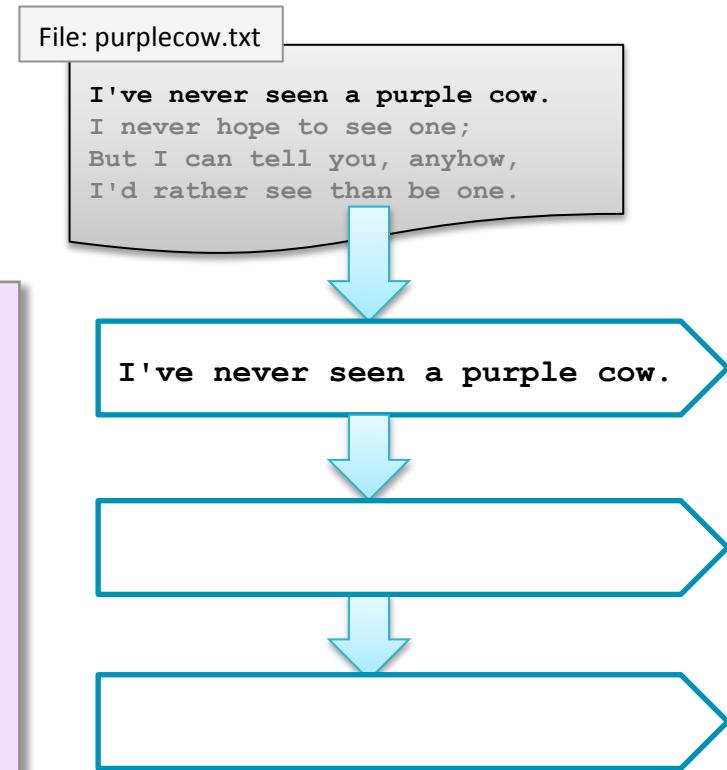
- Use `print` for prettier output

```
> print mydata_filt.toDebugString()
(1) PythonRDD[8] at RDD at ...
| purplecow.txt MappedRDD[7] at textFile at ...
| purplecow.txt HadoopRDD[6] at textFile at ...
```

Pipelining (1)

- When possible, Spark will perform sequences of transformations by row so no data is stored

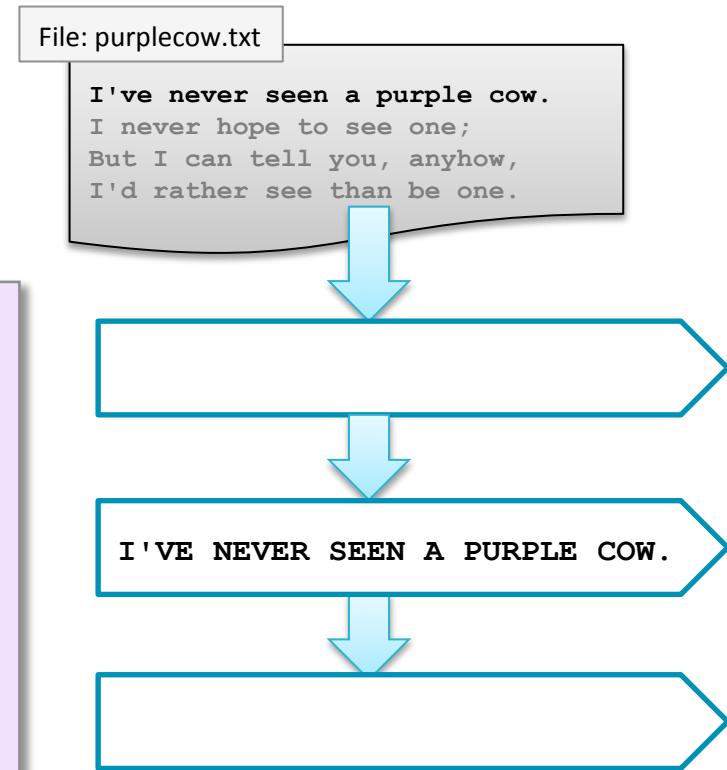
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```



Pipelining (2)

- When possible, Spark will perform sequences of transformations by row so no data is stored

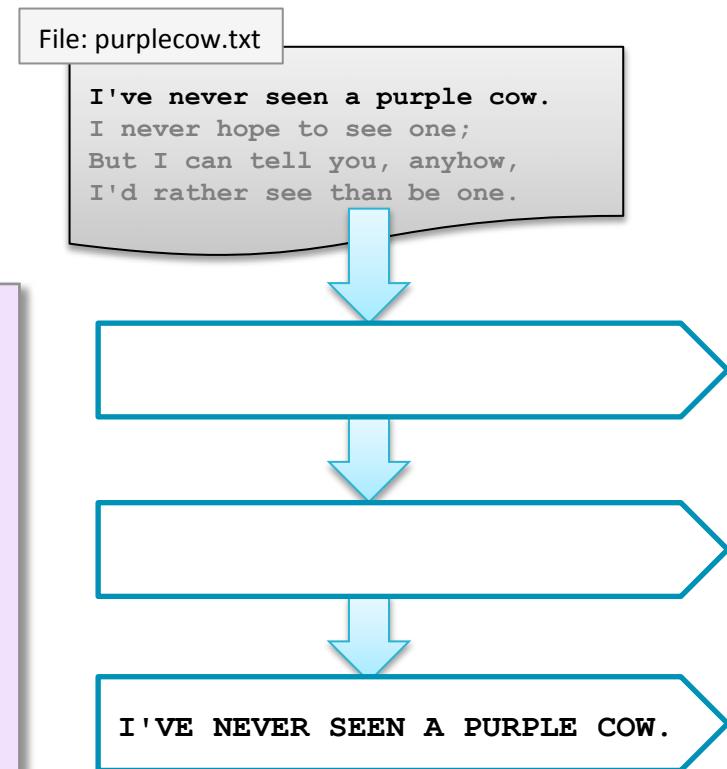
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```



Pipelining (3)

- When possible, Spark will perform sequences of transformations by row so no data is stored

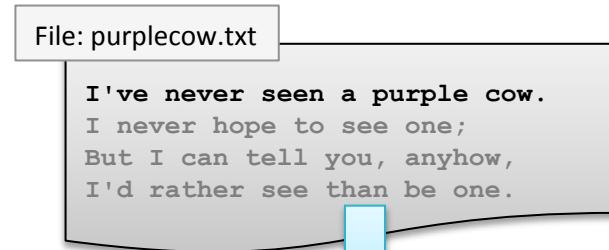
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
```



Pipelining (4)

- When possible, Spark will perform sequences of transformations by row so no data is stored

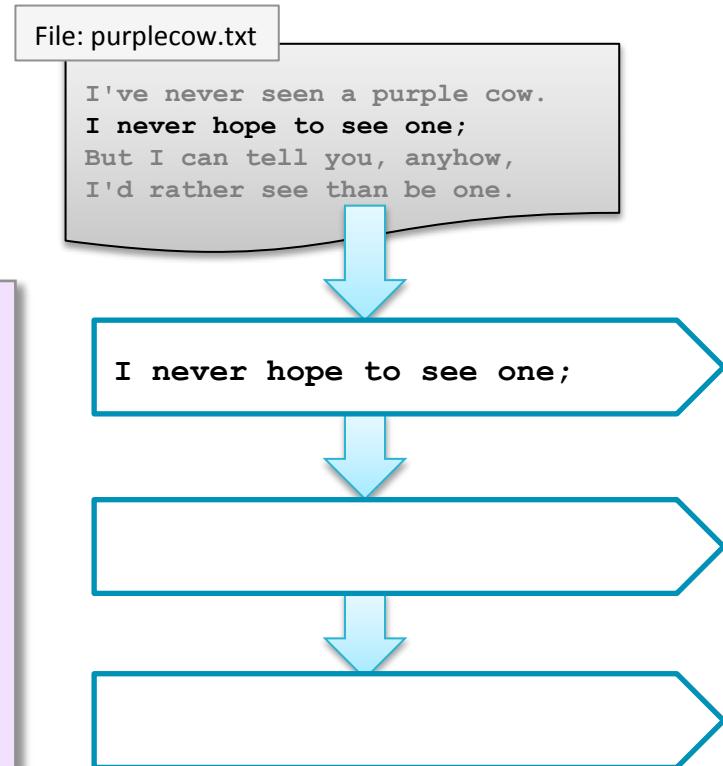
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (5)

- When possible, Spark will perform sequences of transformations by row so no data is stored

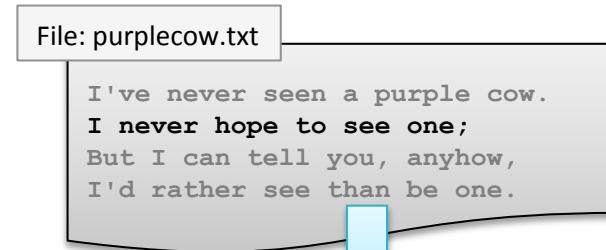
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (6)

- When possible, Spark will perform sequences of transformations by row so no data is stored

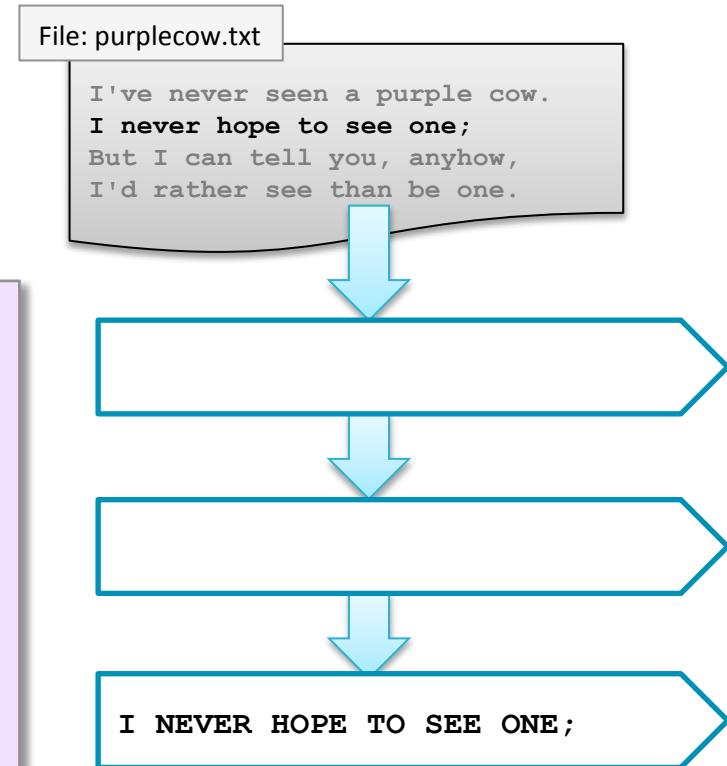
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



Pipelining (7)

- When possible, Spark will perform sequences of transformations by row so no data is stored

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
```



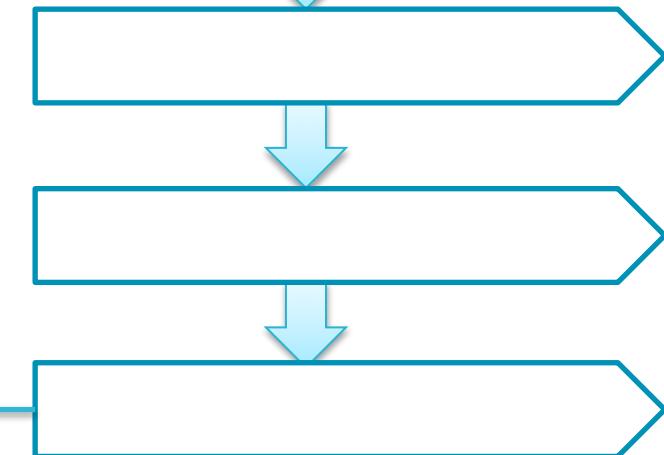
Pipelining (8)

- When possible, Spark will perform sequences of transformations by row so no data is stored

```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.take(2)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```

File: purplecow.txt

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.



Chapter Topics

Spark Basics

Distributed Data Processing with Spark

- What is Apache Spark?
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- **Functional Programming in Spark**
- Conclusion
- Hands-On Exercises

Functional Programming in Spark

- **Spark depends heavily on the concepts of *functional programming***
 - Functions are the fundamental unit of programming
 - Functions have input and output only
 - No state or side effects
- **Key concepts**
 - Passing functions as input to other functions
 - Anonymous functions

Passing Functions as Parameters

- Many RDD operations take functions as parameters
- Pseudocode for the RDD map operation
 - Applies function `fn` to each record in the RDD

```
RDD {  
    map(fn(x)) {  
        foreach record in rdd  
        emit fn(record)  
    }  
}
```

Example: Passing Named Functions

- Python

```
> def toUpper(s):  
    return s.upper()  
> mydata = sc.textFile("purplecow.txt")  
> mydata.map(toUpper).take(2)
```

- Scala

```
> def toUpper(s: String): String =  
    { s.toUpperCase }  
> val mydata = sc.textFile("purplecow.txt")  
> mydata.map(toUpper).take(2)
```

Anonymous Functions

- **Functions defined in-line without an identifier**
 - Best for short, one-off functions
- **Supported in many programming languages**
 - Python: `lambda x: ...`
 - Scala: `x => ...`
 - Java 8: `x -> ...`

Example: Passing Anonymous Functions

- Python:

```
> mydata.map(lambda line: line.upper()).take(2)
```

- Scala:

```
> mydata.map(line => line.toUpperCase()).take(2)
```

OR

```
> mydata.map(_.toUpperCase()).take(2)
```

Scala allows anonymous parameters
using underscore (_)

Example: Java

Java 7

```
...
JavaRDD<String> lines = sc.textFile("file");
JavaRDD<String> lines_uc = lines.map(
    new MapFunction<String, String>() {
        public String call(String line) {
            return line.toUpperCase();
        }
    });
...
...
```

Java 8

```
...
JavaRDD<String> lines = sc.textFile("file");
JavaRDD<String> lines_uc = lines.map(
    line -> line.toUpperCase());
...
...
```

Chapter Topics

Spark Basics

Distributed Data Processing with Spark

- What is Apache Spark?
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- Functional Programming With Spark
- **Conclusion**
- Hands-On Exercises

Essential Points

- **Spark can be used interactively via the Spark Shell**
 - Python or Scala
 - Writing non-interactive Spark applications will be covered later
- **RDDs (Resilient Distributed Datasets) are a key concept in Spark**
- **RDD Operations**
 - Transformations create a new RDD based on an existing one
 - Actions return a value from an RDD
- **Lazy Execution**
 - Transformations are not executed until required by an action
- **Spark uses functional programming**
 - Passing functions as parameters
 - Anonymous functions in supported languages (Python and Scala)

Chapter Topics

Spark Basics

Distributed Data Processing with Spark

- What is Apache Spark?
- Using the Spark Shell
- RDDs (Resilient Distributed Datasets)
- Functional Programming With Spark
- Conclusion
- **Hands-On Exercises**

Introduction to Spark Exercises: Pick Your Language

- **Your choice: Python or Scala**
 - For the Spark-based exercises in this course, you may choose to work with either Python or Scala
- **Solution and example files**
 - **.pyspark** – Python shell commands
 - **.scalaspark** – Scala shell commands
 - **.py** – complete Python Spark applications
 - **.scala** – complete Scala Spark applications

Hands-On Exercises

- Now, please do the following three Hands-On Exercises

1. *View the Spark Documentation*

- Familiarize yourself with the Spark documentation; you will refer to this documentation frequently during the course

2. *Explore RDDs Using the Spark Shell*

- Follow the instructions for either the Python or Scala shell

3. *Use RDDs to Transform a Dataset*

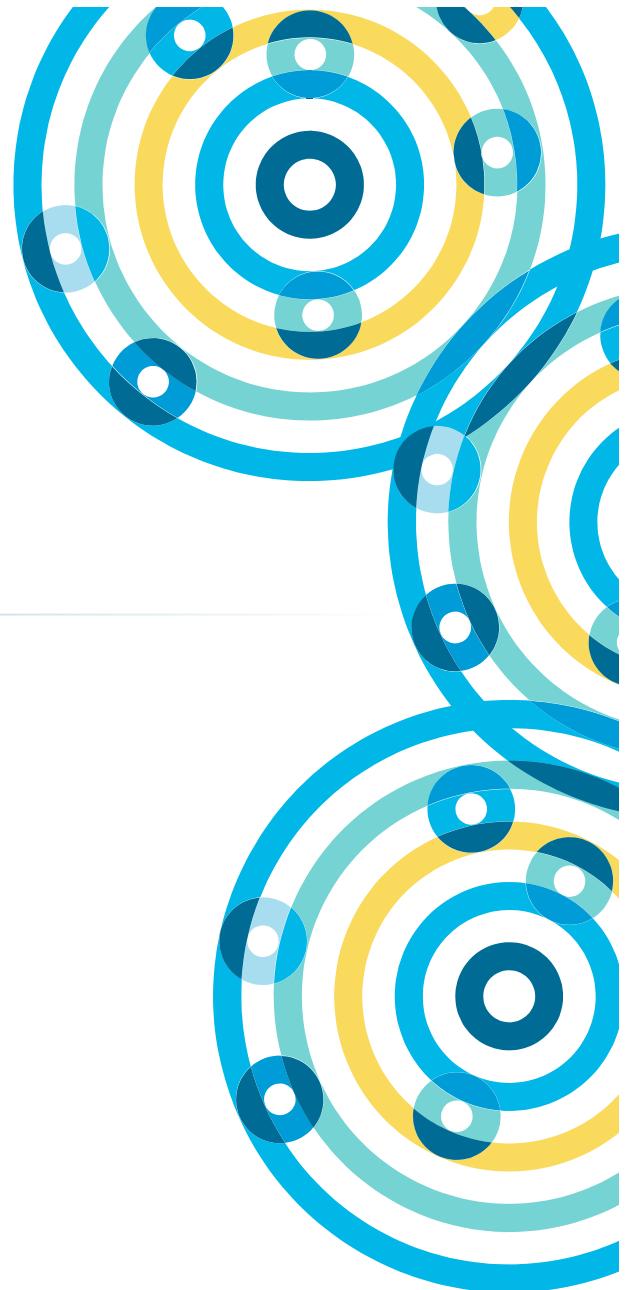
- Explore Loudacre web log files

- Please refer to your Hands-On Exercise Manual



Working With RDDs in Spark

Chapter 11



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
 - **Working with RDDs in Spark**
 - Aggregating Data with Pair RDDs
 - Writing and Deploying Spark Applications
 - Parallel Processing in Spark
 - Spark RDD Persistence
 - Common Patterns in Spark Data Processing
 - Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

Working With RDDs

In this chapter you will learn

- **How RDDs are created from files or data in memory**
- **How to handle file formats with multi-line records**
- **How to use some additional operations on RDDs**

Chapter Topics

Working With RDDs in Spark

Distributed Data Processing with Spark

- **Creating RDDs**
- Other General RDD Operations
- Conclusion
- Hands-On Exercise: Process Data Files with Spark

RDDs

- **RDDs can hold any type of element**
 - Primitive types: integers, characters, booleans, etc.
 - Sequence types: strings, lists, arrays, tuples, dicts, etc. (including nested data types)
 - Scala/Java Objects (if serializable)
 - Mixed types
- **Some types of RDDs have additional functionality**
 - Pair RDDs
 - RDDs consisting of Key-Value pairs
 - Double RDDs
 - RDDs consisting of numeric data

Creating RDDs From Collections

- You can create RDDs from collections instead of files
 - `sc.parallelize(collection)`

```
> myData = ["Alice", "Carlos", "Frank", "Barbara"]
> myRdd = sc.parallelize(myData)
> myRdd.take(2)
['Alice', 'Carlos']
```

- Useful when
 - Testing
 - Generating data programmatically
 - Integrating

Creating RDDs from Files (1)

- For file-based RDDs, use **SparkContext.textFile**
 - Accepts a single file, a wildcard list of files, or a comma-separated list of files
 - Examples
 - `sc.textFile("myfile.txt")`
 - `sc.textFile("mydata/*.log")`
 - `sc.textFile("myfile1.txt,myfile2.txt")`
 - Each line in the file(s) is a separate record in the RDD
- Files are referenced by absolute or relative URI
 - Absolute URI:
 - `file:/home/training/myfile.txt`
 - `hdfs://localhost/loudacre/myfile.txt`
 - Relative URI (uses default file system): `myfile.txt`

Creating RDDs from Files (2)

- **textFile** maps each line in a file to a separate RDD element

```
I've never seen a purple cow.\nI never hope to see one;\nBut I can tell you, anyhow,\nI'd rather see than be one.\n
```



```
I've never seen a purple cow.\nI never hope to see one;\nBut I can tell you, anyhow,\nI'd rather see than be one.
```

- **textFile** only works with line-delimited text files
- What about other formats?

Input and Output Formats (1)

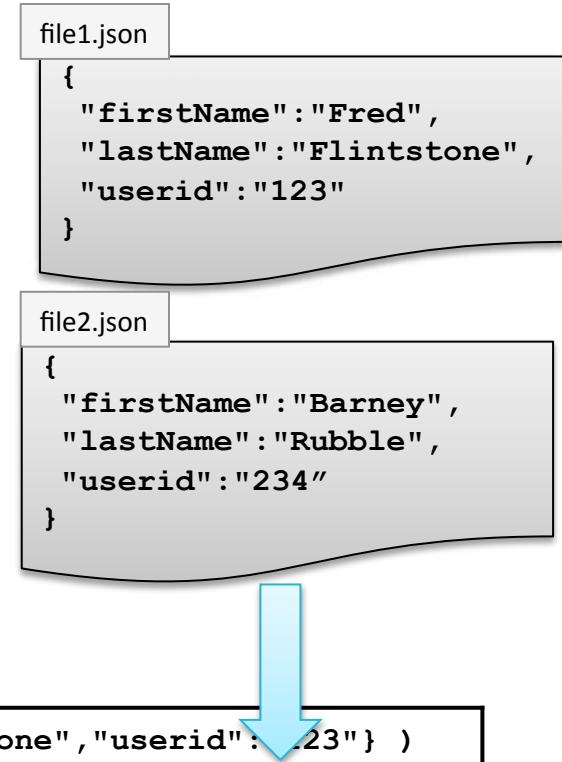
- Spark uses Hadoop `InputFormat` and `OutputFormat` Java classes
 - Some examples from core Hadoop
 - `TextInputFormat` / `TextOutputFormat` – newline delimited text files
 - `SequenceInputFormat` / `SequenceOutputFormat`
 - `FixedLengthInputFormat`
 - Many implementations available in additional libraries
 - e.g. `AvroInputFormat` / `AvroOutputFormat` in the Avro library

Input and Output Formats (2)

- Specify any input format using `sc.hadoopFile`
 - or `newAPIhadoopFile` for New API classes
- Specify any output format using `rdd.saveAsHadoopFile`
 - or `saveAsNewAPIhadoopFile` for New API classes
- `textFile` and `saveAsTextFile` are convenience functions
 - `textFile` just calls `hadoopFile` specifying `TextInputFormat`
 - `saveAsTextFile` calls `saveAsHadoopFile` specifying `TextOutputFormat`

Whole File-Based RDDs (1)

- **sc.textFile** maps each line in a file to a separate RDD element
 - What about files with a multi-line input format, e.g. XML or JSON?
- **sc.wholeTextFiles (*directory*)**
 - Maps entire contents of each file in a directory to a single RDD element
 - Works only for small files (element must fit in memory)



```
(file1.json,{"firstName":"Fred","lastName":"Flintstone","userid":"123"} )  
(file2.json,{"firstName":"Barney","lastName":"Rubble","userid":"234"} )  
(file3.xml,... )  
(file4.xml,... )
```

Whole File-Based RDDs (2)

```
> import json
> myrdd1 = sc.wholeTextFiles(mydir)
> myrdd2 = myrdd1
>     .map(lambda (fname,s): json.loads(s))
> for record in myrdd2.take(2):
>     print record["firstName"]
```

Output:

```
Fred
Barney
```

```
> import scala.util.parsing.json.JSON
> val myrdd1 = sc.wholeTextFiles(mydir)
> val myrdd2 = myrdd1
>     .map(pair => JSON.parseFull(pair._2).get.
>             asInstanceOf[Map[String, String]])
> for (record <- myrdd2.take(2))
>     println(record.getOrElse("firstName", null))
```

Chapter Topics

Working With RDDs in Spark

Distributed Data Processing with Spark

- Creating RDDs
- **Other General RDD Operations**
- Conclusion
- Hands-On Exercise: Process Data Files with Spark

Some Other General RDD Operations

- **Single-RDD Transformations**

- **flatMap** – maps one element in the base RDD to multiple elements
 - **distinct** – filter out duplicates
 - **sortBy** – use provided function to sort

- **Multi-RDD Transformations**

- **intersection** – create a new RDD with all elements in both original RDDs
 - **union** – add all elements of two RDDs into a single new RDD
 - **zip** – pair each element of the first RDD with the corresponding element of the second

Example: flatMap and distinct

Python

```
> sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .distinct()
```

Scala

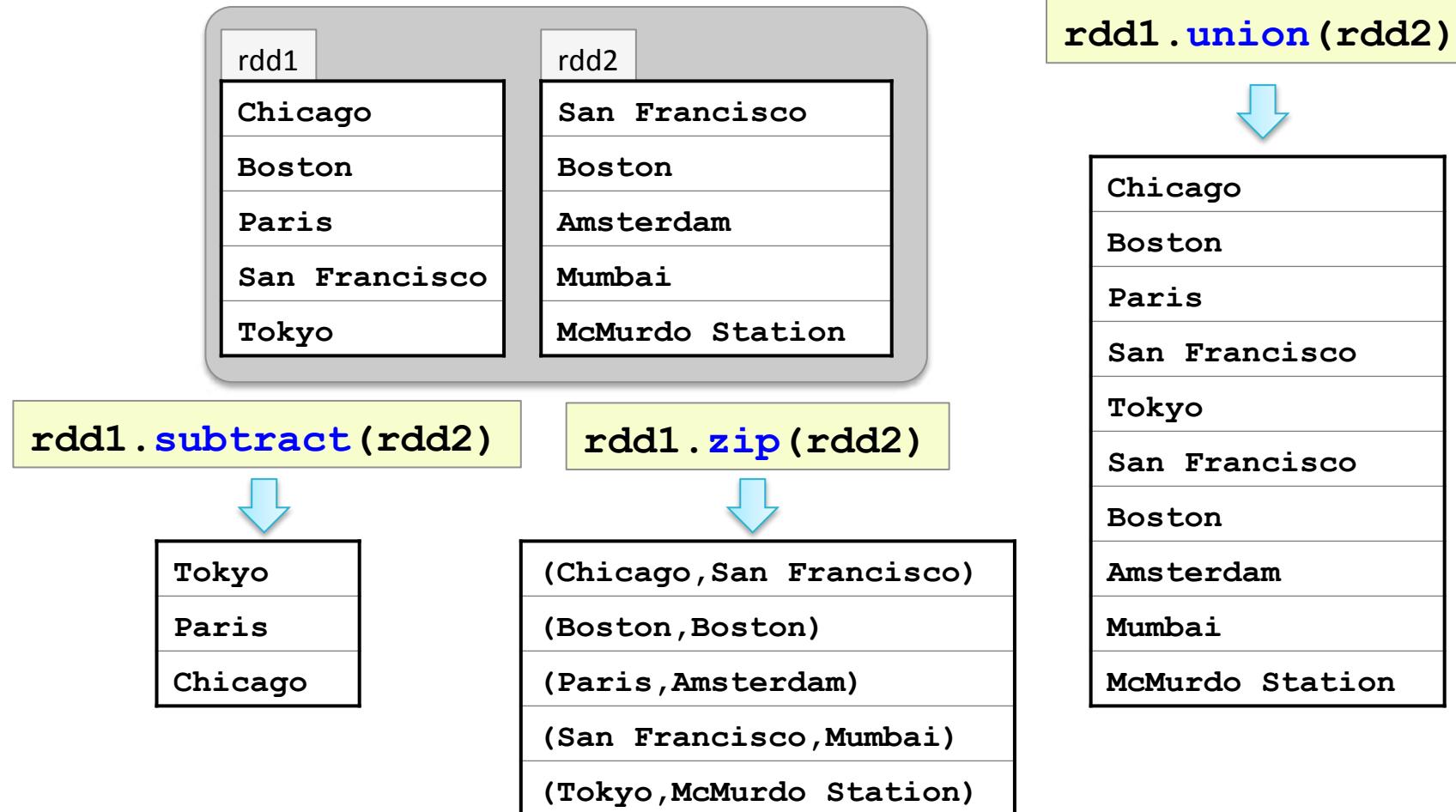
```
> sc.textFile(file).  
    flatMap(line => line.split(' ')).  
    distinct()
```

```
I've never seen a purple cow.  
I never hope to see one;  
But I can tell you, anyhow,  
I'd rather see than be one.
```

I've
never
seen
a
purple
cow
I
never
hope
to
...

I've
never
seen
a
purple
cow
I
hope
to
...

Examples: Multi-RDD Transformations



Some Other General RDD Operations

- **Other RDD operations**

- **first** – return the first element of the RDD
 - **foreach** – apply a function to each element in an RDD
 - **top (n)** – return the largest n elements using natural ordering

- **Sampling operations**

- **sample** – create a new RDD with a sampling of elements
 - **takeSample** – return an array of sampled elements

- **Double RDD operations**

- Statistical functions, e.g., **mean**, **sum**, **variance**, **stdev**

Chapter Topics

Working With RDDs in Spark

Distributed Data Processing with Spark

- Creating RDDs
- Other General RDD Operations
- **Conclusion**
- Hands-On Exercise: Process Data Files with Spark

Essential Points

- RDDs can be created from files, parallelized data in memory, or other RDDs
- `sc.textFile` reads newline delimited text, one line per RDD record
- `sc.wholeTextFile` reads entire files into single RDD records
- Generic RDDs can consist of any type of data
- Generic RDDs provide a wide range of transformation operations

Chapter Topics

Working With RDDs in Spark

Distributed Data Processing with Spark

- Creating RDDs
- Other General RDD Operations
- Conclusion
- **Hands-On Exercise: Process Data Files with Spark**

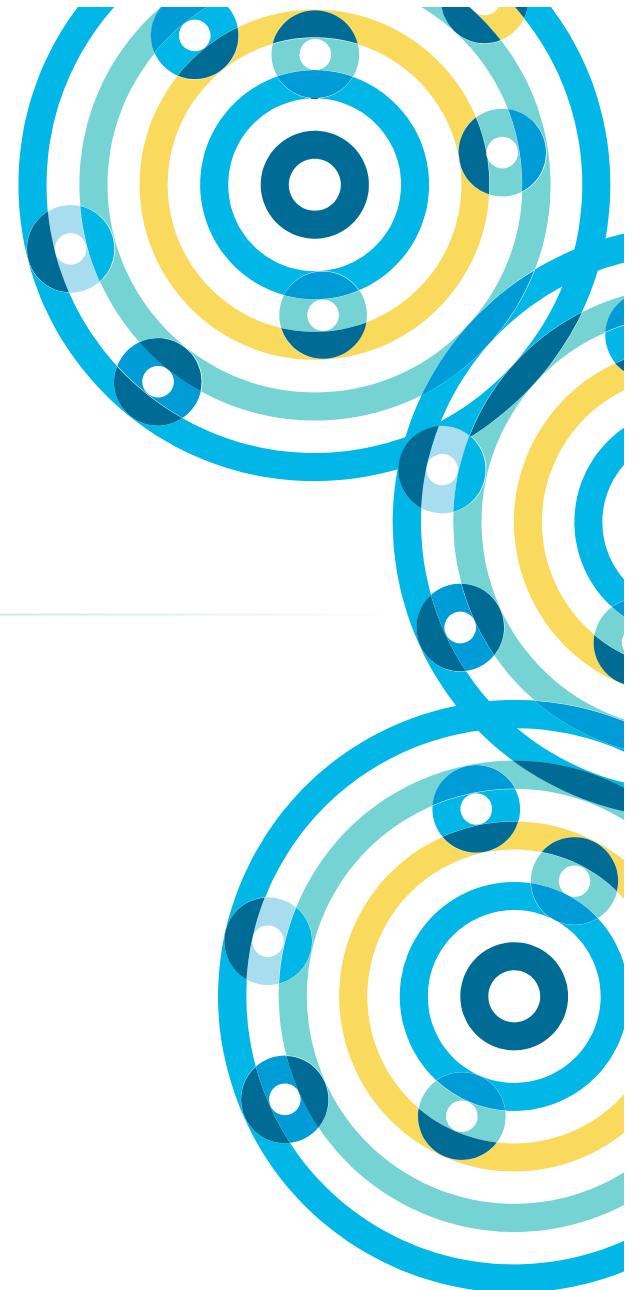
Hands-On Exercise: Process Data Files with Spark

- **In this exercise you will**
 - Process a set of XML files using `wholeTextFiles`
 - Reformat a dataset to standardize format (bonus)
- **Please refer to the Hands-On Exercise Manual**



Aggregating Data with Pair RDDs

Chapter 12



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs**
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

Aggregating Data with Pair RDDs

In this chapter you will learn

- **How to create Pair RDDs of key-value pairs from generic RDDs**
- **Special operations available on Pair RDDs**
- **How map-reduce algorithms are implemented in Spark**

Chapter Topics

Aggregating Data with Pair RDDs

Distributed Data Processing with Spark

- **Key-Value Pair RDDs**
- Map-Reduce
- Other Pair RDD Operations
- Conclusion
- Hands-On Exercise: Use Pair RDDs to Join Two Datasets

Pair RDDs

- **Pair RDDs are a special form of RDD**
 - Each element must be a key-value pair (a two-element tuple)
 - Keys and values can be any type
- **Why?**
 - Use with map-reduce algorithms
 - Many additional functions are available for common data processing needs
 - e.g., sorting, joining, grouping, counting, etc.

Pair RDD
(key1,value1)
(key2,value2)
(key3,value3)
...

Creating Pair RDDs

- **The first step in most workflows is to get the data into key/value form**
 - What should the RDD should be keyed on?
 - What is the value?
- **Commonly used functions to create Pair RDDs**
 - `map`
 - `flatMap / flatMapValues`
 - `keyBy`

Example: A Simple Pair RDD

- Example: Create a Pair RDD from a tab-separated file

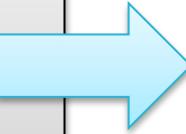
Python

```
> users = sc.textFile(file) \
    .map(lambda line: line.split('\t')) \
    .map(lambda fields: (fields[0],fields[1]))
```

Scala

```
> val users = sc.textFile(file) \
    .map(line => line.split('\t')) \
    .map(fields => (fields(0),fields(1)))
```

user001\tFred Flintstone
user090\tBugs Bunny
user111\tHarry Potter
...



(user001,Fred Flintstone)
(user090,Bugs Bunny)
(user111,Harry Potter)
...

Example: Keying Web Logs by User ID

Python

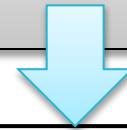
```
> sc.textFile(logfile) \
    .keyBy(lambda line: line.split(' ')[2])
```

Scala

```
> sc.textFile(logfile) \
    .keyBy(line => line.split(' ')(2))
```

User ID

```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" ...
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" ...
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0" ...
...
```



(99788,56.38.234.188 - 99788 "GET /KBDOC-00157.html...")

(99788,56.38.234.188 - 99788 "GET /theme.css...")

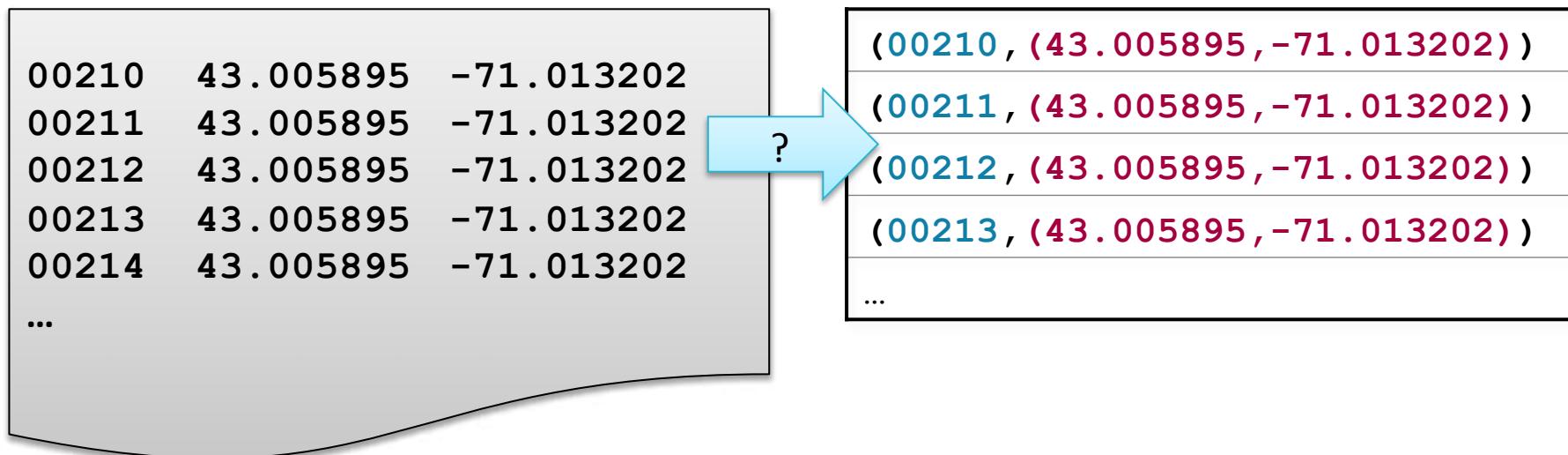
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html...")

...

Question 1: Pairs With Complex Values

- How would you do this?

- Input: a list of postal codes with latitude and longitude
- Output: postal code (key) and lat/long pair (value)



Answer 1: Pairs With Complex Values

```
> sc.textFile(file) \
    .map(lambda line: line.split()) \
    .map(lambda fields: (fields[0], (fields[1], fields[2])))
```

```
> sc.textFile(file) .
    map(line => line.split('\t')) .
    map(fields => (fields(0), (fields(1), fields(2))))
```

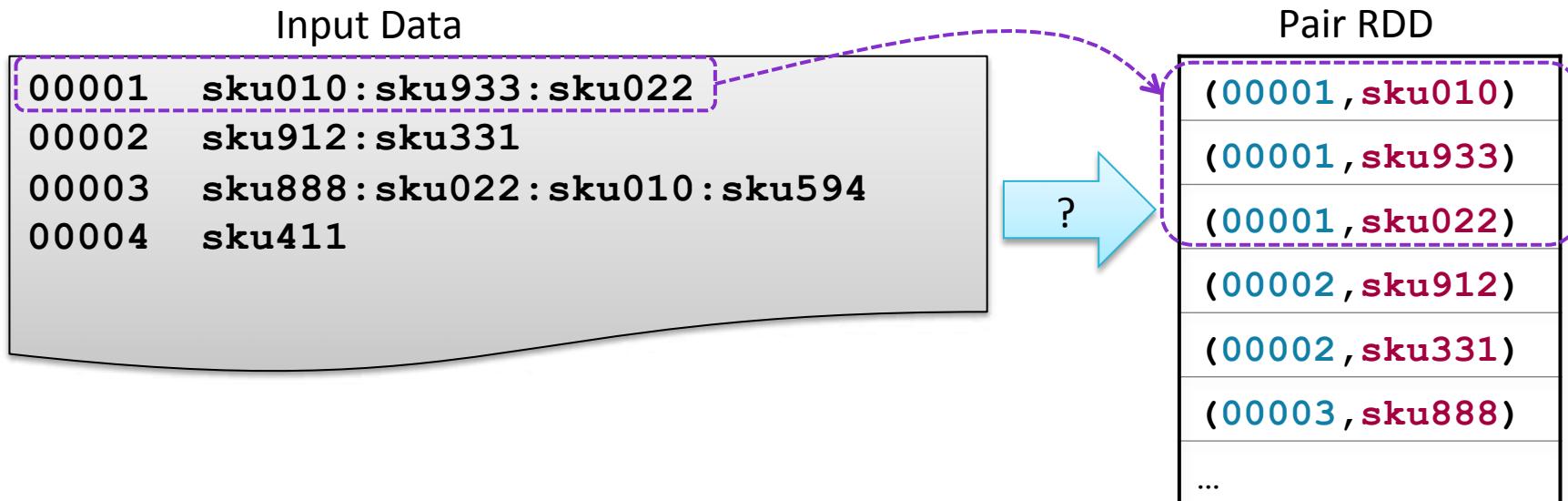
00210	43.005895	-71.013202
01014	42.170731	-72.604842
01062	42.324232	-72.67915
01263	42.3929	-73.228483
...		

(00210, (43.005895, -71.013202))
(01014, (42.170731, -72.604842))
(01062, (42.324232, -72.67915))
(01263, (42.3929, -73.228483))
...

Question 2: Mapping Single Rows to Multiple Pairs (1)

- **How would you do this?**

- Input: order numbers with a list of SKUs in the order
- Output: **order** (key) and **sku** (value)



Question 2: Mapping Single Rows to Multiple Pairs (2)

- Hint: map alone won't work

```
00001  sku010:sku933:sku022  
00002  sku912:sku331  
00003  sku888:sku022:sku010:sku594  
00004  sku411
```



(00001, (sku010,sku933,sku022))
(00002, (sku912,sku331))
(00003, (sku888,sku022,sku010,sku594))
(00004, (sku411))

Answer 2: Mapping Single Rows to Multiple Pairs (1)

```
> sc.textFile(file)
```

00001	sku010:sku933:sku022
00002	sku912:sku331
00003	sku888:sku022:sku010:sku594
00004	sku411

Answer 2: Mapping Single Rows to Multiple Pairs (2)

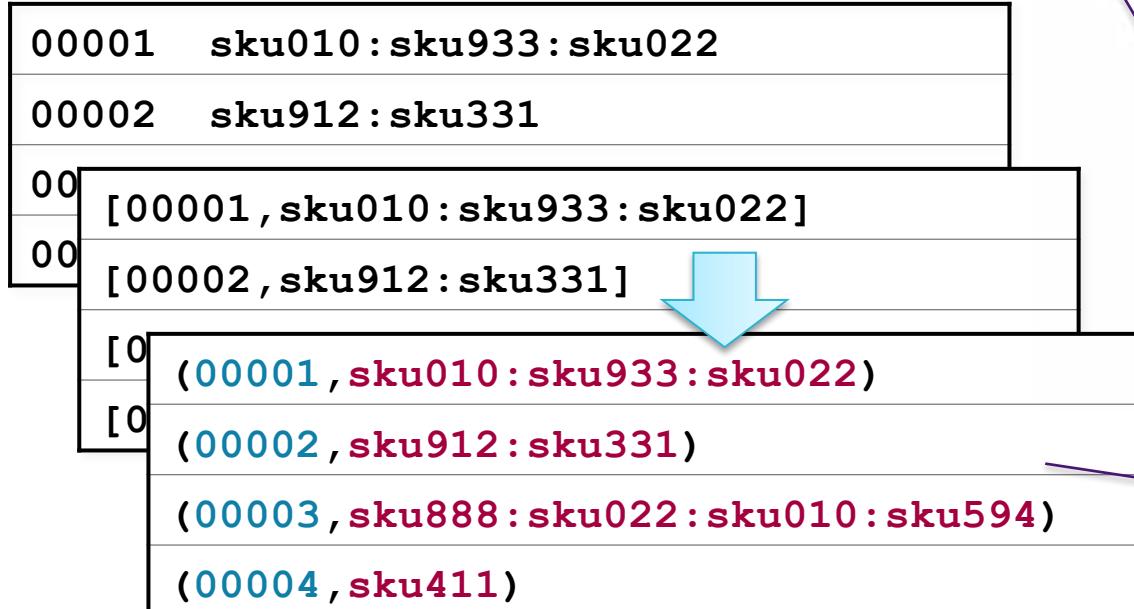
```
> sc.textFile(file) \
    .map(lambda line: line.split('\t'))
```

00001	sku010:sku933:sku022
00002	sku912:sku331
00003	[00001,sku010:sku933:sku022]
00004	[00002,sku912:sku331]
00005	[00003,sku888:sku022:sku010:sku594]
00006	[00004,sku411]

Note that `split` returns
2-element arrays, not
pairs/tuples

Answer 2: Mapping Single Rows to Multiple Pairs (3)

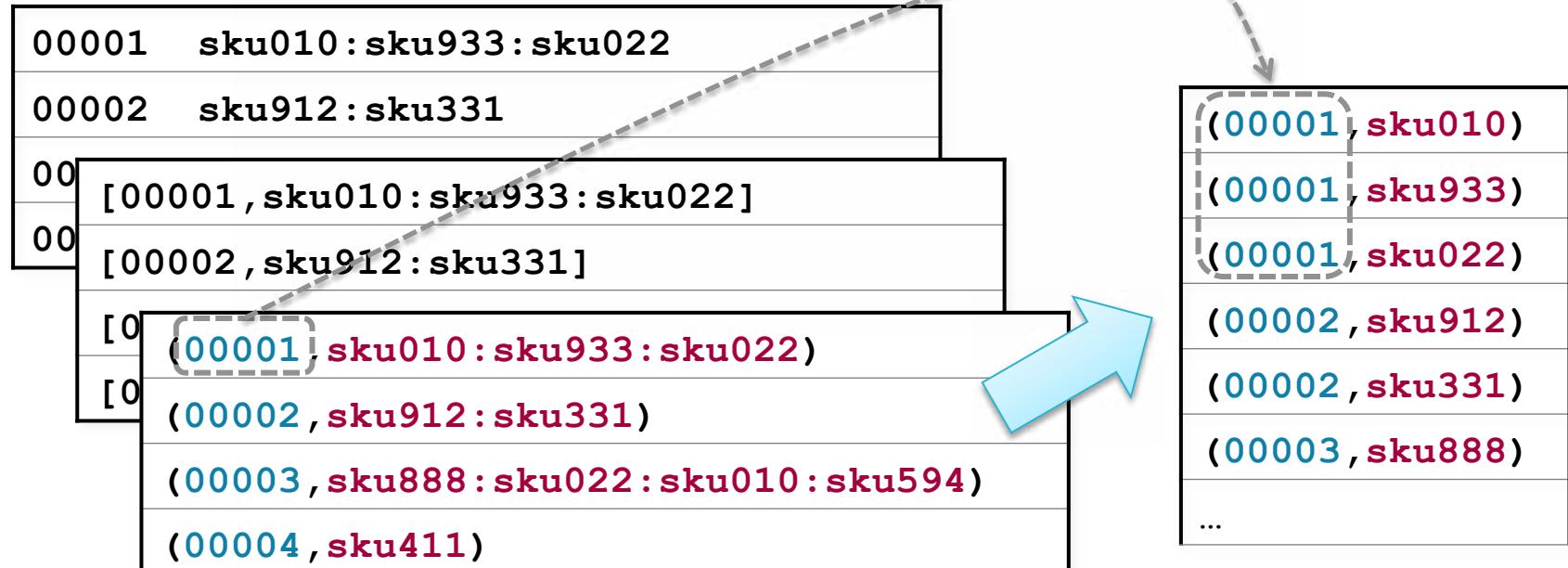
```
> sc.textFile(file) \
    .map(lambda line: line.split('\t')) \
    .map(lambda fields: (fields[0],fields[1]))
```



Map array elements to
tuples to produce a
Pair RDD

Answer 2: Mapping Single Rows to Multiple Pairs (4)

```
> sc.textFile(file) \
    .map(lambda line: line.split('\t')) \
    .map(lambda fields: (fields[0], fields[1])) \
    .flatMapValues(lambda skus: skus.split(':'))
```



Chapter Topics

Aggregating Data with Pair RDDs

Distributed Data Processing with Spark

- Key-Value Pair RDDs
- **Map-Reduce**
- Other Pair RDD Operations
- Conclusion
- Hands-On Exercise: Use Pair RDDs to Join Two Datasets

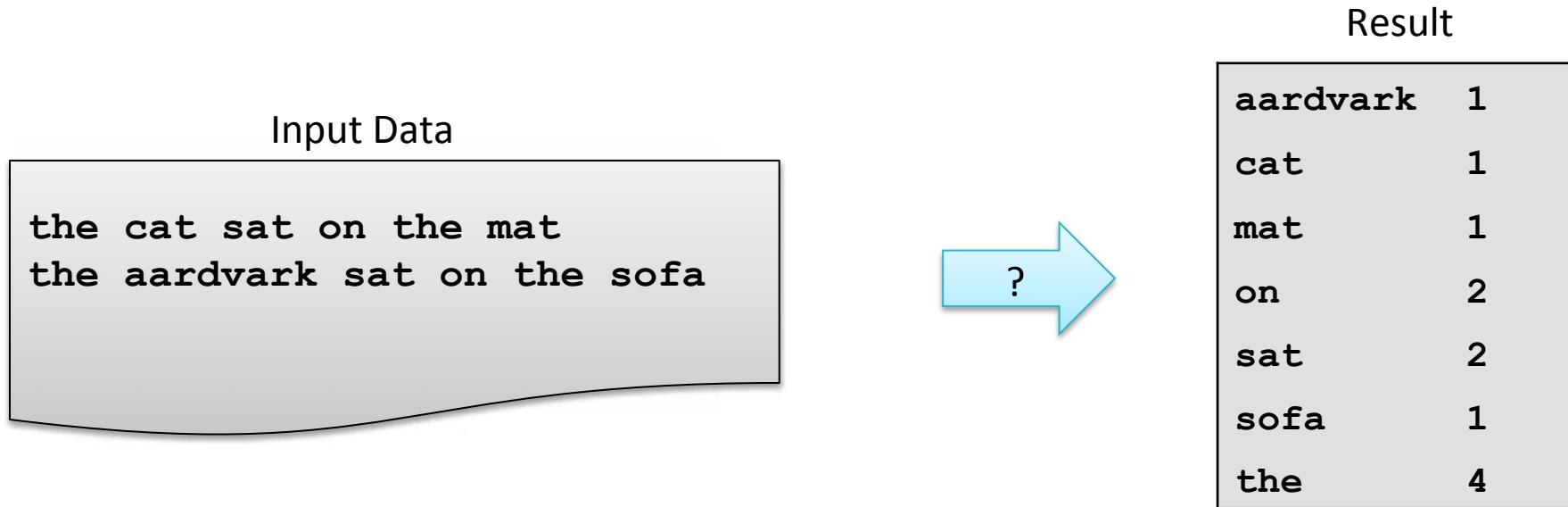
Map-Reduce

- **Map-reduce is a common programming model**
 - Easily applicable to distributed processing of large data sets
- **Hadoop MapReduce is the major implementation**
 - Somewhat limited
 - Each job has one Map phase, one Reduce phase
 - Job output is saved to files
- **Spark implements map-reduce with much greater flexibility**
 - Map and reduce functions can be interspersed
 - Results can be stored in memory
 - Operations can easily be chained

Map-Reduce in Spark

- Map-reduce in Spark works on Pair RDDs
- Map phase
 - Operates on one record at a time
 - “Maps” each record to one or more new records
 - e.g. `map`, `flatMap`, `filter`, `keyBy`
- Reduce phase
 - Works on map output
 - Consolidates multiple records
 - e.g. `reduceByKey`, `sortByKey`, `mean`

Map-Reduce Example: Word Count



Example: Word Count (1)

```
> counts = sc.textFile(file)
```

```
the cat sat on the  
mat
```

```
the aardvark sat on  
the sofa
```

Example: Word Count (2)

```
> counts = sc.textFile(file) \  
.flatMap(lambda line: line.split())
```

the cat sat on the mat
the aardvark sat on the sofa



the
cat
sat
on
the
mat
the
aardvark
...

Example: Word Count (3)

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word, 1))
```

Key-Value Pairs

the cat sat on the
mat
the aardvark sat on
the sofa



the
cat
sat
on
the
mat
the
aardvark
...



(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
...

Example: Word Count (4)

```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```

the cat sat on the
mat

the aardvark sat on
the sofa



the
cat
sat
on
the
mat
the
aardvark
...



(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
...

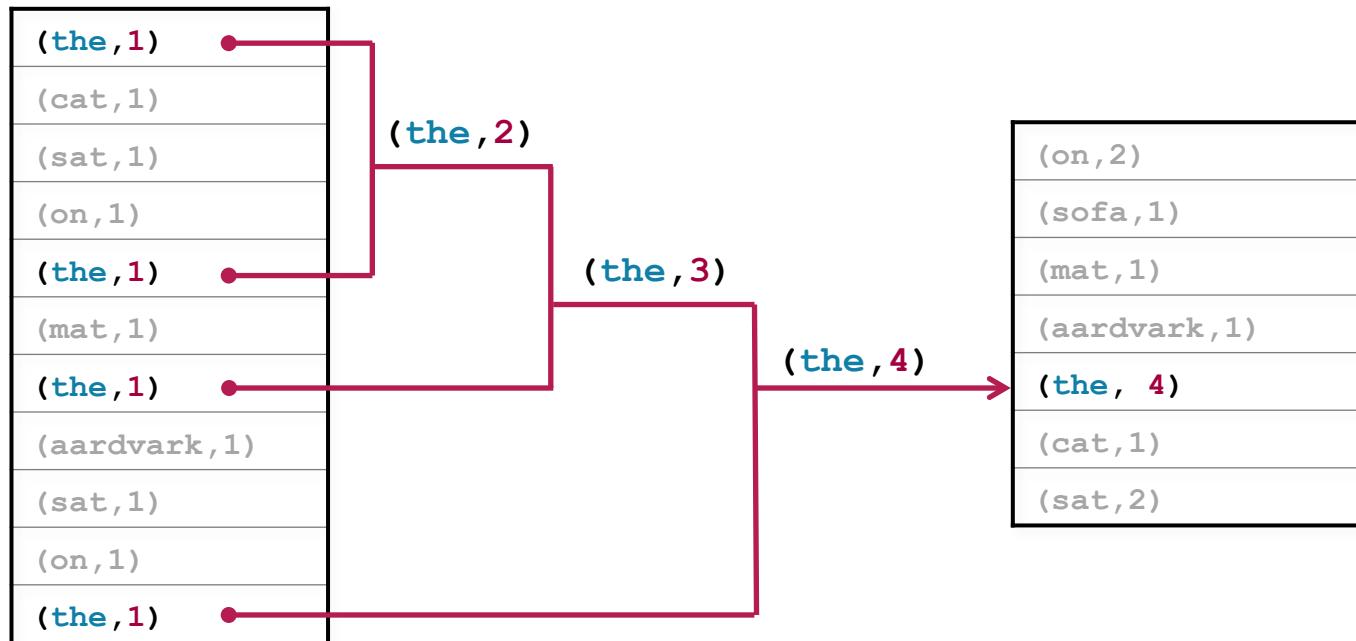


(aardvark, 1)
(cat, 1)
(mat, 1)
(on, 2)
(sat, 2)
(sofa, 1)
(the, 4)

ReduceByKey (1)

- The function passed to `reduceByKey` combines values from two keys
 - Function must be binary

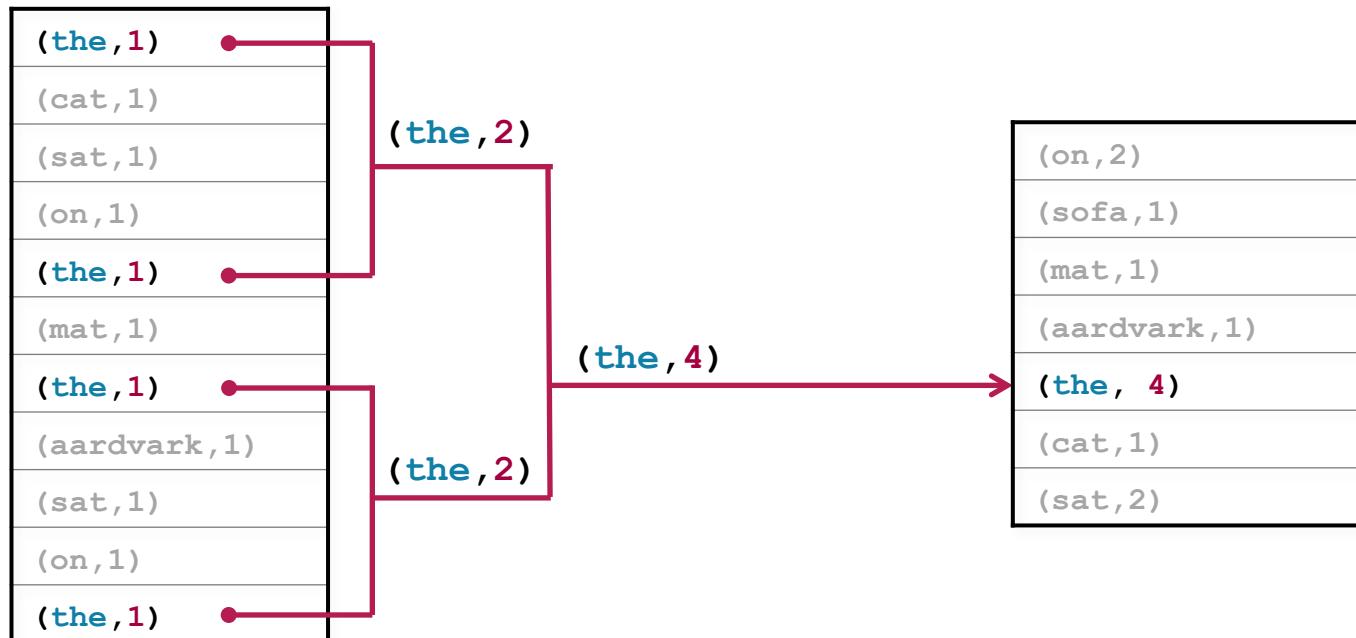
```
> counts = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word,1)) \
    .reduceByKey(lambda v1,v2: v1+v2)
```



ReduceByKey (2)

- The function might be called in any order, therefore must be
 - Commutative – $x+y = y+x$
 - Associative – $(x+y)+z = x+(y+z)$

```
> counts = sc.textFile(file) \
  .flatMap(lambda line: line.split()) \
  .map(lambda word: (word,1)) \
  .reduceByKey(lambda v1,v2: v1+v2)
```



Word Count Recap (the Scala Version)

```
> val counts = sc.textFile(file).  
  flatMap(line => line.split("\\w+")).  
  map(word => (word, 1)).  
  reduceByKey((v1, v2) => v1+v2)
```

OR

```
> val counts = sc.textFile(file).  
  flatMap(_.split("\\w+")).  
  map((_, 1)).  
  reduceByKey(_+_)
```

Why Do We Care About Counting Words?

- **Word count is challenging over massive amounts of data**
 - Using a single compute node would be too time-consuming
 - Number of unique words could exceed available memory
- **Statistics are often simple aggregate functions**
 - Distributive in nature
 - e.g., max, min, sum, count
- **Map-reduce breaks complex tasks down into smaller elements which can be executed in parallel**
- **Many common tasks are very similar to word count**
 - e.g., log file analysis

Chapter Topics

Aggregating Data with Pair RDDs

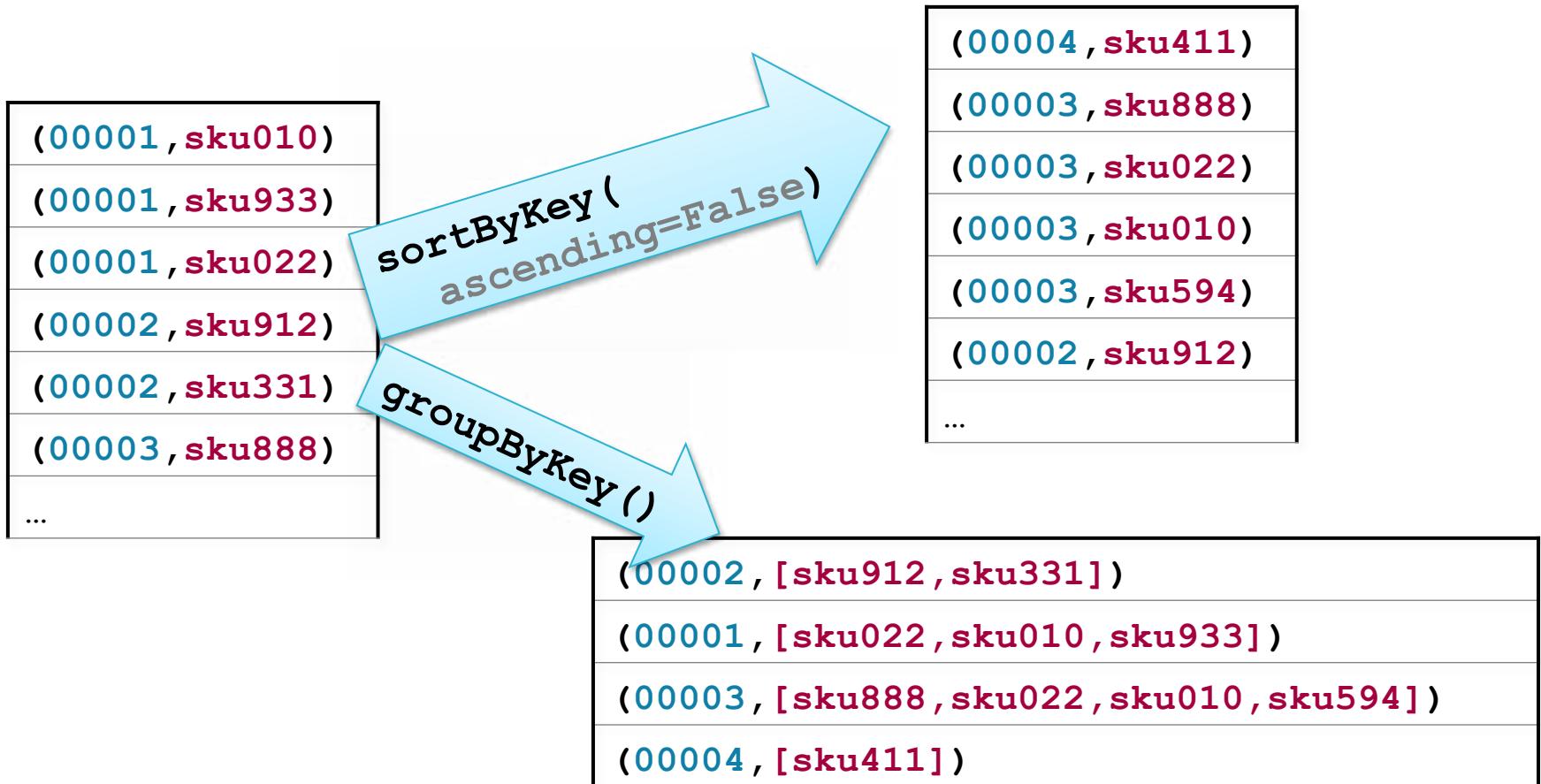
Distributed Data Processing with Spark

- Key-Value Pair RDDs
- Map-Reduce
- **Other Pair RDD Operations**
- Conclusion
- Hands-On Exercise: Use Pair RDDs to Join Two Datasets

Pair RDD Operations

- In addition to `map` and `reduce` functions, Spark has several operations specific to Pair RDDs
- Examples
 - `countByKey` – return a map with the count of occurrences of each key
 - `groupByKey` – group all the values for each key in an RDD
 - `sortByKey` – sort in ascending or descending order
 - `join` – return an RDD containing all pairs with matching keys from two RDDs

Example: Pair RDD Operations



Example: Joining by Key

```
> movies = moviegross.join(movieyear)
```

RDD: moviegross
(Casablanca, \$3.7M)
(Star Wars, \$775M)
(Annie Hall, \$38M)
(Argo, \$232M)
...

RDD: movieyear
(Casablanca, 1942)
(Star Wars, 1977)
(Annie Hall, 1977)
(Argo, 2012)
...

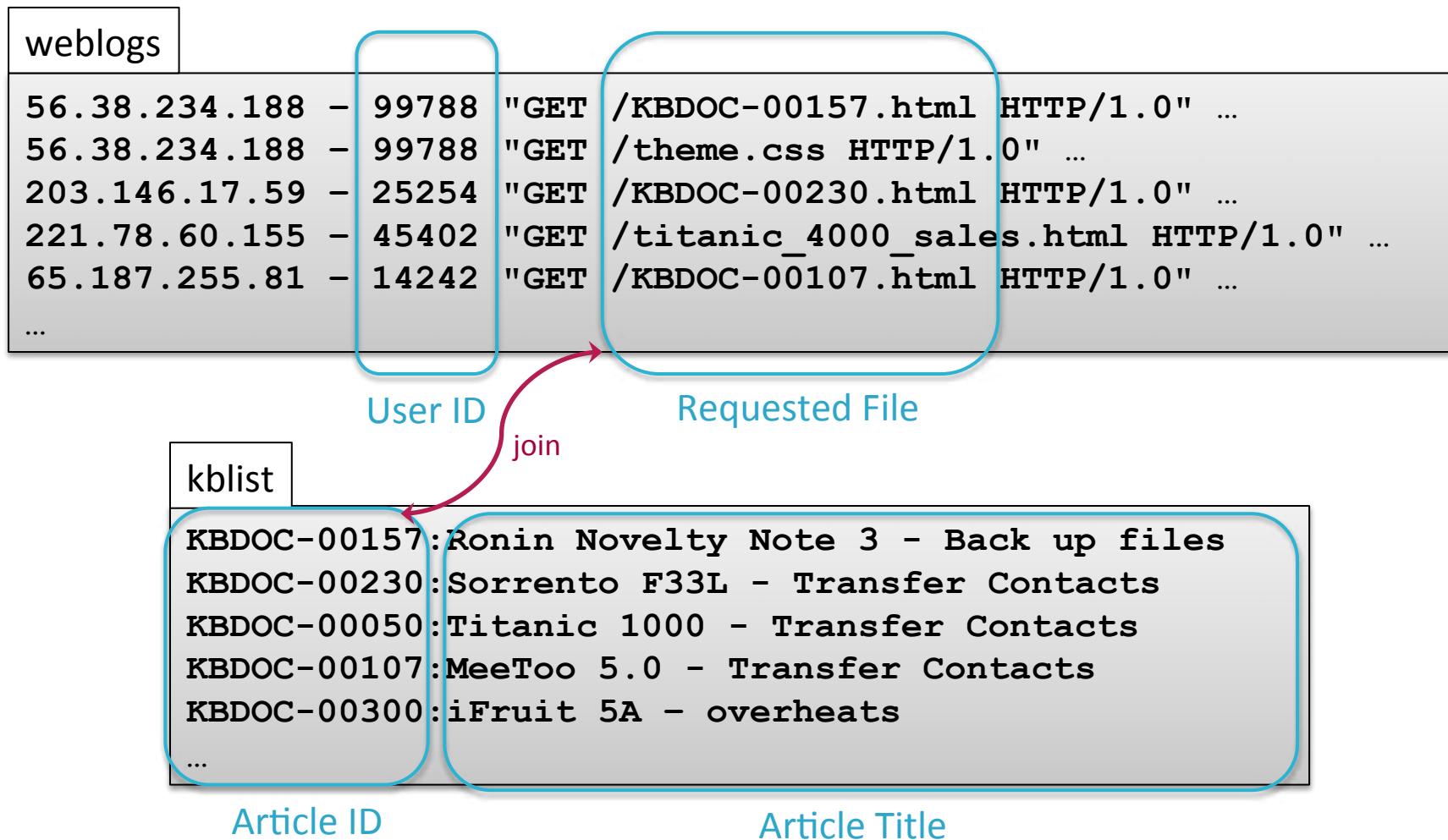
(Casablanca, (\$3.7M, 1942))
(Star Wars, (\$775M, 1977))
(Annie Hall, (\$38M, 1977))
(Argo, (\$232M, 2012))
...

Using Join

- A common programming pattern

1. Map separate datasets into key-value Pair RDDs
2. Join by key
3. Map joined data into the desired format
4. Save, display, or continue processing...

Example: Join Web Log With Knowledge Base Articles (1)



Example: Join Web Log With Knowledge Base Articles (2)

- Steps

1. Map separate datasets into key-value Pair RDDs
 - a. Map web log requests to (**docid,userid**)
 - b. Map KB Doc index to (**docid,title**)
2. Join by key: **docid**
3. Map joined data into the desired format: (**userid,title**)
4. Further processing: group titles by User ID

Step 1a: Map Web Log Requests to **(docid, userid)**

```
> import re
> def getRequestDoc(s):
    return re.search(r'KBDOC-[0-9]*',s).group()

> kbreqs = sc.textFile(logfile) \
    .filter(lambda line: 'KBDOC-' in line) \
    .map(lambda line: (getRequestDoc(line),line.split(' ')[2])) \
    .distinct()
```

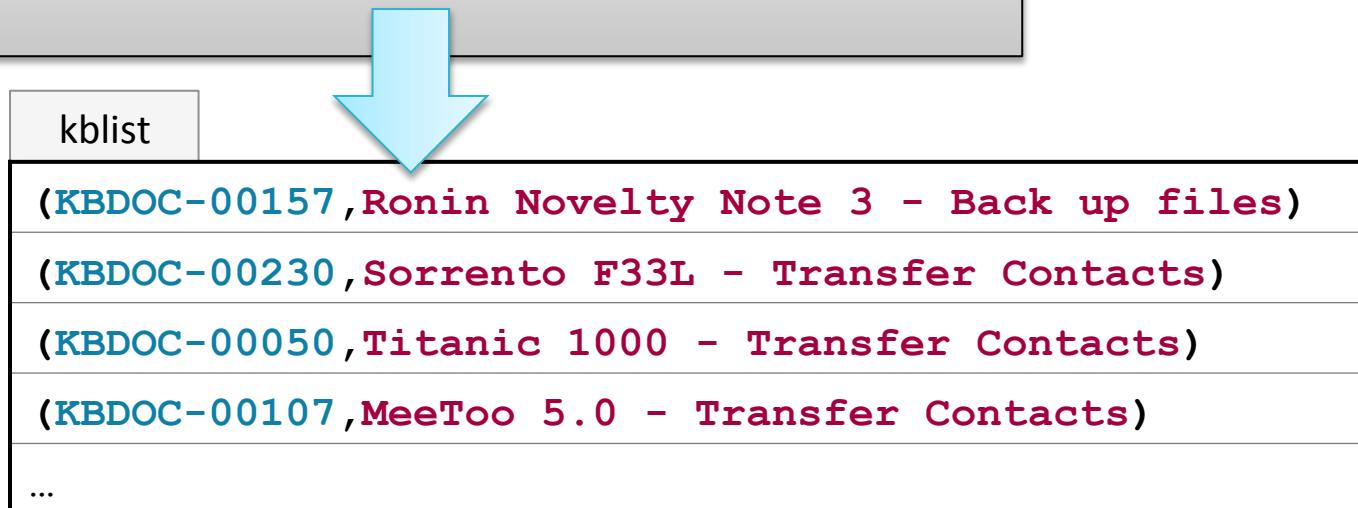
```
56.38.234.188 - 99788 "GET /KBDOC-00157.html HTTP/1.0" ...
56.38.234.188 - 99788 "GET /theme.css HTTP/1.0" ...
203.146.17.59 - 25254 "GET /KBDOC-00230.html HTTP/1.0"
221.78.60.155 - 45402 "GET /titanic_4000_sales.html"
65.187.255.81 - 14242 "GET /KBDOC-00107.html HTTP/1.0"
...
```

kbreqs	...
(KBDOC-00157, 99788)	
(KBDOC-00203, 25254)	
(KBDOC-00107, 14242)	
...	

Step 1b: Map KB Index to (docid, title)

```
> kblist = sc.textFile(kblistfile) \
    .map(lambda line: line.split(':')) \
    .map(lambda fields: (fields[0], fields[1]))
```

```
KBDOC-00157:Ronin Novelty Note 3 - Back up files
KBDOC-00230:Sorrento F33L - Transfer Contacts
KBDOC-00050:Titanic 1000 - Transfer Contacts
KBDOC-00107:MeeToo 5.0 - Transfer Contacts
KBDOC-00206:iFruit 5A - overheats
...
```



Step 2: Join By Key **docid**

```
> titlereqs = kbreqs.join(kblist)
```

kbreqs
(KBDOC-00157, 99788)
(KBDOC-00230, 25254)
(KBDOC-00107, 14242)
...



kblist
(KBDOC-00157, Ronin Novelty Note 3 - Back up files)
(KBDOC-00230, Sorrento F33L - Transfer Contacts)
(KBDOC-00050, Titanic 1000 - Transfer Contacts)
(KBDOC-00107, MeeToo 5.0 - Transfer Contacts)
...

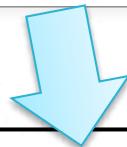


(KBDOC-00157, (99788, Ronin Novelty Note 3 - Back up files))
(KBDOC-00230, (25254, Sorrento F33L - Transfer Contacts))
(KBDOC-00107, (14242, MeeToo 5.0 - Transfer Contacts))
...

Step 3: Map Result to Desired Format (`userid, title`)

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid, (userid, title)): (userid, title))
```

(KBDOC-00157, (99788, Ronin Novelty Note 3 - Back up files))
(KBDOC-00230, (25254, Sorrento F33L - Transfer Contacts))
(KBDOC-00107, (14242, MeeToo 5.0 - Transfer Contacts))
...



(99788, Ronin Novelty Note 3 - Back up files)
(25254, Sorrento F33L - Transfer Contacts)
(14242, MeeToo 5.0 - Transfer Contacts)
...

Step 4: Continue Processing – Group Titles by User ID

```
> titlereqs = kbreqs.join(kblist) \
    .map(lambda (docid, (userid, title)): (userid, title)) \
    .groupByKey()
```

```
(99788, Ronin Novelty Note 3 - Back up files)
(25254, Sorrento F33L - Transfer Contacts)
(14242, MeeToo 5.0 - Transfer Contacts)
...
```



Note: values
are grouped
into Iterables

```
(99788, [Ronin Novelty Note 3 - Back up files,
          Ronin S3 - overheating])
(25254, [Sorrento F33L - Transfer Contacts])
(14242, [MeeToo 5.0 - Transfer Contacts,
          MeeToo 5.1 - Back up files,
          iFruit 1 - Back up files,
          MeeToo 3.1 - Transfer Contacts])
...
```

Example Output

```
> for (userid,titles) in titlereqs.take(10):  
    print 'user id: ',userid  
    for title in titles: print '\t',title
```

user id: 99788

Ronin Novelty Note 3 - Back up files

Ronin S3 - overheating

user id: 25254

Sorrento F33L - Transfer Contacts

user id: 14242

MeeToo 5.0 - Transfer Contacts

MeeToo 5.1 - Back up files

iFruit 1 - Back up files

MeeToo 3.1 - Transfer Contacts

...

(99788, [Ronin Novelty Note 3 - Back up files,
Ronin S3 - overheating])

(25254, [Sorrento F33L - Transfer Contacts])

(14242, [MeeToo 5.0 - Transfer Contacts,
MeeToo 5.1 - Back up files,
iFruit 1 - Back up files,
MeeToo 3.1 - Transfer Contacts])

...

Aside: Anonymous Function Parameters

- Python and Scala pattern matching can help improve code readability

Python

```
> map(lambda (docid, (userid, title)): (userid, title))
```

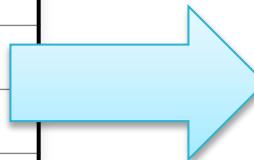
Scala

```
> map(pair => (pair._2._1, pair._2._2))
```

OR

```
> map{case (docid, (userid, title)) => (userid, title)}
```

(KBDOC-00157, (99788, ...title...))
(KBDOC-00230, (25254, ...title...))
(KBDOC-00107, (14242, ...title...))
...



(99788, ...title...)
(25254, ...title...)
(14242, ...title...))
...

Other Pair Operations

- Some other pair operations
 - **keys** – return an RDD of just the keys, without the values
 - **values** – return an RDD of just the values, without keys
 - **lookup (key)** – return the value(s) for a key
 - **leftOuterJoin, rightOuterJoin, fullOuterJoin** – join, including keys defined in the left, right or either RDD respectively
 - **mapValues, flatMapValues** – execute a function on just the values, keeping the key the same
- See the **PairRDDFunctions** class Scaladoc for a full list

Chapter Topics

Aggregating Data with Pair RDDs

Distributed Data Processing with Spark

- Key-Value Pair RDDs
- Map-Reduce
- Other Pair RDD Operations
- **Conclusion**
- Hands-On Exercise: Use Pair RDDs to Join Two Datasets

Essential Points

- **Pair RDDs are a special form of RDD consisting of Key-Value pairs (tuples)**
- **Spark provides several operations for working with Pair RDDs**
- **Map-reduce is a generic programming model for distributed processing**
 - Spark implements map-reduce with Pair RDDs
 - Hadoop MapReduce and other implementations are limited to a single map and single reduce phase per job
 - Spark allows flexible chaining of map and reduce operations
 - Spark provides operations to easily perform common map-reduce algorithms like joining, sorting, and grouping

Chapter Topics

Aggregating Data with Pair RDDs

Distributed Data Processing with Spark

- Key-Value Pair RDDs
- Map-Reduce
- Other Pair RDD Operations
- Conclusion
- **Hands-On Exercise: Use Pair RDDs to Join Two Datasets**

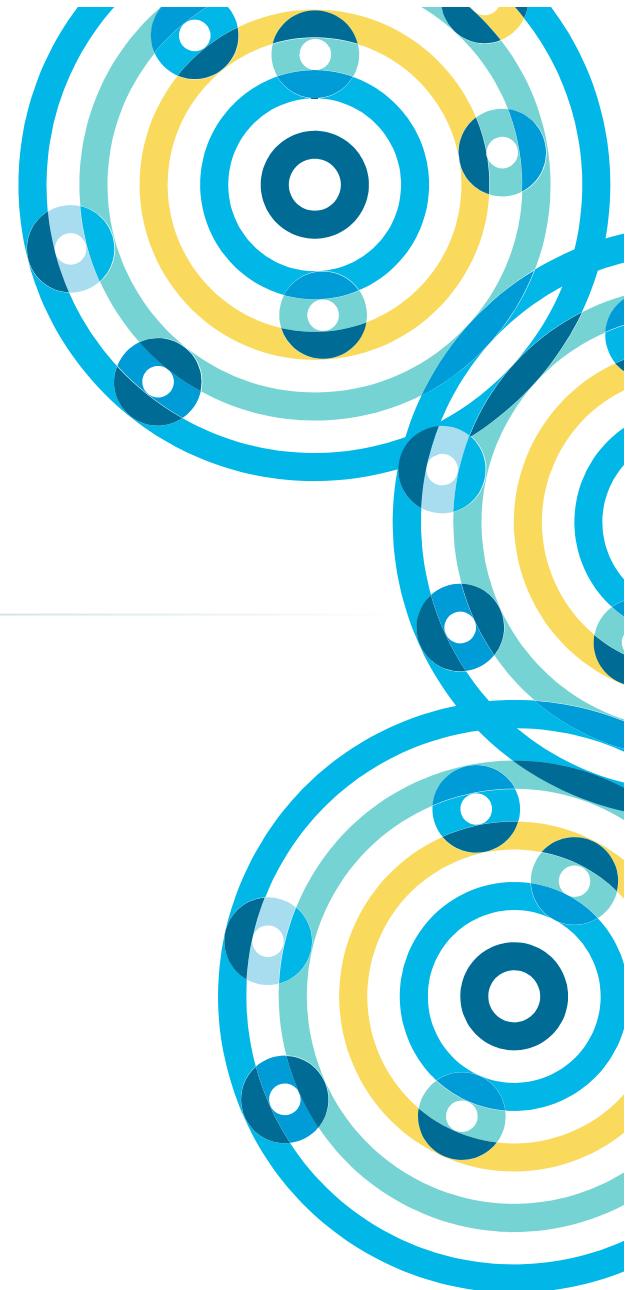
Hands-On Exercise: Use Pair RDDs to Join Two Datasets

- **In this exercise you will**
 - Continue exploring web server log files using key-value Pair RDDs
 - Join log data with user account data
- **Please refer to the Hands-On Exercise Manual**



Writing and Deploying Spark Applications

Chapter 13



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications**
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

Spark on a Cluster

In this chapter you will learn

- **How to write a Spark Application**
- **How to run a Spark Application or the Spark Shell on a YARN cluster**
- **How to access and use the Spark Application Web UI**
- **How to configure application properties and logging**

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- **Spark Applications vs. Spark Shell**
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Spark Shell vs. Spark Applications

- **The Spark Shell allows interactive exploration and manipulation of data**
 - REPL using Python or Scala
- **Spark applications run as independent programs**
 - Python, Scala, or Java
 - e.g., ETL processing, Streaming, and so on

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- **Creating the SparkContext**
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

The SparkContext

- **Every Spark program needs a SparkContext**
 - The interactive shell creates one for you
- **In your own Spark application you create your own SparkContext**
 - Named `sc` by convention
 - Call `sc.stop` when program terminates

Python Example: WordCount

```
import sys
from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print >> sys.stderr, "Usage: WordCount <file>"
        exit(-1)

    sc = SparkContext()

    counts = sc.textFile(sys.argv[1]) \
        .flatMap(lambda line: line.split()) \
        .map(lambda word: (word,1)) \
        .reduceByKey(lambda v1,v2: v1+v2)

    for pair in counts.take(5): print pair

    sc.stop()
```

Scala Example: WordCount

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

object WordCount {
    def main(args: Array[String]) {
        if (args.length < 1) {
            System.err.println("Usage: WordCount <file>")
            System.exit(1)
        }

        val sc = new SparkContext()

        val counts = sc.textFile(args(0)) .
            flatMap(line => line.split("\\W")) .
            map(word => (word,1)) .reduceByKey(_ + _)
        counts.take(5) .foreach(println)

        sc.stop()
    }
}
```

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- **Building a Spark Application (Scala and Java)**
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Building a Spark Application: Scala or Java

- **Scala or Java Spark applications must be compiled and assembled into JAR files**
 - JAR file will be passed to worker nodes
- **Apache Maven is a popular build tool**
 - For specific setting recommendations, see
<http://spark.apache.org/docs/latest/building-with-maven.html>
- **Build details will differ depending on**
 - Version of Hadoop (HDFS)
 - Deployment platform (Spark Standalone, YARN, Mesos)
- **Consider using an IDE**
 - IntelliJ or Eclipse are two popular examples
 - Can run Spark locally in a debugger

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- **Running a Spark Application**
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Running a Spark Application

- The easiest way to run a Spark Application is using the **spark-submit** script

Python

```
$ spark-submit WordCount.py fileURL
```

Scala

Java

```
$ spark-submit --class WordCount \
MyJarFile.jar fileURL
```

Spark Application Cluster Options

- **Spark can run**
 - Locally
 - No distributed processing
 - Locally with multiple worker threads
 - On a cluster
- **Local mode is useful for development and testing**
- **Production use is almost always on a cluster**

Supported Cluster Resource Managers

- **Hadoop YARN**

- Included in CDH
 - Most common for production sites
 - Allows sharing cluster resources with other applications (e.g. MapReduce, Impala)

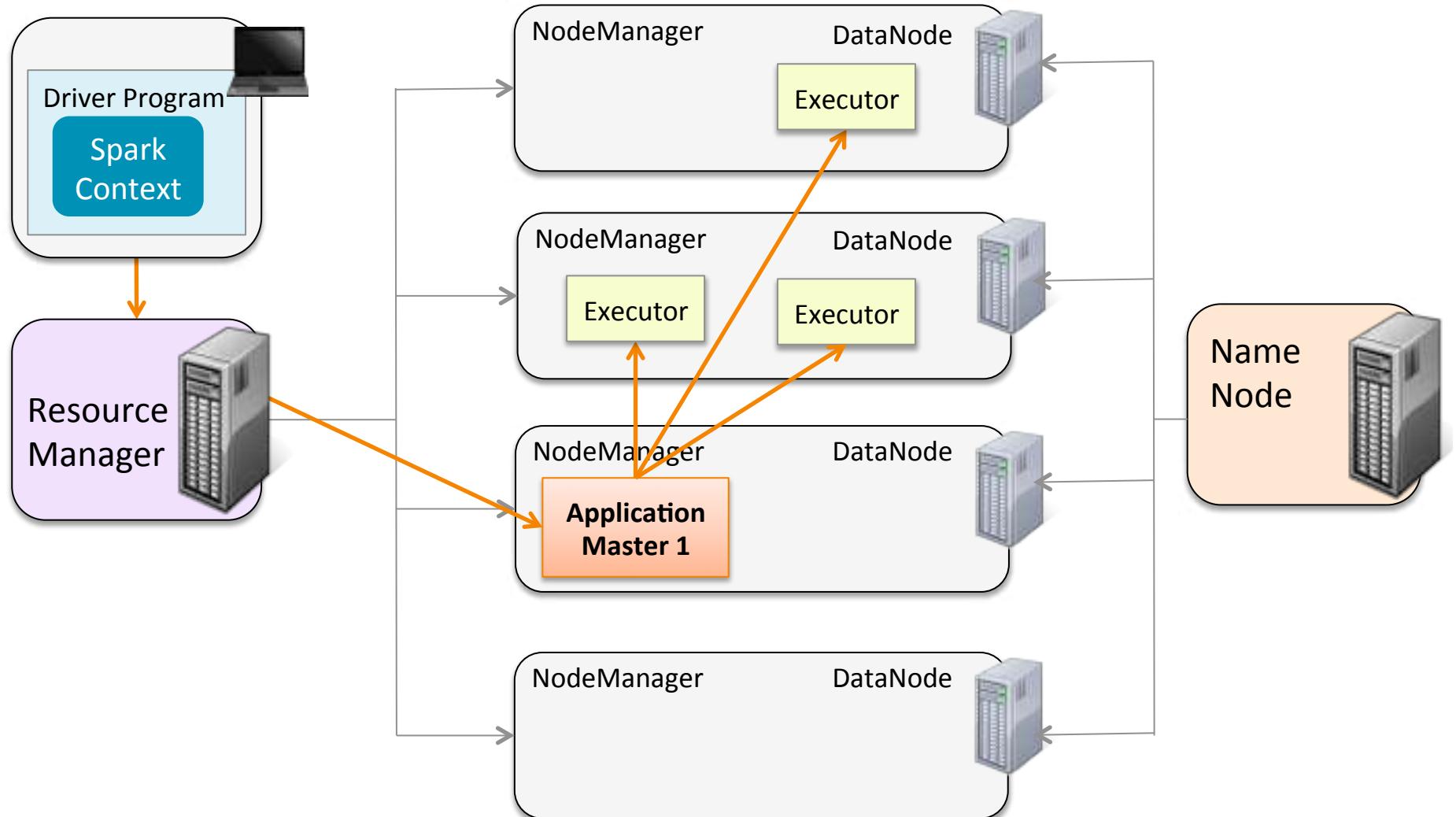
- **Spark Standalone**

- Included with Spark
 - Easy to install and run
 - Limited configurability and scalability
 - Useful for testing, development, or small systems

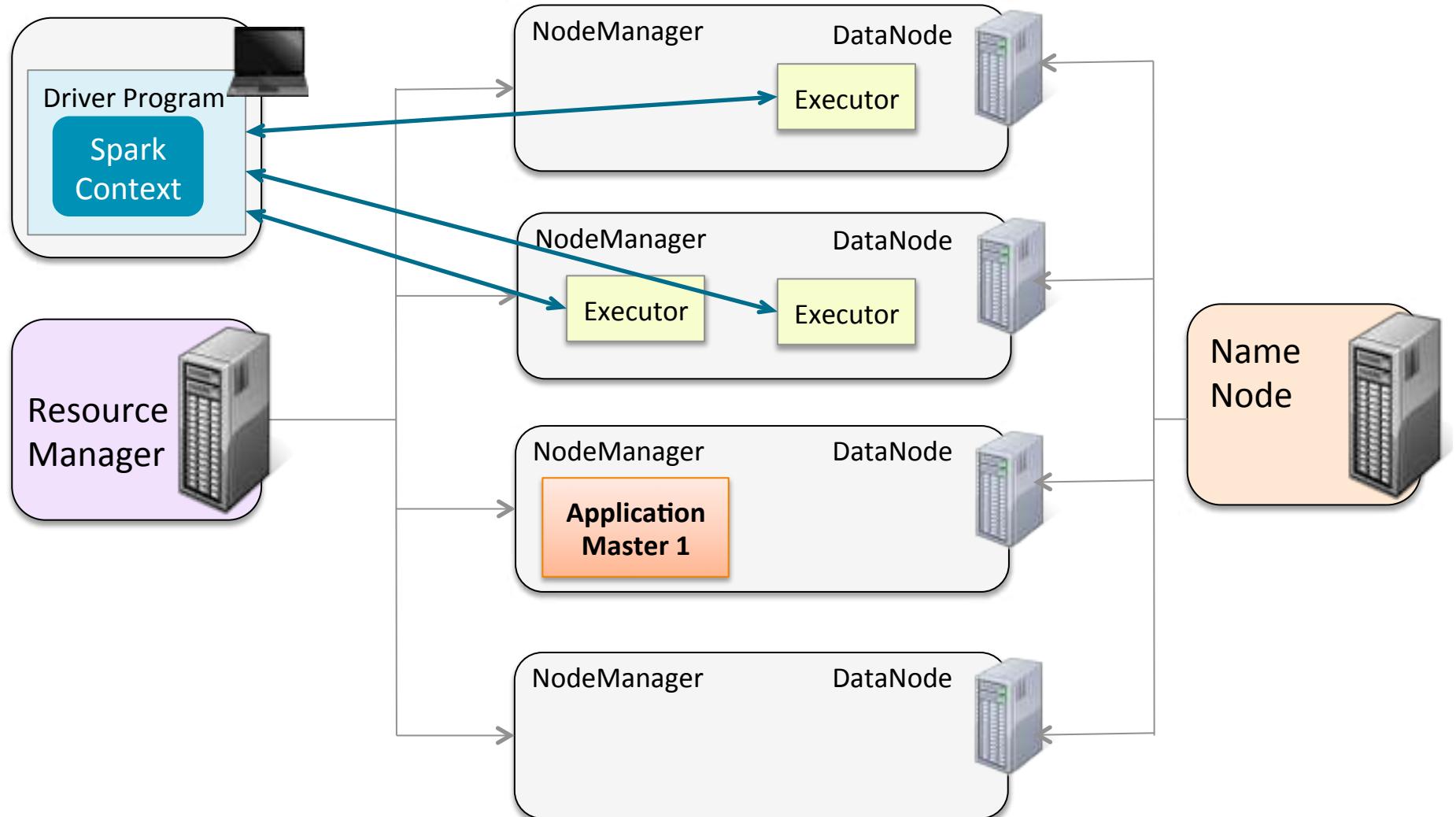
- **Apache Mesos**

- First platform supported by Spark
 - Now used less often

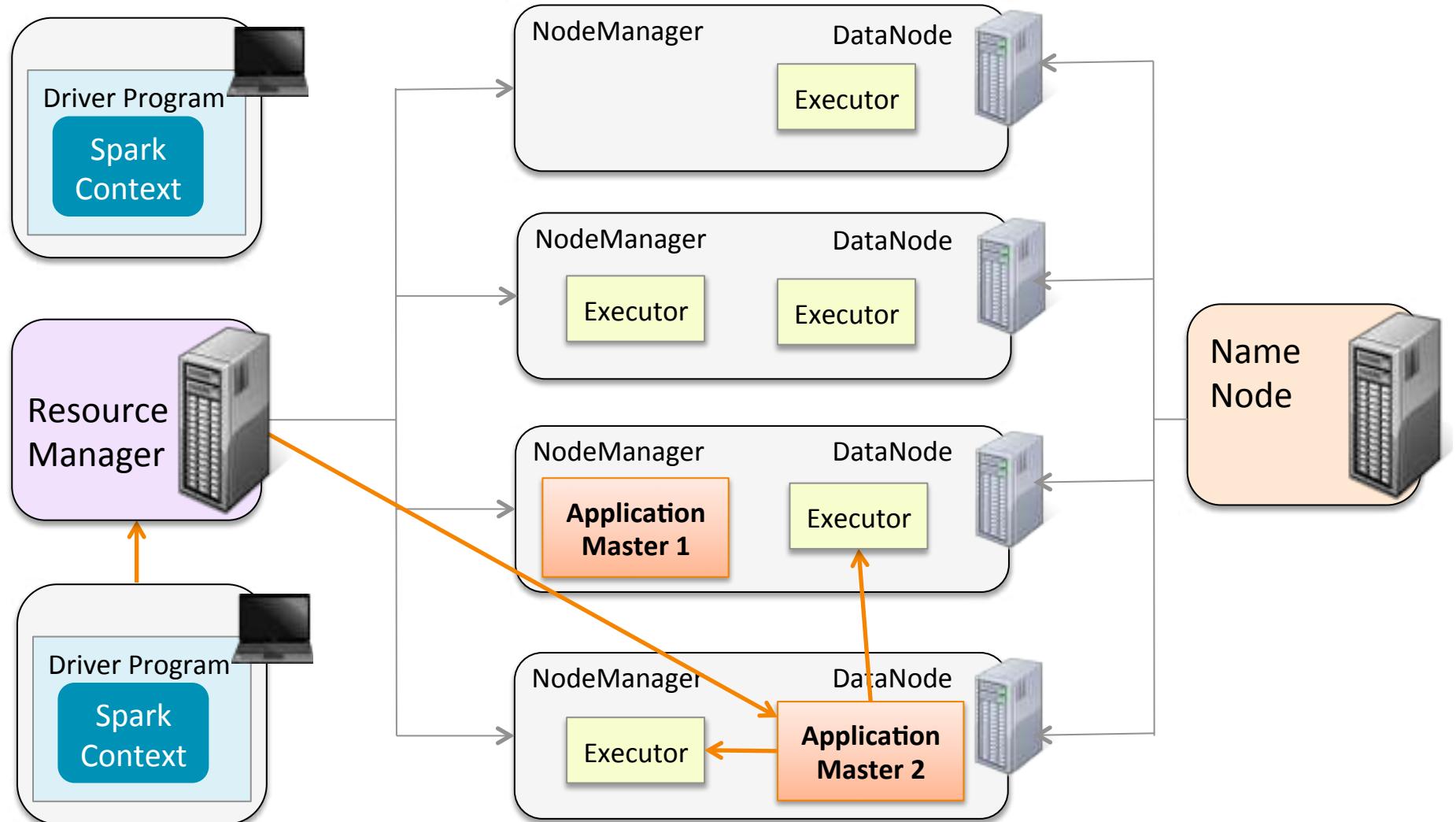
How Spark Runs on YARN: Client Mode (1)



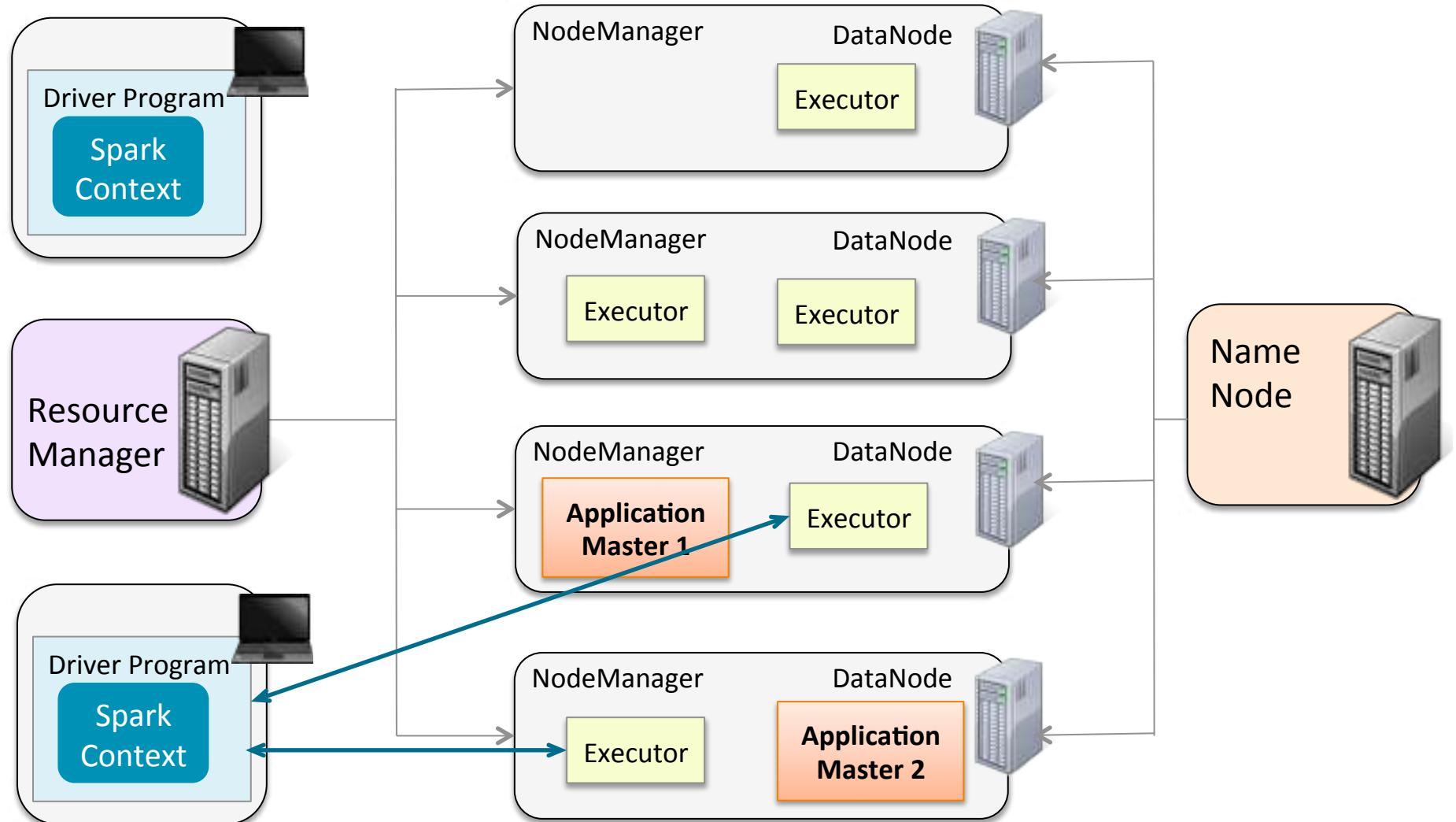
How Spark Runs on YARN: Client Mode (2)



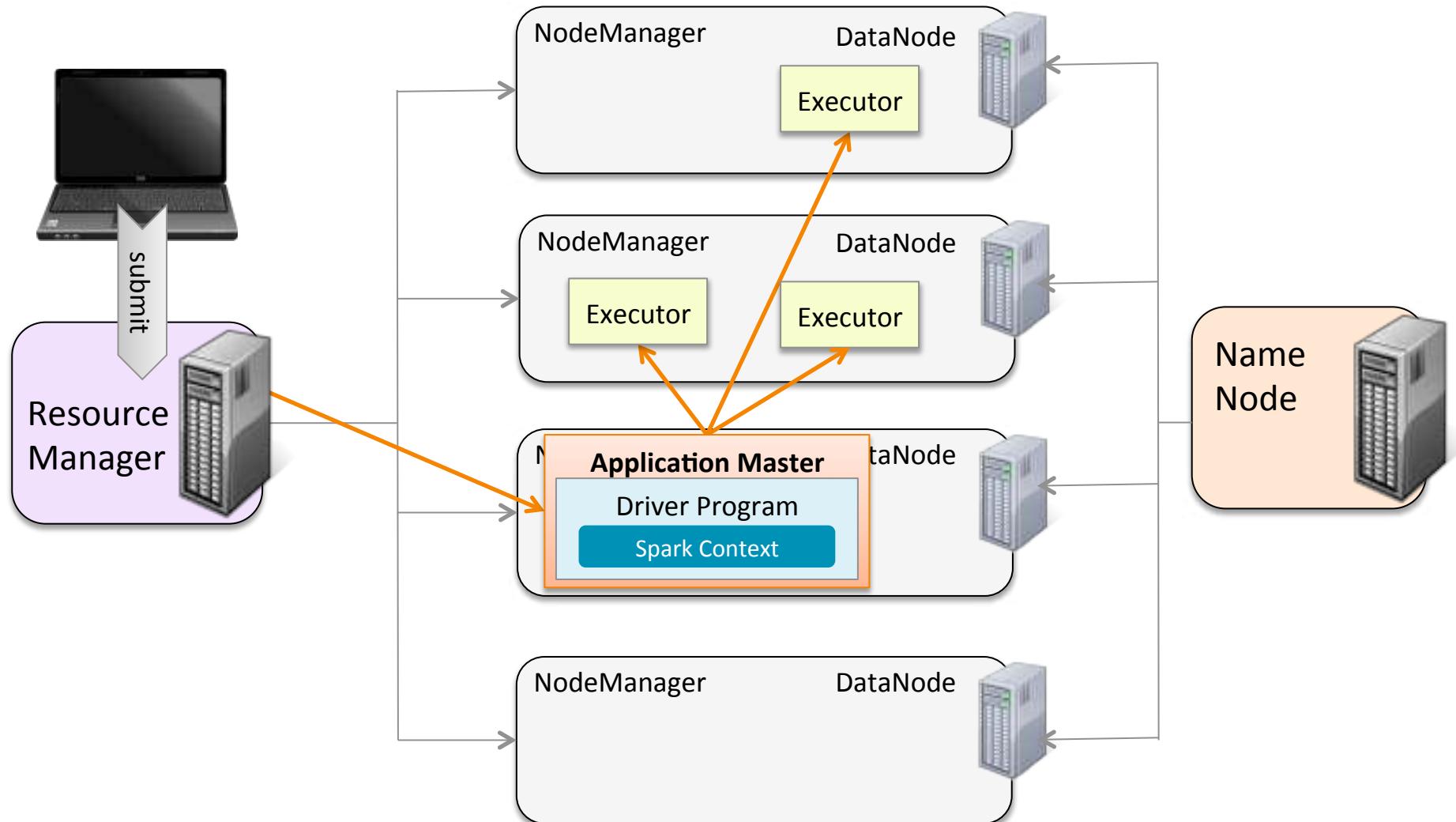
How Spark Runs on YARN: Client Mode (3)



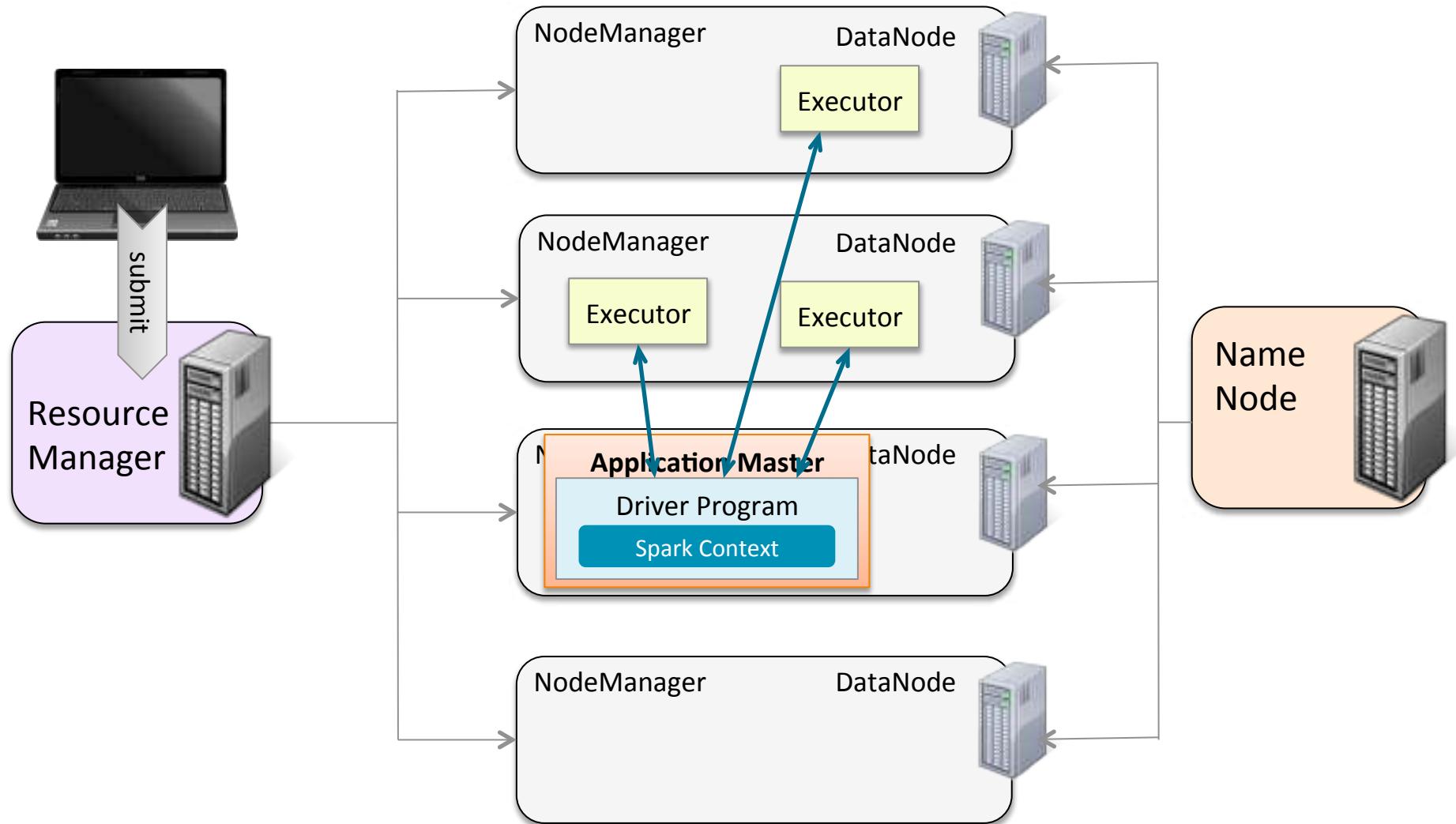
How Spark Runs on YARN: Client Mode (4)



How Spark Runs on YARN: Cluster Mode (1)



How Spark Runs on YARN: Cluster Mode (2)



Running a Spark Application Locally

- Use **spark-submit --master** to specify cluster option
 - Local options
 - **local [*]** – run locally with as many threads as cores (default)
 - **local [n]** – run locally with n threads
 - **local** – run locally with a single thread

Python

```
$ spark-submit --master local[3] \
WordCount.py fileURL
```

Scala

```
$ spark-submit --master local[3] --class \
WordCount MyJarFile.jar fileURL
```

Java

Running a Spark Application on a Cluster

- Use **spark-submit --master** to specify cluster option
 - Cluster options
 - **yarn-client**
 - **yarn-cluster**
 - **spark://masternode:port** (Spark Standalone)
 - **mesos://masternode:port** (Mesos)

Python

```
$ spark-submit --master yarn-cluster \
WordCount.py fileURL
```

Scala

```
$ spark-submit --master yarn-cluster --class \
WordCount MyJarFile.jar fileURL
```

Java

Starting the Spark Shell on a Cluster

- The Spark Shell can also be run on a cluster
- Pyspark and spark-shell both have a `--master` option
 - `yarn` (client mode only)
 - Spark or Mesos cluster manager URL
 - `local[*]` – run with as many threads as cores (default)
 - `local[n]` – run locally with n worker threads
 - `local` – run locally without distributed processing

Python

```
$ pyspark --master yarn
```

Scala

```
$ spark-shell --master yarn
```

Options when Submitting a Spark Application to a Cluster

- Some other **spark-submit** options for clusters
 - **--jars** – additional JAR files (Scala and Java only)
 - **--py-files** – additional Python files (Python only)
 - **--driver-java-options** – parameters to pass to the driver JVM
 - **--executor-memory** – memory per executor (e.g. 1000M, 2G)
(Default: 1G)
 - **--packages** -- Maven coordinates of an external library to include
- Plus several YARN-specific options
 - **--num-executors**
 - **--queue**
- Show all available options
 - **--help**

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- **The Spark Application Web UI**
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

The Spark Application Web UI

The Spark UI lets you monitor running jobs, and view statistics and configuration



The Executors (3) page displays the following information:

Memory: 0.0 B Used (684.9 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Memory Used	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs
1	localhost:38882	0	0.0 B / 208.8 MB	0.0 B	0	0	157	157	2.4 m	78.0 MB	463.0 KB	465.1 KB	stdout stderr
2	localhost:58187	0	0.0 B / 208.8 MB	0.0 B	0	0	155	155	2.3 m	78.0 MB	0.0 B	463.0 KB	stdout stderr
<driver>	192.168.234.139:37578	0	0.0 B / 267.3 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B	

Spark Jobs (?)

Total Duration: 16 s

Scheduling Mode: FIFO

Active Jobs: 1

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	runJob at PythonRDD.scala:356	2015/05/21 06:24:38	7 s	0/2	<div style="width: 25%; background-color: #0072bc; height: 10px;"></div> 36/312

Accessing the Spark UI

- The Web UI is run by the Spark drivers
 - When running locally: `http://localhost:4040`
 - When running on a cluster, access via the cluster UI, e.g. YARN UI

Cluster Metrics																	
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes		
24	0	1	23	2	2 GB	8 GB	0 B	2	8	0	1	0	0	0	0	0	
User Metrics for dr.who																	
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending	Memory Reserved	VCores Used	VCores Pending	VCores Reserved					
0	0	1	23	0	0	0	0 B	0 B	0 B	0	0	0					

Show 20	entries	Search:								
ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	progress	Tracking URL
application_1431967875241_0024	training	topArticles.py	SPARK	root.training	Thu May 21 06:30:05 -0700 2015	N/A	RUNNING	UNDEFINED		ApplicationMaster

Showing 1 to 1 of 1 entries	Last	Previous	Next	First
-----------------------------	------	----------	------	-------

Viewing Spark Job History (1)

■ Viewing Spark Job History

- Spark UI is only available while the application is running
- Use Spark History Server to view metrics for a completed application
 - Optional Spark component

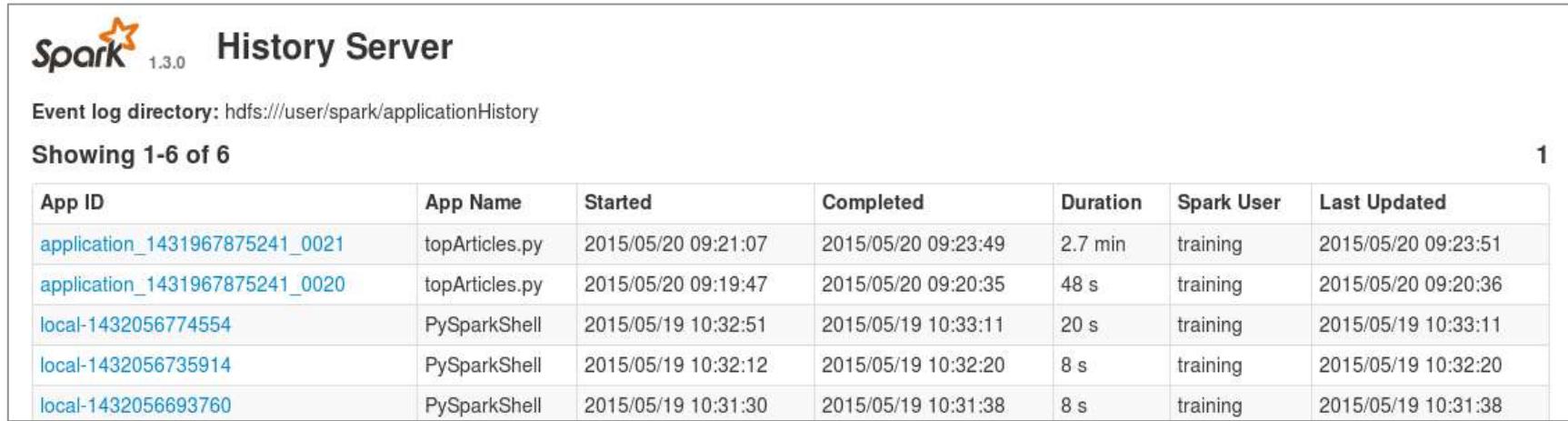
■ Accessing the History Server

- For local jobs, access by URL
 - E.g. **localhost:18080**
- For YARN Jobs, click History link in YARN UI

Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
SPARK	root.training	Thu May 21 07:02:18 -0700 2015	N/A	RUNNING	UNDEFINED	<div style="width: 0%;"></div>	ApplicationMaster
SPARK	root.training	Thu May 21 06:30:05 -0700 2015	Thu May 21 06:30:49 -0700 2015	FINISHED	SUCCEEDED	<div style="width: 100%;"></div>	History
SPARK	root.training	Thu May 21	Thu May 21	FINISHED	SUCCEEDED	<div style="width: 100%;"></div>	History

Viewing Spark Job History (2)

■ Spark History Server



The screenshot shows the Spark History Server interface. At the top left is the Spark logo with the text "1.3.0". To its right is the title "History Server". Below the title is the text "Event log directory: hdfs://user/spark/applicationHistory". Underneath that, it says "Showing 1-6 of 6". On the far right, there is a small number "1". The main area is a table with the following data:

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated
application_1431967875241_0021	topArticles.py	2015/05/20 09:21:07	2015/05/20 09:23:49	2.7 min	training	2015/05/20 09:23:51
application_1431967875241_0020	topArticles.py	2015/05/20 09:19:47	2015/05/20 09:20:35	48 s	training	2015/05/20 09:20:36
local-1432056774554	PySparkShell	2015/05/19 10:32:51	2015/05/19 10:33:11	20 s	training	2015/05/19 10:33:11
local-1432056735914	PySparkShell	2015/05/19 10:32:12	2015/05/19 10:32:20	8 s	training	2015/05/19 10:32:20
local-1432056693760	PySparkShell	2015/05/19 10:31:30	2015/05/19 10:31:38	8 s	training	2015/05/19 10:31:38

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- **Hands-On Exercise: Write and Run a Spark Application**
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Building and Running Scala Applications in the Hands-On Exercises

- Basic Maven projects are provided in the `exercises/spark/projects` directory with two packages
 - `stubs` – starter Scala file, do exercises here
 - `solution` – final exercise solution

```
$ mvn package  
  
$ spark-submit \  
  --class stubs.CountJPGs \  
  target/countjpgs-1.0.jar \  
  weblogs.*
```

Project Directory Structure

```
+countjpgs  
  -pom.xml  
  +src  
    +main  
      +scala  
        +solution  
          -CountJPGs.scala  
        +stubs  
          -CountJPGs.scala  
  +target  
  -countjpgs-1.0.jar
```

Hands-On Exercise: Write and Run a Spark Application

- **In this exercise you will**

- Write a Spark application to count JPG requests in a web server log
 - If you choose to use Scala, compile and package the application in a JAR file
 - Run the application locally to test
 - Submit the application to run on the YARN cluster

- **Please refer to the Hands-On Exercise Manual**

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- **Configuring Spark Properties**
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Spark Application Configuration

- Spark provides numerous properties for configuring your application
- Some example properties
 - `spark.master`
 - `spark.app.name`
 - `spark.local.dir` – where to store local files such as shuffle output (default `/tmp`)
 - `spark.ui.port` – port to run the Spark Application UI (default `4040`)
 - `spark.executor.memory` – how much memory to allocate to each Executor (default `512m`)
 - And many more...
 - See Spark Configuration page for more details

Spark Application Configuration

- **Spark Applications can be configured**
 - Declaratively or
 - Programmatically

Declarative Configuration Options

- **spark-submit script**
 - e.g., **spark-submit --driver-memory 500M**
- **Properties file**
 - Tab- or space-separated list of properties and values
 - Load with **spark-submit --properties-file filename**
 - Example:

```
spark.master      spark://masternode:7077
spark.local.dir  /tmp
spark.ui.port    4444
```
- **Site defaults properties file**
 - **\$SPARK_HOME/conf/spark-defaults.conf**
 - Template file provided

Setting Configuration Properties Programmatically

- Spark configuration settings are part of the `SparkContext`
- Configure using a `SparkConf` object
- Some example functions
 - `setAppName (name)`
 - `setMaster (master)`
 - `set (property-name, value)`
- `set` functions return a `SparkConf` object to support chaining

SparkConf Example (Python)

```
import sys
from pyspark import SparkContext
from pyspark import SparkConf

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print >> sys.stderr, "Usage: WordCount <file>"
        exit(-1)

    sconf = SparkConf() \
        .setAppName("Word Count") \
        .set("spark.ui.port","4141")
    sc = SparkContext(conf=sconf)

    counts = sc.textFile(sys.argv[1]) \
        .flatMap(lambda line: line.split()) \
        .map(lambda w: (w,1)) \
        .reduceByKey(lambda v1,v2: v1+v2)

    for pair in counts.take(5): print pair
```

SparkConf Example (Scala)

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
import org.apache.spark.SparkConf

object WordCount {
    def main(args: Array[String]) {
        if (args.length < 1) {
            System.err.println("Usage: WordCount <file>")
            System.exit(1)
        }

        val sconf = new SparkConf()
            .setAppName("Word Count")
            .set("spark.ui.port", "4141")
        val sc = new SparkContext(sconf)

        val counts = sc.textFile(args(0)) .
            flatMap(line => line.split("\\W")) .
            map(word => (word,1)) .
            reduceByKey(_ + _)
        counts.take(5).foreach(println)
    }
}
```

Viewing Spark Properties

- You can view the Spark property setting in the Spark Application UI

The screenshot shows the Spark Application UI interface. At the top, there are tabs for Stages, Storage, Environment (which is highlighted with a red box), and Executors. Below the tabs, the title "Environment" is displayed, followed by "Runtime Information". A table lists the following runtime information:

Name	Value
Java Home	/usr/java/jdk1.7.0_51/jre
Java Version	1.7.0_51 (Oracle Corporation)
Scala Home	
Scala Version	version 2.10.3

Below this is a section titled "Spark Properties" which contains another table:

Name	Value
spark.app.name	PySparkShell
spark.driver.host	master
spark.driver.port	33121
spark.filesServer.uri	http://master:34670
spark.httpBroadcast.uri	http://master:38591
spark.master	spark://master:7077

Chapter Topics

Writing Spark Applications

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- **Logging**
- Conclusion
- Hands-On Exercise: Configure a Spark Application

Spark Logging

- **Spark uses Apache Log4j for logging**
 - Allows for controlling logging at runtime using a properties file
 - Enable or disable logging, set logging levels, select output destination
 - For more info see <http://logging.apache.org/log4j/1.2/>
- **Log4j provides several logging levels**
 - Fatal
 - Error
 - Warn
 - Info
 - Debug
 - Trace
 - Off

Spark Log Files (1)

- Log file locations depend on your cluster management platform
- YARN
 - If log aggregation off, logs are stored locally on each worker node
 - If log aggregation is on, logs are stored in HDFS
 - Default `/var/log/hadoop-yarn`
 - Access via `yarn logs` command or YARN RM UI

```
$ yarn application -list

Application-Id          Application-Name Application-Type...
application_1441395433148_0003  Spark shell      SPARK    ...
application_1441395433148_0001  myapp.jar       MAPREDUCE  ...

$ yarn logs -applicationId <appid>
...
```

Spark Log Files (2)

The screenshot shows the Hadoop Cluster Overview page. At the top right, it says "Logged in as: dr.who". The left sidebar has a "Cluster" section with links for About, Nodes, Applications (with sub-links for NEW, NEW_SAVING, SUBMITTED, ACCEPTED, RUNNING, FINISHED, FAILED, KILLED), and Scheduler. Below that is a "Tools" section. The main content area has two tabs: "Application Overview" and "Application Metrics".

Application Overview:

User:	training
Name:	Spark shell
Application Type:	SPARK
Application Tags:	
State:	RUNNING
FinalStatus:	UNDEFINED
Started:	Mon Mar 09 08:29:45 -0700 2015
Elapsed:	3mins, 46sec
Tracking URL:	ApplicationMaster
Diagnostics:	

Application Metrics:

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	1095144 MB-seconds, 645 vcore-seconds

ApplicationMaster:

Attempt Number	Start Time	Node	Logs
1	Mon Mar 09 08:29:46 -0700 2015	localhost:8042	logs

Configuring Spark Logging (1)

- Logging levels can be set for the cluster, for individual applications, or even for specific components or subsystems
- Default for machine: `$SPARK_HOME/conf/log4j.properties`
 - Start by copying `log4j.properties.template`

`log4j.properties.template`

```
# Set everything to be logged to the console
log4j.rootCategory=INFO, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
...
```

Configuring Spark Logging (2)

- Spark will use the first `log4j.properties` file it finds in the Java classpath
- Spark Shell will read `log4j.properties` from the current directory
 - Copy `log4j.properties` to the working directory and edit

...my-working-directory/log4j.properties

```
# Set everything to be logged to the console
log4j.rootCategory=DEBUG, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
...
```

Chapter Topics

Writing Spark Applications

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- **Conclusion**
- Hands-On Exercise: Configure a Spark Application

Essential Points (1)

- Use the Spark Shell application for interactive data exploration
- Write a Spark application to run independently
- Spark applications require a Spark Context object
- Spark applications are run using the `spark-submit` script
- Spark configuration parameters can be set at runtime using the `spark-submit` script or programmatically using a `SparkConf` object
- Spark uses log4j for logging
 - Configure using a `log4j.properties` file

Essential Points (2)

- **Spark is designed to run on a cluster**
 - Spark includes a basic cluster management platform called Spark Standalone
 - Can also run on Hadoop YARN and Mesos
- **The master distributes tasks to individual workers in the cluster**
 - Tasks run in *executors* – JVMs running on worker nodes
- **Spark clusters work closely with HDFS**
 - Tasks are assigned to workers where the data is physically stored when possible

Chapter Topics

Writing and Deploying a Spark Application

Distributed Data Processing with Spark

- Spark Applications vs. Spark Shell
- Creating the SparkContext
- Building a Spark Application (Scala and Java)
- Running a Spark Application
- The Spark Application Web UI
- Hands-On Exercise: Write and Run a Spark Application
- Configuring Spark Properties
- Logging
- Conclusion
- Hands-On Exercise: Configure a Spark Application**

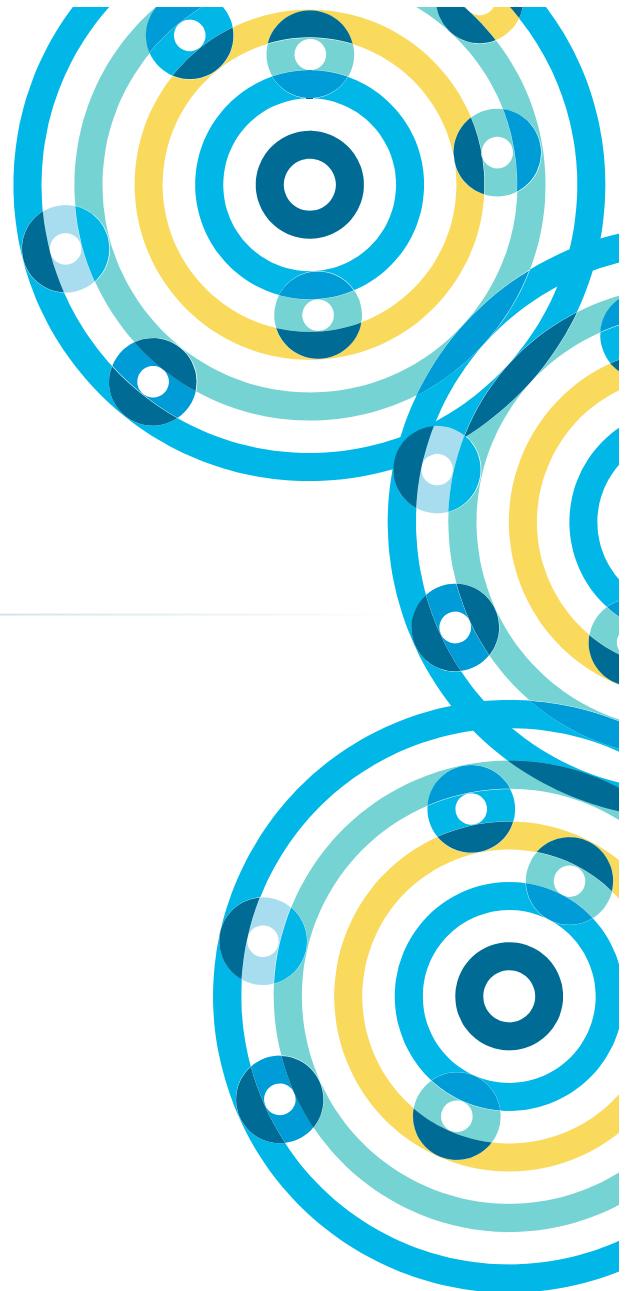
Hands-On Exercise: Configure Spark Applications

- **In this exercise you will**
 - Set properties using **spark-submit**
 - Set properties in a properties file
 - Change the logging levels in a **log4j.properties** file
- **Please refer to the Hands-On Exercise Manual**



Parallel Processing in Spark

Chapter 14



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- **Parallel Processing in Spark**
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

Parallel Programming with Spark

In this chapter you will learn

- **How RDDs are distributed across a cluster**
- **How Spark executes RDD operations in parallel**

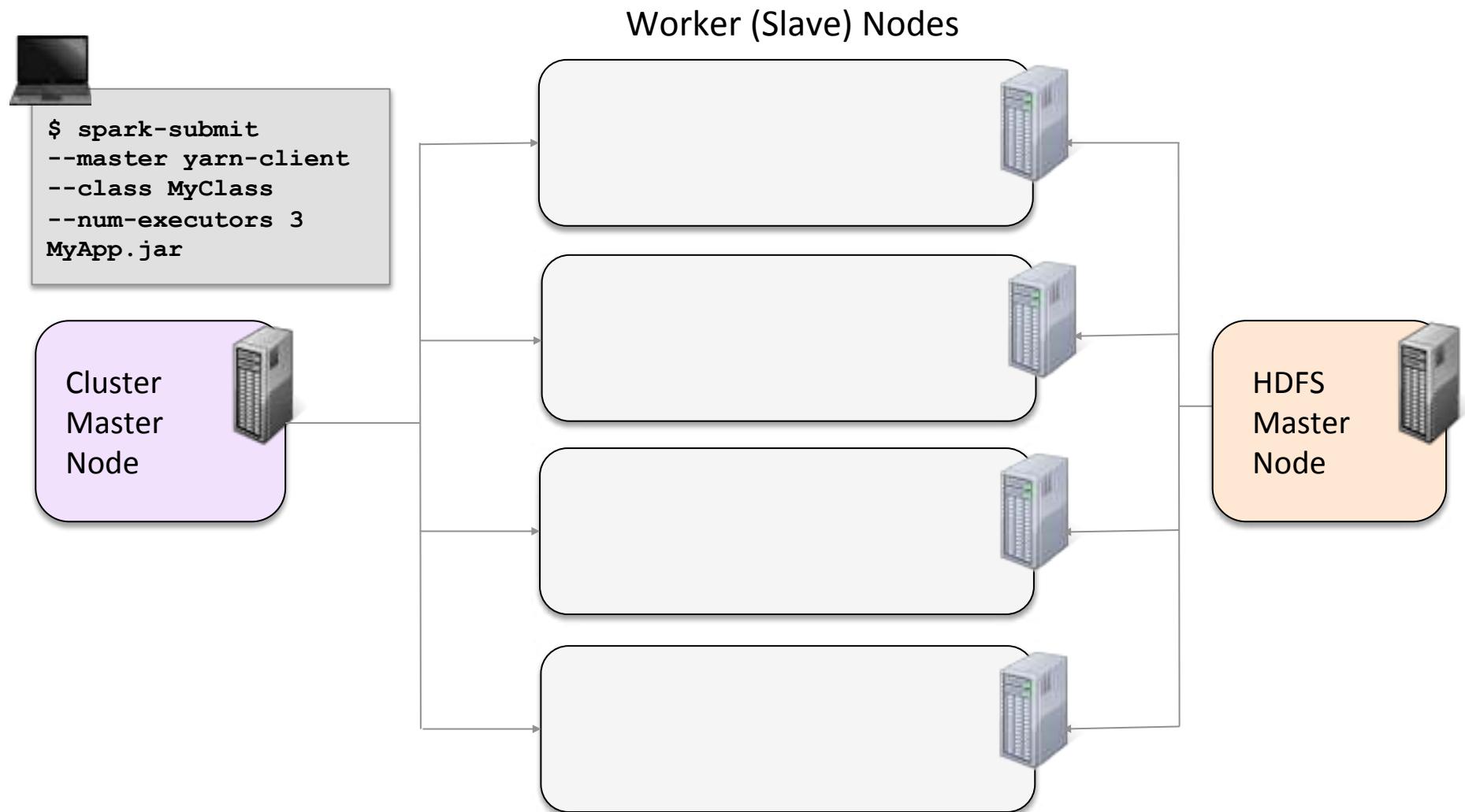
Chapter Topics

Parallel Processing in Spark

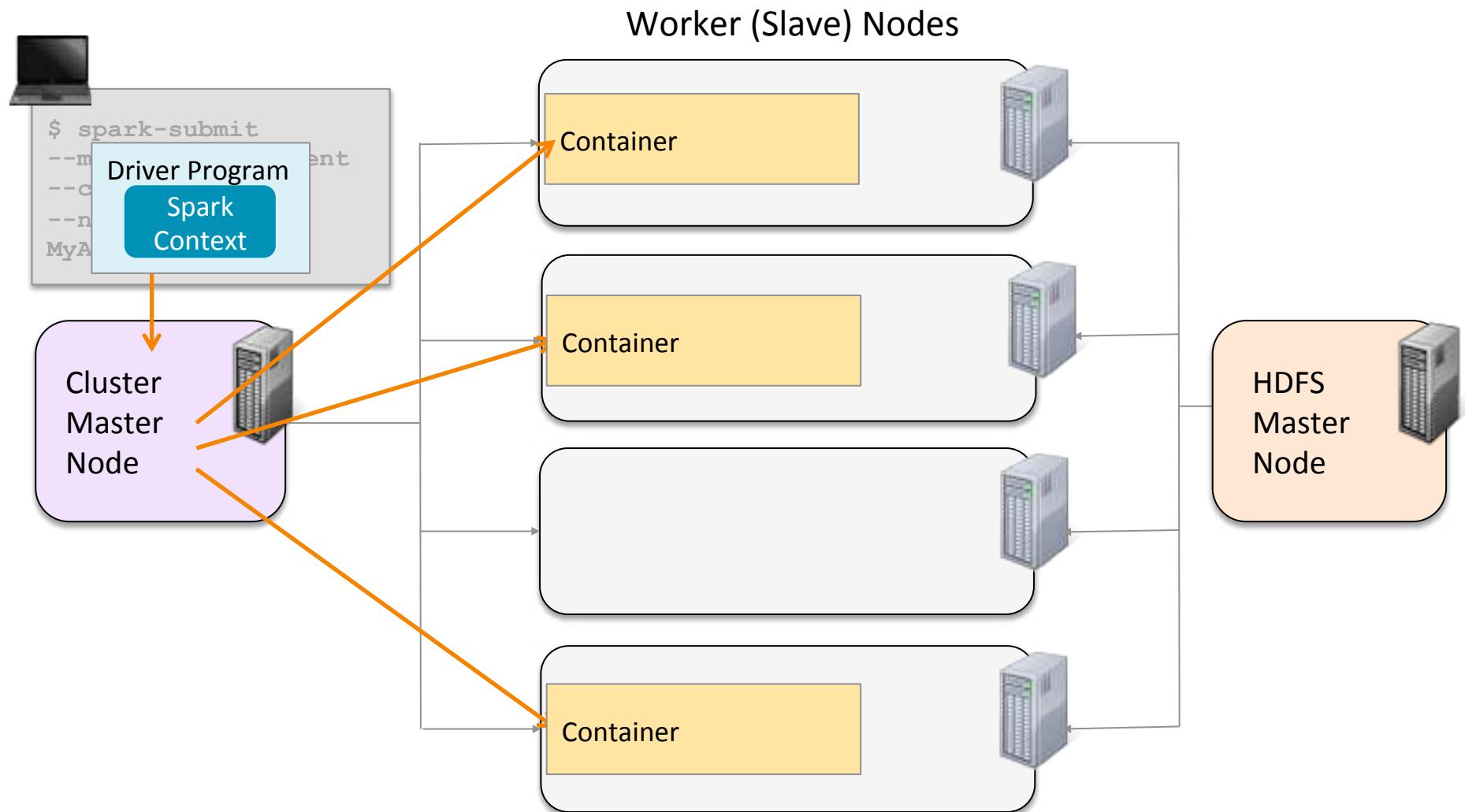
Distributed Data Processing with Spark

- **Review: Spark on a Cluster**
 - RDD Partitions
 - Partitioning of File-based RDDs
 - HDFS and Data Locality
 - Executing Parallel Operations
 - Stages and Tasks
 - Conclusion
 - Hands-On Exercise: View Jobs and Stages in the Spark Application UI

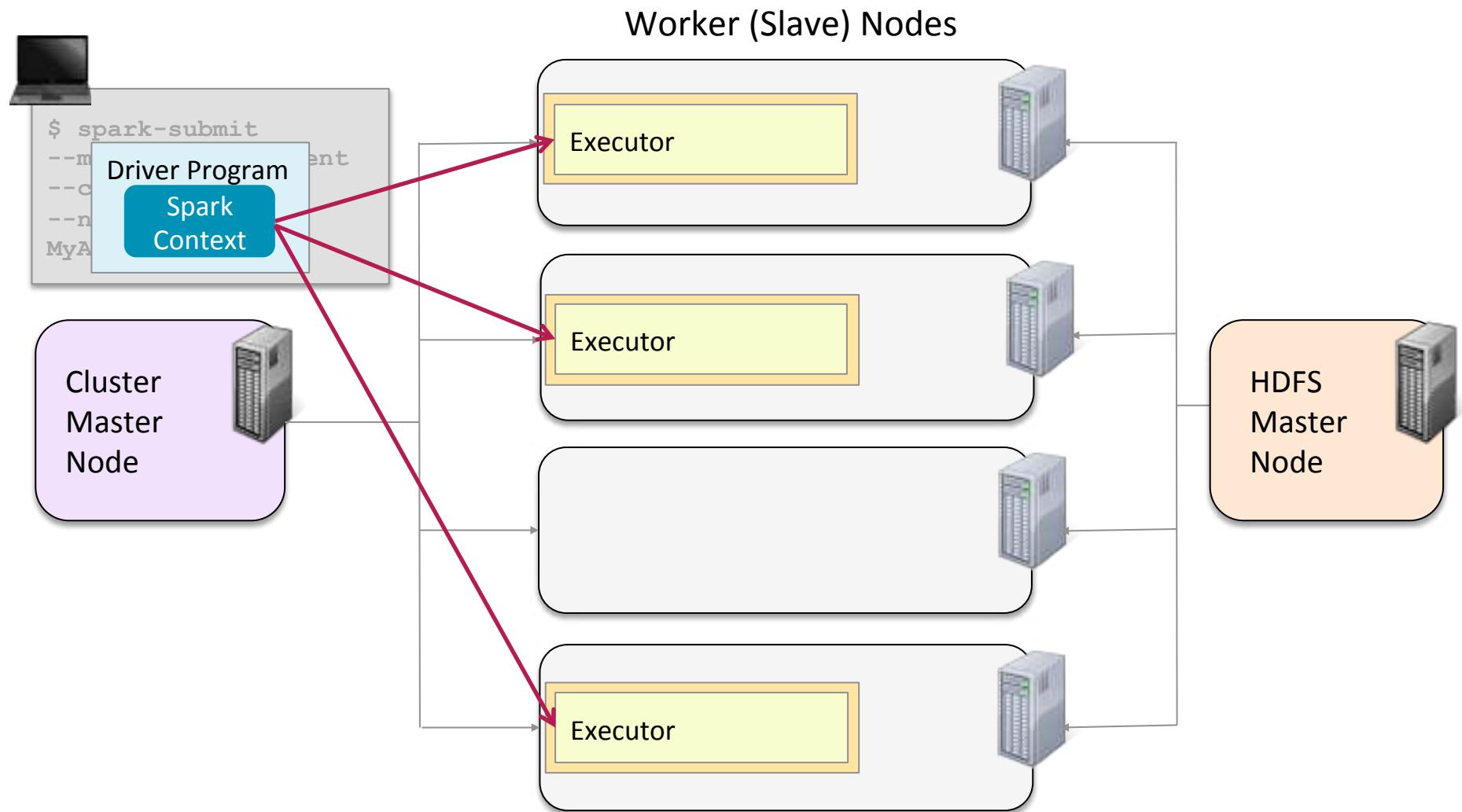
Spark Cluster Review



Spark Cluster Review



Spark Cluster Review



Chapter Topics

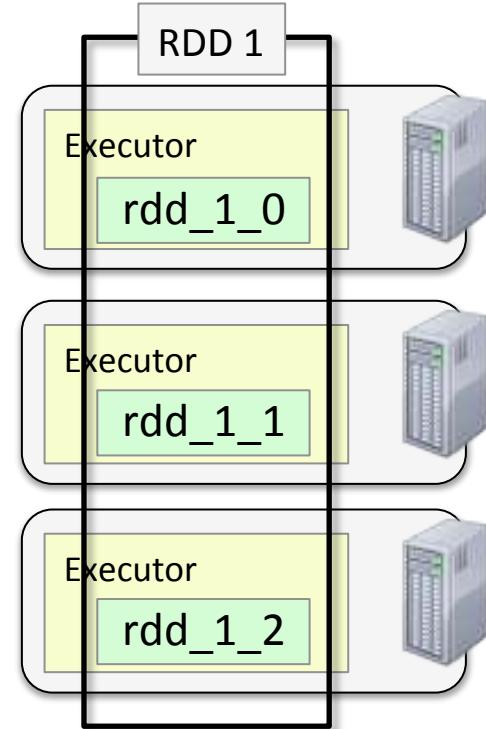
Parallel Processing in Spark

Distributed Data Processing with Spark

- Review: Spark on a Cluster
- **RDD Partitions**
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

RDDs on a Cluster

- **Resilient *Distributed* Datasets**
 - Data is *partitioned* across worker nodes
- **Partitioning is done automatically by Spark**
 - Optionally, you can control how many partitions are created



Chapter Topics

Parallel Processing in Spark

Distributed Data Processing with Spark

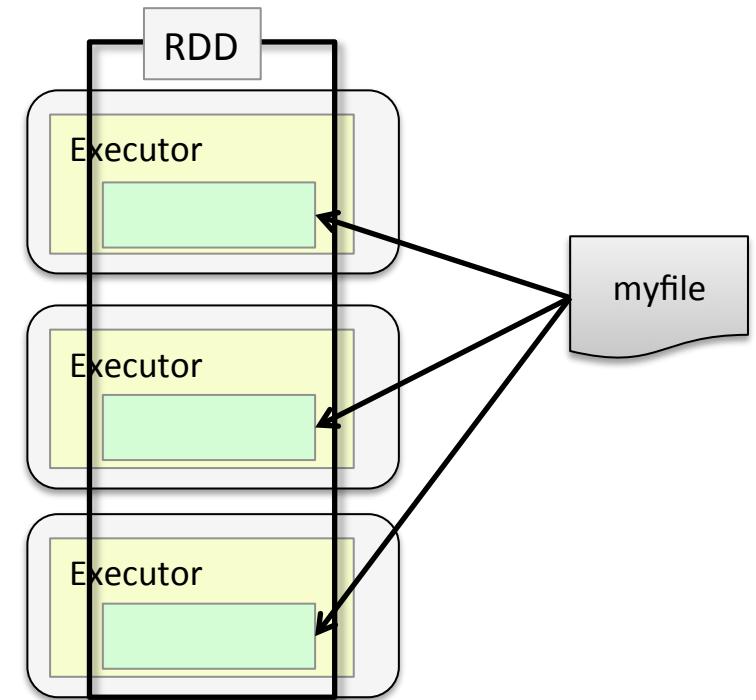
- Review: Spark on a Cluster
- RDD Partitions
- **Partitioning of File-based RDDs**
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

File Partitioning: Single Files

- **Partitions from single files**

- Partitions based on size
 - You can optionally specify a minimum number of partitions
 - `textFile(file, minPartitions)`
 - Default is 2
 - More partitions = more parallelization

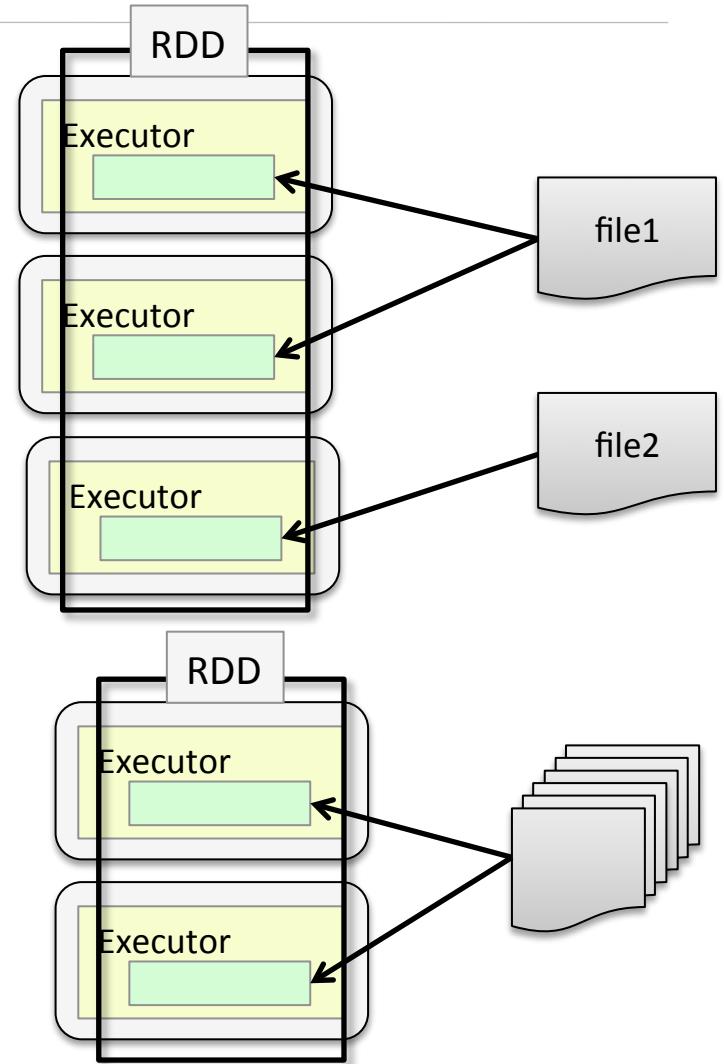
```
sc.textFile("myfile", 3)
```



File Partitioning: Multiple Files

- **sc.textFile("mydir/*")**
 - Each file becomes (at least) one partition
 - File-based operations can be done per-partition, for example parsing XML

- **sc.wholeTextFiles ("mydir")**
 - For many small files
 - Creates a key-value PairRDD
 - **key = file name**
 - **value = file contents**



Operating on Partitions

- Most RDD operations work on each *element* of an RDD
- A few work on each *partition*
 - `foreachPartition` – call a function for each partition
 - `mapPartitions` – create a new RDD by executing a function on each partition in the current RDD
 - `mapPartitionsWithIndex` – same as `mapPartitions` but includes index of the partition
- Functions for partition operations take iterators

Chapter Topics

Parallel Processing in Spark

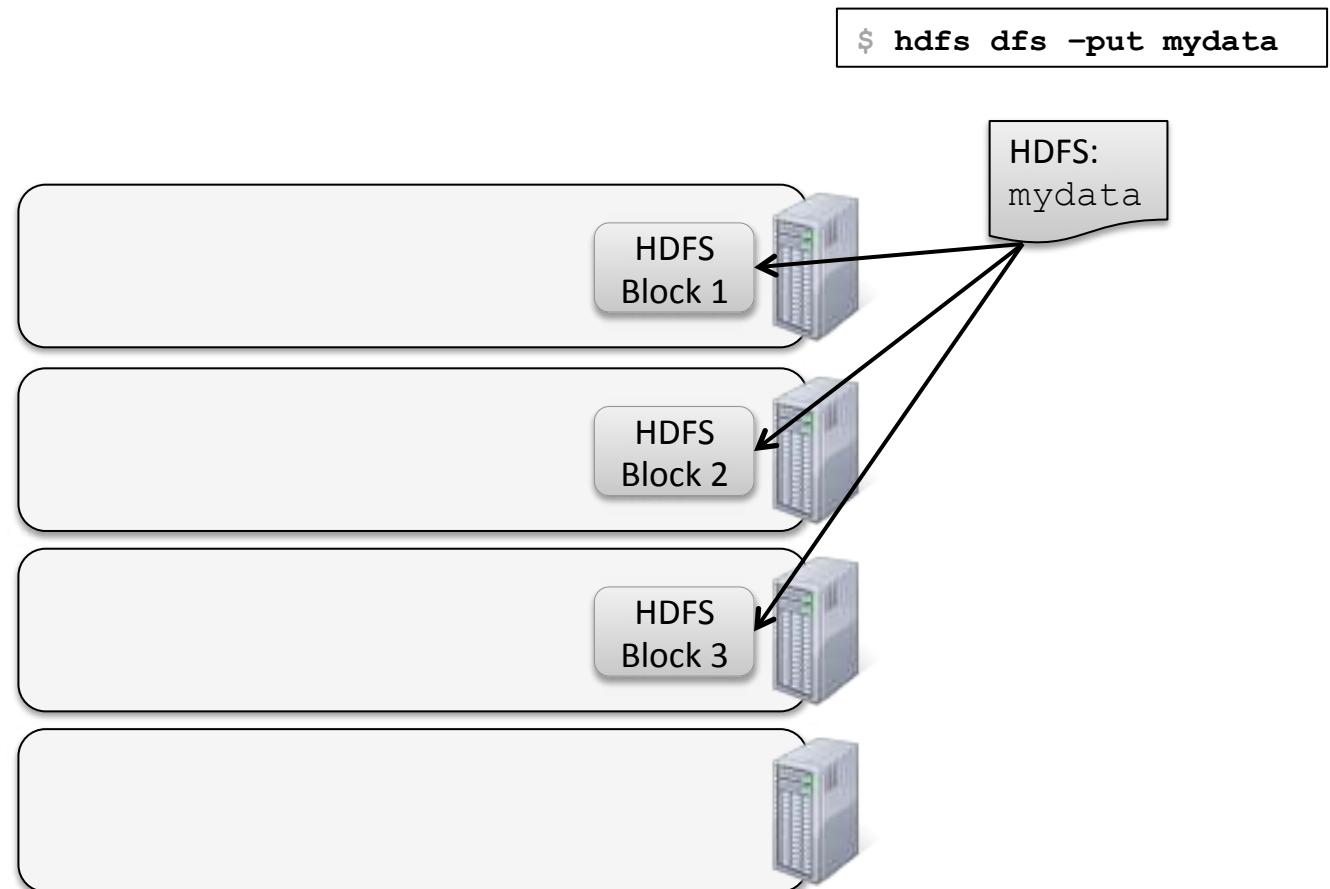
Distributed Data Processing with Spark

- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- **HDFS and Data Locality**
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

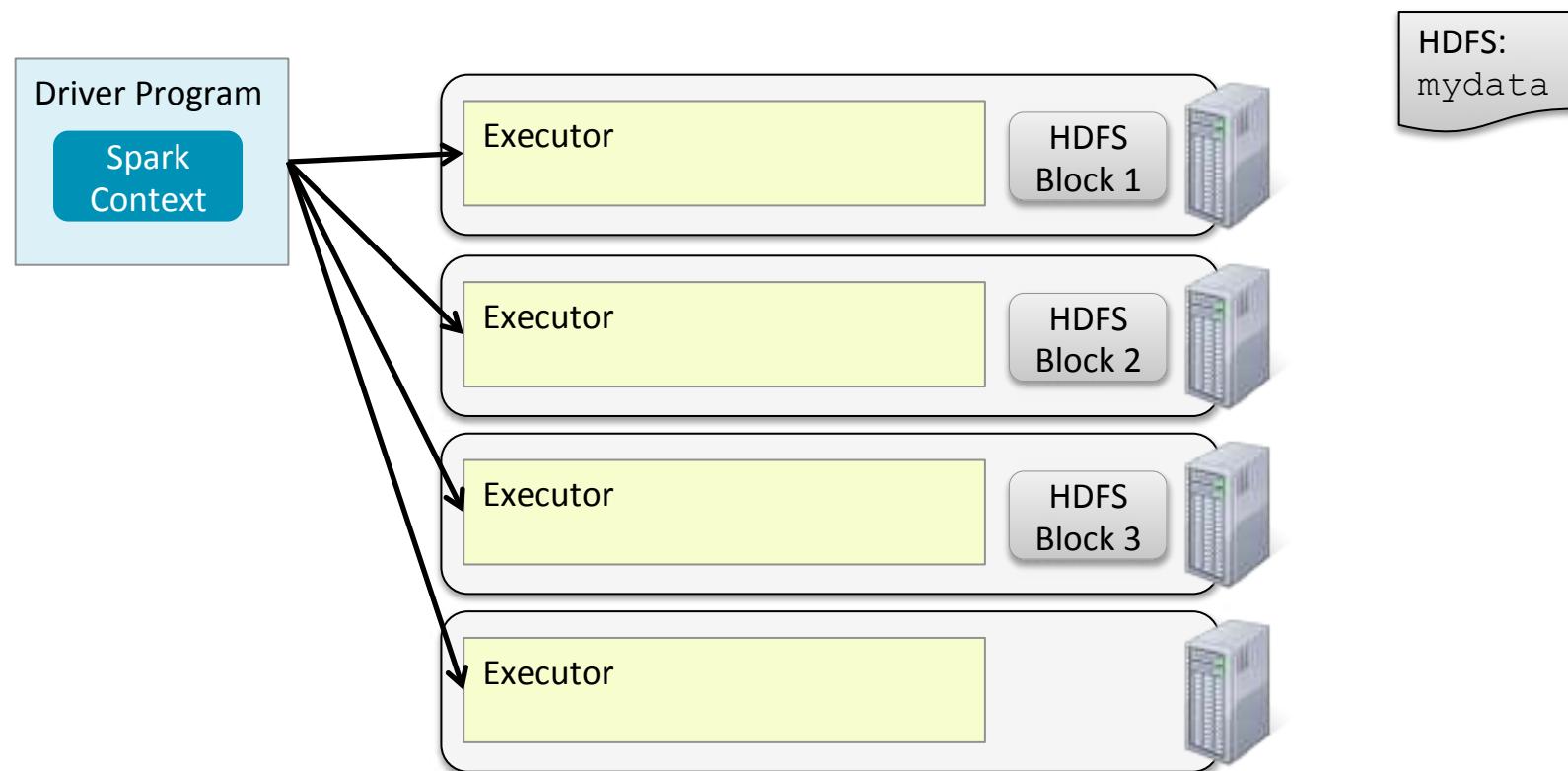
HDFS and Data Locality (1)



HDFS and Data Locality (2)



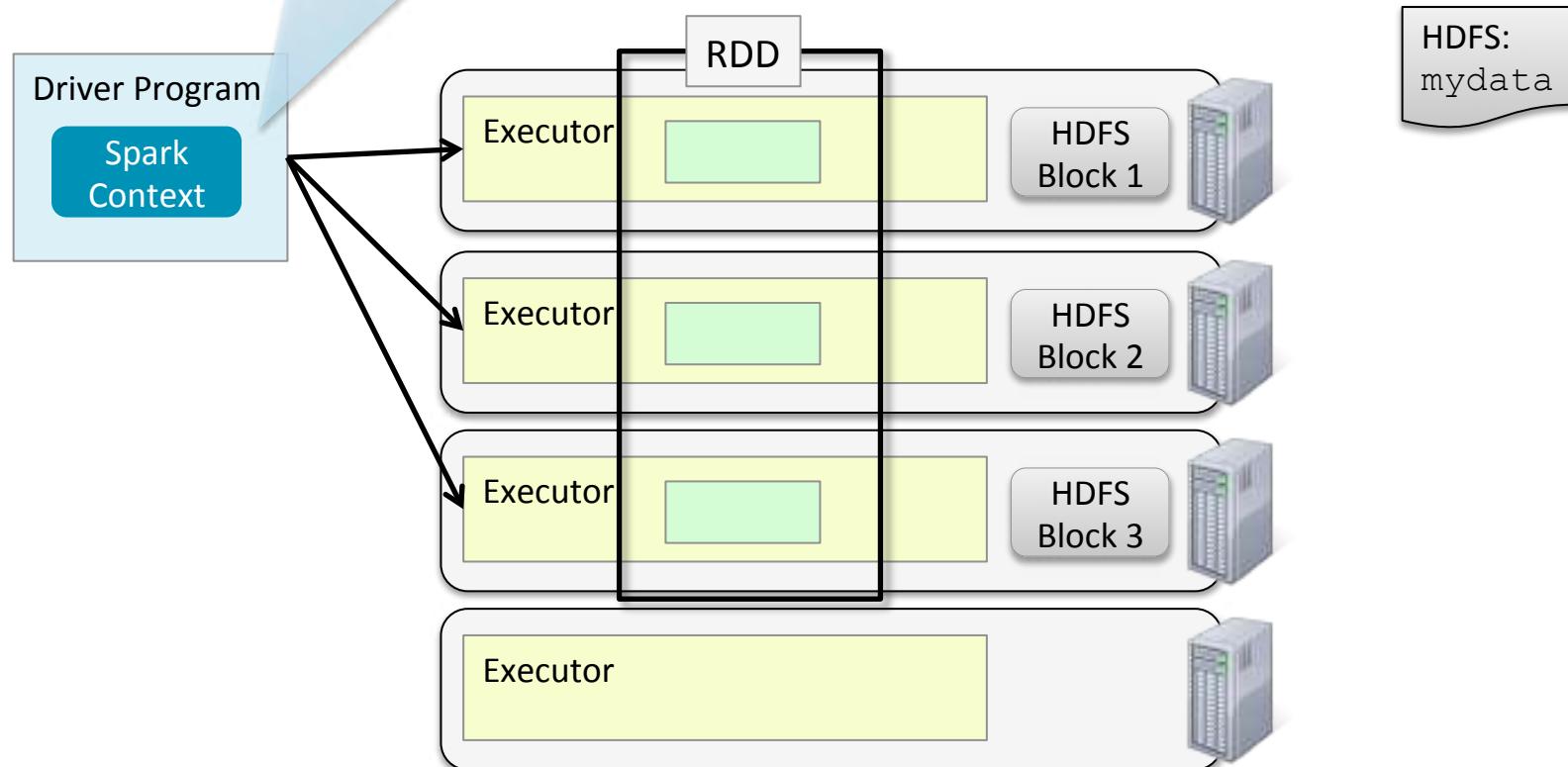
HDFS and Data Locality (3)



HDFS and Data Locality (4)

```
sc.textFile("hdfs://...mydata").collect()
```

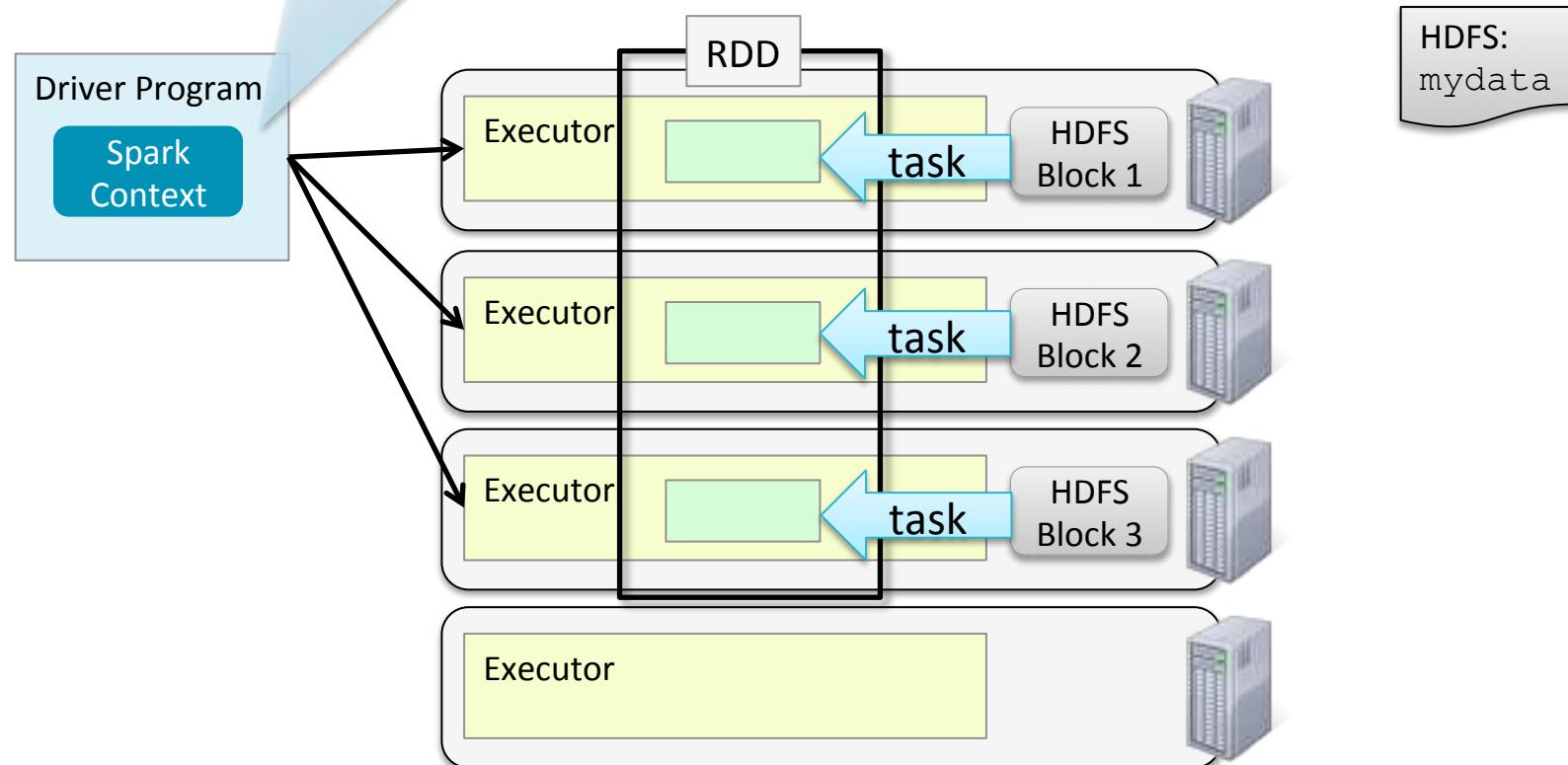
By default, Spark partitions file-based RDDs by block. Each block loads into a single partition.



HDFS and Data Locality (5)

```
sc.textFile("hdfs://...mydata").collect()
```

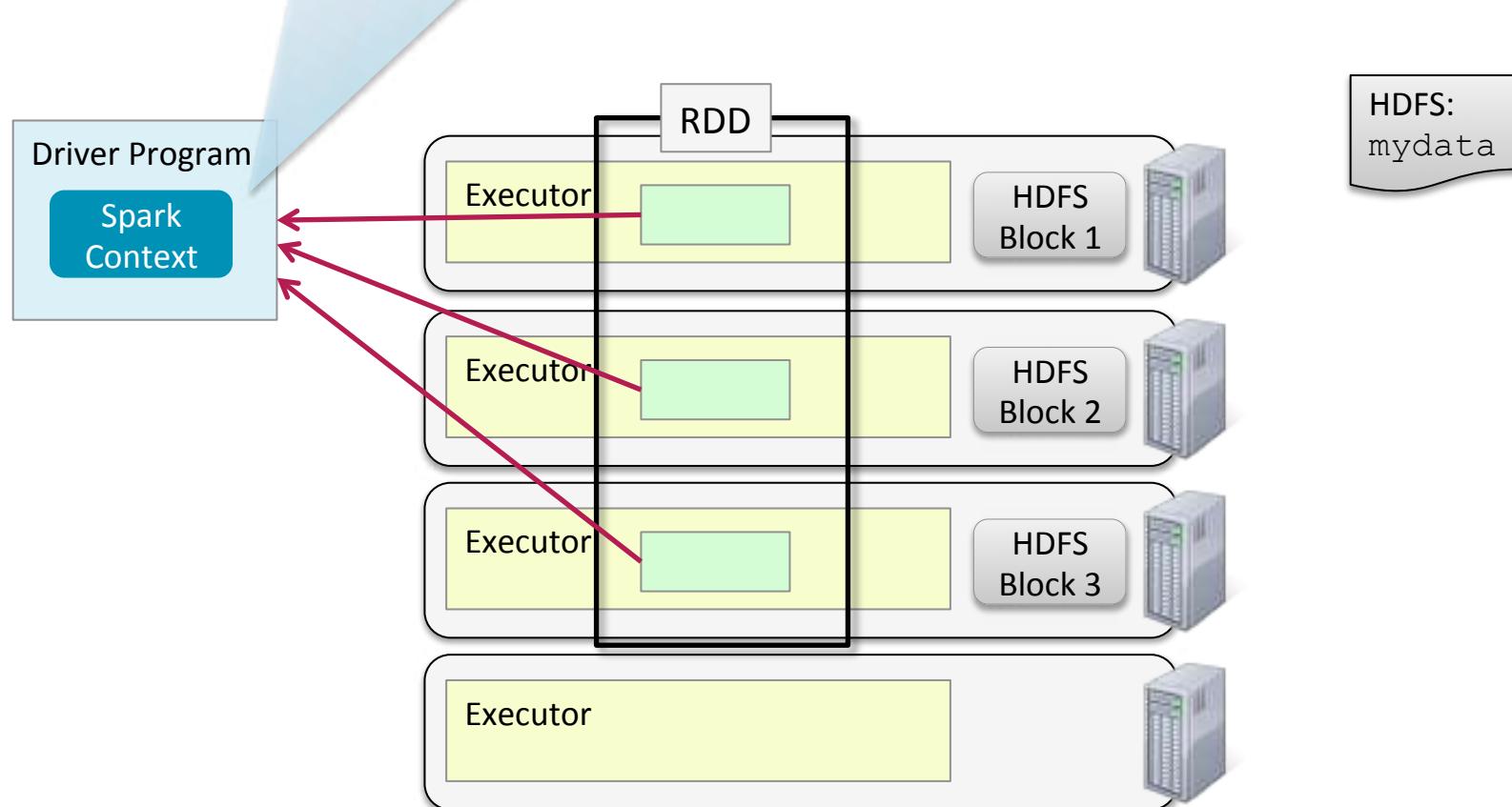
An action triggers execution: tasks on executors load data from blocks into partitions



HDFS and Data Locality (6)

```
sc.textFile("hdfs://...mydata").collect()
```

Data is distributed across executors until an action returns a value to the driver



Chapter Topics

Parallel Processing in Spark

Distributed Data Processing with Spark

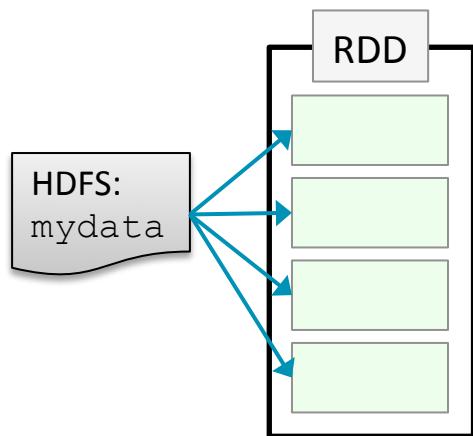
- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- **Executing Parallel Operations**
- Stages and Tasks
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

Parallel Operations on Partitions

- **RDD operations are executed in parallel on each partition**
 - When possible, tasks execute on the worker nodes where the data is in memory
- **Some operations preserve partitioning**
 - e.g., `map`, `flatMap`, `filter`
- **Some operations repartition**
 - e.g., `reduce`, `sort`, `group`

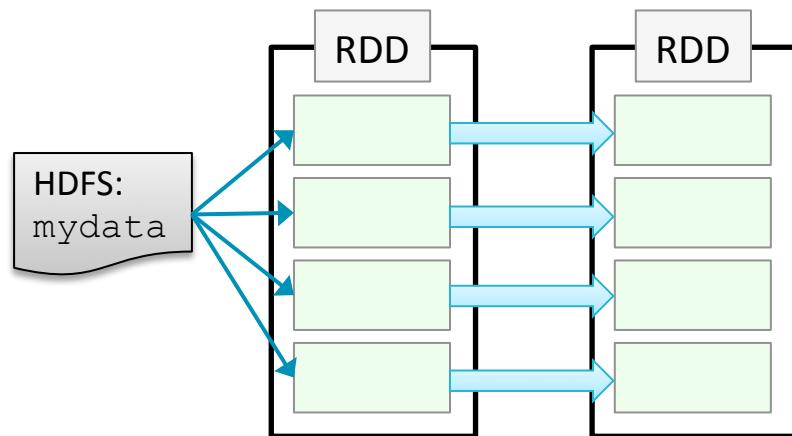
Example: Average Word Length by Letter (1)

```
> avgLens = sc.textFile(file)
```



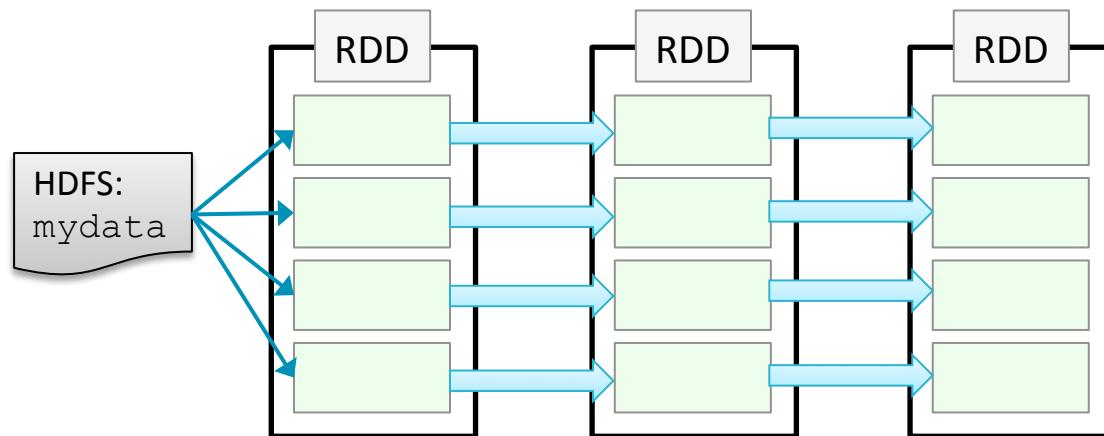
Example: Average Word Length by Letter (2)

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split())
```



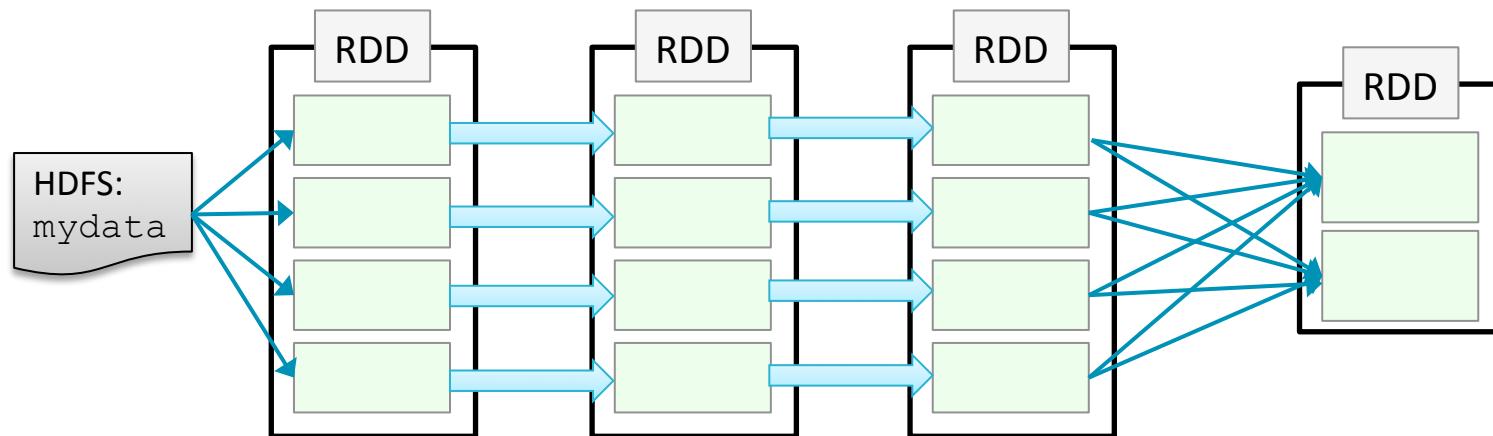
Example: Average Word Length by Letter (3)

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0],len(word)))
```



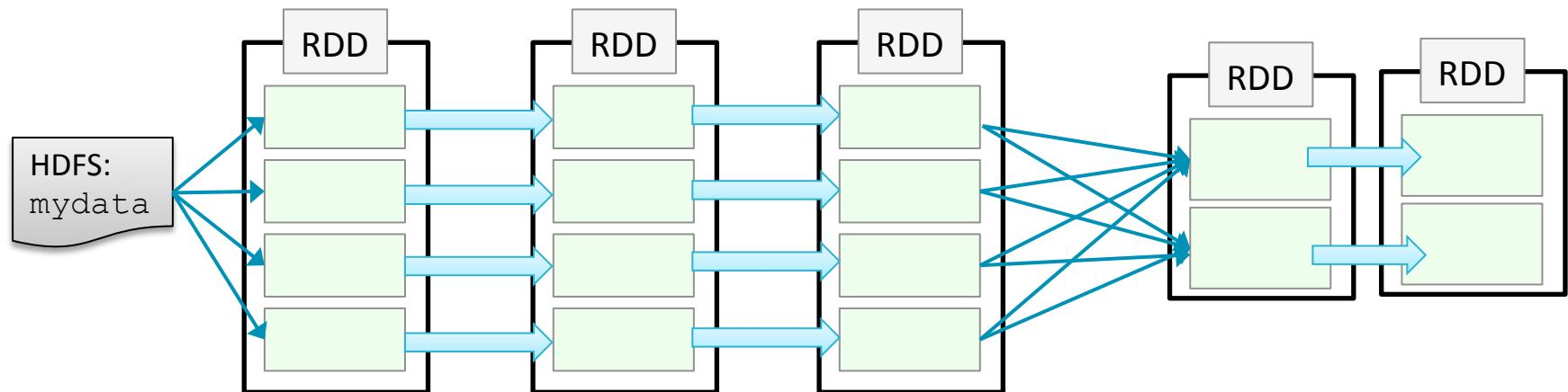
Example: Average Word Length by Letter (4)

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0],len(word))) \
    .groupByKey()
```



Example: Average Word Length by Letter (5)

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0],len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
        (k, sum(values)/len(values)))
```



Chapter Topics

Parallel Processing in Spark

Distributed Data Processing with Spark

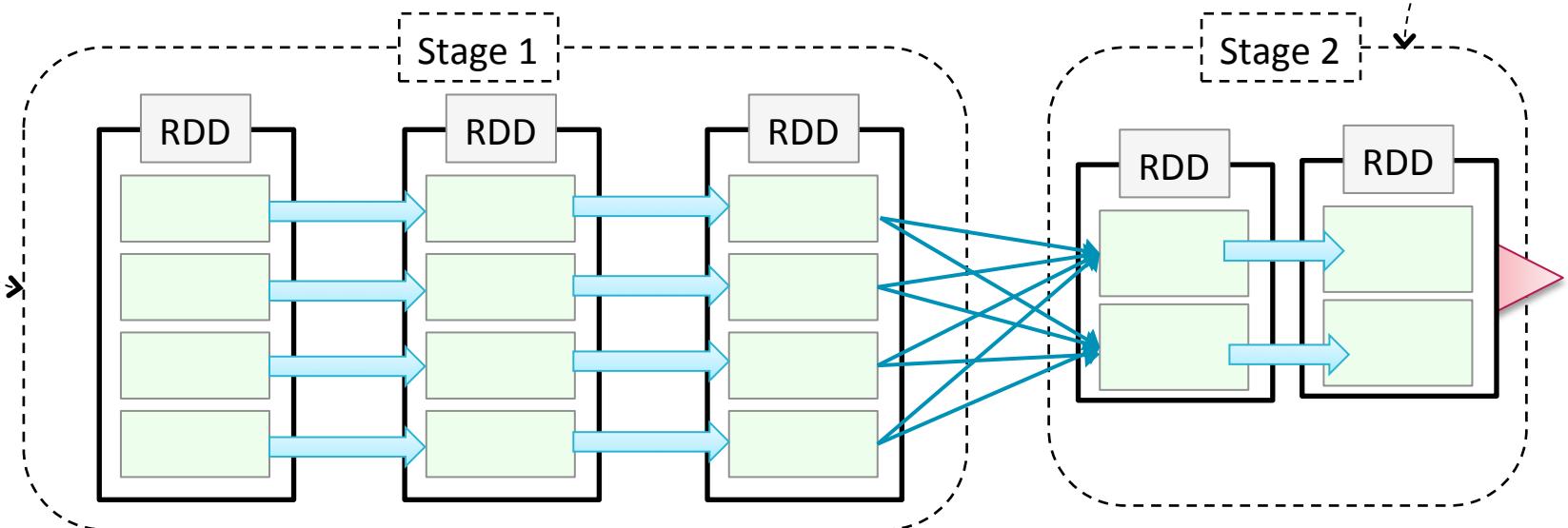
- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- **Stages and Tasks**
- Conclusion
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

Stages

- Operations that can run on the same partition are executed in *stages*
- Tasks within a stage are pipelined together
- Developers should be aware of stages to improve performance

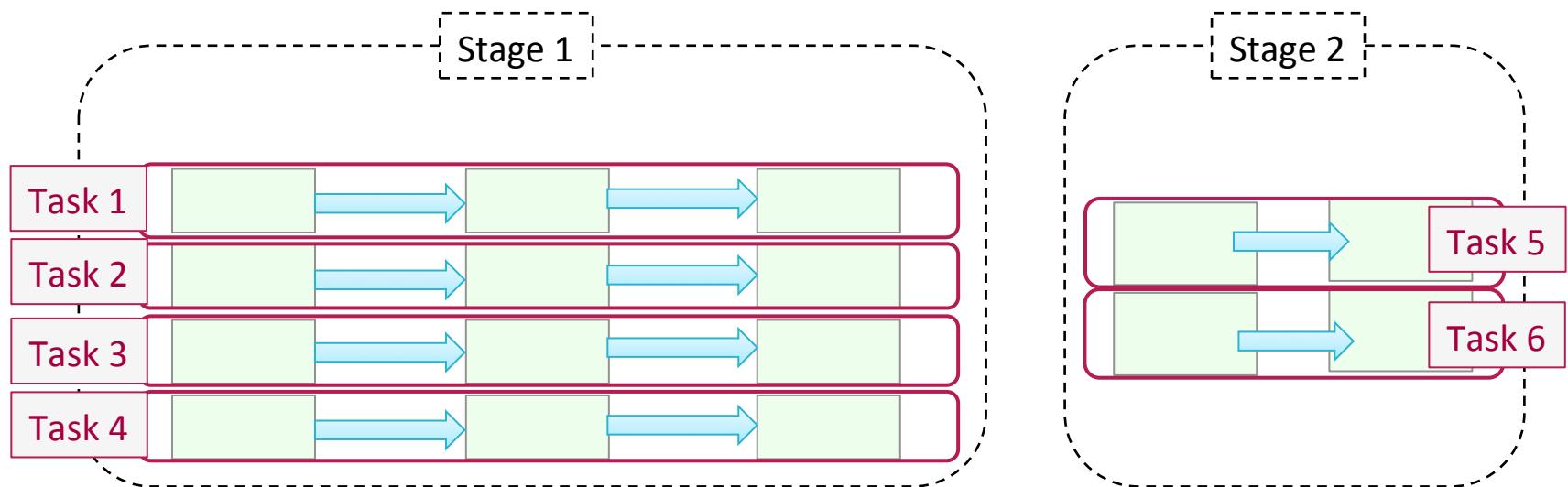
Spark Execution: Stages (1)

```
> val avglen = sc.textFile("myfile") .  
    flatMap(line => line.split("\\W")) .  
    map(word => (word(0), word.length)) .  
    groupByKey() .  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```



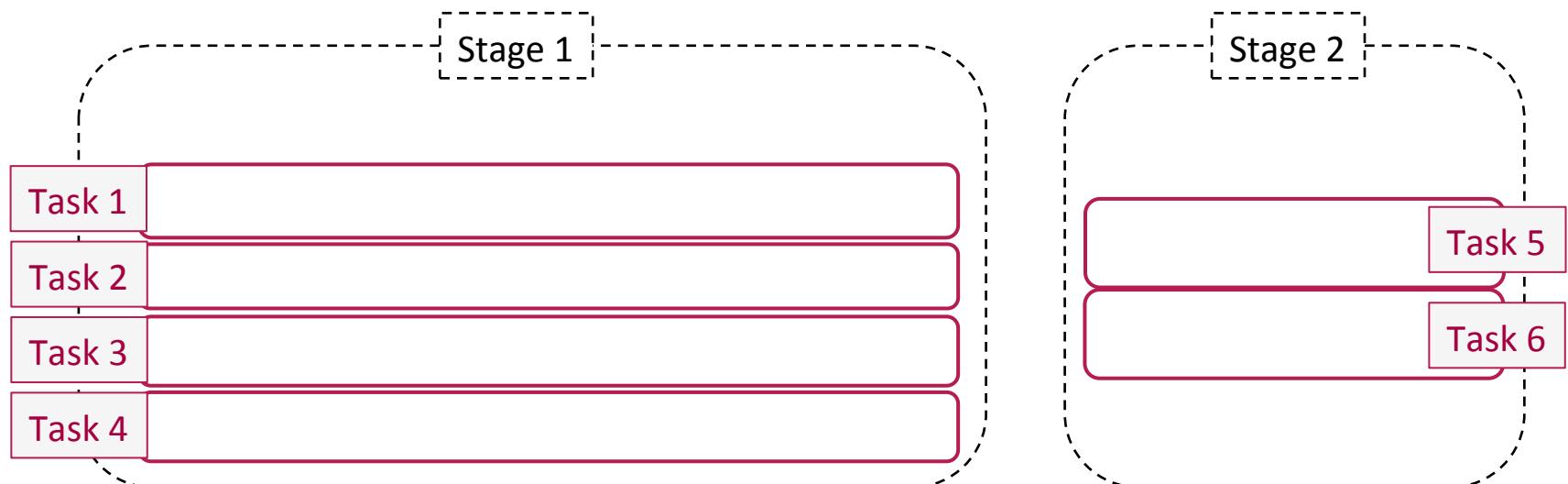
Spark Execution: Stages (2)

```
> val avglen = sc.textFile(myfile) .  
    flatMap(line => line.split("\\W")) .  
    map(word => (word(0), word.length)) .  
    groupByKey() .  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```



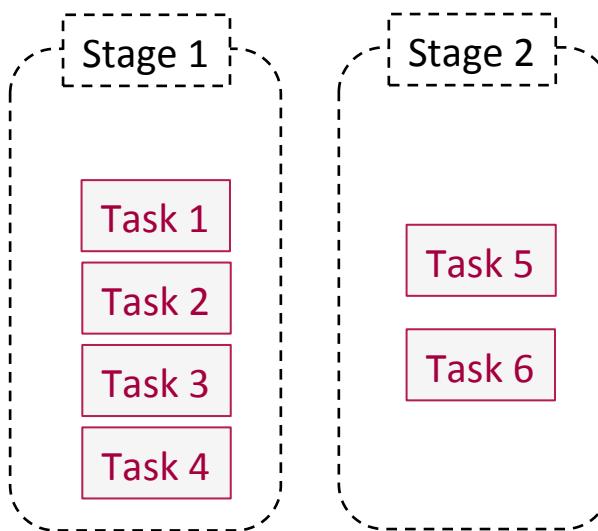
Spark Execution: Stages (3)

```
> val avglen = sc.textFile(myfile) .  
    flatMap(line => line.split("\\W")) .  
    map(word => (word(0), word.length)) .  
    groupByKey() .  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```



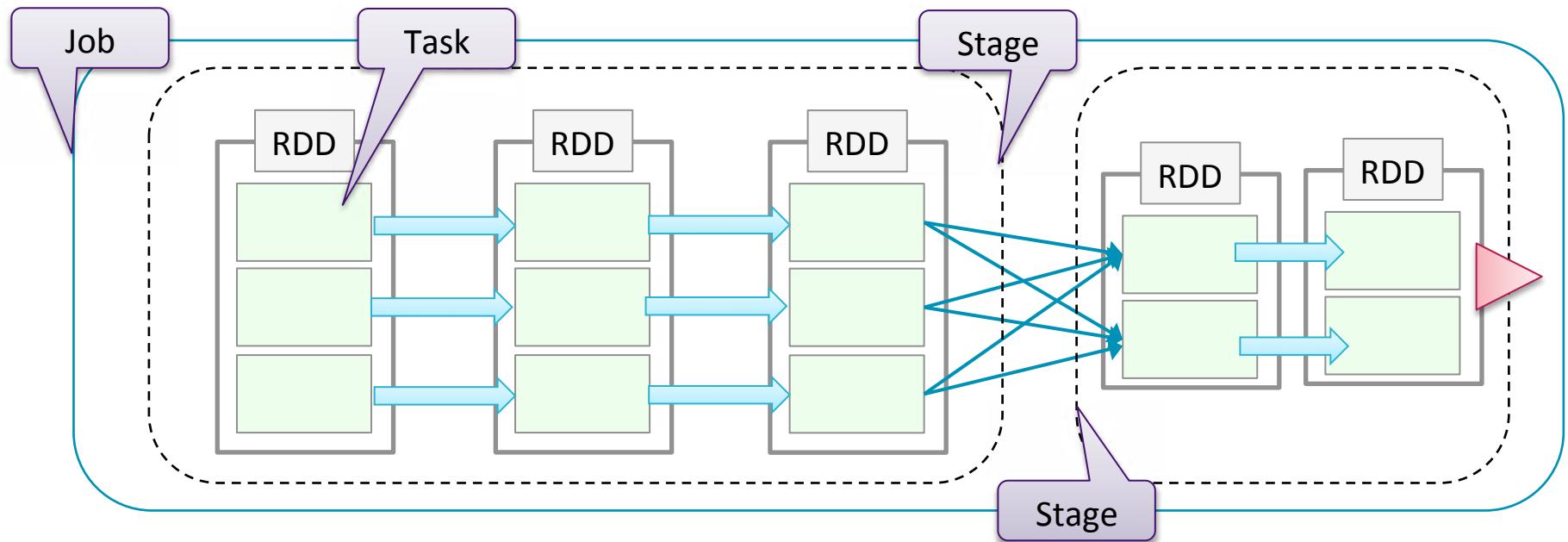
Spark Execution: Stages (4)

```
> val avglen = sc.textFile(myfile) .  
    flatMap(line => line.split("\\W")) .  
    map(word => (word(0), word.length)) .  
    groupByKey() .  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```



Summary of Spark Terminology

- **Job** – a set of tasks executed as a result of an *action*
- **Stage** – a set of tasks in a job that can be executed in parallel
- **Task** – an individual unit of work sent to one executor
- **Application** – can contain any number of jobs managed by a single driver

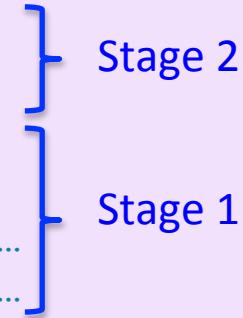


How Spark Calculates Stages

- **Spark constructs a DAG (Directed Acyclic Graph) of RDD dependencies**
- ***Narrow dependencies***
 - Only one child depends on the RDD
 - No shuffle required between nodes
 - Can be collapsed into a single stage
 - e.g., `map`, `filter`, `union`
- ***Wide (or shuffle) dependencies***
 - Multiple children depend on the RDD
 - Defines a new stage
 - e.g., `reduceByKey`, `join`, `groupByKey`

Viewing the Stages using `toDebugString` (Scala)

```
> val avglangs = sc.textFile(myfile) .  
    flatMap(line => line.split("\\W")) .  
    map(word => (word(0), word.length)) .  
    groupByKey() .  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglangs.toDebugString()  
  
(2) MappedRDD[5] at map at ...  
| ShuffledRDD[4] at groupByKey at ...  
+- (4) MappedRDD[3] at map at ...  
| FlatMappedRDD[2] at flatMap at ...  
| myfile MappedRDD[1] at textFile at ...  
| myfile HadoopRDD[0] at textFile at ...
```



Indents indicate
stages (shuffle
boundaries)

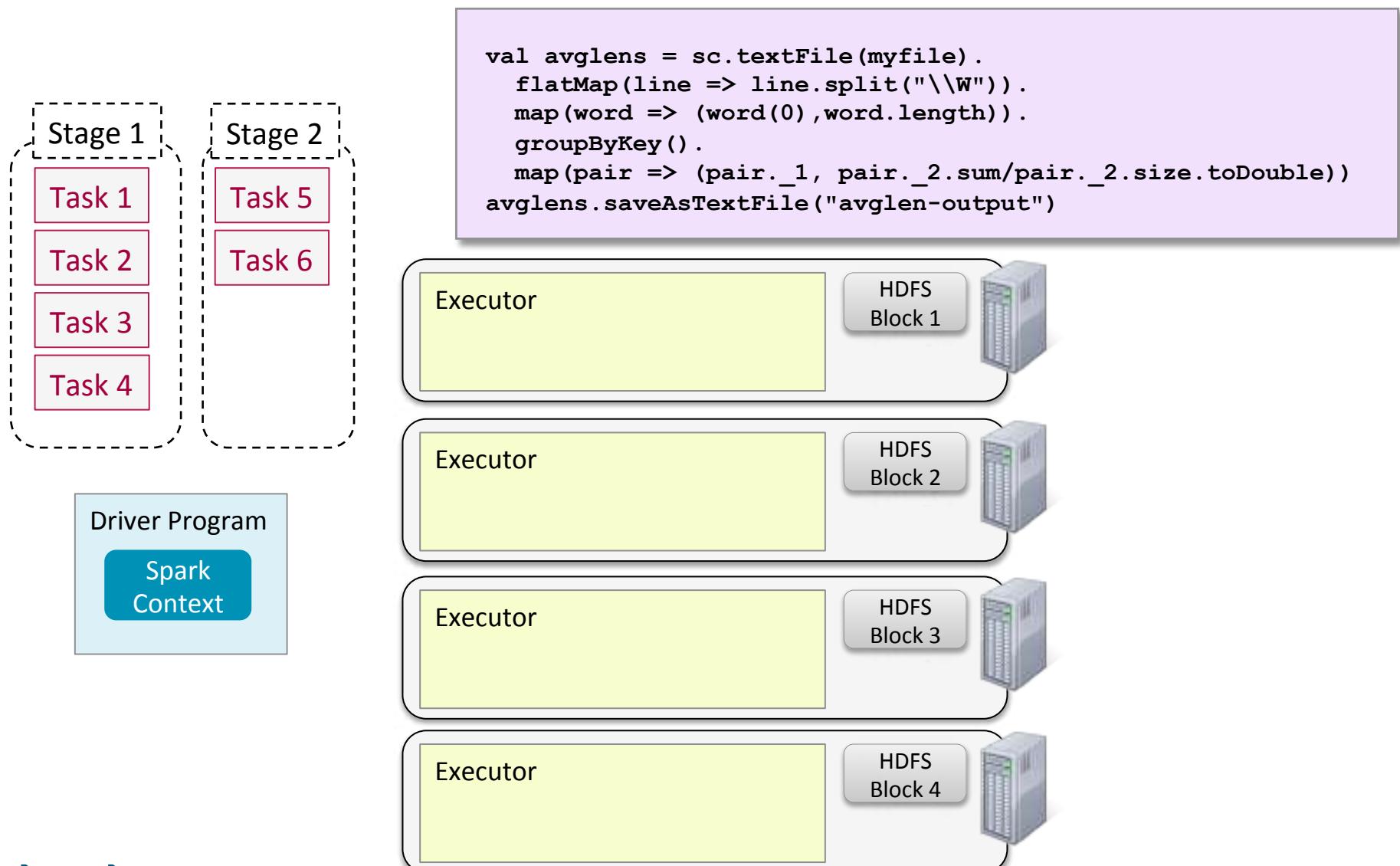
Viewing the Stages using `toDebugString` (Python)

```
> avglens = sc.textFile(myfile) \
    .flatMap(lambda line: line.split()) \
    .map(lambda word: (word[0],len(word))) \
    .groupByKey() \
    .map(lambda (k, values): \
        (k, sum(values)/len(values)))\n\n> print avglens.toDebugString()\n(2) PythonRDD[13] at RDD at ...  
|  MappedRDD[12] at values at ...  
|  ShuffledRDD[11] at partitionBy at ...  
+- (4) PairwiseRDD[10] at groupByKey at ...  
   |  PythonRDD[9] at groupByKey at ...  
   |  myfile MappedRDD[7] at textFile at ...  
   |  myfile HadoopRDD[6] at textFile at ...
```

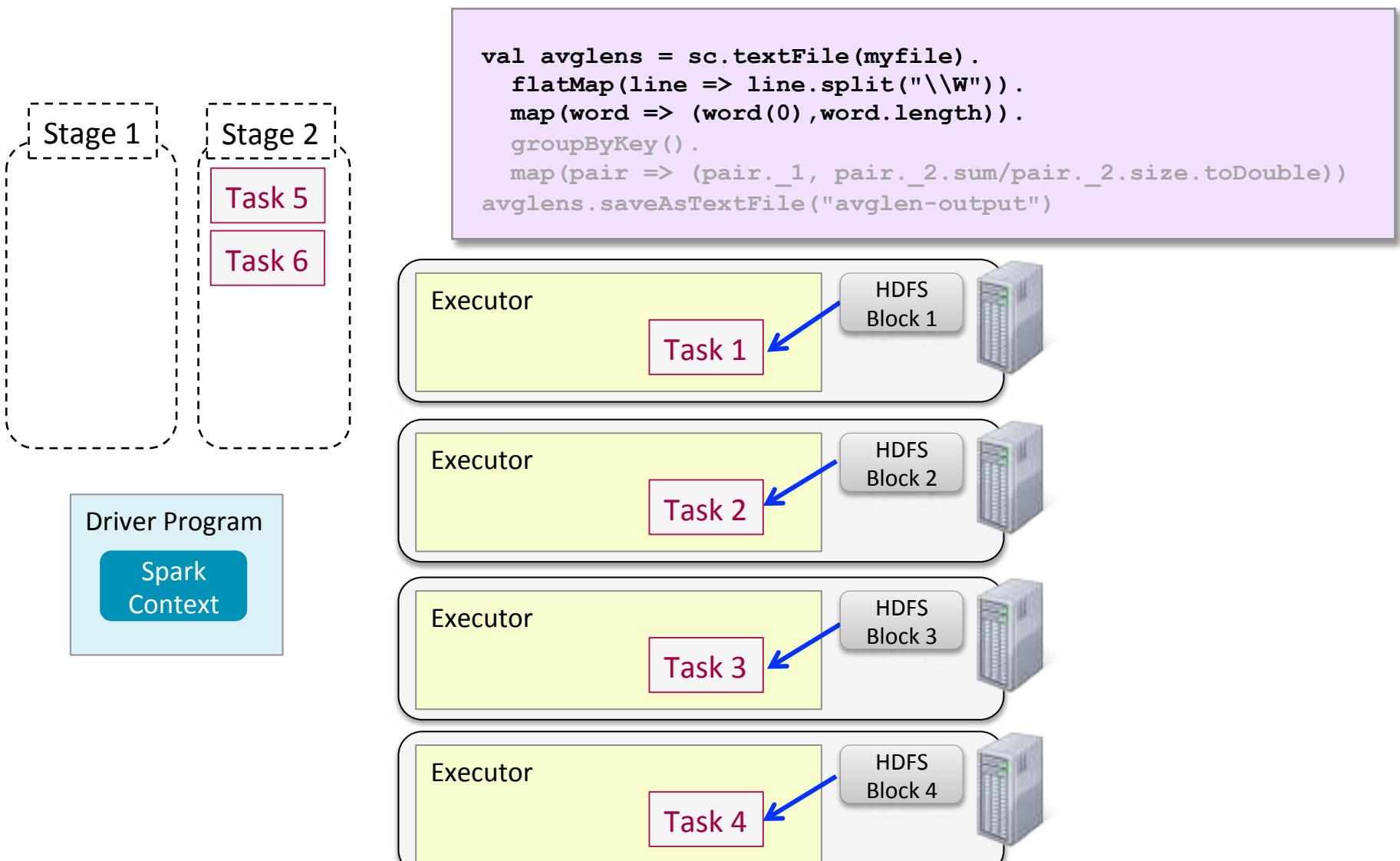
Indents indicate
stages (shuffle
boundaries)

} Stage 2
} Stage 1

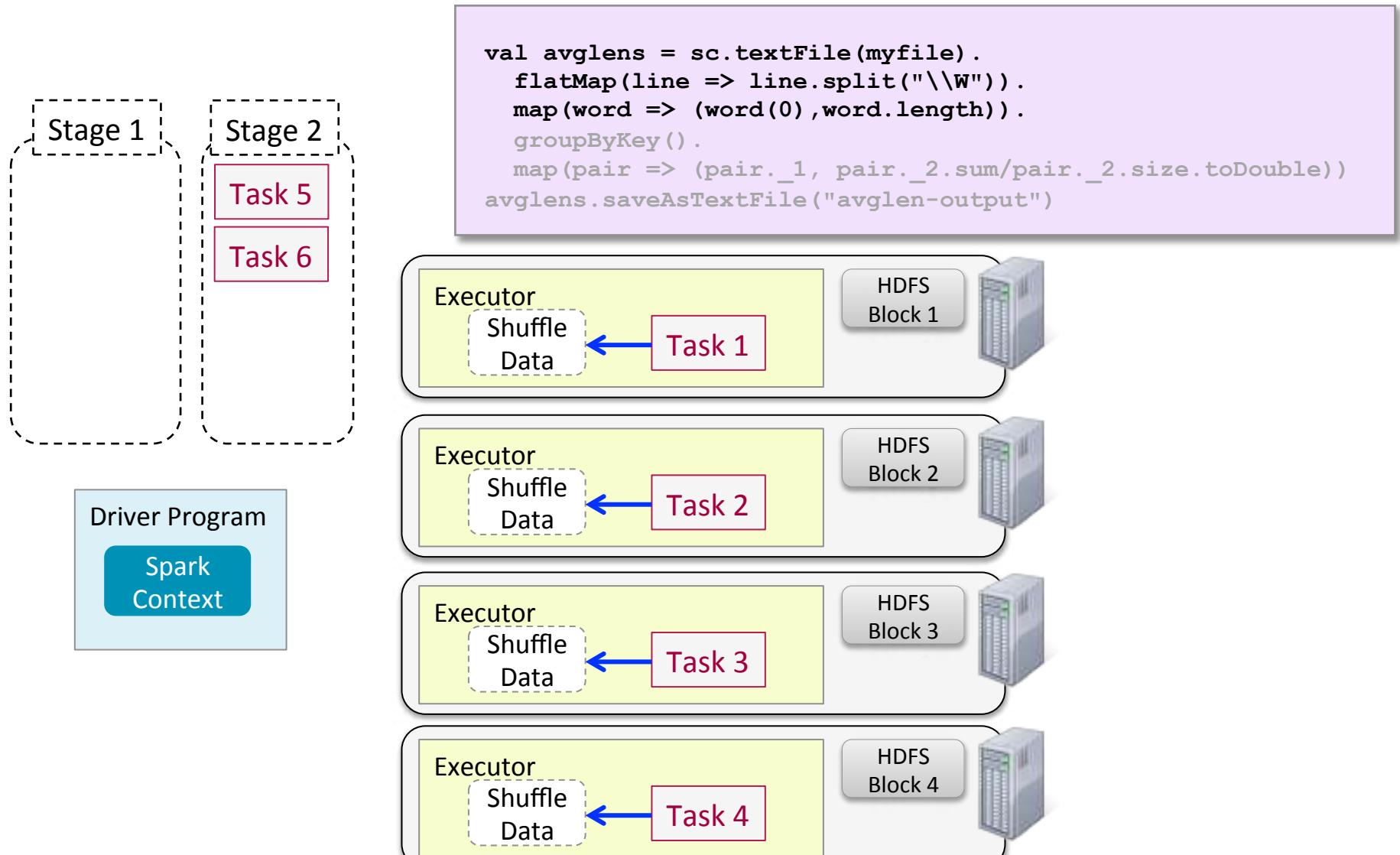
Spark Task Execution (1)



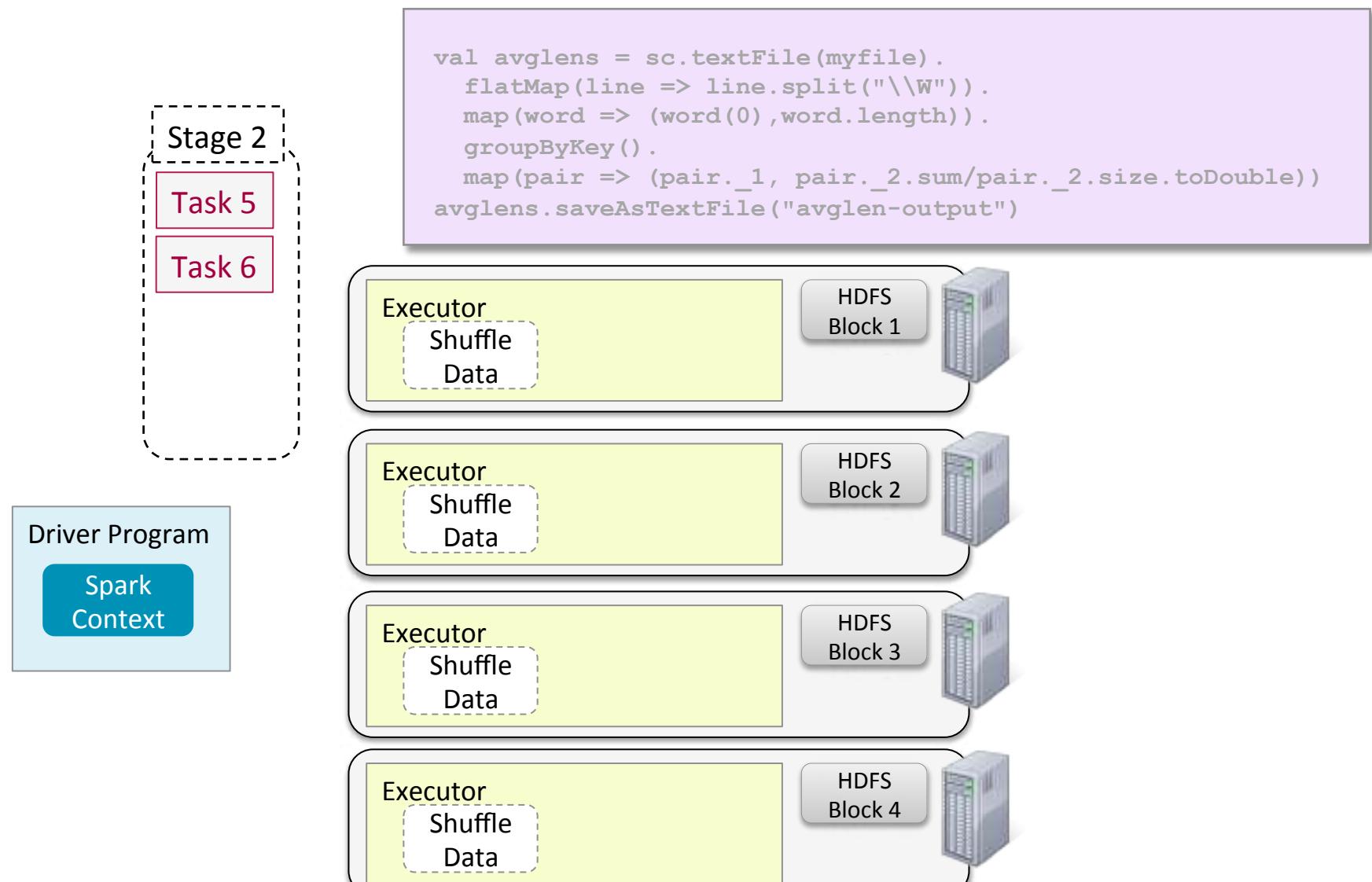
Spark Task Execution (2)



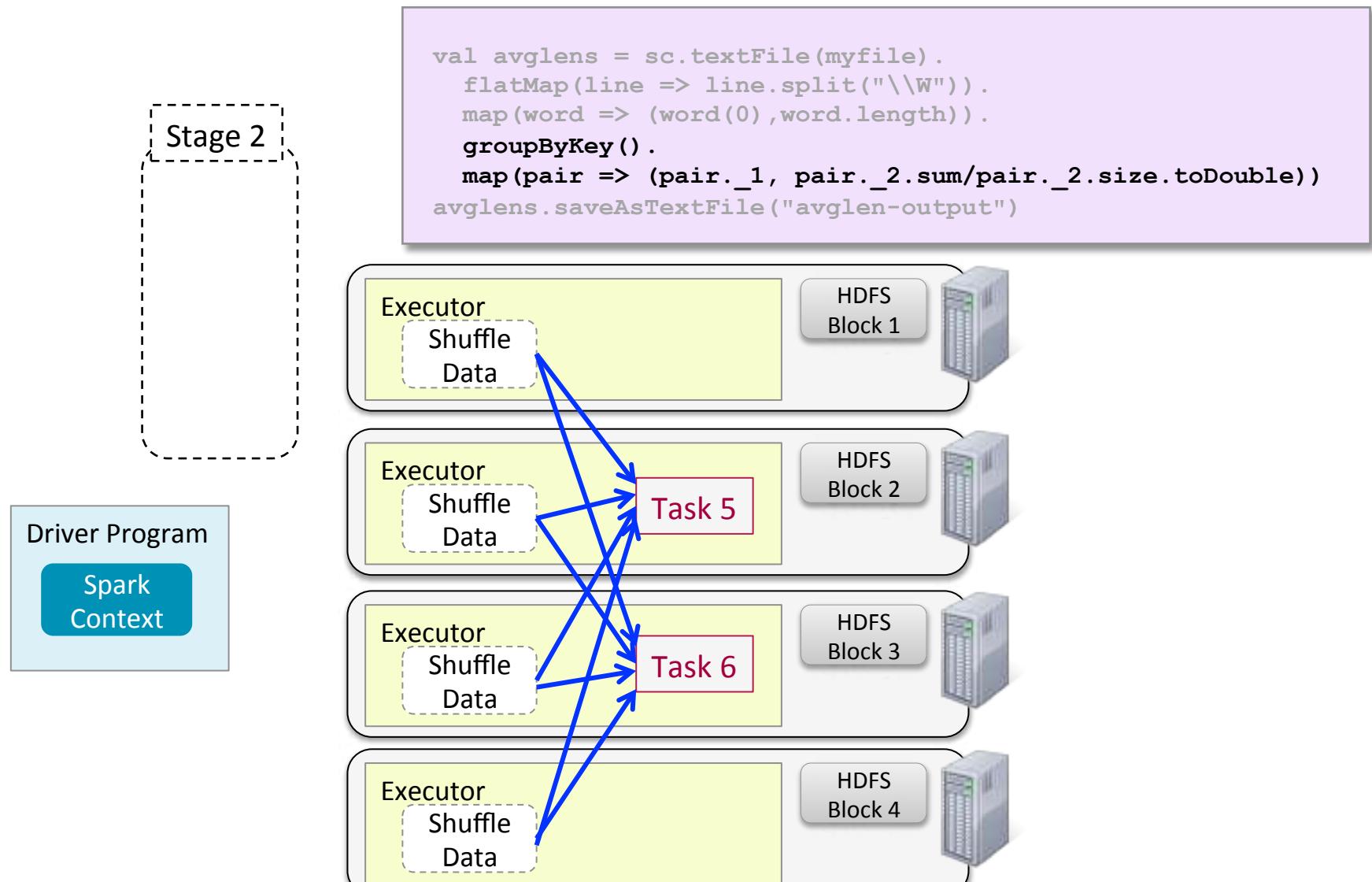
Spark Task Execution (3)



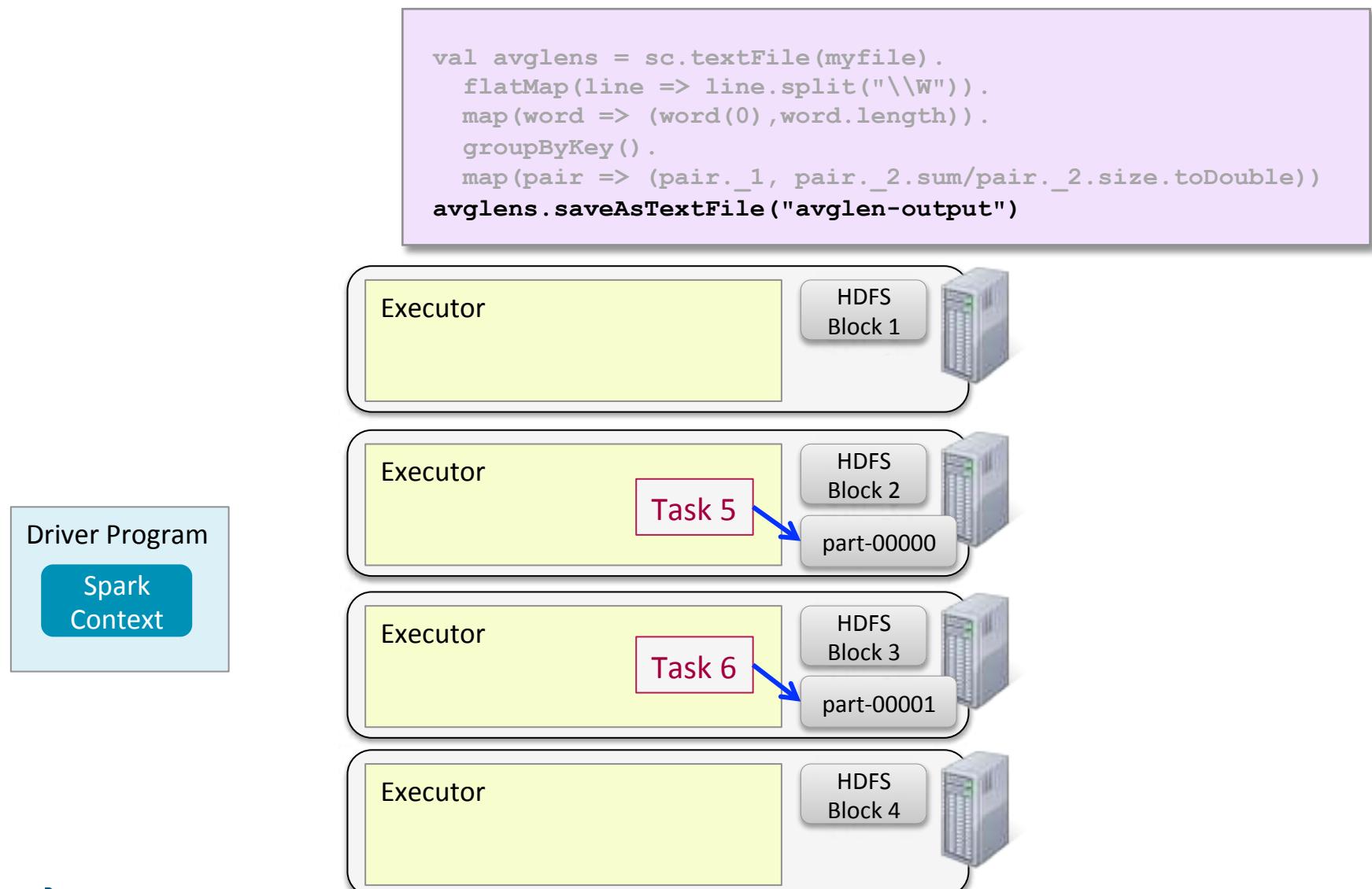
Spark Task Execution (4)



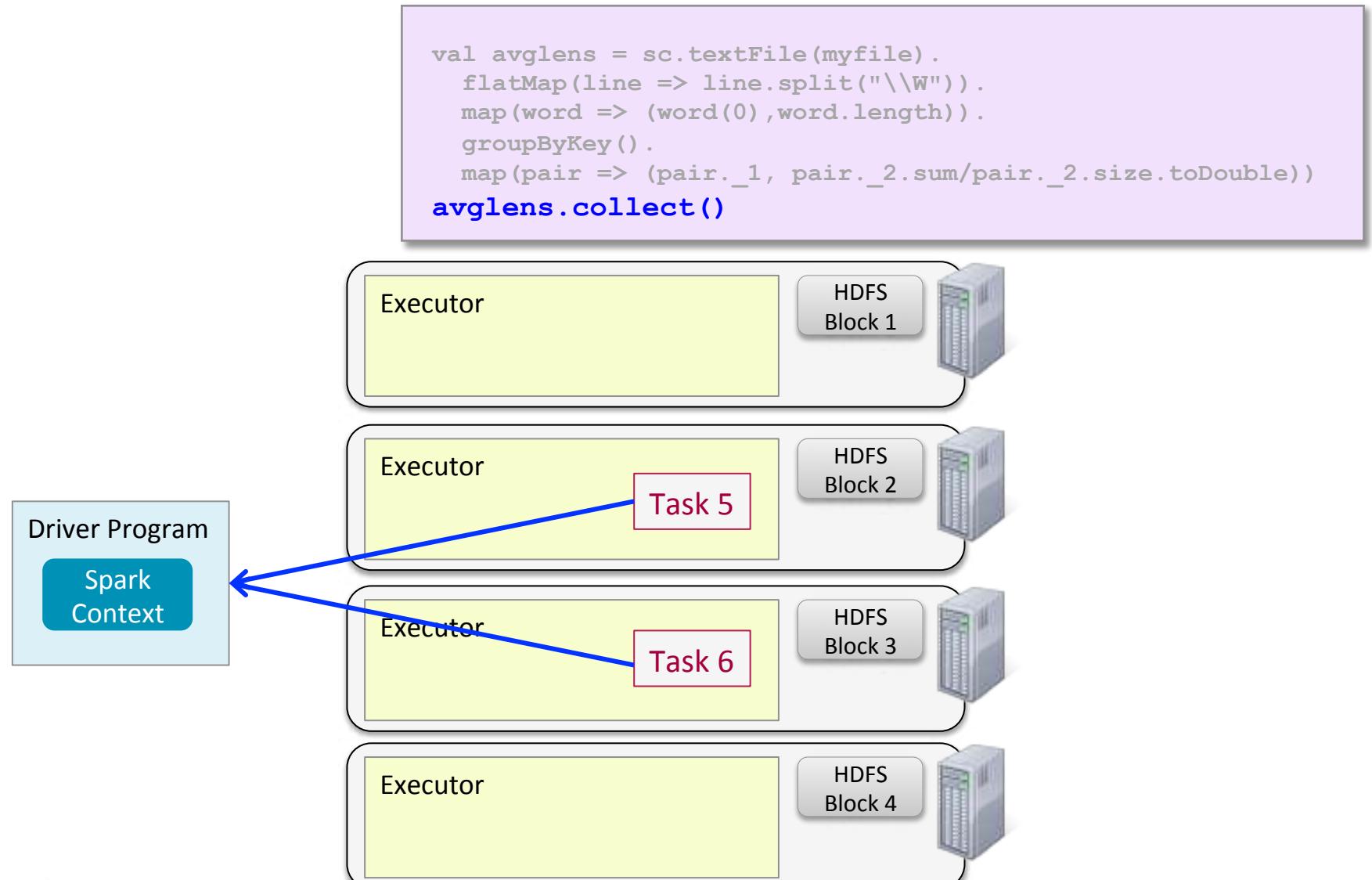
Spark Task Execution (5)



Spark Task Execution (6)



Spark Task Execution (alternate ending)



Controlling the Level of Parallelism

- “Wide” operations (e.g., `reduceByKey`) partition result RDDs
 - More partitions = more parallel tasks
 - Cluster will be under-utilized if there are too few partitions
- You can control how many partitions
 - Configure with the `spark.default.parallelism` property

```
spark.default.parallelism      10
```

- Optional `numPartitions` parameter in function call

```
> words.reduceByKey(lambda v1, v2: v1 + v2, 15)
```

Viewing Stages in the Spark Application UI (1)

- You can view jobs and stages in the Spark Application UI

The screenshot shows the Spark Application UI interface. At the top, there is a navigation bar with tabs: Jobs (which is selected), Stages, Storage, Environment, and Executors. The main content area is titled "Spark Jobs (?)". It displays the following statistics:

- Total Duration: 59 s
- Scheduling Mode: FIFO
- Completed Jobs: 1

A callout bubble points to the "Completed Jobs: 1" text, containing the explanatory text: "Jobs are identified by the action that triggered the job execution". Below this, a table titled "Completed Jobs (1)" lists the single completed job. The table has columns: Job Id, Description, Submitted, Duration, Stages: Succeeded/Total, and Tasks (for all stages): Succeeded/Total. The data for the single job is:

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	saveAsTextFile at <console>:26	2015/09/01 08:56:46	17 s	2/2	7/7

Viewing Stages in the Spark Application UI (2)

- Select the job to view execution stages

The screenshot shows the Spark Application UI interface for version 1.3.0. The top navigation bar includes tabs for Jobs, Stages, Storage, Environment, and Executors. Below this, a section titled "Details for Job" displays the status as "SUCCEEDED" and indicates "Completed Stages: 2". A table titled "Completed Stages (2)" lists two stages:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	saveAsTextFile at <console>:26 +details	2015/09/01 08:56:57	5 s	3/3		433.0 B	18.3 MB	
0	map at <console>:26 +details	2015/09/01 08:56:46	12 s	4/4	61.2 MB			18.3 MB

Three callout boxes highlight specific features:

- "Stages are identified by the last operation"
- "Number of tasks = number of partitions"
- "Data shuffled between stages"

Chapter Topics

Parallel Processing in Spark

Distributed Data Processing with Spark

- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- **Conclusion**
- Hands-On Exercise: View Jobs and Stages in the Spark Application UI

Essential Points

- **RDDs are stored in the memory of Spark executor JVMs**
- **Data is split into partitions – each partition in a separate executor**
- **RDD operations are executed on partitions in parallel**
- **Operations that depend on the same partition are pipelined together in stages**
 - e.g., `map`, `filter`
- **Operations that depend on multiple partitions are executed in separate stages**
 - e.g., `join`, `reduceByKey`

Chapter Topics

Parallel Processing in Spark

Distributed Data Processing with Spark

- Review: Spark on a Cluster
- RDD Partitions
- Partitioning of File-based RDDs
- HDFS and Data Locality
- Executing Parallel Operations
- Stages and Tasks
- Conclusion
- **Hands-On Exercise: View Jobs and Stages in the Spark Application UI**

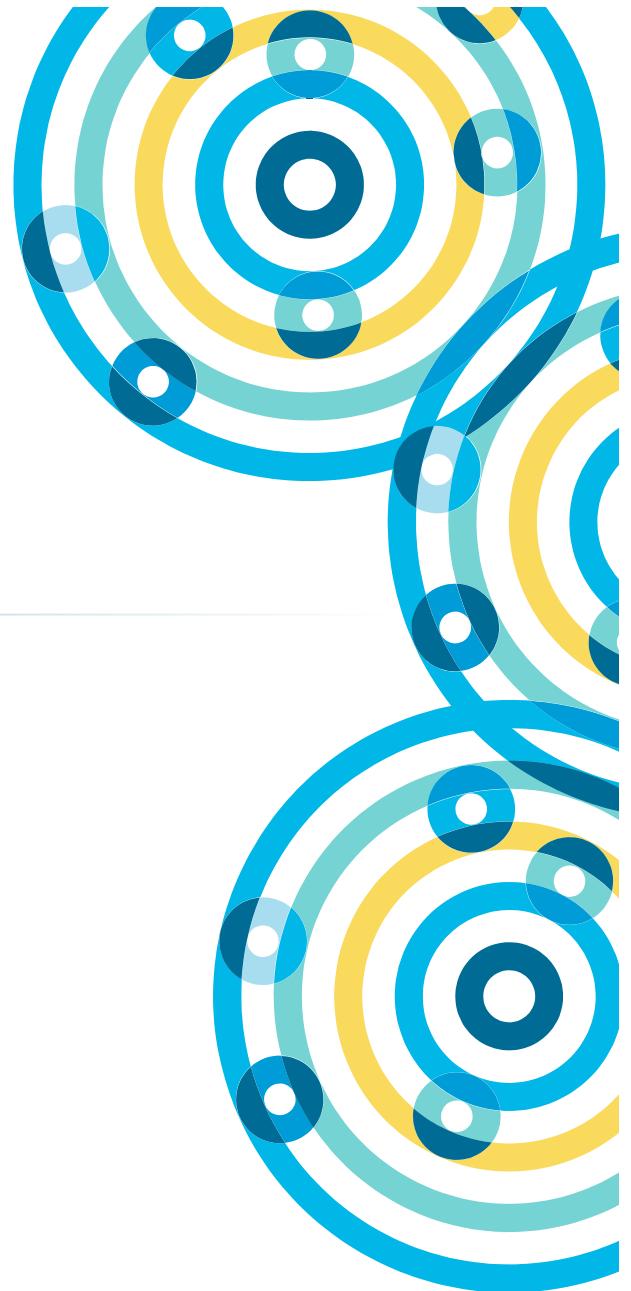
Hands-On Exercise: View Jobs and Stages in the Spark Application UI

- **In this Hands-On Exercise, you will**
 - Use the Spark Application UI to view how jobs, stages and tasks are executed in a job
- **Please refer to the Hands-On Exercise Manual**



Spark RDD Persistence

Chapter 15



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence**
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

Spark RDD Persistence

In this chapter you will learn

- **How Spark uses an RDD's lineage in operations**
- **How to persist RDDs to improve performance**

Chapter Topics

Spark RDD Persistence

Distributed Data Processing with Spark

- **RDD Lineage**
- RDD Persistence Overview
- Distributed Persistence
- Conclusion
- Hands-On Exercise: Persist an RDD

Lineage Example (1)

- Each *transformation* operation creates a new *child* RDD

File: purplecow.txt

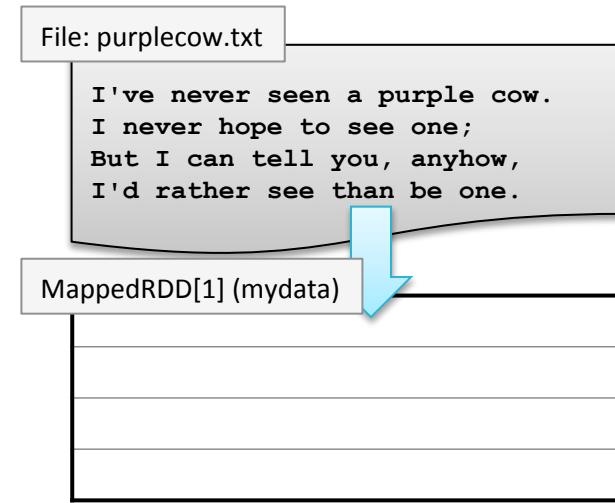
```
I've never seen a purple cow.  
I never hope to see one;  
But I can tell you, anyhow,  
I'd rather see than be one.
```



Lineage Example (2)

- Each *transformation* operation creates a new *child* RDD

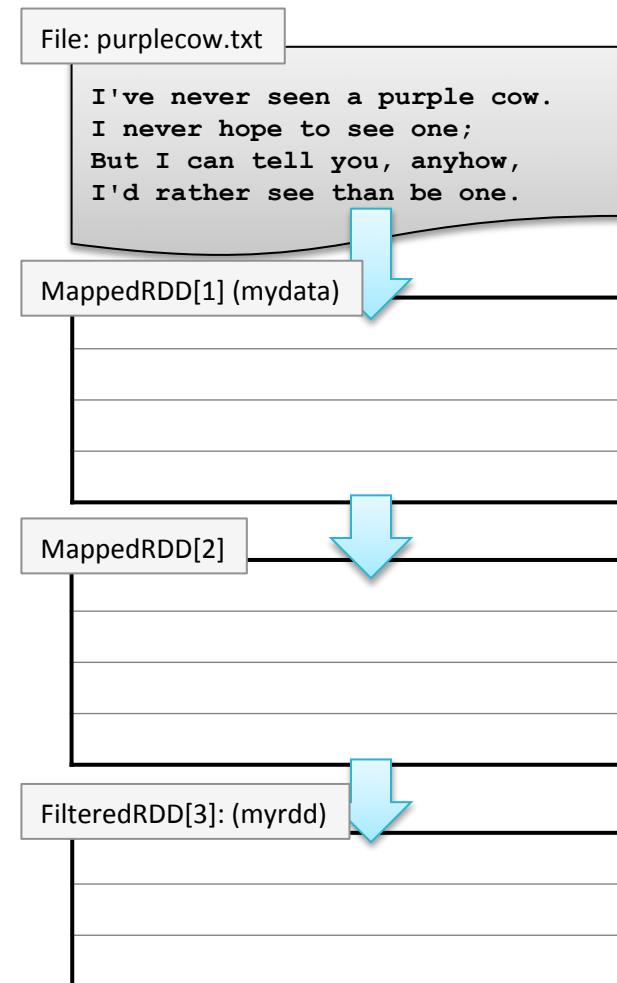
```
> mydata = sc.textFile("purplecow.txt")
```



Lineage Example (3)

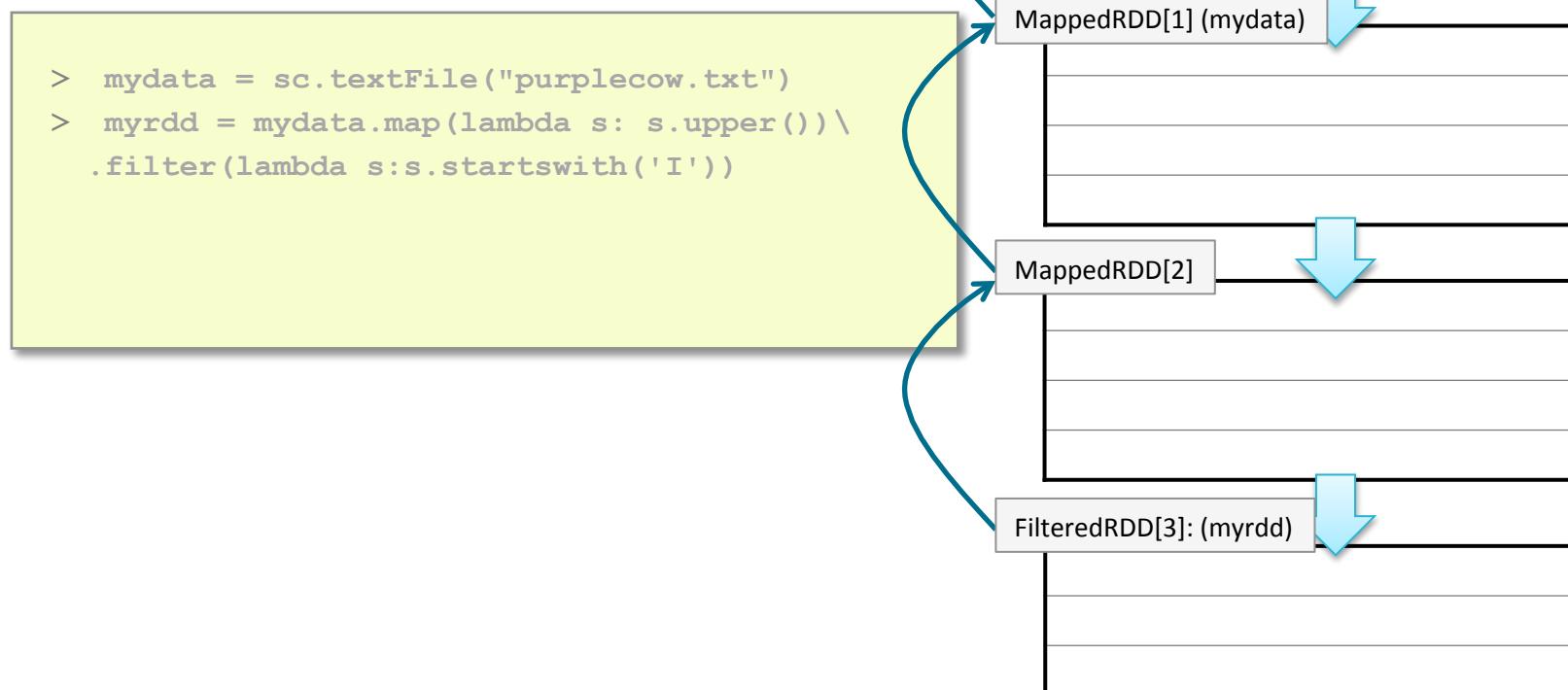
- Each *transformation* operation creates a new *child* RDD

```
> mydata = sc.textFile("purplecow.txt")
> myrdd = mydata.map(lambda s: s.upper()) \
    .filter(lambda s:s.startswith('I'))
```



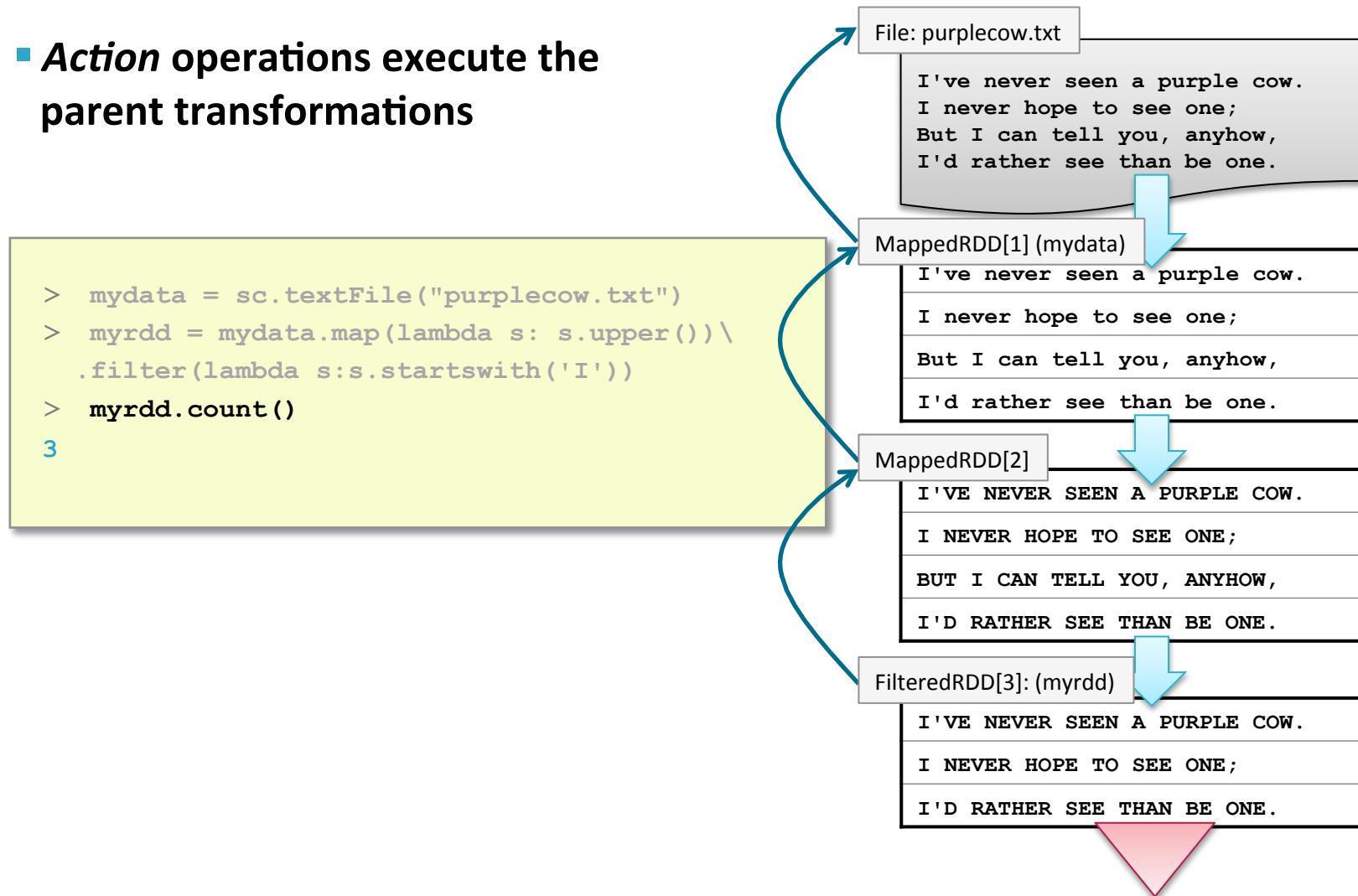
Lineage Example (4)

- Spark keeps track of the *parent* RDD for each new RDD
- Child RDDs *depend on* their parents



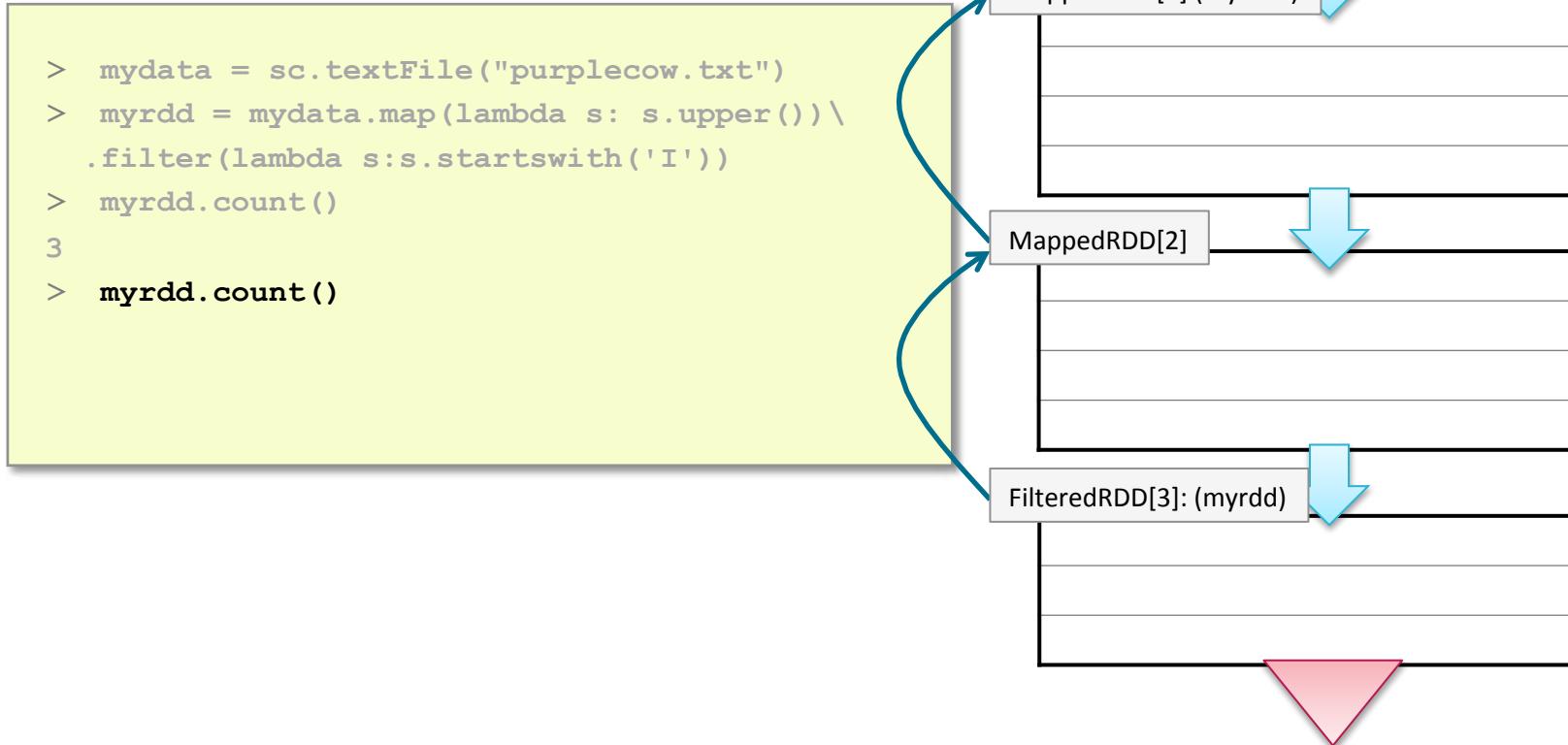
Lineage Example (5)

- Action operations execute the parent transformations



Lineage Example (6)

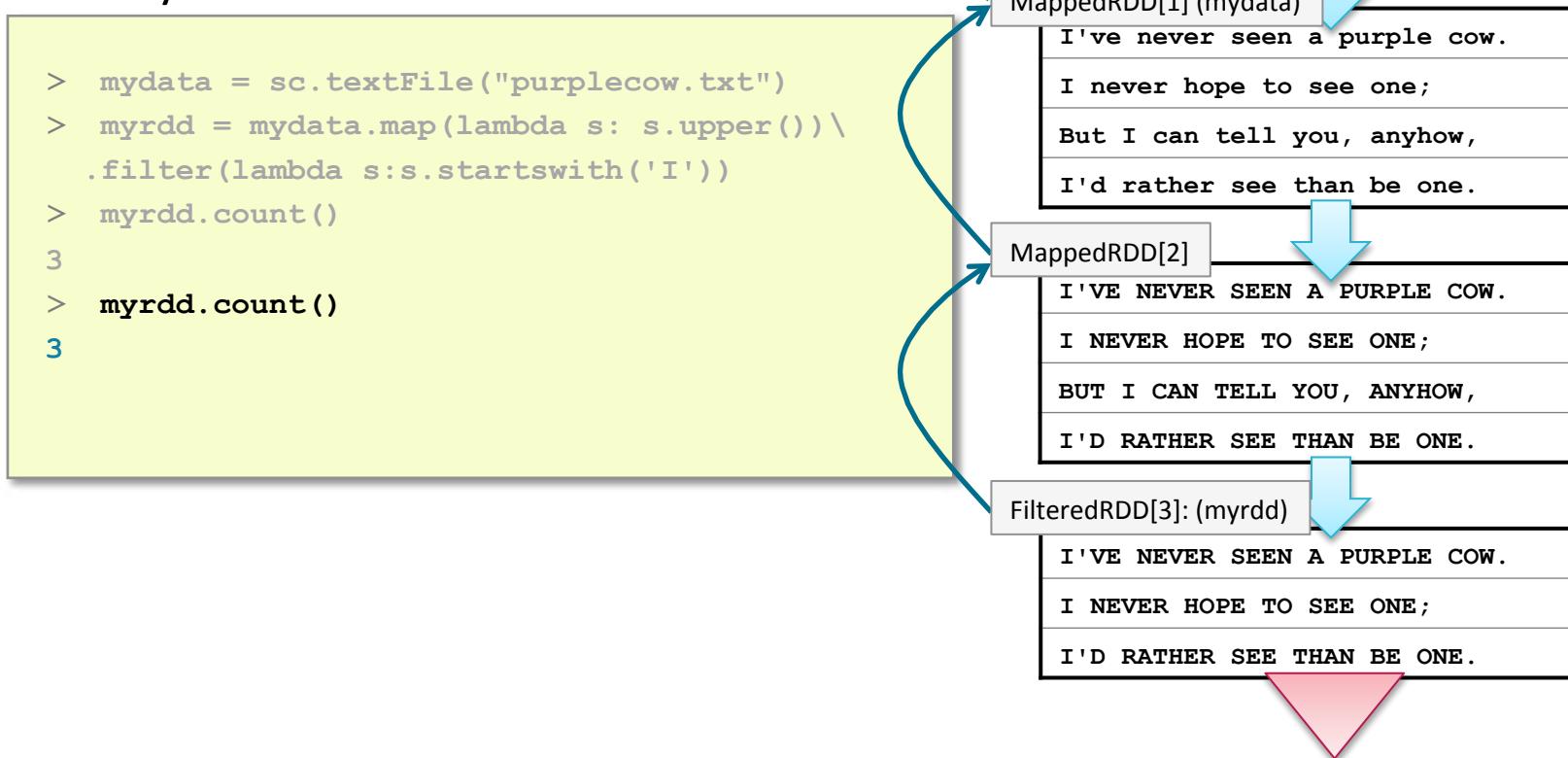
- Each action re-executes the lineage transformations starting with the base
 - By default



Lineage Example (7)

- Each action re-executes the lineage transformations starting with the base

- By default



Chapter Topics

Spark RDD Persistence

Distributed Data Processing with Spark

- RDD Lineage
- **RDD Persistence Overview**
- Distributed Persistence
- Conclusion
- Hands-On Exercise: Persist an RDD

RDD Persistence

- Persisting an RDD saves the data (by default in memory)

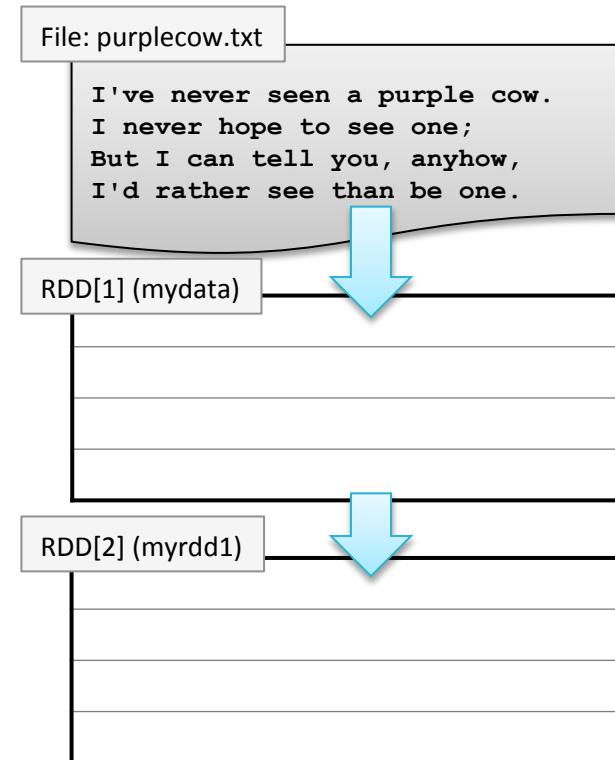
File: purplecow.txt

```
I've never seen a purple cow.  
I never hope to see one;  
But I can tell you, anyhow,  
I'd rather see than be one.
```

RDD Persistence

- Persisting an RDD saves the data (by default in memory)

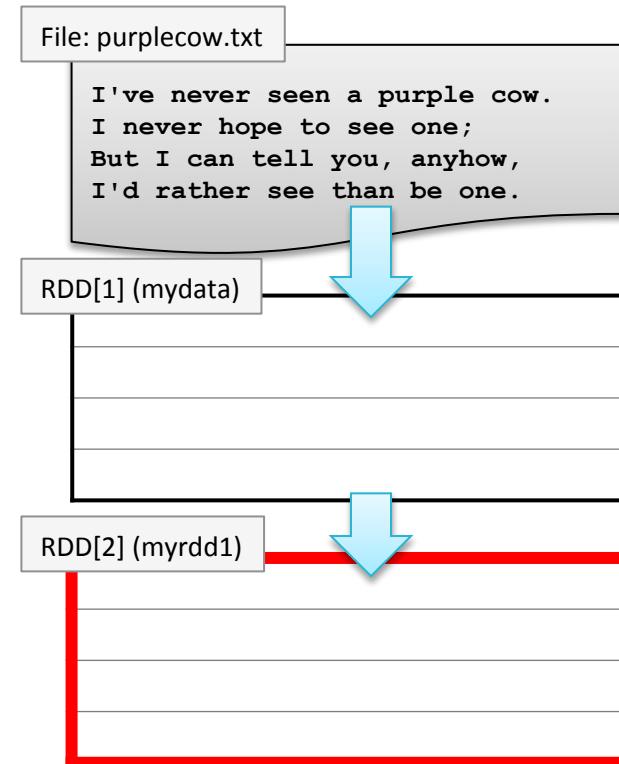
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
    s.upper())
```



RDD Persistence

- Persisting an RDD saves the data (by default in memory)

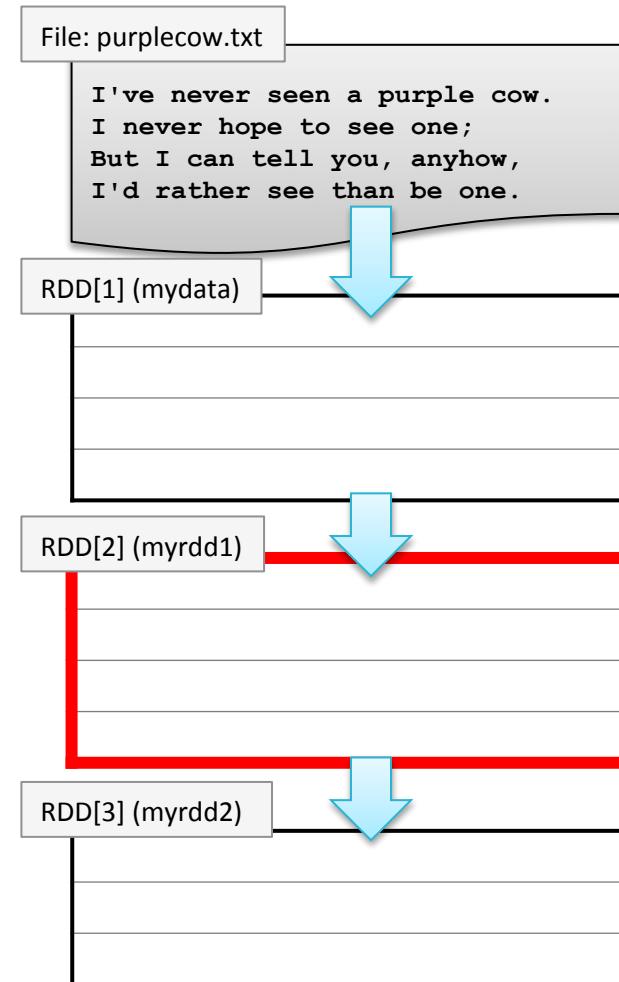
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
```



RDD Persistence

- Persisting an RDD saves the data (by default in memory)

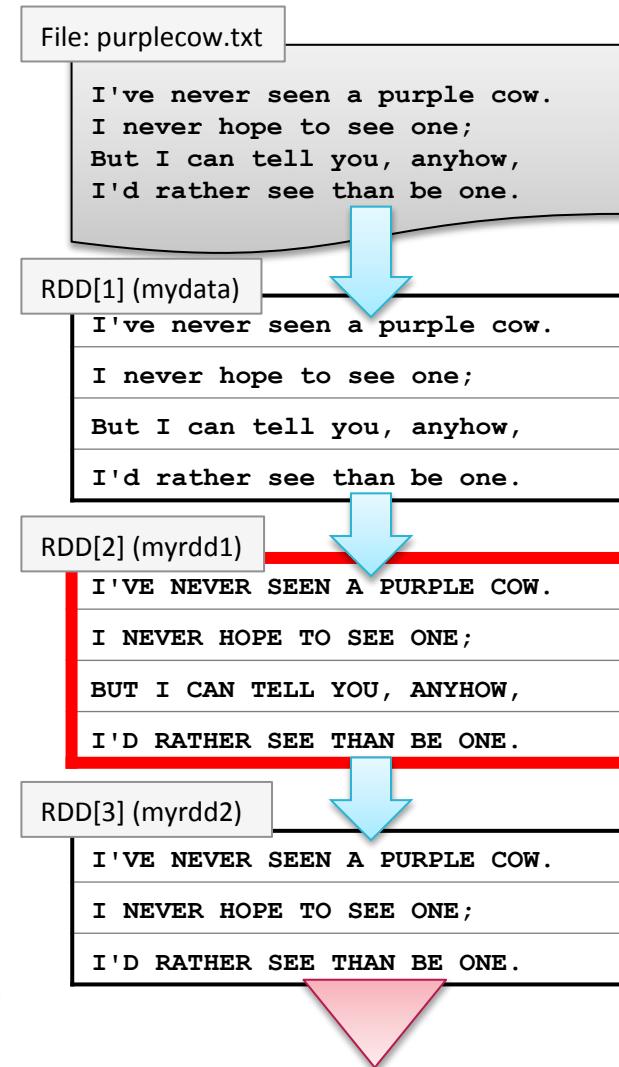
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
```



RDD Persistence

- Persisting an RDD saves the data (by default in memory)

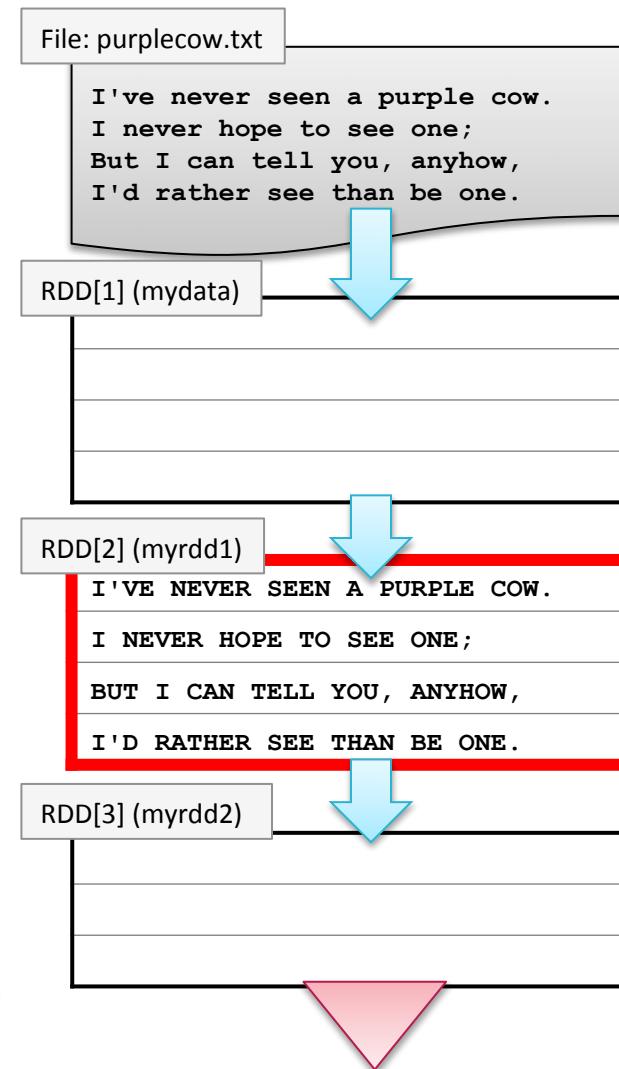
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
```



RDD Persistence

- Subsequent operations use saved data

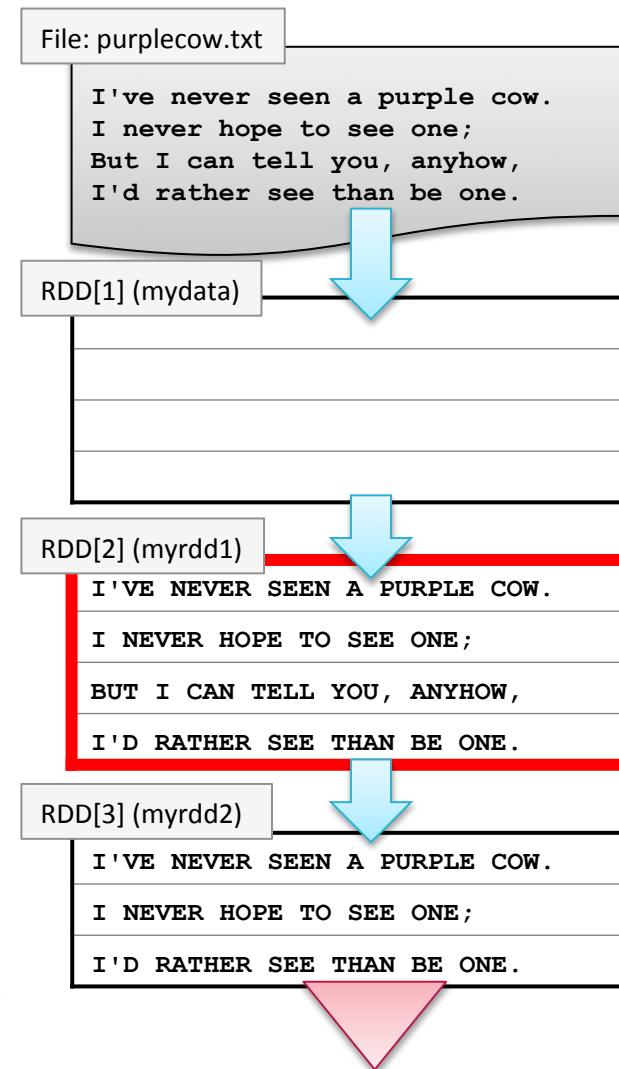
```
> mydata = sc.textFile("purplecow.txt")
> myrdd1 = mydata.map(lambda s:
  s.upper())
> myrdd1.persist()
> myrdd2 = myrdd1.filter(lambda \
  s:s.startswith('I'))
> myrdd2.count()
3
> myrdd2.count()
```



RDD Persistence

- Subsequent operations use saved data

```
> my data =  
  sc.textFile("purplecow.txt")  
> myrdd1 = mydata.map(lambda s:  
  s.upper())  
> myrdd1.persist()  
> myrdd2 = myrdd1.filter(lambda \  
  s:s.startswith('I'))  
> myrdd2.count()  
3  
> myrdd2.count()  
3
```



Memory Persistence

- **In-memory persistence is a *suggestion* to Spark**
 - If not enough memory is available, persisted partitions will be cleared from memory
 - Least recently used partitions cleared first
 - Transformations will be re-executed using the lineage when needed

Chapter Topics

Spark RDD Persistence

Distributed Data Processing with Spark

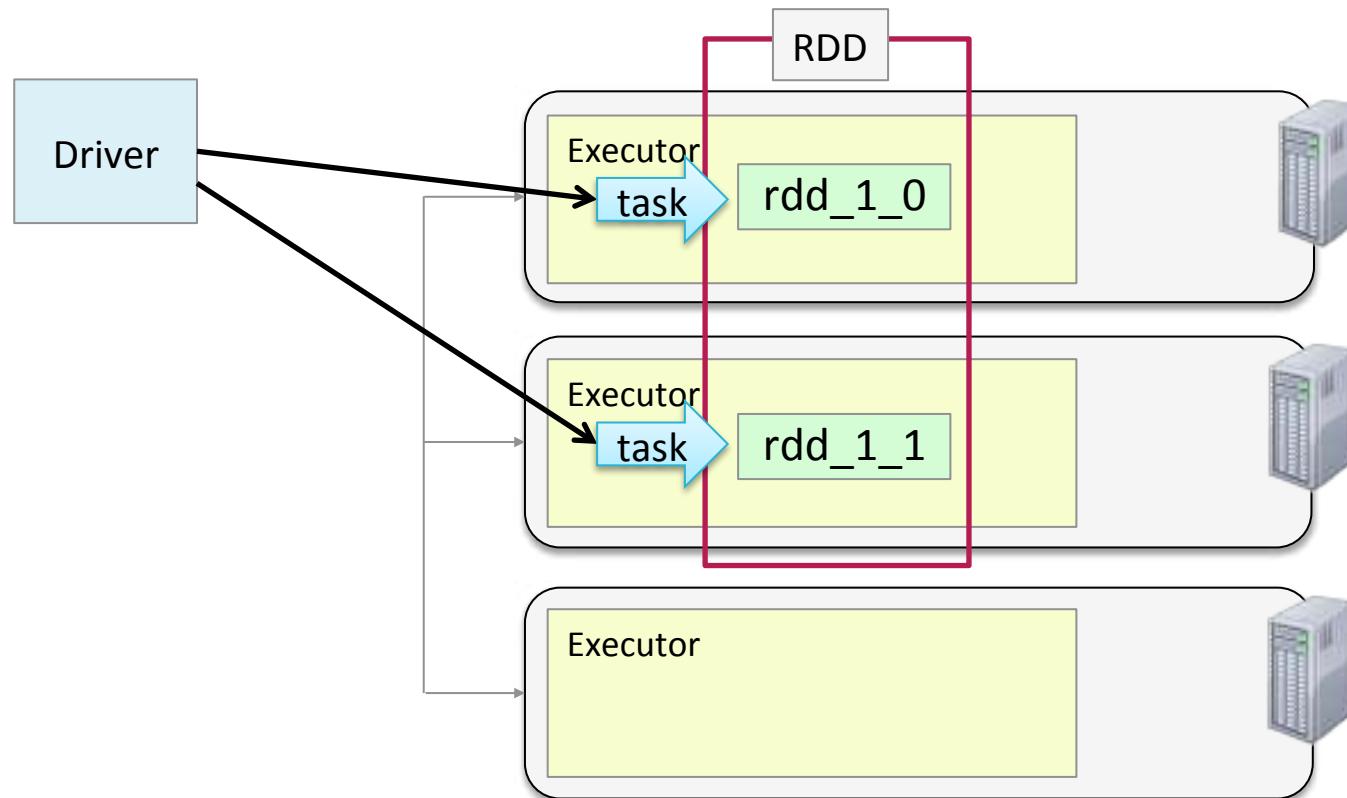
- RDD Lineage
- RDD Persistence Overview
- **Distributed Persistence**
- Conclusion
- Hands-On Exercise: Persist an RDD

Persistence and Fault-Tolerance

- **RDD = *Resilient Distributed Dataset***
 - Resiliency is a product of tracking lineage
 - RDDs can always be recomputed from their base if needed

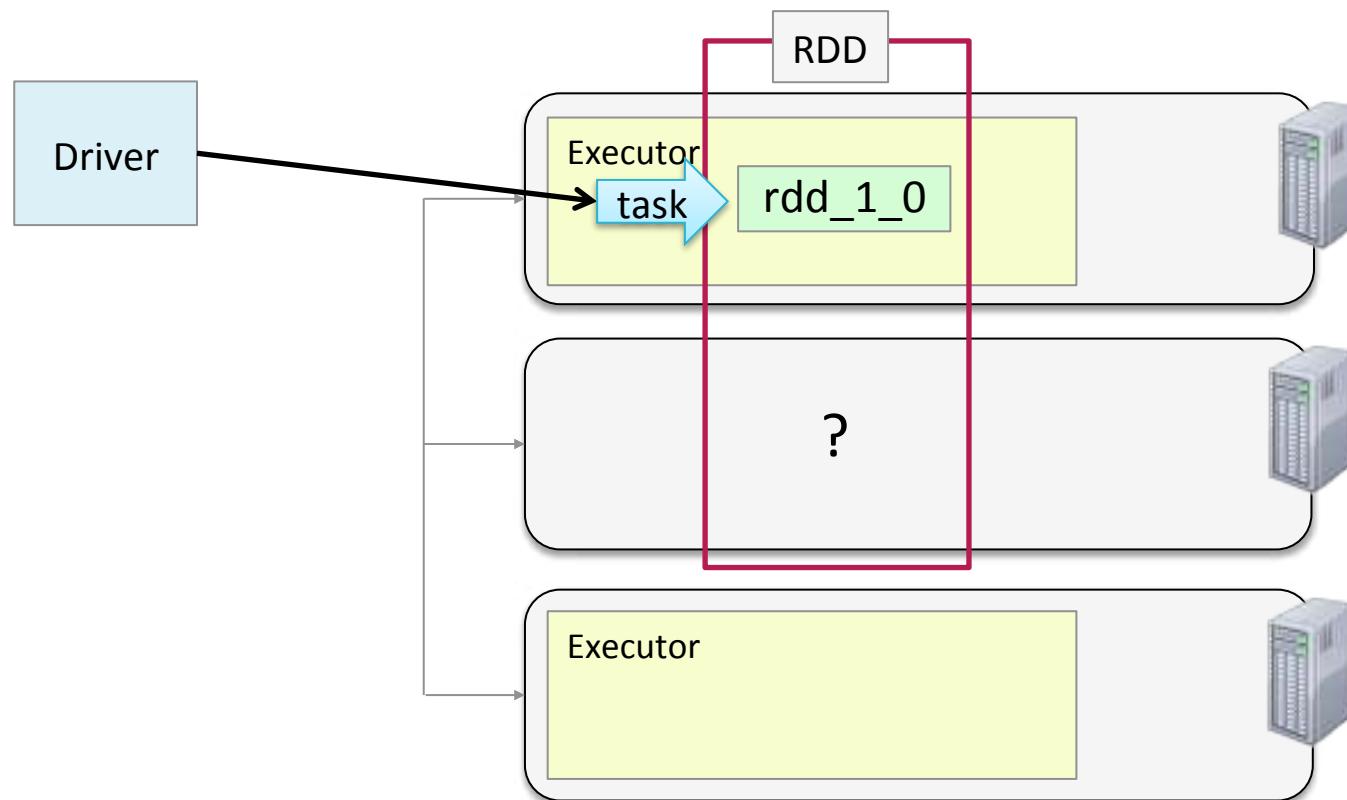
Distributed Persistence

- RDD partitions are distributed across a cluster
- By default, partitions are persisted in memory in Executor JVMs



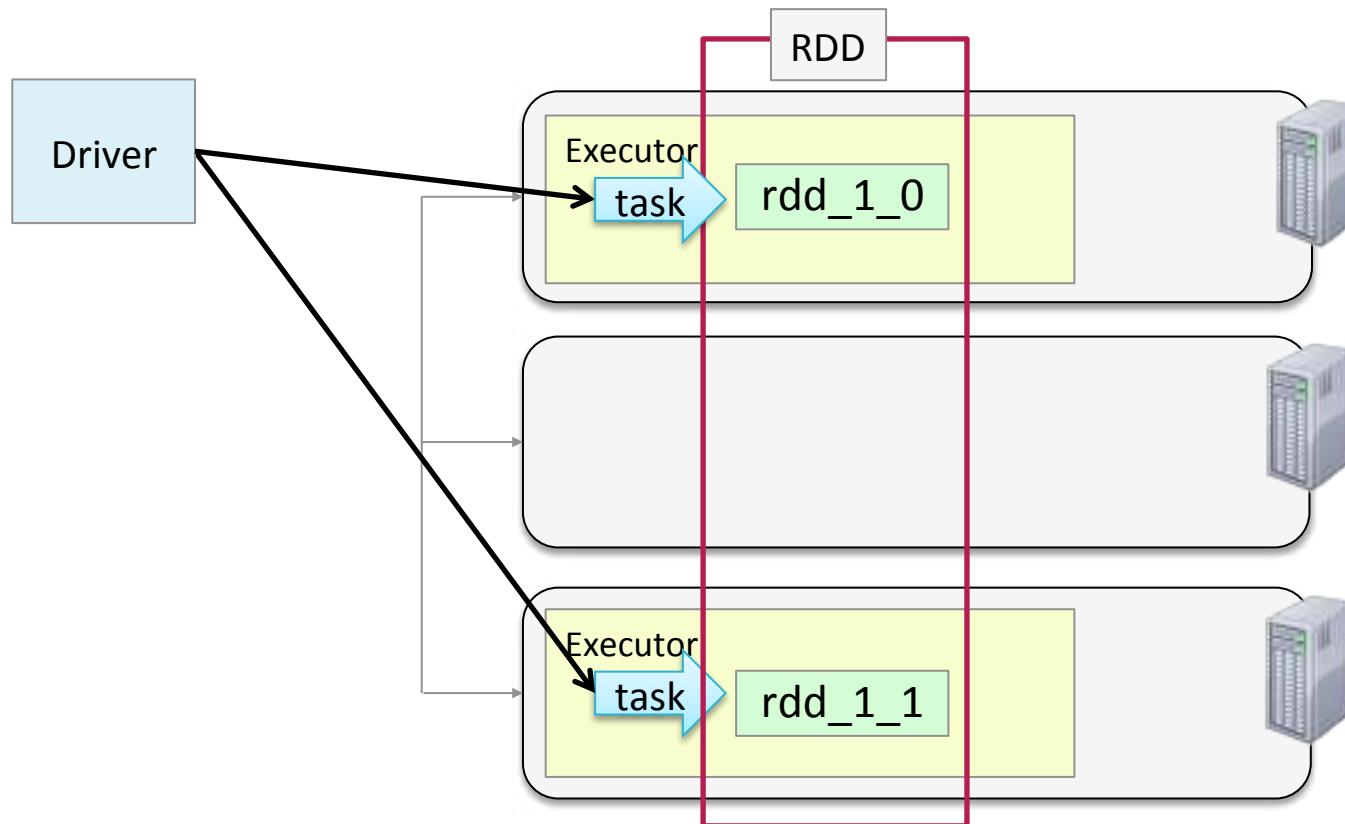
RDD Fault-Tolerance (1)

- What happens if a partition persisted in memory becomes unavailable?



RDD Fault-Tolerance (2)

- The driver starts a new task to recompute the partition on a different node
- Lineage is preserved, data is never lost



Persistence Levels

- By default, the **persist** method stores data in memory only
 - The **cache** method is a synonym for default (memory) persist
- The **persist** method offers other options called **Storage Levels**
- Storage Levels let you control
 - Storage location
 - Format in memory
 - Partition replication

Persistence Levels: Storage Location

- Storage location – where is the data stored?
 - **MEMORY_ONLY** (default) – same as **cache**
 - **MEMORY_AND_DISK** – Store partitions on disk if they do not fit in memory
 - Called *spilling*
 - **DISK_ONLY** – Store all partitions on disk

Python

```
> from pyspark import StorageLevel  
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Scala

```
> import org.apache.spark.storage.StorageLevel  
> myrdd.persist(StorageLevel.DISK_ONLY)
```

Persistence Levels: Memory Format

- **Serialization – you can choose to serialize the data in memory**
 - **MEMORY_ONLY_SER** and **MEMORY_AND_DISK_SER**
 - Much more space efficient
 - Less time efficient
 - If using Java or Scala, choose a fast serialization library (e.g. Kryo)

Persistence Levels: Partition Replication

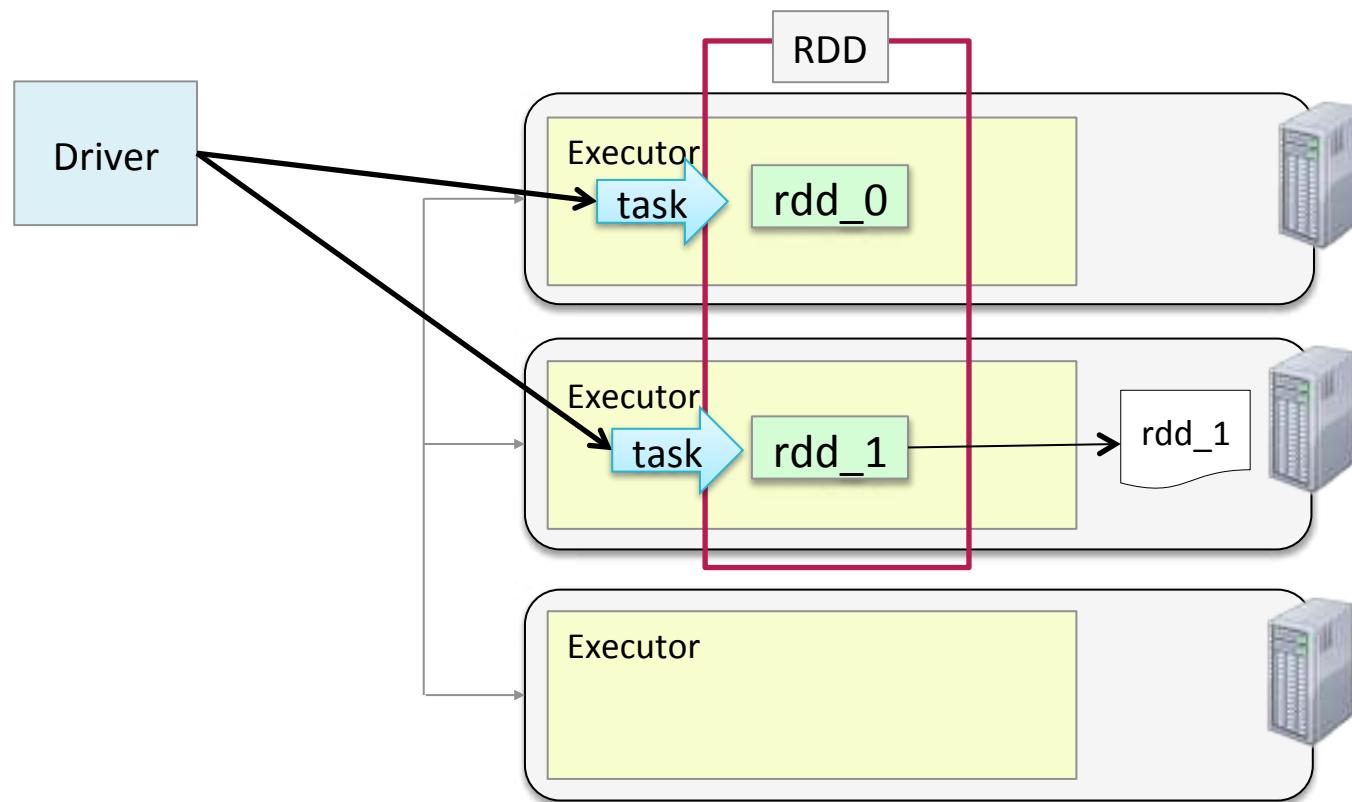
- **Replication – store partitions on two nodes**
 - **MEMORY_ONLY_2**
 - **MEMORY_AND_DISK_2**
 - **DISK_ONLY_2**
 - **MEMORY_AND_DISK_SER_2**
 - **DISK_ONLY_2**
 - You can also define custom storage levels

Changing Persistence Options

- To stop persisting and remove from memory and disk
 - `rdd.unpersist()`
- To change an RDD to a different persistence level
 - Unpersist first

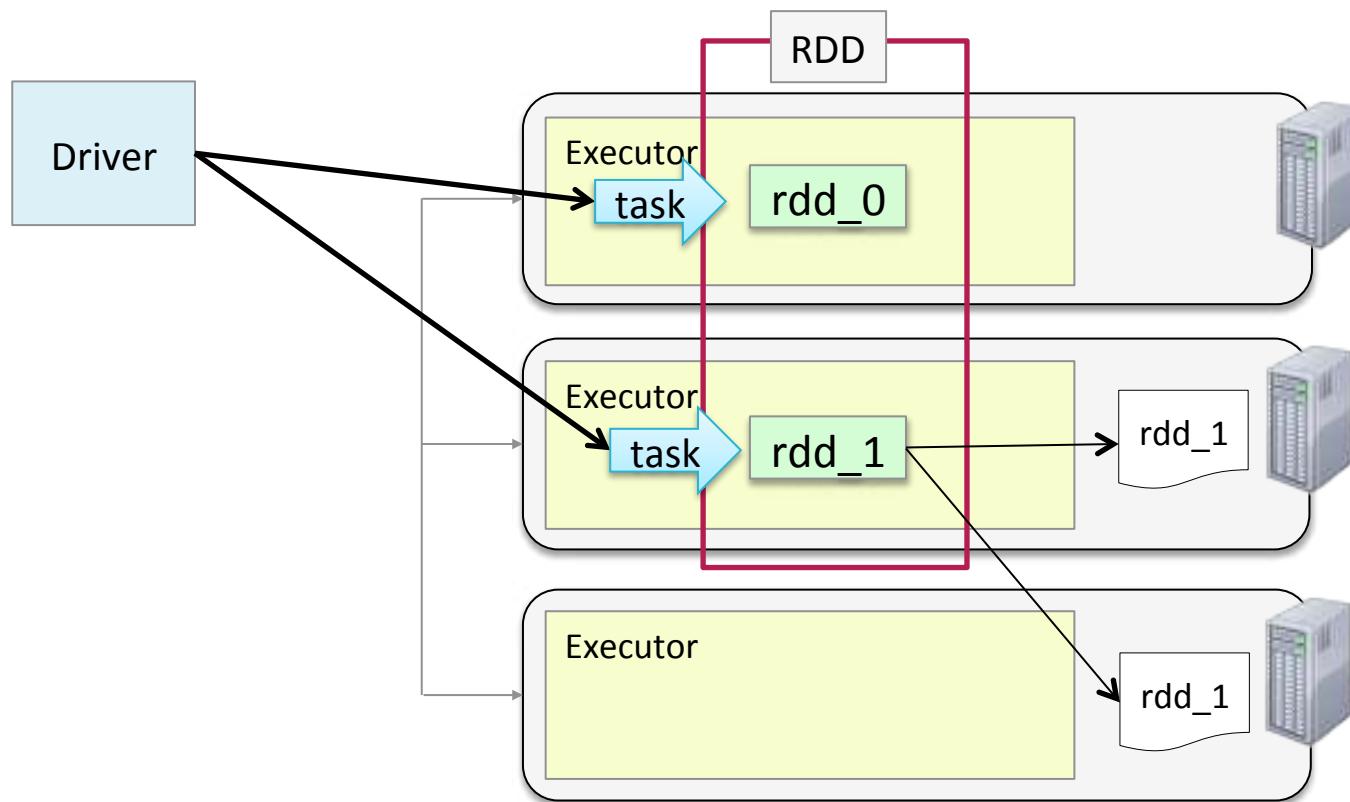
Disk Persistence

- Disk-persisted partitions are stored in local files



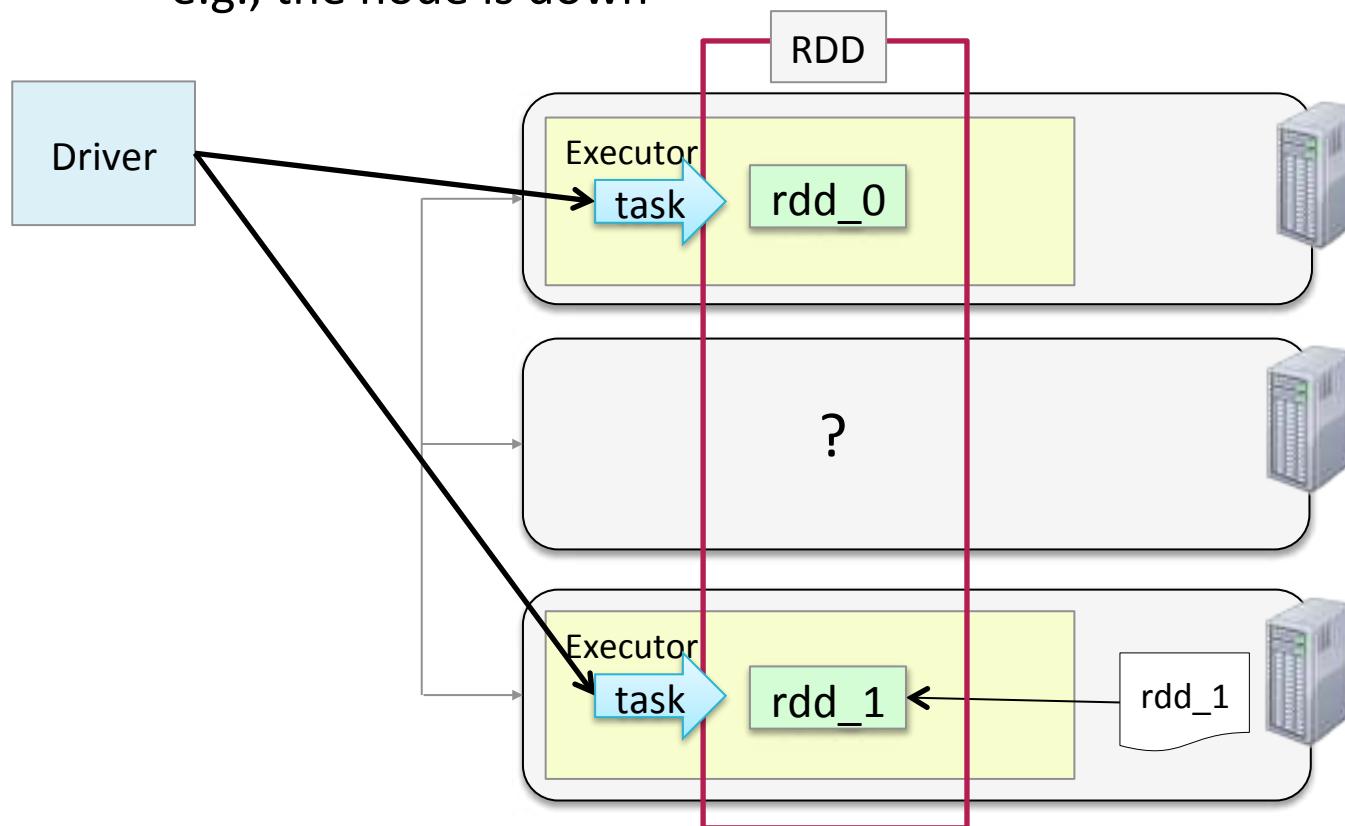
Disk Persistence with Replication (1)

- Persistence replication makes recomputation less likely to be necessary



Disk Persistence with Replication (2)

- Replicated data on disk will be used to recreate the partition if possible
 - Will be recomputed if the data is unavailable
 - e.g., the node is down



When and Where to Persist

- **When should you persist a dataset?**
 - When a dataset is likely to be re-used
 - e.g., iterative algorithms, machine learning
- **How to choose a persistence level**
 - Memory only – when possible, best performance
 - Save space by saving as serialized objects in memory if necessary
 - Disk – choose when recomputation is more expensive than disk read
 - e.g., expensive functions or filtering large datasets
 - Replication – choose when recomputation is more expensive than memory

Chapter Topics

Spark RDD Persistence

Distributed Data Processing with Spark

- RDD Lineage
- RDD Persistence Overview
- Distributed Persistence
- **Conclusion**
- Hands-On Exercise: Persist an RDD

Essential Points

- **Spark keeps track of each RDD's lineage**
 - Provides fault tolerance
- **By default, every RDD operation executes the entire lineage**
- **If an RDD will be used multiple times, persist it to avoid re-computation**
- **Persistence options**
 - Location – memory only, memory and disk , disk only
 - Format – in-memory data can be serialized to save memory (but at the cost of performance)
 - Replication – saves data on multiple nodes in case a node goes down, for job recovery without recomputation

Chapter Topics

Spark RDD Persistence

Distributed Data Processing with Spark

- RDD Lineage
- RDD Persistence Overview
- Distributed Persistence
- Conclusion
- **Hands-On Exercise: Persist an RDD**

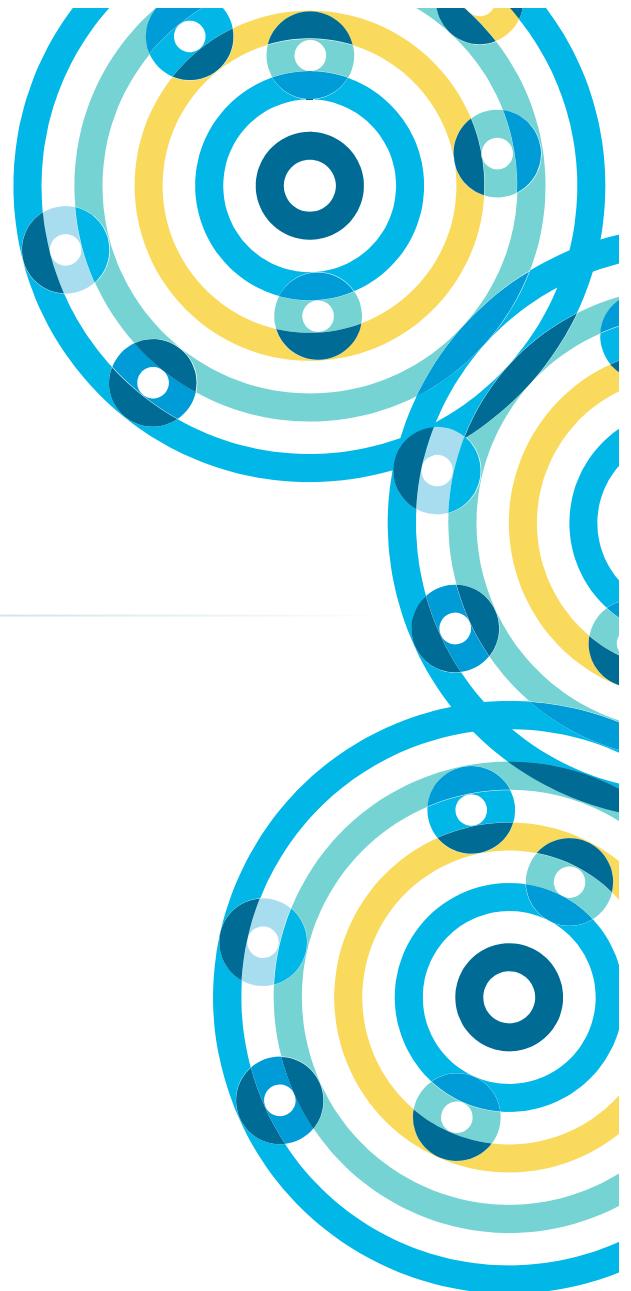
Hands-On Exercises: Persist an RDD

- **In this exercise you will**
 - Persist an RDD before reusing it
 - Use the Spark Application UI to see how an RDD is persisted
- **Please refer to the Hands-On Exercise Manual**



Common Patterns in Spark Data Processing

Chapter 16



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data File Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- **Common Patterns in Spark Data Processing**
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

Common Patterns in Spark Programming

In this chapter you will learn

- **What kinds of processing and analysis Spark is best at**
- **How to implement an iterative algorithm in Spark**
- **How GraphX and MLlib work with Spark**

Chapter Topics

Common Patterns in Spark Data Processing

Distributed Data Processing with Spark

■ Common Spark Use Cases

- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Common Spark Use Cases (1)

- **Spark is especially useful when working with any combination of:**
 - Large amounts of data
 - Distributed storage
 - Intensive computations
 - Distributed computing
 - Iterative algorithms
 - In-memory processing and pipelining

Common Spark Use Cases (2)

- Examples

- Risk analysis
 - “How likely is this borrower to pay back a loan?”
 - Recommendations
 - “Which products will this customer enjoy?”
 - Predictions
 - “How can we prevent service outages instead of simply reacting to them?”
 - Classification
 - “How can we tell which mail is spam and which is legitimate?”

Spark Examples

- Spark includes many example programs that demonstrate some common Spark programming patterns and algorithms
 - k-means
 - Logistic regression
 - Calculate pi
 - Alternating least squares (ALS)
 - Querying Apache web logs
 - Processing Twitter feeds
- Examples
 - `$SPARK_HOME/examples/lib`
 - `spark-examples-version.jar` – Java and Scala examples
 - `python.tar.gz` – Pyspark examples

Chapter Topics

Common Patterns in Spark Data Processing

Distributed Data Processing with Spark

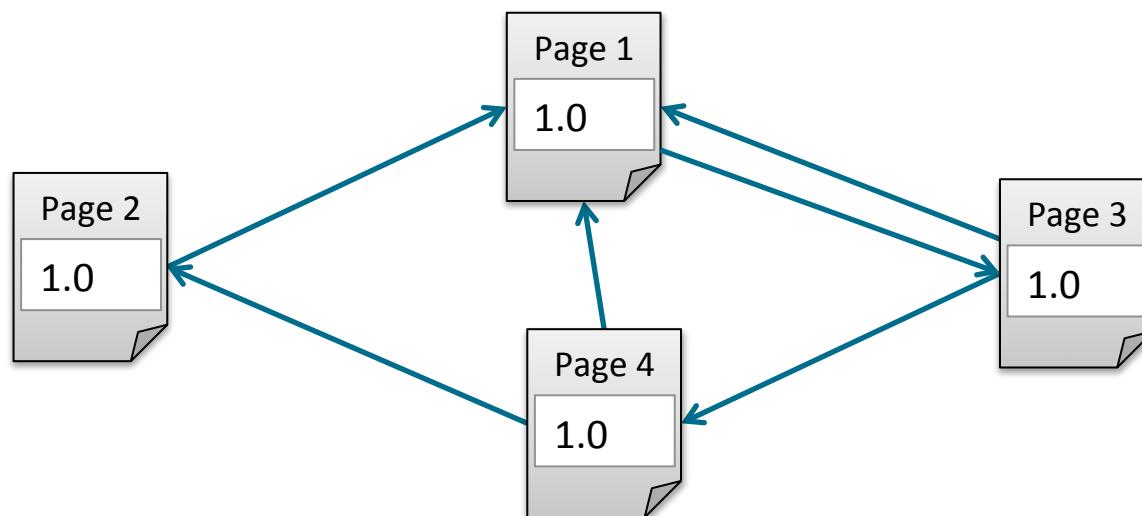
- Common Spark Use Cases
- **Iterative Algorithms in Spark**
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Example: PageRank

- **PageRank gives web pages a ranking score based on links from other pages**
 - Higher scores given for more links, and links from other high ranking pages
- **Why do we care?**
 - PageRank is a classic example of big data analysis (like WordCount)
 - Lots of data – needs an algorithm that is distributable and scalable
 - Iterative – the more iterations, the better than answer

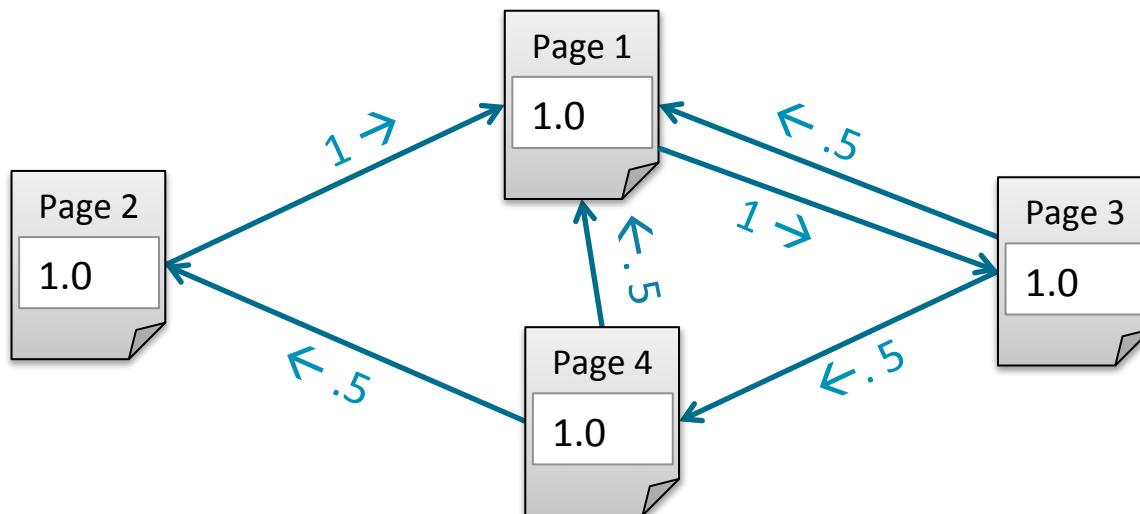
PageRank Algorithm (1)

1. Start each page with a rank of 1.0



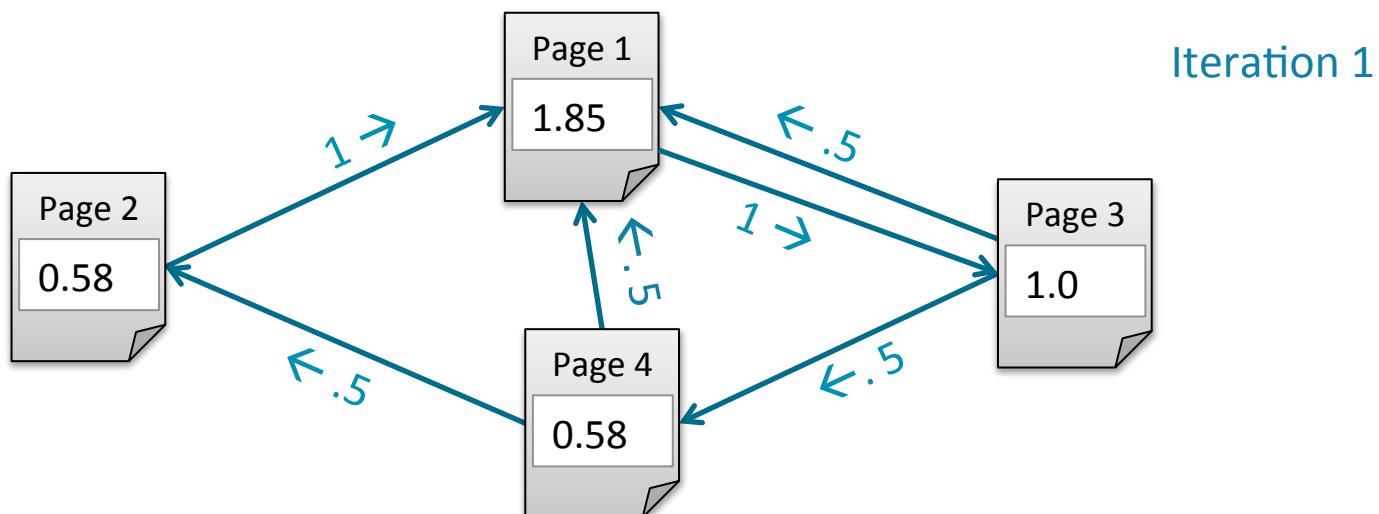
PageRank Algorithm (2)

1. Start each page with a rank of 1.0
2. On each iteration:
 1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$



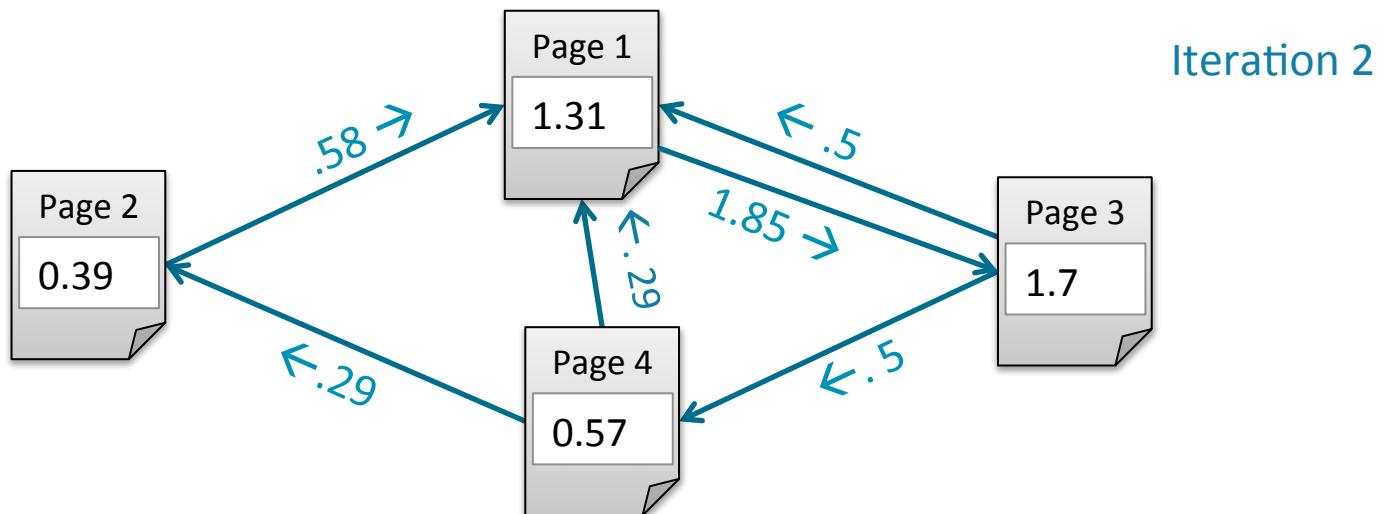
PageRank Algorithm (3)

1. Start each page with a rank of 1.0
2. On each iteration:
 1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$
 2. Set each page's new rank based on the sum of its neighbors contribution: $\text{new-rank} = \sum \text{contribs} * .85 + .15$



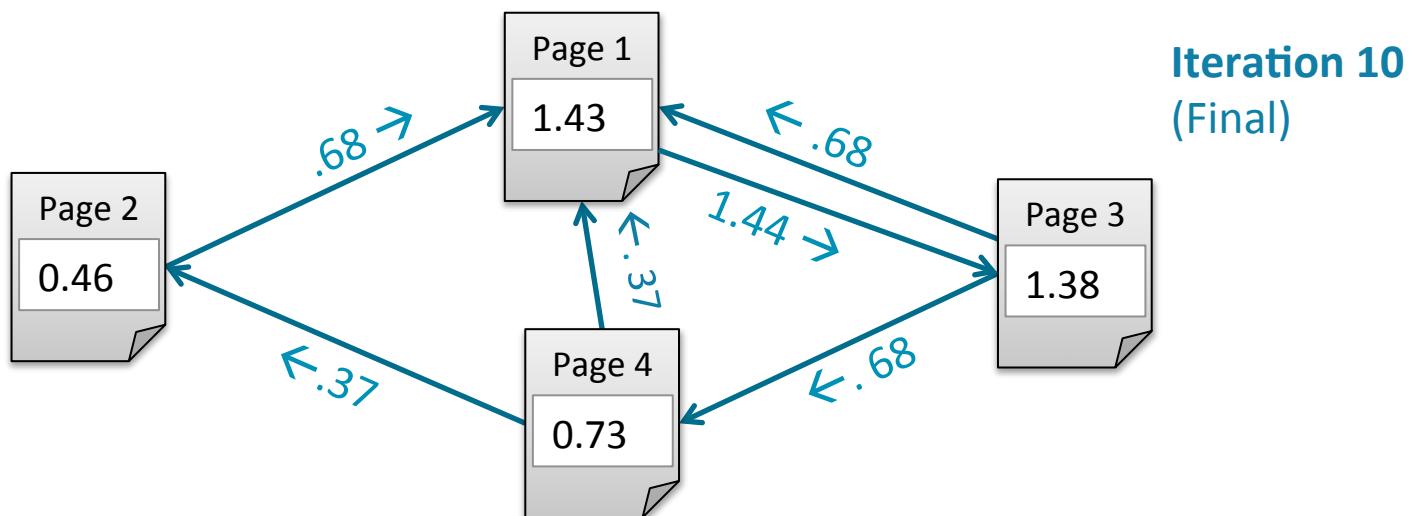
PageRank Algorithm (4)

- 1. Start each page with a rank of 1.0**
- 2. On each iteration:**
 - 1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$**
 - 2. Set each page's new rank based on the sum of its neighbors contribution: $\text{new-rank} = \sum \text{contris} * .85 + .15$**
- 3. Each iteration incrementally improves the page ranking**



PageRank Algorithm (5)

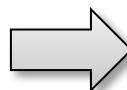
- 1. Start each page with a rank of 1.0**
- 2. On each iteration:**
 - 1. each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$**
 - 2. Set each page's new rank based on the sum of its neighbors contribution: $\text{new-rank} = \sum \text{contris} * .85 + .15$**
- 3. Each iteration incrementally improves the page ranking**



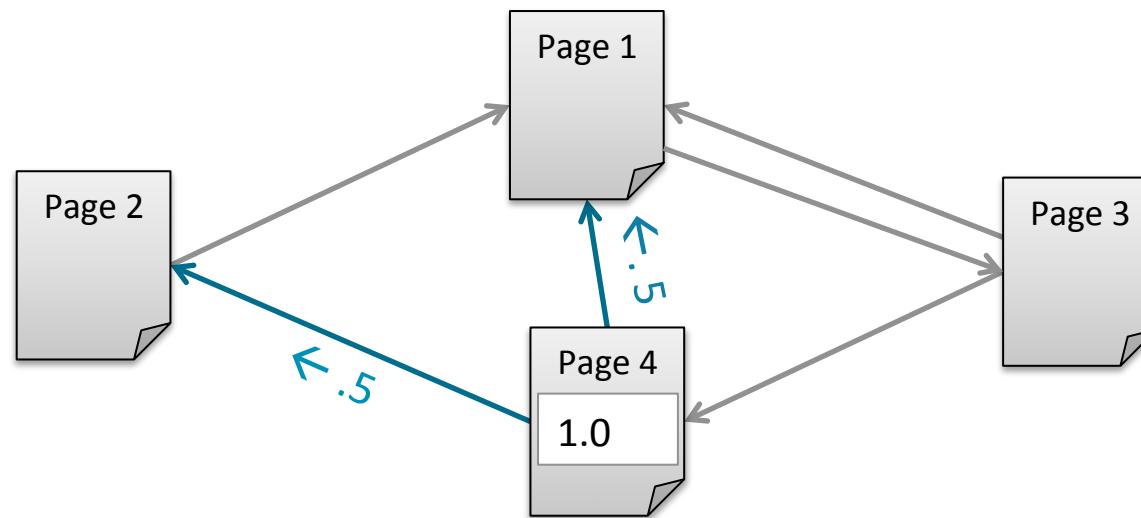
PageRank in Spark: Neighbor Contribution Function

```
def computeContribs(neighbors, rank):
    for neighbor in neighbors: yield(neighbor, rank/len(neighbors))
```

neighbors: [page1,page2]
rank: 1.0



(page1,.5)
(page2,.5)



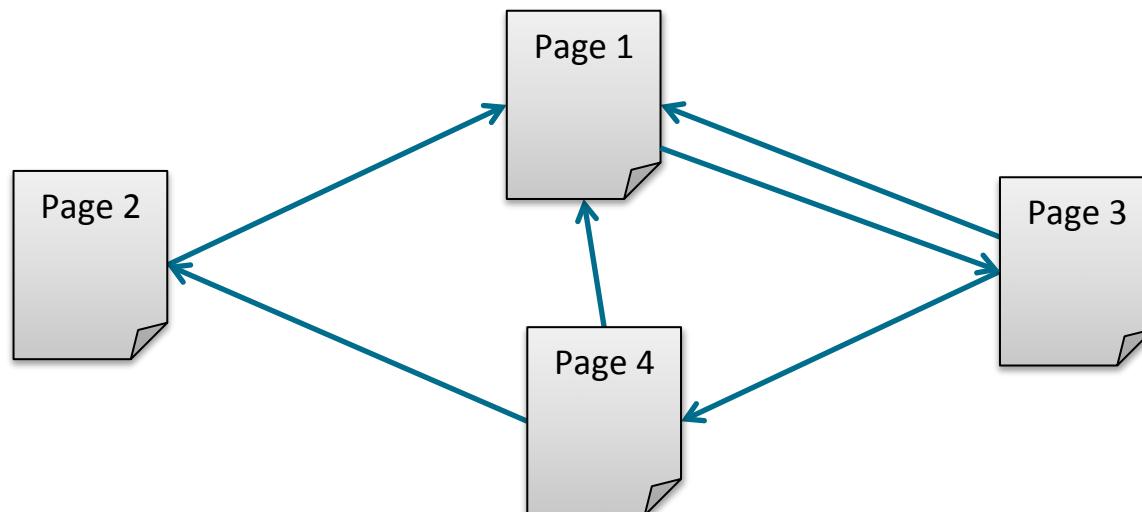
PageRank in Spark: Example Data

Data Format:

source-page destination-page

...

```
page1 page3  
page2 page1  
page4 page1  
page3 page1  
page4 page2  
page3 page4
```



PageRank in Spark: Pairs of Page Links

```
def computeContribs(neighbors, rank):...

links = sc.textFile(file) \
    .map(lambda line: line.split()) \
    .map(lambda pages: (pages[0],pages[1])) \
    .distinct()
```

page1 page3
page2 page1
page4 page1
page3 page1
page4 page2
page3 page4

(page1,page3)
(page2,page1)
(page4,page1)
(page3,page1)
(page4,page2)
(page3,page4)

PageRank in Spark: Page Links Grouped by Source Page

```
def computeContribs(neighbors, rank):...

links = sc.textFile(file) \
    .map(lambda line: line.split()) \
    .map(lambda pages: (pages[0], pages[1])) \
    .distinct() \
    .groupByKey()
```

page1 page3
page2 page1
page4 page1
page3 page1
page4 page2
page3 page4

(page1,page3)
(page2,page1)
(page4,page1)
(page3,page1)
(page4,page2)
(page3,page4)

links
(page4, [page2,page1])
(page2, [page1])
(page3, [page1,page4])
(page1, [page3])

PageRank in Spark: Persisting the Link Pair RDD

```
def computeContribs(neighbors, rank):...

links = sc.textFile(file) \
    .map(lambda line: line.split()) \
    .map(lambda pages: (pages[0], pages[1])) \
    .distinct() \
    .groupByKey() \
    .persist()
```

page1 page3
page2 page1
page4 page1
page3 page1
page4 page2
page3 page4

(page1,page3)
(page2,page1)
(page4,page1)
(page3,page1)
(page4,page2)
(page3,page4)

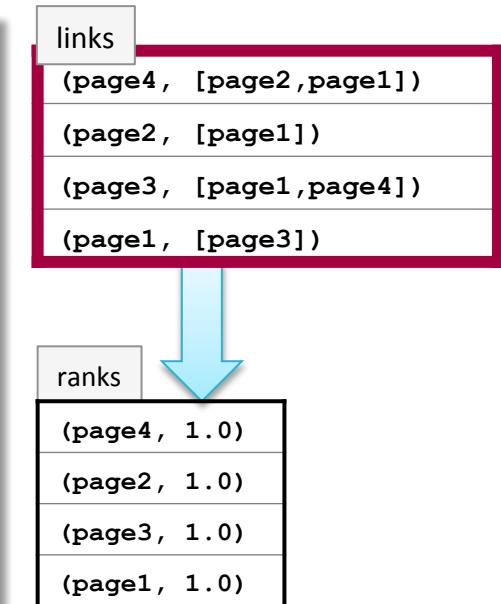
links
(page4, [page2,page1])
(page2, [page1])
(page3, [page1,page4])
(page1, [page3])

PageRank in Spark: Set Initial Ranks

```
def computeContribs(neighbors, rank):...

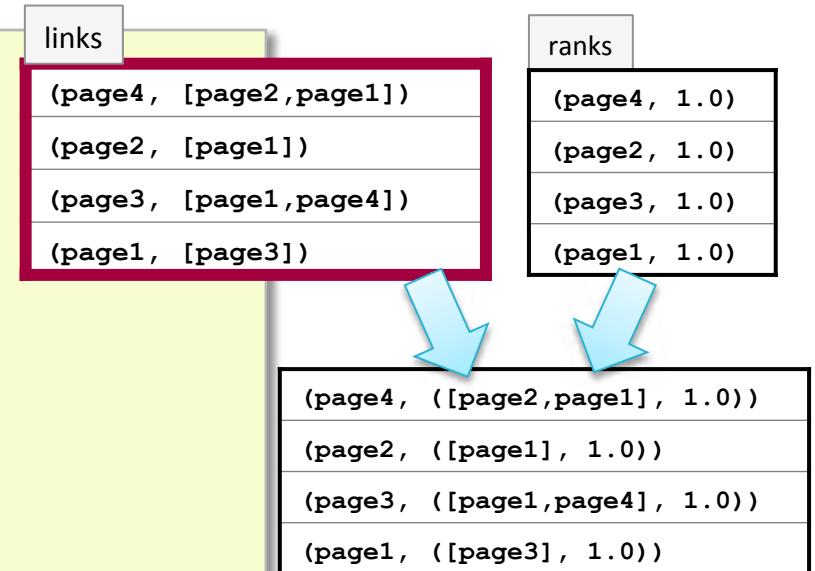
links = sc.textFile(file) \
    .map(lambda line: line.split()) \
    .map(lambda pages: (pages[0], pages[1])) \
    .distinct() \
    .groupByKey() \
    .persist()

ranks=links.map(lambda (page,neighbors): (page,1.0))
```



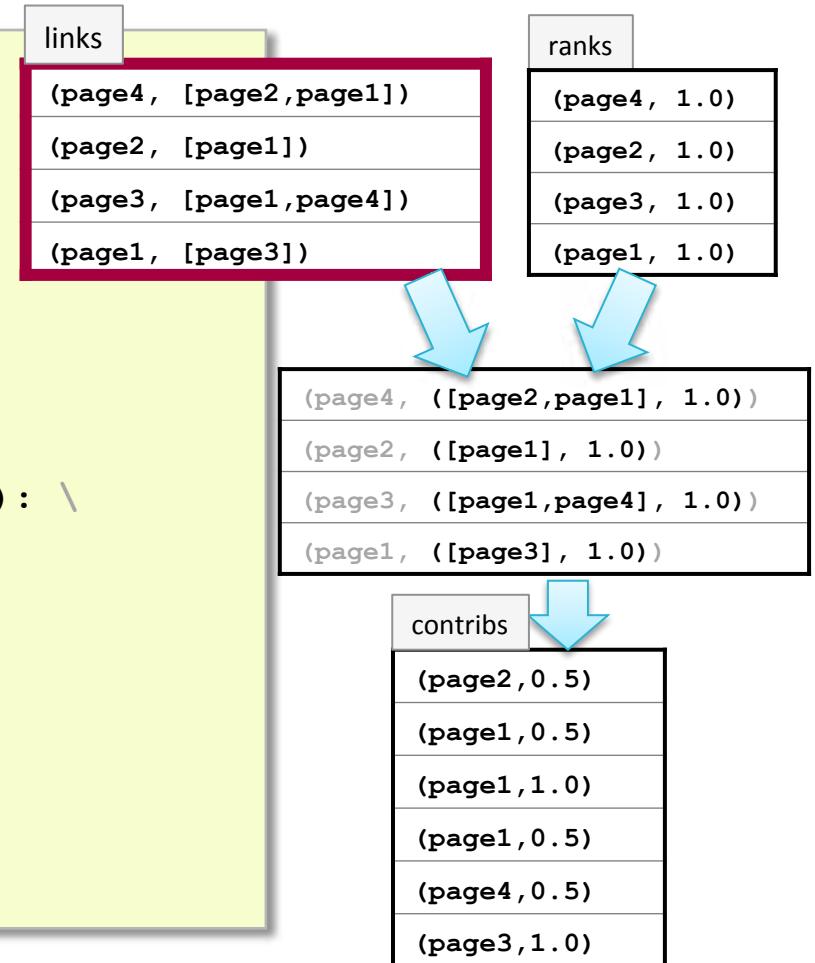
PageRank in Spark: First Iteration (1)

```
def computeContribs(neighbors, rank):...  
  
links = ...  
  
ranks = ...  
  
for x in xrange(10):  
    contribs=links\  
        .join(ranks)
```



PageRank in Spark: First Iteration (2)

```
def computeContribs(neighbors, rank):...  
  
links = ...  
  
ranks = ...  
  
for x in xrange(10):  
    contribs=links\  
        .join(ranks)\  
        .flatMap(lambda (page,(neighbors,rank)): \  
            computeContribs(neighbors,rank))
```



PageRank in Spark: First Iteration (3)

```
def computeContribs(neighbors, rank):...  
  
links = ...  
  
ranks = ...  
  
for x in xrange(10):  
    contribs=links\  
        .join(ranks)\  
        .flatMap(lambda (page, (neighbors,rank)): \  
            computeContribs(neighbors,rank))  
    ranks=contribs\  
        .reduceByKey(lambda v1,v2: v1+v2)
```

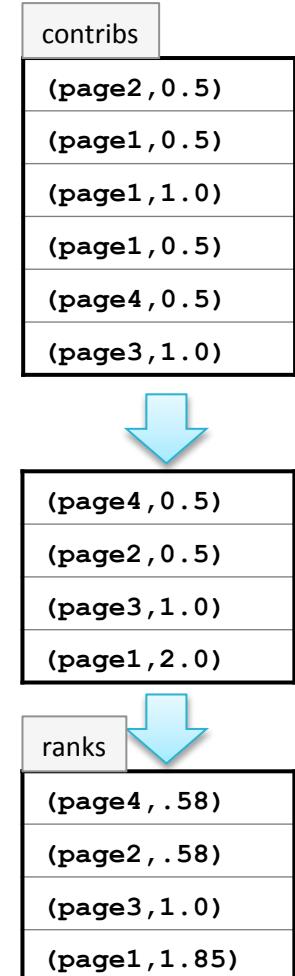
contribs
(page2, 0.5)
(page1, 0.5)
(page1, 1.0)
(page1, 0.5)
(page4, 0.5)
(page3, 1.0)

↓

(page4, 0.5)
(page2, 0.5)
(page3, 1.0)
(page1, 2.0)

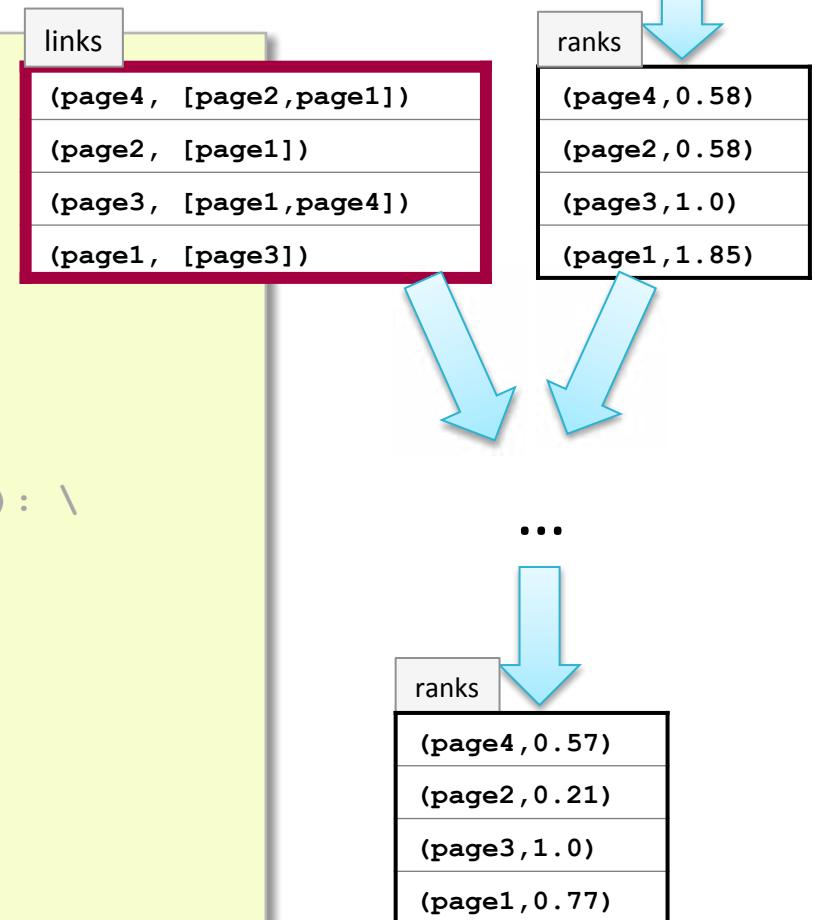
PageRank in Spark: First Iteration (4)

```
def computeContribs(neighbors, rank):...  
  
links = ...  
  
ranks = ...  
  
for x in xrange(10):  
    contribs=links\  
        .join(ranks)\\  
        .flatMap(lambda (page, (neighbors,rank)): \  
            computeContribs(neighbors,rank))  
    ranks=contribs\  
        .reduceByKey(lambda v1,v2: v1+v2)\\  
        .map(lambda (page,contrib): \  
            (page,contrib * 0.85 + 0.15))
```



PageRank in Spark: Second Iteration

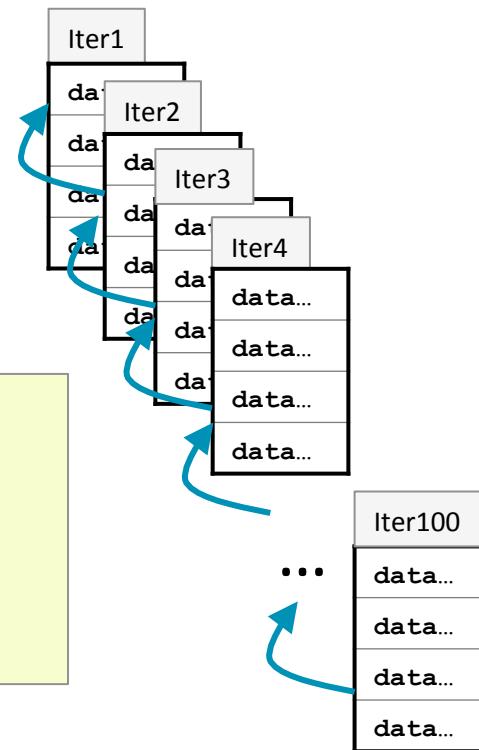
```
def computeContribs(neighbors, rank):...  
  
links = ...  
  
ranks = ...  
  
for x in xrange(10):  
    contribs=links\  
        .join(ranks)\\  
        .flatMap(lambda (page, (neighbors,rank)): \  
            computeContribs(neighbors,rank))  
    ranks=contribs\  
        .reduceByKey(lambda v1,v2: v1+v2)\\  
        .map(lambda (page,contrib): \  
            (page,contrib * 0.85 + 0.15))  
  
for rank in ranks.collect(): print rank
```



Checkpointing (1)

- Maintaining RDD lineage provides resilience but can also cause problems when the lineage gets very long
 - e.g., iterative algorithms, streaming
- Recovery can be very expensive
- Potential stack overflow

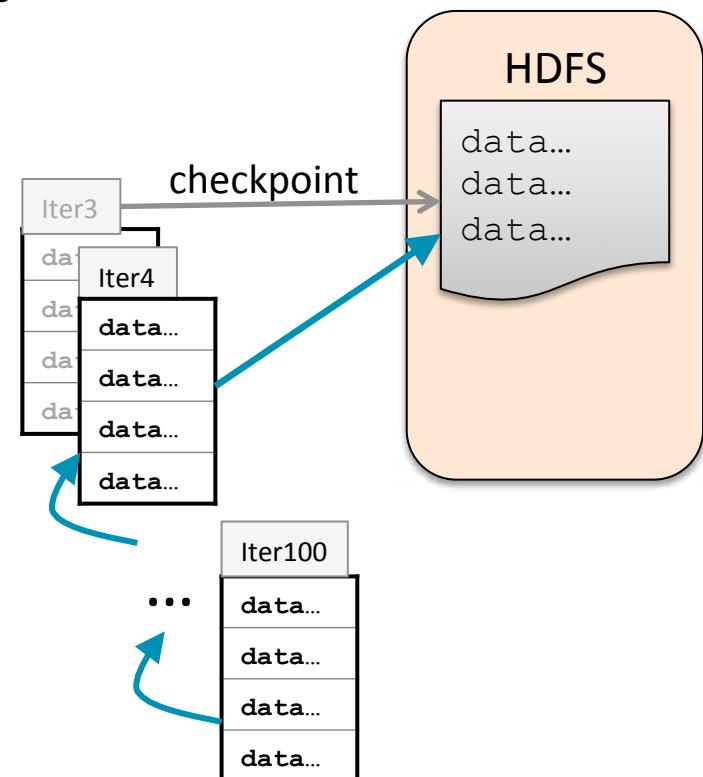
```
myrdd = ...initial-value...
while x in xrange(100):
    myrdd = myrdd.transform(...)
myrdd.saveAsTextFile(dir)
```



Checkpointing (2)

- Checkpointing saves the data to HDFS
 - Provides fault-tolerant storage across nodes
- Lineage is not saved
- Must be checkpointed before any actions on the RDD

```
sc.setCheckpointDir(directory)
myrdd = ...initial-value...
while x in xrange(100):
    myrdd = myrdd.transform(...)
    if x % 3 == 0:
        myrdd.checkpoint()
        myrdd.count()
myrdd.saveAsTextFile(dir)
```



Chapter Topics

Common Patterns in Spark Data Processing

Distributed Data Processing with Spark

- Common Spark Use Cases
- Iterative Algorithms in Spark
- **Graph Processing and Analysis**
- Machine Learning
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Graph Analytics

- **Many data analytics problems work with “data parallel” algorithms**
 - Records can be processed independently of each other
 - Very well suited to parallelizing
- **Some problems focus on the relationships between the individual data items. For example:**
 - Social networks
 - Web page hyperlinks
 - Roadmaps
- **These relationships can be represented by graphs**
 - Requires “graph parallel” algorithms

Graph Analysis Challenges at Scale

- **Graph Creation**

- Extracting relationship information from a data source
 - For example, extracting links from web pages

- **Graph Representation**

- e.g., adjacency lists in a table

- **Graph Analysis**

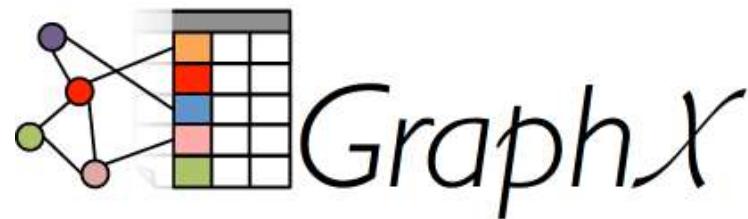
- Inherently iterative, hard to parallelize
 - This is the focus of specialized libraries like Pregel, GraphLab

- **Post-analysis processing**

- e.g., incorporating product recommendations into a retail site

Graph Analysis in Spark

- **Spark is very well suited to graph parallel algorithms**
- **GraphX**
 - UC Berkeley AMPLab project on top of Spark
 - Unifies optimized graph computation with Spark's fast data parallelism and interactive abilities
 - Supersedes predecessor Bagel (Pregel on Spark)



Chapter Topics

Common Patterns in Spark Data Processing

Distributed Data Processing with Spark

- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- **Machine Learning**
- Example: k-means
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Machine Learning

- **Most programs tell computers exactly what to do**
 - Database transactions and queries
 - Controllers
 - Phone systems, manufacturing processes, transport, weaponry, etc.
 - Media delivery
 - Simple search
 - Social systems
 - Chat, blogs, email, etc.
- **An alternative technique is to have computers *learn* what to do**
- **Machine Learning refers to programs that leverage collected data to drive future program behavior**
- **This represents another major opportunity to gain value from data**

The ‘Three Cs’

- **Machine Learning is an active area of research and new applications**
- **There are three well-established categories of techniques for exploiting data**
 - Collaborative filtering (recommendations)
 - Clustering
 - Classification

Collaborative Filtering

- Collaborative Filtering is a technique for recommendations
- Example application: given people who each like certain books, learn to suggest what someone may like in the future based on what they already like
- Helps users navigate data by expanding to topics that have affinity with their established interests
- Collaborative Filtering algorithms are agnostic to the different types of data items involved
 - Useful in many different domains

Clustering

- **Clustering algorithms discover structure in collections of data**
 - Where no formal structure previously existed
- **They discover what clusters, or groupings, naturally occur in data**
- **Examples**
 - Finding related news articles
 - Computer vision (groups of pixels that cohere into objects)

Classification

- **The previous two techniques are considered ‘unsupervised’ learning**
 - The algorithm discovers groups or recommendations itself
- **Classification is a form of ‘supervised’ learning**
- **A classification system takes a set of data records with known labels**
 - Learns how to label new records based on that information
- **Examples**
 - Given a set of e-mails identified as spam/not spam, label new e-mails as spam/not spam
 - Given tumors identified as benign or malignant, classify new tumors

Machine Learning Challenges

- Highly computation intensive and iterative
- Many traditional numerical processing systems do not scale to very large datasets
 - e.g., MatLab

MLlib: Machine Learning on Spark

- **MLlib is part of Apache Spark**
- **Includes many common ML functions**
 - ALS (alternating least squares)
 - k-means
 - Logistic Regression
 - Linear Regression
 - Gradient Descent
- **Still a ‘work in progress’**

Chapter Topics

Common Patterns in Spark Data Processing

Distributed Data Processing with Spark

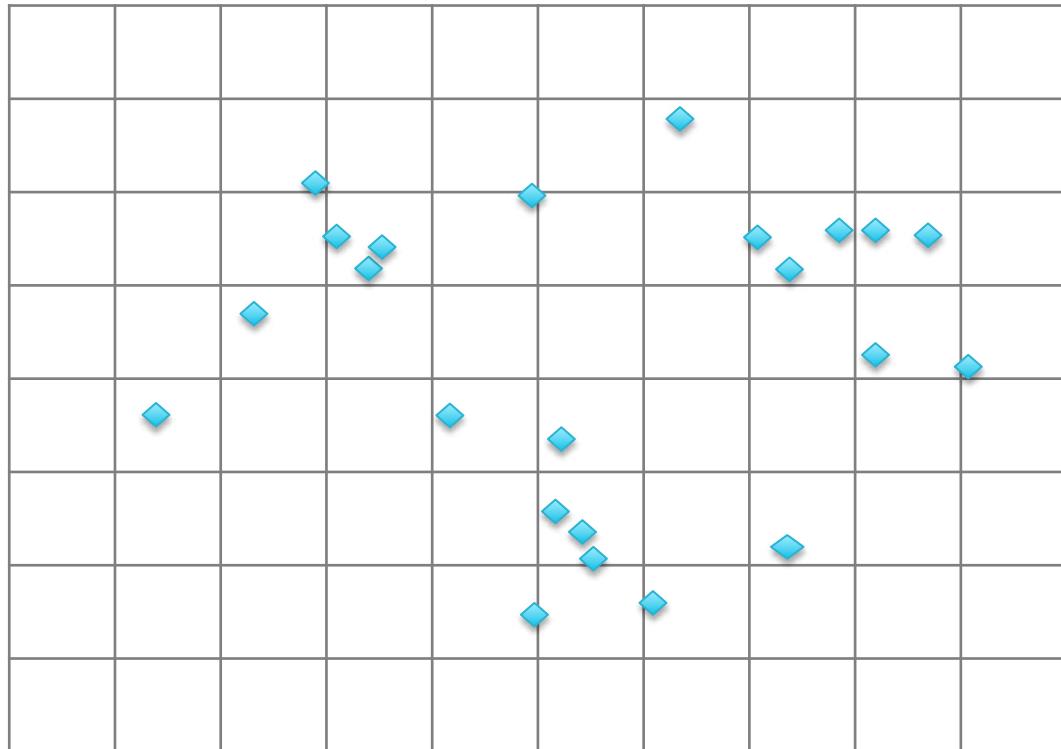
- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- **Example: k-means**
- Conclusion
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

k-means Clustering

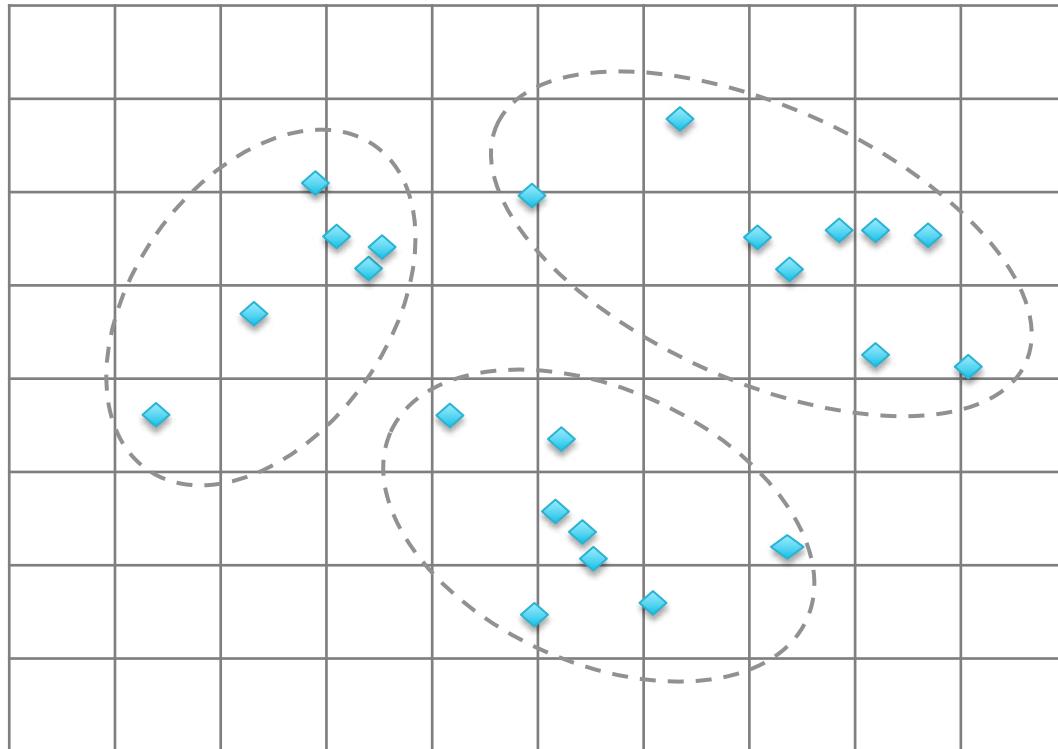
- **k-means Clustering**

- A common iterative algorithm used in graph analysis and machine learning
 - You will implement a simplified version in the Hands-On Exercises

Clustering (1)

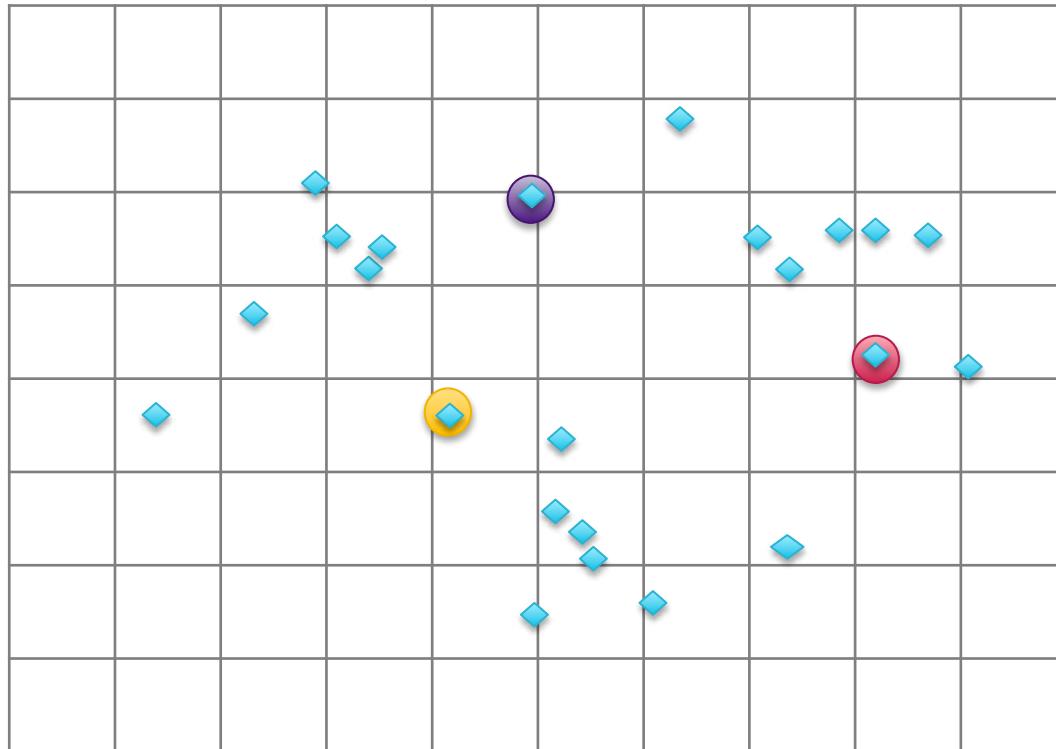


Clustering (2)



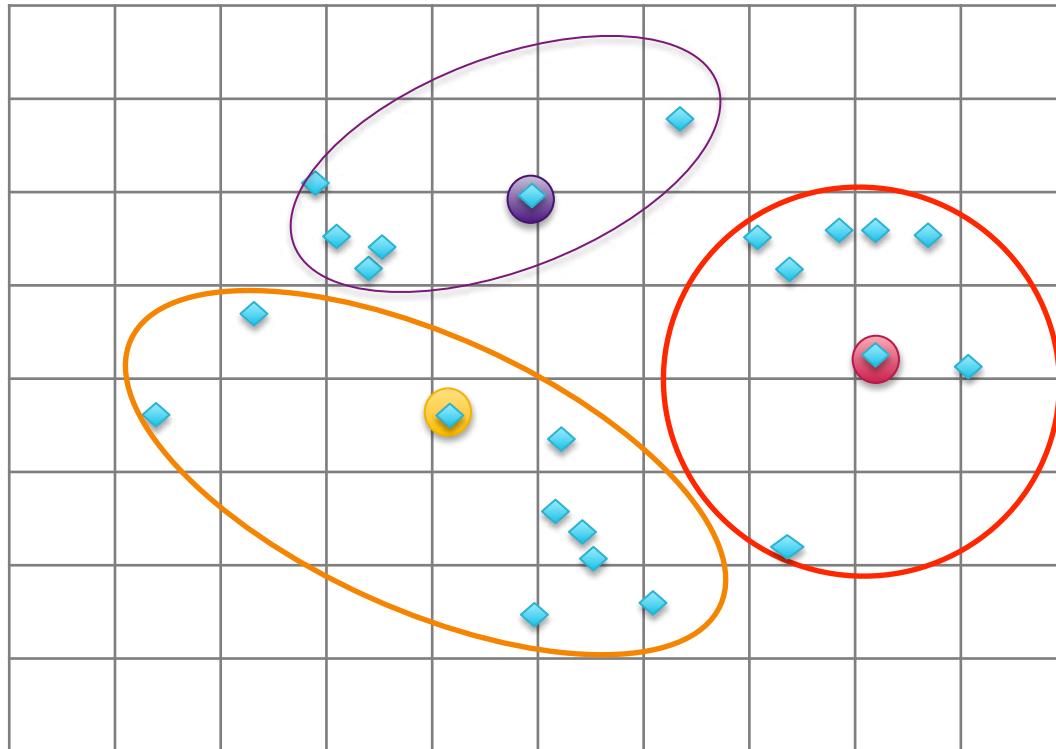
Goal: Find “clusters” of data points

Example: k-means Clustering (1)



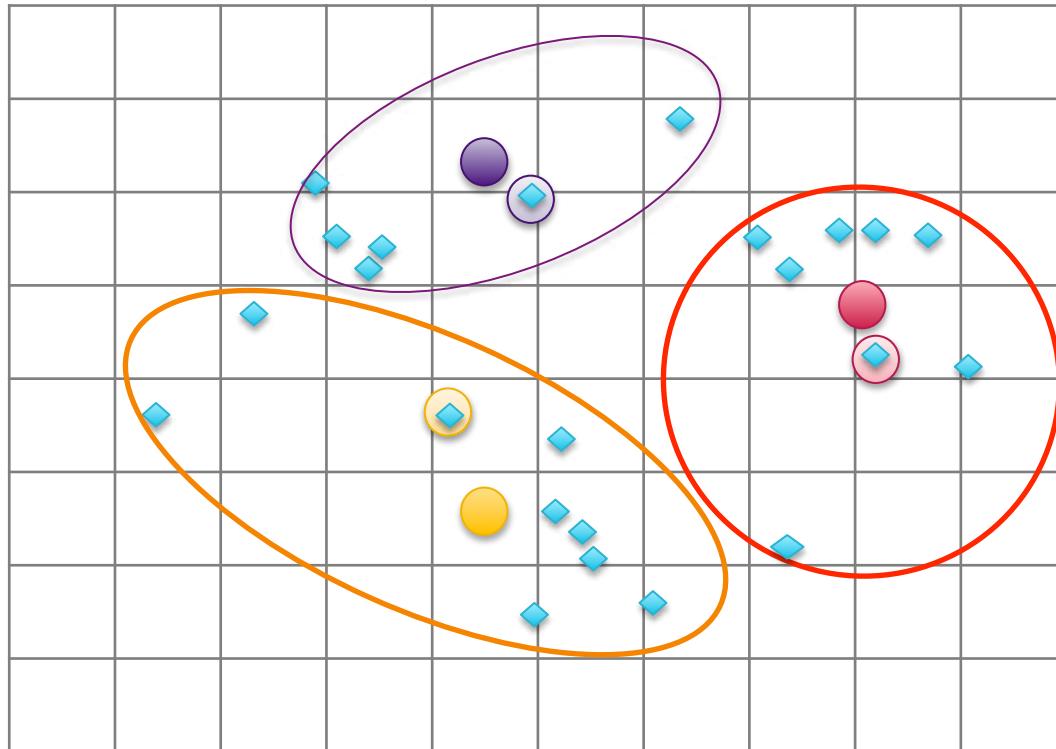
1. Choose K random points as starting centers

Example: k-means Clustering (2)



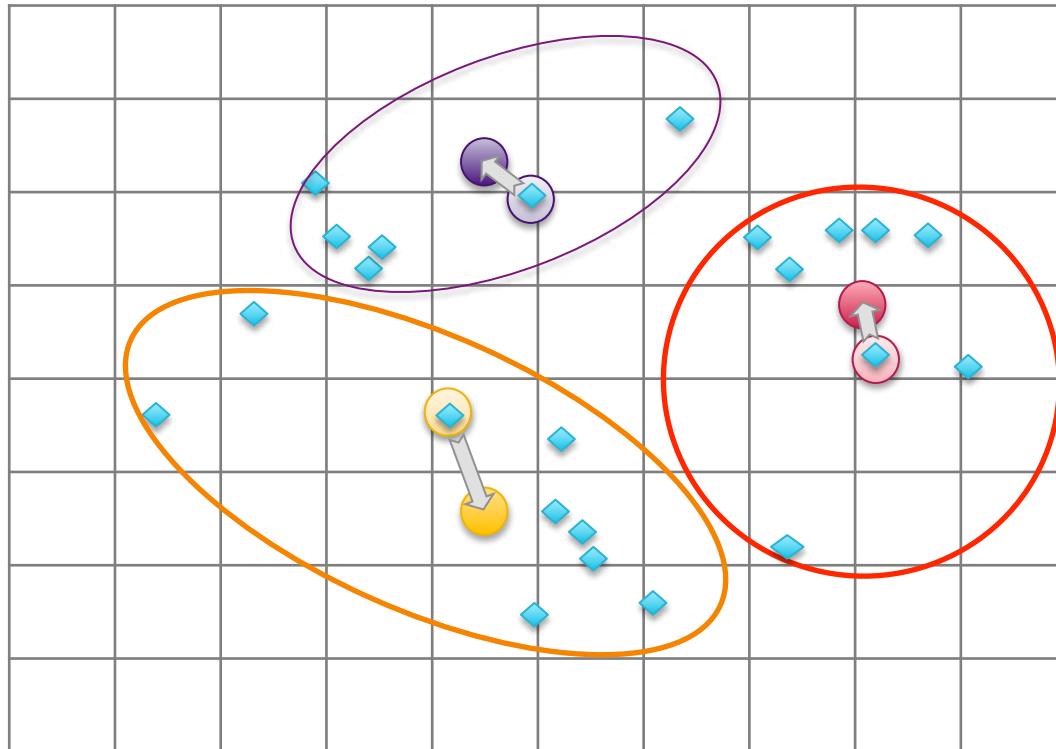
1. Choose K random points as starting centers
2. Find all points closest to each center

Example: k-means Clustering (3)



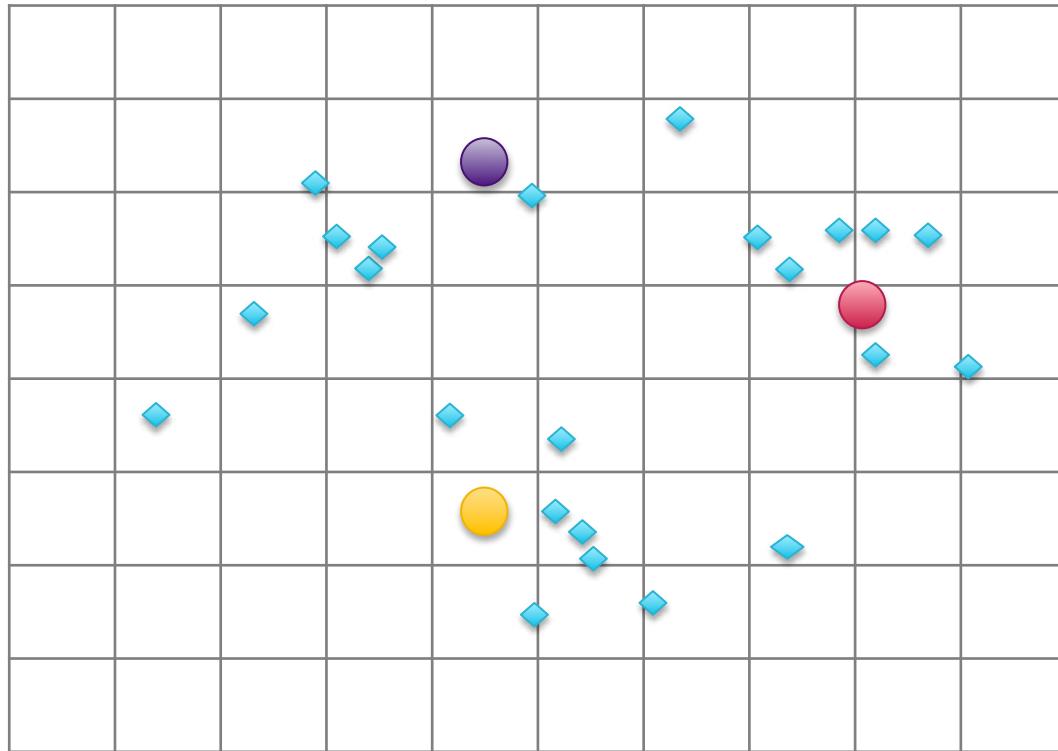
1. Choose K random points as starting centers
2. Find all points closest to each center
3. **Find the center (mean) of each cluster**

Example: k-means Clustering (4)



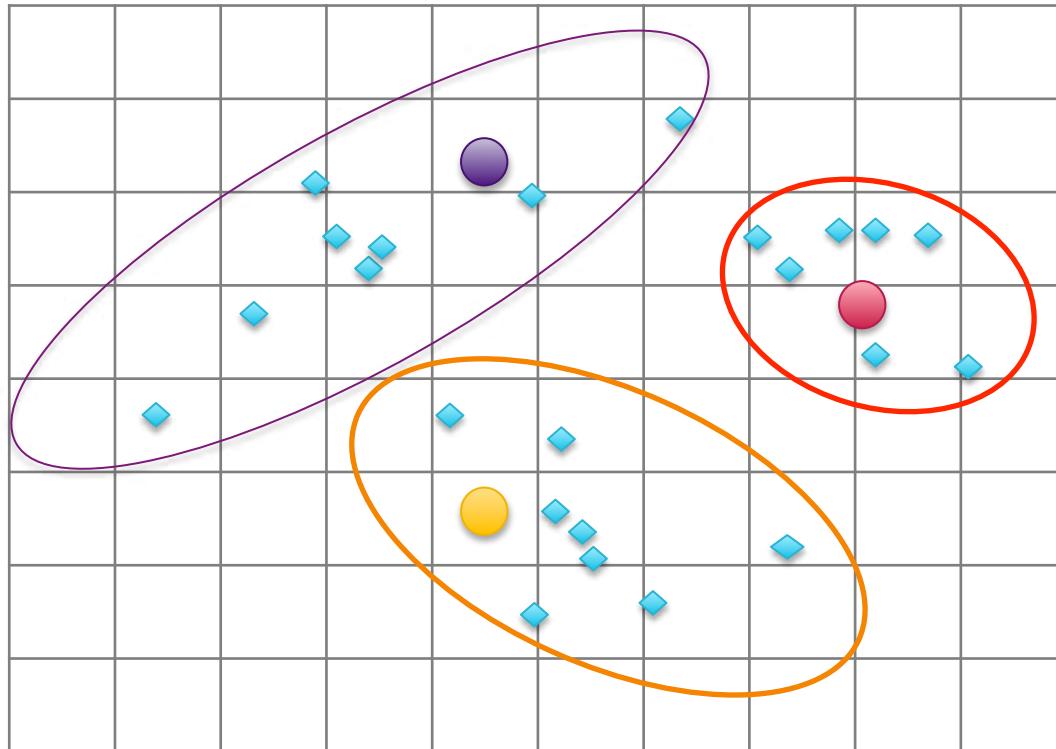
1. Choose K random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (5)



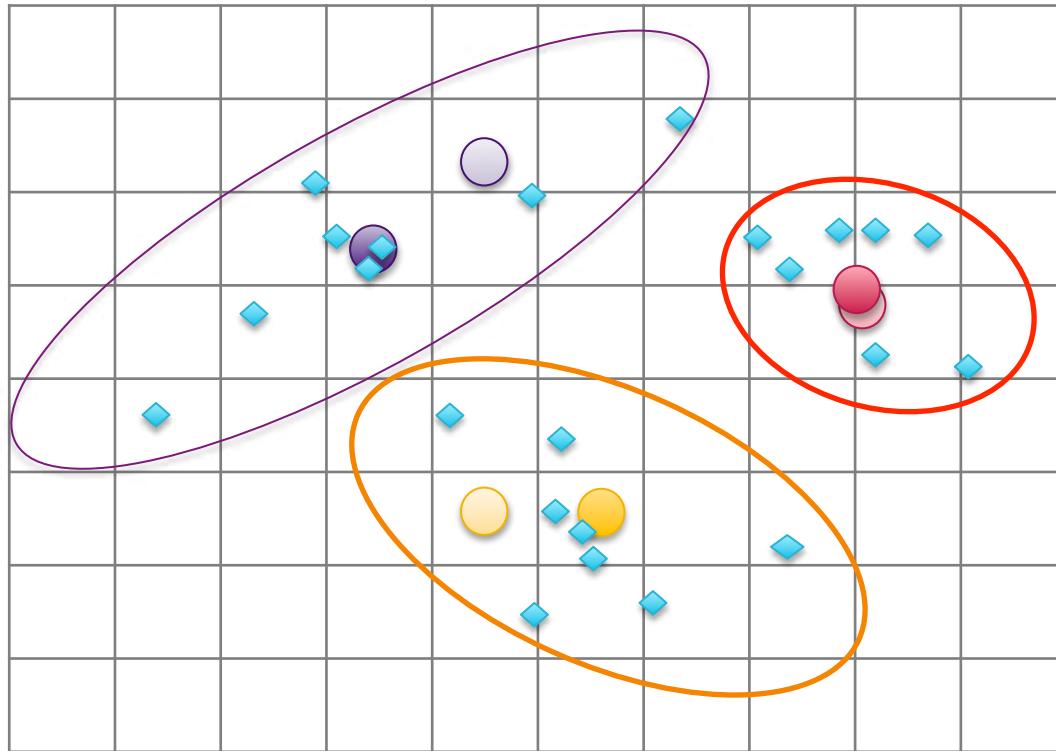
1. Choose K random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (6)



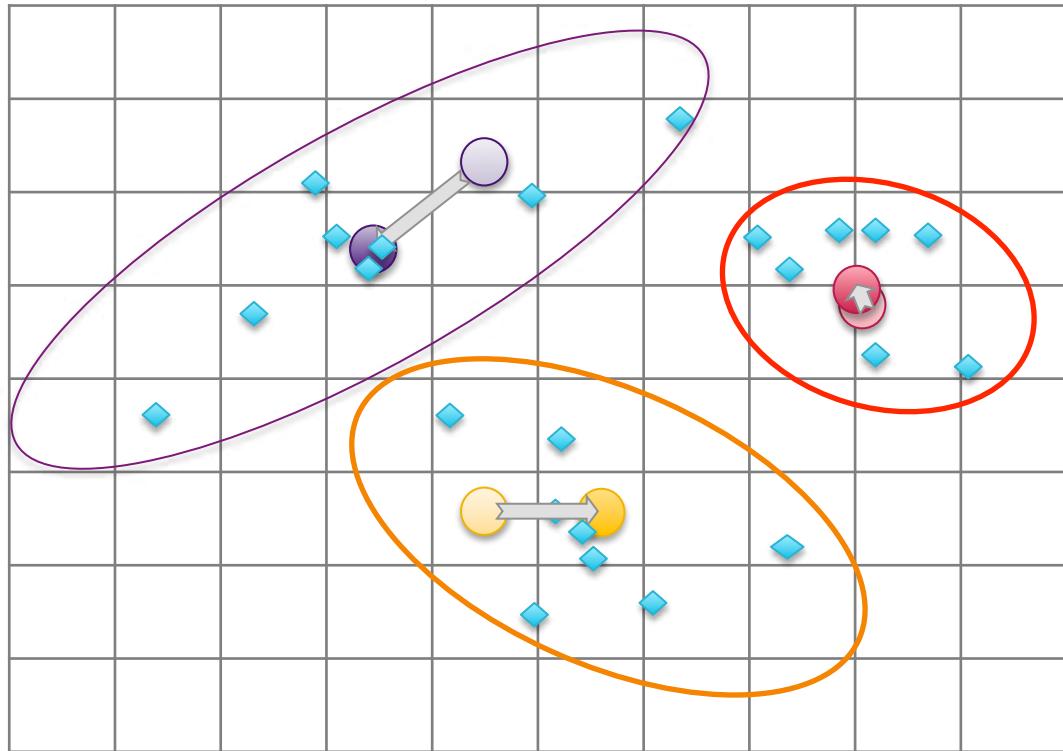
1. Choose K random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (7)



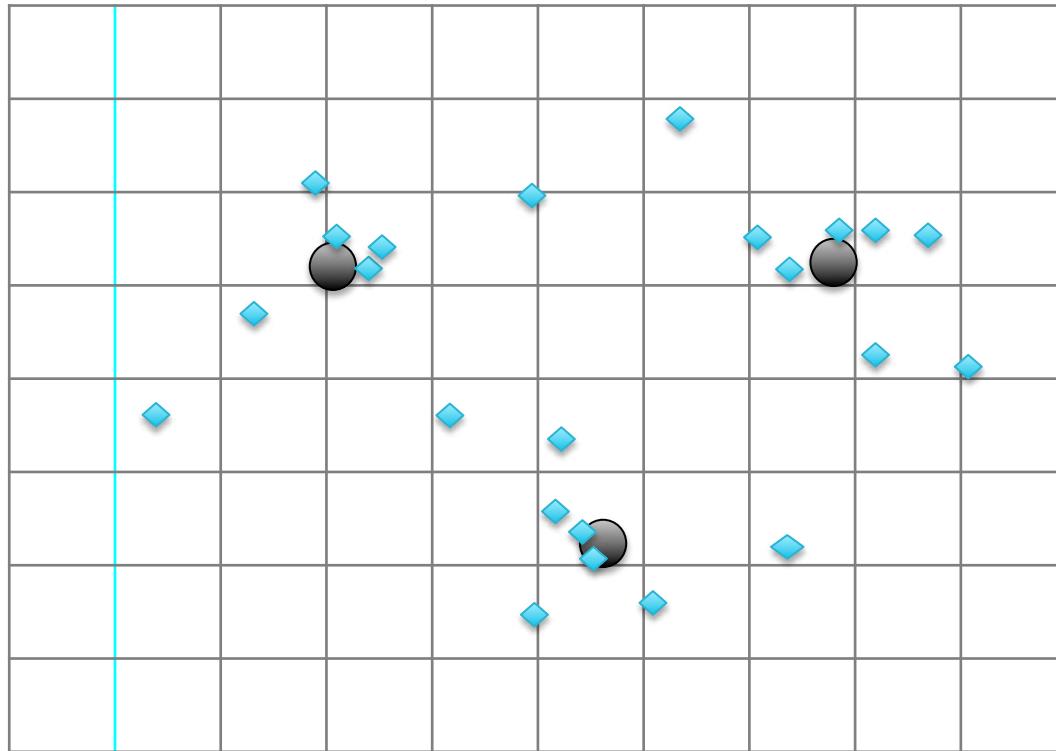
1. Choose K random points as starting centers
2. Find all points closest to each center
3. **Find the center (mean) of each cluster**
4. If the centers changed, iterate again

Example: k-means Clustering (8)



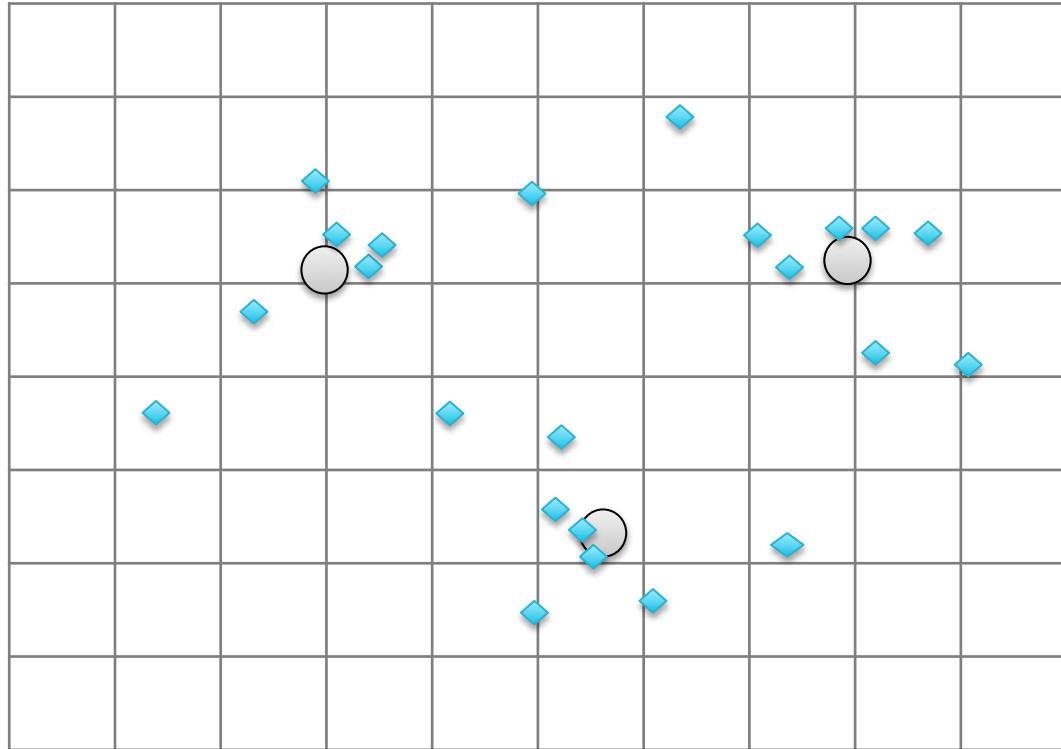
1. Choose K random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (9)



1. Choose K random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again
...
5. Done!

Example: Approximate k-means Clustering



1. Choose K random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed by more than c , iterate again
...
5. Close enough!

Chapter Topics

Common Patterns in Spark Data Processing

Distributed Data Processing with Spark

- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- **Conclusion**
- Hands-On Exercise: Implement an Iterative Algorithm with Spark
- Bonus Hands-On Exercise: Partition Data Files Using Spark

Essential Points

- **Spark is especially suited to big data problems that require iteration**
 - In-memory persistence makes this very efficient
- **Common in many types of analysis**
 - e.g., common algorithms such as PageRank and k-means
- **Spark includes specialized libraries to implement many common functions**
 - GraphX
 - MLlib
- **GraphX**
 - Highly efficient graph analysis (similar to Pregel et al.) and graph construction, representation and post-processing
- **MLlib**
 - Efficient, scalable functions for machine learning (e.g., logistic regression, k-means)

Chapter Topics

Common Patterns in Spark Data Processing

Distributed Data Processing with Spark

- Common Spark Use Cases
- Iterative Algorithms in Spark
- Graph Processing and Analysis
- Machine Learning
- Example: k-means
- Conclusion
- **Hands-On Exercise: Implement an Iterative Algorithm with Spark**
- **Bonus Hands-On Exercise: Partition Data Files Using Spark**

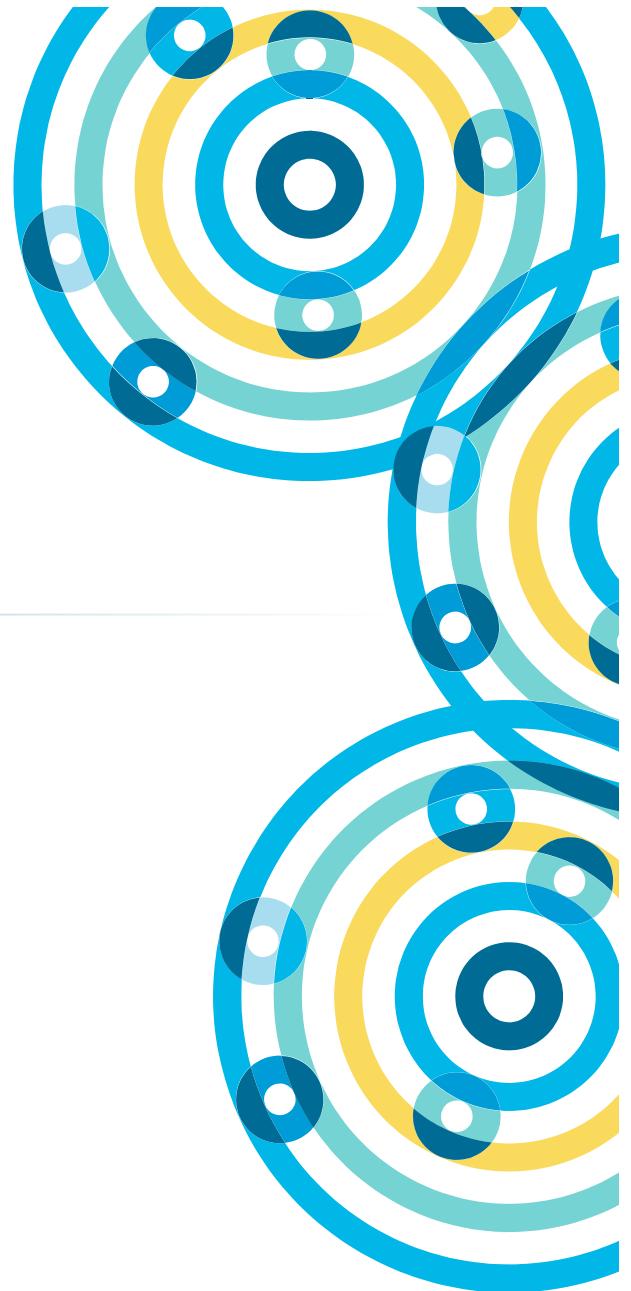
Hands-On Exercises

- **Iterative Processing in Spark**
 - In this exercise you will
 - Implement k-means in Spark in order to identify clustered location data points from Loudacre device status logs
 - Find the geographic centers of device activity
- **Bonus: Partition Data Files Using Spark**
 - In this exercise you will
 - Define “regions” according to the k-means points identified above
 - Use Spark to create a dataset for device status data, partitioned by region
- **Please refer to the Hands-On Exercise Manual**



Spark SQL and DataFrames

Chapter 17



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- **Spark SQL and DataFrames**
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured
Data

Ingesting Streaming Data

**Distributed Data Processing with
Spark**

Course Conclusion

DataFrames and SparkSQL

In this chapter you will learn

- **What Spark SQL is**
- **What features the DataFrame API provides**
- **How to create a SQLContext**
- **How to load existing data into a DataFrame**
- **How to query data in a DataFrame**
- **How to convert from DataFrames to Pair RDDs**

Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- **Spark SQL and the SQL Context**
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

What is Spark SQL?

- **What is Spark SQL?**

- Spark module for structured data processing
 - Replaces Shark (a prior Spark module, now deprecated)
 - Built on top of core Spark

- **What does Spark SQL provide?**

- The DataFrame API – a library for working with data as tables
 - Defines DataFrames containing Rows and Columns
 - DataFrames are the focus of this chapter!
 - Catalyst Optimizer – an extensible optimization framework
 - A SQL Engine and command line interface

SQL Context

- **The main Spark SQL entry point is a SQL Context object**
 - Requires a SparkContext
 - The SQL Context in Spark SQL is similar to Spark Context in core Spark
- **There are two implementations**
 - **SQLContext**
 - basic implementation
 - **HiveContext**
 - Reads and writes Hive/HCatalog tables directly
 - Supports full HiveQL language
 - Requires the Spark application be linked with Hive libraries
 - Recommended starting with Spark 1.5

Creating a SQL Context

- **SQLContext is created based on the SparkContext**

Python

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

Scala

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

Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- Spark SQL and the SQL Context
- **Creating DataFrames**
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

DataFrames

- **DataFrames are the main abstraction in Spark SQL**
 - Analogous to RDDs in core Spark
 - A distributed collection of data organized into named columns
 - Built on a base RDD containing **Row** objects

Creating DataFrames

- **DataFrames can be created**
 - From an existing structured data source (Parquet file, JSON file, etc.)
 - From an existing RDD
 - By performing an operation or query on another DataFrame
 - By programmatically defining a schema

Example: Creating a DataFrame from a JSON File

Python

```
from pyspark.sql import SQLContext
sqlCtx = SQLContext(sc)
peopleDF = sqlCtx.jsonFile("people.json")
```

Scala

```
val sqlCtx = new SQLContext(sc)
import sqlCtx._

val peopleDF = sqlCtx.jsonFile("people.json")
```

File: people.json

```
{"name": "Alice", "PCODE": "94304"}
{"name": "Brayden", "age": 30, "PCODE": "94304"}
{"name": "Carla", "age": 19, "PCODE": "10036"}
{"name": "Diana", "age": 46}
{"name": "Étienne", "PCODE": "94104"}
```



age	name	PCODE
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

Creating a DataFrame from a Data Source

- Methods on the **SQLContext** object
- Convenience functions
 - **jsonFile(filename)**
 - **parquetFile(filename)**
- Generic base function: **load**
 - **load(filename, source)** – load **filename** of type **source** (default Parquet)
 - **load(source, options...)** – load from a source of type **source** using options
 - Convenience functions are implemented by calling **load**
 - **jsonFile("people.json") = load("people.json", "json")**

Data Sources

- **Spark SQL 1.3 includes three data source types**
 - json
 - parquet
 - jdbc
- **You can also use third party data source libraries, such as**
 - Avro
 - HBase
 - CSV
 - MySQL
 - and more being added all the time

Generic Load Function Example: JDBC

- Example: Loading from a MySQL database

```
val accountsDF = sqlCtx.load("jdbc",
  Map("url" -> "jdbc:mysql://dbhost/dbname?user=...&password=..." ,
  "dbtable" -> "accounts"))
```

```
accountsDF = sqlCtx.load(source="jdbc", \
  url="jdbc:mysql://dbhost/dbname?user=...&password=..." , \
  dbtable="accounts")
```

Warning: Avoid direct access to databases in production environments, which may overload the DB or be interpreted as service attacks

- Use Sqoop to import instead

Generic Load Function Example: Third-party or Custom Sources

- You can also use custom or third party data sources
- Example: Read from an Avro file using the avro source in the Databricks Spark Avro package

```
$ spark-shell --packages com.databricks:spark-avro_2.10:1.0.0  
> ...  
> val myDF =  
sqlCtx.load("myfile.avro","com.databricks.spark.avro")
```

```
$ pyspark --packages com.databricks:spark-avro_2.10:1.0.0  
> ...  
> myDF = sqlCtx.load("myfile.avro","com.databricks.spark.avro")
```

Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- Spark SQL and the SQL Context
- Creating DataFrames
- **Transforming and Querying DataFrames**
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

DataFrame Basic Operations (1)

- **Basic Operations deal with DataFrame metadata (rather than its data), e.g.**
 - **schema** – returns a Schema object describing the data
 - **printSchema** – displays the schema as a visual tree
 - **cache / persist** – persists the DataFrame to disk or memory
 - **columns** – returns an array containing the names of the columns
 - **dtypes** – returns an array of (column-name,type) pairs
 - **explain** – prints debug information about the DataFrame to the console

DataFrame Basic Operations (2)

- Example: Displaying column data types using `dtypes`

```
> peopleDF = sqlCtx.jsonFile("people.json")
> for item in peopleDF.dtypes(): print item
('age', 'bigint')
('name', 'string')
('pcode', 'string')
```

```
> val peopleDF = sqlCtx.jsonFile("people.json")
> peopleDF.dtypes.foreach(println)
(age,LongType)
(name,StringType)
(pcode,StringType)
```

Working with Data in a DataFrame

- **Queries – create a new DataFrame**
 - DataFrames are immutable
 - Queries are analogous to RDD transformations
- **Actions – return data to the Driver**
 - Actions trigger “lazy” execution of queries

DataFrame Actions

- Some DataFrame actions

- **collect** – return all rows as an array of **Row** objects
- **take (n)** – return the first **n** rows as an array of **Row** objects
- **count** – return the number of rows
- **show (n)** – display the first **n** rows (default=20)

```
> peopleDF.count()
5L

> peopleDF.show(3)
age  name    pcode
null Alice  94304
30   Brayden 94304
19   Carla   10036
```

```
> peopleDF.count()
res7: Long = 5

> peopleDF.show(3)
age  name    pcode
null Alice  94304
30   Brayden 94304
19   Carla   10036
```

DataFrame Queries (1)

- **DataFrame query methods return new DataFrames**
 - Queries can be chained like transformations
- **Some query methods**
 - **distinct** – returns a new DataFrame with distinct elements of this DF
 - **join** – joins this DataFrame with a second DataFrame
 - several variants for inside, outside, left, right, etc.
 - **limit** – a new DF with the first **n** rows of this DataFrame
 - **select** – a new DataFrame with data from one or more columns of the base DataFrame
 - **filter** – a new DataFrame with rows meeting a specified condition

DataFrame Queries (2)

- Example: A basic query with limit

```
> peopleDF.limit(3).show
```

```
> peopleDF.limit(3).show()
```

Output
of show

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

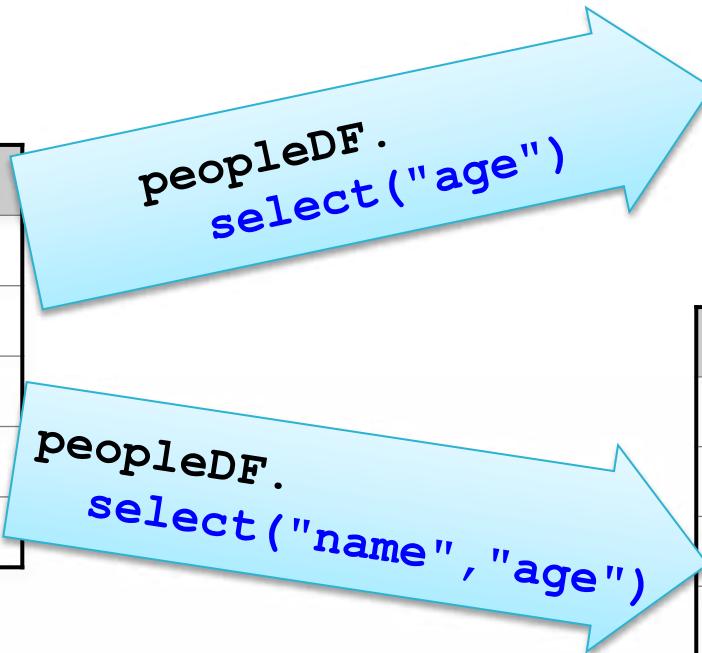


age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036

DataFrame Query Strings (1)

- Some query operations take strings containing simple query expressions
 - Such as `select` and `where`
- Example: `select`

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



age
null
30
19
46
null

name	age
Alice	null
Brayden	30
Carla	19
Diana	46
Étienne	null

DataFrame Query Strings (2)

- Example: `where`



```
peopleDF.  
where("age > 21")
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

age	name	pcode
30	Brayden	94304
46	Diana	null

Querying DataFrames using Columns (1)

- Some DF queries take one or more *columns* or *column expressions*
 - Required for more sophisticated operations
- Some examples
 - `select`
 - `sort`
 - `join`
 - `where`

Querying DataFrames using Columns (2)

- In Python, reference columns by name using *dot notation*

```
ageDF = peopleDF.select(peopleDF.age)
```

- In Scala, columns can be referenced in two ways

```
val ageDF = peopleDF.select($"age")
```

- OR

```
val ageDF = peopleDF.select(peopleDF("age"))
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



age
null
30
19
46
null

Querying DataFrames using Columns (3)

- Column references can also be *column expressions*

```
peopleDF.select(peopleDF.name, peopleDF.age+10)
```

```
peopleDF.select(peopleDF("name"), peopleDF("age") + 10)
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



name	age+10
Alice	null
Brayden	40
Carla	29
Diana	56
Étienne	null

Querying DataFrames using Columns (4)

- Example: Sorting in by columns (descending)

```
peopleDF.sort(peopleDF.age.desc())
```

```
peopleDF.sort(peopleDF("age").desc)
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



age	name	pcode
46	Diana	null
30	Brayden	94304
19	Carla	10036
null	Alice	94304
null	Étienne	94104

.asc and .desc
are column expression
methods used with
sort

SQL Queries

- Spark SQL also supports the ability to perform SQL queries
 - First, register the DataFrame as a “table” with the SQL Context

```
peopleDF.registerTempTable("people")
sqlCtx.sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```

```
peopleDF.registerTempTable("people")
sqlCtx.sql("""SELECT * FROM people WHERE name LIKE "A%" """)
```

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104



age	name	pcode
null	Alice	94304

Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- **Saving DataFrames**
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

Saving DataFrames

- Data in DataFrames can be saved to a data source
 - Built in support for JDBC and Parquet File
 - `createJDBCTable` – create a new table in a database
 - `insertInto` – save to an existing table in a database
 - `saveAsParquetFile` – save as a Parquet file (including schema)
 - `saveAsTable` – save as a Hive table (HiveContext only)
 - Can also use third party and custom data sources
 - `save` – generic base function

Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- **DataFrames and RDDs**
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

DataFrames and RDDs (1)

- **DataFrames are built on RDDs**
 - Base RDDs contain **Row** objects
 - Use **rdd** to get the underlying RDD

```
peopleRDD = peopleDF.rdd
```

peopleDF

age	name	pcode
null	Alice	94304
30	Brayden	94304
19	Carla	10036
46	Diana	null
null	Étienne	94104

peopleRDD

Row[null, Alice, 94304]
Row[30, Brayden, 94304]
Row[19, Carla, 10036]
Row[46, Diana, null]
Row[null, Étienne, 94104]

DataFrames and RDDs (2)

- Row RDDs have all the standard Spark actions and transformations
 - Actions – `collect`, `take`, `count`, etc.
 - Transformations – `map`, `flatMap`, `filter`, etc.
- Row RDDs can be transformed into PairRDDs to use map-reduce methods

Working with Row Objects

- The syntax for extracting data from Rows depends on language
- Python
 - Column names are object attributes
 - `row.age` – return age column value from row
- Scala
 - Use Array-like syntax
 - `row(0)` – returns element in the first column
 - `row(1)` – return element in the second column
 - etc.
 - Use type-specific `get` methods to return typed values
 - `row.getString(n)` – returns nth column as a String
 - `row.getInt(n)` – returns nth column as an Integer
 - etc.

Example: Extracting Data from Rows

- Extract data from Rows

```
peopleRDD = peopleDF.rdd  
peopleByPCode = peopleRDD \  
.map(lambda row(row.pcode, row.name)) \  
.groupByKey()
```

```
val peopleRDD = peopleDF.rdd  
peopleByPCode = peopleRDD.  
map(row => (row(2), row(1))).  
groupByKey()
```

Row[null, Alice, 94304]
Row[30, Brayden, 94304]
Row[19, Carla, 10036]
Row[46, Diana, null]
Row[null, Étienne, 94104]

(94304, Alice)
(94304, Brayden)
(10036, Carla)
(null, Diana)
(94104, Étienne)

(null, [Diana])
(94304, [Alice, Brayden])
(10036, [Carla])
(94104, [Étienne])

Converting RDDs to DataFrames

- You can also create a DF from an RDD
 - `sqlCtx.createDataFrame(rdd)`

Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- **Comparing Spark SQL, Impala and Hive-on-Spark**
- Conclusion
- Hands-On Exercises: Use Spark SQL for ETL

Comparing Impala to Spark SQL

- **Spark SQL is built on Spark, a *general purpose* processing engine**
 - Provides convenient SQL-like access to structured data in a Spark application
- **Impala is a *specialized* SQL engine**
 - Much better performance for querying
 - Much more mature than Spark SQL
 - Robust security via Sentry
- **Impala is better for**
 - Interactive queries
 - Data analysis
- **Use Spark SQL for**
 - ETL
 - Access to structured data required by a Spark application



Comparing Spark SQL with Hive on Spark

- **Spark SQL**

- Provides the DataFrame API to allow structured data processing *in a Spark application*
 - Programmers can mix SQL with procedural processing

- **Hive-on-Spark**

- Hive provides a SQL abstraction layer over MapReduce or Spark
 - Allows non-programmers to analyze data using familiar SQL
 - Hive-on-Spark replaces MapReduce as the engine underlying Hive
 - Does not affect the user experience of Hive
 - Except many times faster queries!



Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- **Conclusion**
- Hands-On Exercises: Use Spark SQL for ETL

Essential Points

- **Spark SQL is a Spark API for handling structured and semi-structured data**
- **Entry point is a SQLContext**
- **DataFrames are the key unit of data**
- **DataFrames are based on an underlying RDD of Row objects**
- **DataFrames query methods return new DataFrames; similar to RDD transformations**
- **The full Spark API can be used with Spark SQL Data by accessing the underlying RDD**
- **Spark SQL is not a replacement for a database, or a specialized SQL engine like Impala**
 - Spark SQL is most useful for ETL or incorporating structured data into other applications

Chapter Topics

Spark SQL and DataFrames

Distributed Data Processing with Spark

- Spark SQL and the SQL Context
- Creating DataFrames
- Transforming and Querying DataFrames
- Saving DataFrames
- DataFrames and RDDs
- Comparing Spark SQL, Impala and Hive-on-Spark
- Conclusion
- **Hands-On Exercises: Use Spark SQL for ETL**

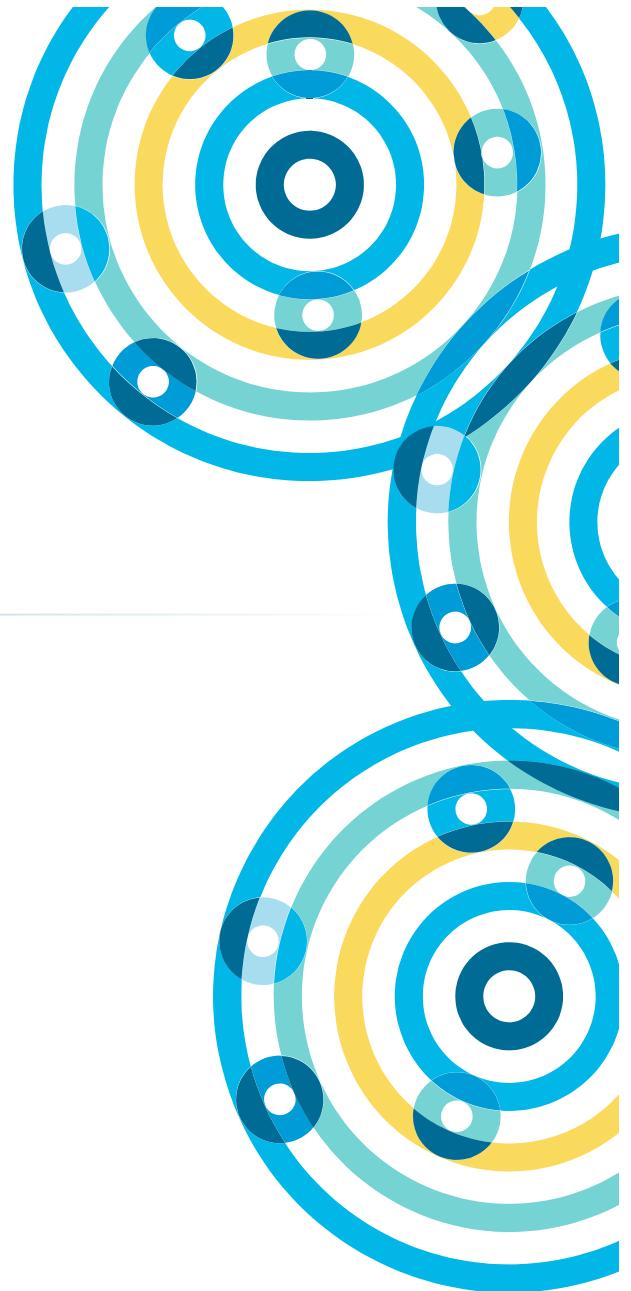
Hands-On Exercise: Use Spark SQL for ETL

- **In this exercise you will**
 - Import the data from MySQL
 - Use Spark to normalize the data
 - Save the data to Parquet format
 - Query the data with Impala or Hive
- **Please refer to the Hands-On Exercise Manual**



Conclusion

Chapter 18



Course Chapters

- Introduction
- Introduction to Hadoop and the Hadoop Ecosystem
- Hadoop Architecture and HDFS
- Importing Relational Data with Apache Sqoop
- Introduction to Impala and Hive
- Modeling and Managing Data with Impala and Hive
- Data Formats
- Data Partitioning
- Capturing Data with Apache Flume
- Spark Basics
- Working with RDDs in Spark
- Aggregating Data with Pair RDDs
- Writing and Deploying Spark Applications
- Parallel Processing in Spark
- Spark RDD Persistence
- Common Patterns in Spark Data Processing
- Spark SQL and DataFrames
- Conclusion

Course Introduction

Introduction to Hadoop

Importing and Modeling Structured Data

Ingesting Streaming Data

Distributed Data Processing with Spark

Course Conclusion

Course Objectives

During this course, you have learned

- **How the Hadoop Ecosystem fits in with the data processing lifecycle**
- **How data is distributed, stored and processed in a Hadoop cluster**
- **How to use Sqoop and Flume to ingest data**
- **How to model structured data as tables in Impala and Hive**
- **Best practices for data storage**
- **How to choose a data storage format for your data usage patterns**
- **How to process distributed data with Spark**

What's Next? (1)

- **Additional training courses you may wish to consider**
 - *Developer Training for Spark and Hadoop II: Advanced Techniques*
 - The follow-on to this course!
 - *Cloudera Administrator Training for Apache Hadoop*
 - *Cloudera Data Analyst Training: Using Pig, Hive, and Impala with Hadoop*
 - *Cloudera Training for Apache HBase*
 - *Introduction to Data Science: Building Recommender Systems*
 - *Cloudera Search Training*
- **Onsite and custom training is also available**
 - <http://go.cloudera.com/privatetrainingrequest.html>

What's Next? (2)

cloudera

DEVELOPER PROGRAM

- **Ramp-up on Apache Hadoop at a low annual cost! Includes:**
 - Prioritized access to professional guidance from Cloudera engineers
 - Complete Cloudera Enterprise Data Hub Edition license (software only)
 - Discounts and other perks
- **Subscribe at cloudera.com/developer**

cloudera®

