

CLOUDERA

Educational Services

Developer Training for Apache Spark and Hadoop



Introduction

Chapter 1

Course Chapters

- **Introduction**
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Trademark Information

- The names and logos of Apache products mentioned in Cloudera training courses, including those listed below, are trademarks of the Apache Software Foundation

Apache Accumulo	Apache Hive	Apache Pig
Apache Avro	Apache Impala	Apache Ranger
Apache Ambari	Apache Kafka	Apache Sentry
Apache Atlas	Apache Knox	Apache Solr
Apache Bigtop	Apache Kudu	Apache Spark
Apache Crunch	Apache Lucene	Apache Sqoop
Apache Druid	Apache Mahout	Apache Storm
Apache Flink	Apache NiFi	Apache Tez
Apache Flume	Apache Oozie	Apache Tika
Apache Hadoop	Apache ORC	Apache Zeppelin
Apache HBase	Apache Parquet	Apache ZooKeeper
Apache HCatalog	Apache Phoenix	

- All other product names, logos, and brands cited herein are the property of their respective owners

Chapter Topics

Introduction

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

Course Objectives

During this course, you will learn

- How the Apache Hadoop ecosystem fits in with the data processing lifecycle
- How data is distributed, stored, and processed in a Hadoop cluster
- How to write, configure, and deploy Apache Spark applications on a Hadoop cluster
- How to use the Spark shell and Spark applications to explore, process, and analyze distributed data
- How to query data using Spark SQL, DataFrames, and Datasets
- How to use Spark Streaming to process a live data stream

Chapter Topics

Introduction

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

Introductions

- **About your instructor**
- **About you**
 - Currently, what do you do at your workplace?
 - What is your experience with database technologies, programming, and query languages?
 - How much experience do you have with UNIX or Linux?
 - What is your experience with big data?
 - What do you expect to gain from this course? What would you like to be able to do at the end that you cannot do now?

Chapter Topics

Introduction

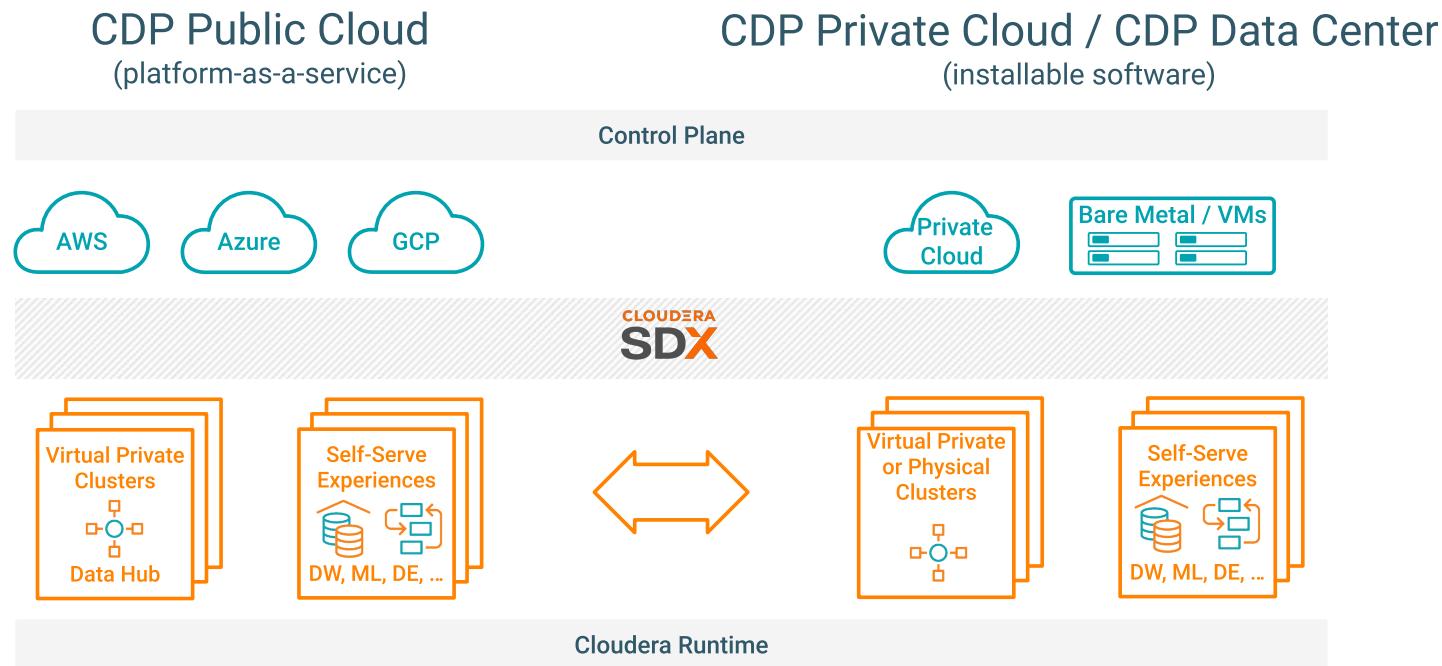
- About This Course
- Introductions
- **About Cloudera**
- About Cloudera Educational Services
- Course Logistics

About Cloudera



- **Cloudera (founded 2008) and Hortonworks (founded 2011) merged in 2019**
- **The new Cloudera improves on the best of both companies**
 - Introduced the world's first Enterprise Data Cloud
 - Delivers a comprehensive platform for any data from the Edge to AI
 - Leads in training, certification, support, and consulting for data professionals
 - Remains committed to open source and open standards

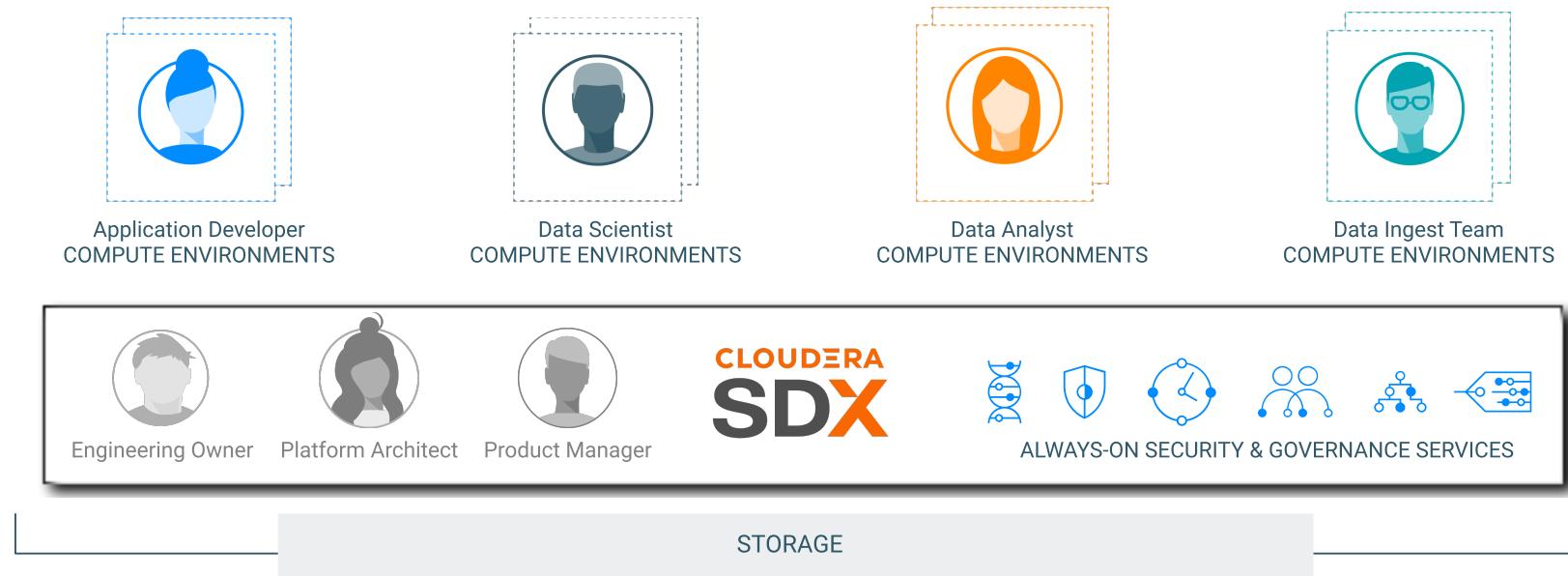
Cloudera Data Platform



A suite of products to collect, curate, report, serve, and predict

- Cloud native or bare metal deployment
- Powered by open source
- Analytics from the Edge to AI
- Unified data control plane
- Shared Data Experience (SDX)

Cloudera Shared Data Experience (SDX)



- **Full data lifecycle:** Manages your data from ingestion to actionable insights
- **Unified security:** Protects sensitive data with consistent controls
- **Consistent governance:** Enables safe self-service access

Self-Serve Experiences for Cloud Form Factors

- Services customized for specific steps in the data lifecycle
 - Emphasize productivity and ease of use
 - Auto-scale compute resources to match changing demands
 - Isolate compute resources to maintain workload performance



Cloudera DataFlow

- Data-in-motion platform
- Reduces data integration development time
- Manages and secures your data from edge to enterprise



EDGE & FLOW MANAGEMENT

High-scale data ingestion, intelligence and monitoring from edge to cloud

Apache NiFi

MiniFi

Edge Flow Manager



STREAM PROCESSING & ANALYTICS

Ultra low-latency event processing for real-time insights

Apache Flink

Spark Streaming

Kafka Streams



STREAMS MESSAGING

High-speed streams messaging, monitoring and replication

Apache Kafka

Streams Messaging Manager

Streams Replication Manager



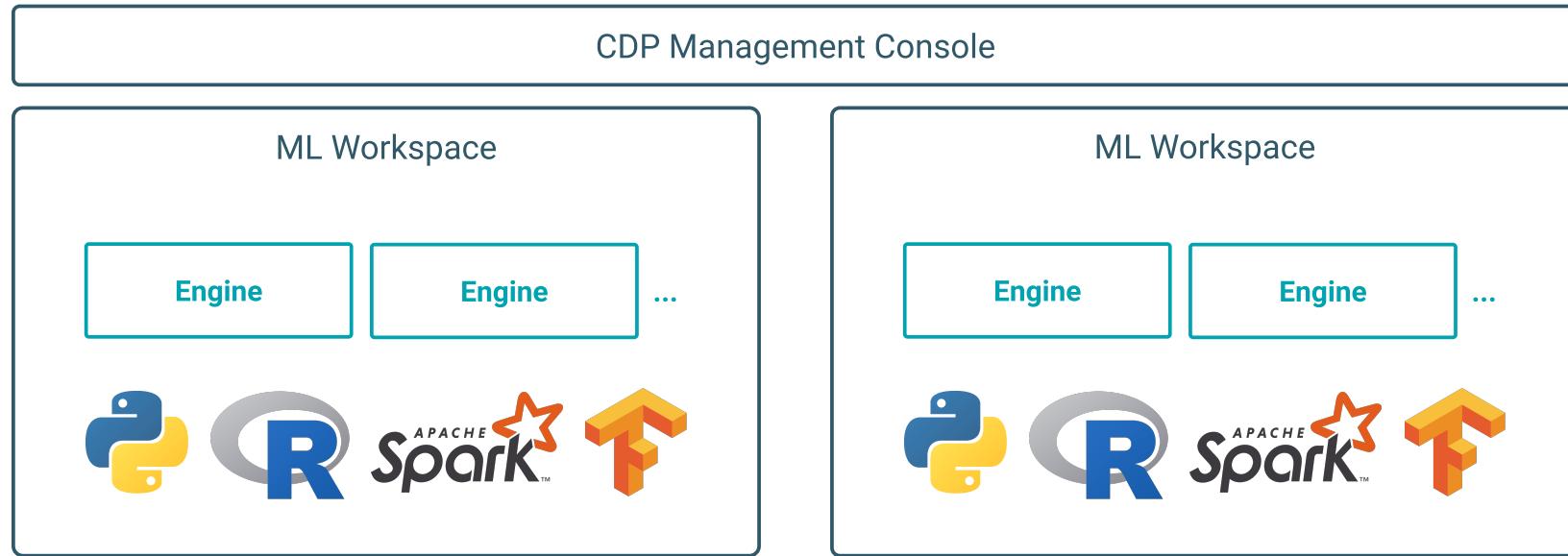
Provisioning,
management and
monitoring

Unified Security

Edge-to-Enterprise Governance

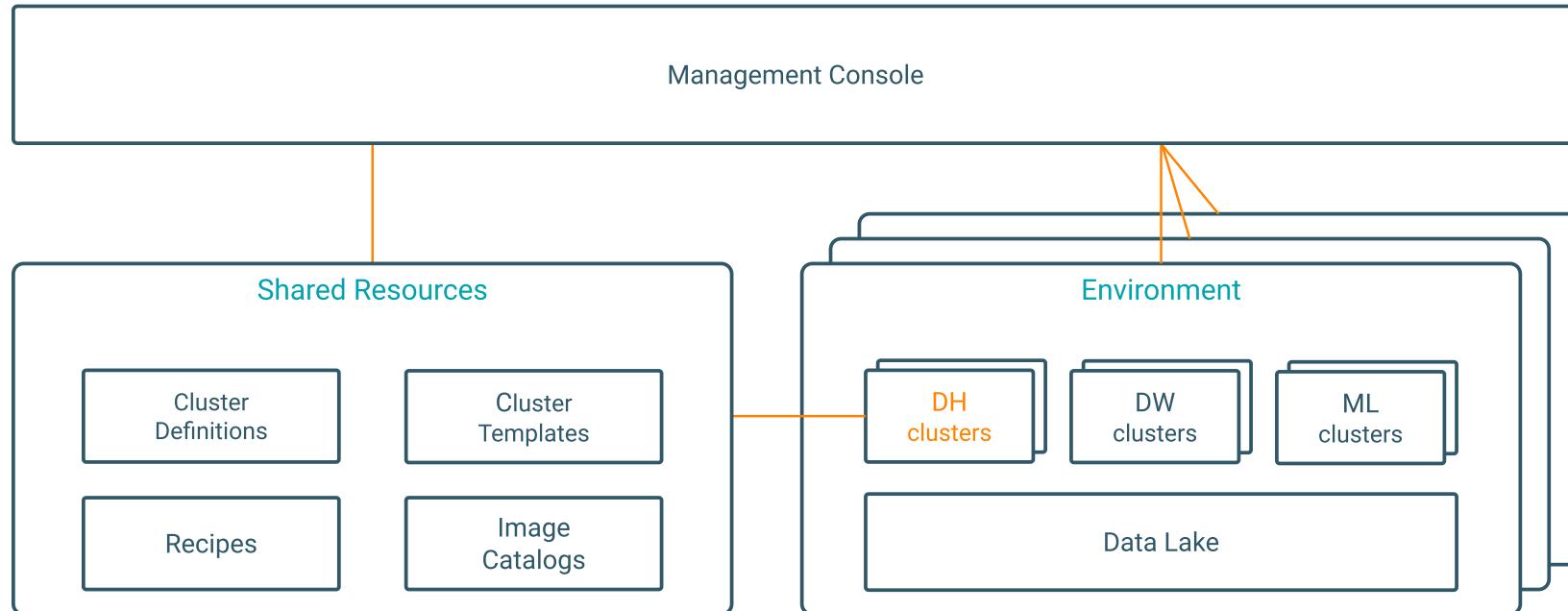
Single Sign-on

Cloudera Machine Learning



- **Cloud-native enterprise machine learning**
 - Fast, easy, and secure self-service data science in enterprise environments
 - Direct access to a secure cluster running Spark and other tools
 - Isolated environments for running Python, R, and Scala code
 - Teams, version control, collaboration, and project sharing

Cloudera Data Hub



Customize your own experience in cloud form factors

- Integrated suite of analytic engines
- Cloudera SDX applies consistent security and governance
- Fueled by open source innovation

Chapter Topics

Introduction

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

Cloudera Educational Services

- **We offer a variety of ways to take our courses**
 - Instructor-led, both in physical and virtual classrooms
 - Private and customized courses also available
 - Self-paced, through Cloudera OnDemand
- **Courses for all kinds of data professionals**
 - Executives and managers
 - Data scientists and machine learning specialists
 - Data analysts
 - Developers and data engineers
 - System administrators
 - Security professionals

Cloudera Education Catalog

- A broad portfolio across multiple platforms
 - Not all courses shown here
 - See [our website](#) for the complete catalog

	Administrator	Security	NiFi	AWS Fundamentals for CDP	
ADMINISTRATOR	CDH HDP 	CDH HDP 	CDF 		
DATA ANALYST	Data Analyst CDH CDP 	Hive 3 HDP 	Kudu CDH 	Cloudera Data Warehouse CDP 	 Private Class  Public Class  OnDemand
DEVELOPER & DATA ENGINEER	Spark CDH HDP 	Spark Performance Tuning CDH 	Stream Developer CDF 	Kafka Operations CDH 	Search Solr CDH 
DATA SCIENTIST	Data Scientist CDH HDP CDP 	Cloudera DS Workbench CDH HDP 		CML CDP 	Architecture Workshop CDH 

Cloudera OnDemand

- Our OnDemand catalog includes
 - Courses for developers, data analysts, administrators, and data scientists, updated regularly
 - Exclusive OnDemand-only courses, such as those covering security and Cloudera Data Science Workbench
 - Free courses such as *Essentials* and *Cloudera Director* available to all with or without an OnDemand account
- Features include
 - Video lectures and demonstrations with searchable transcripts
 - Hands-on exercises through a browser-based virtual environment
 - Discussion forums monitored by Cloudera course instructors
 - Searchable content within and across courses
- Purchase access to a library of courses or individual courses
- See the [Cloudera OnDemand information page](#) for more details or to make a purchase, or go directly to [the OnDemand Course Catalog](#)

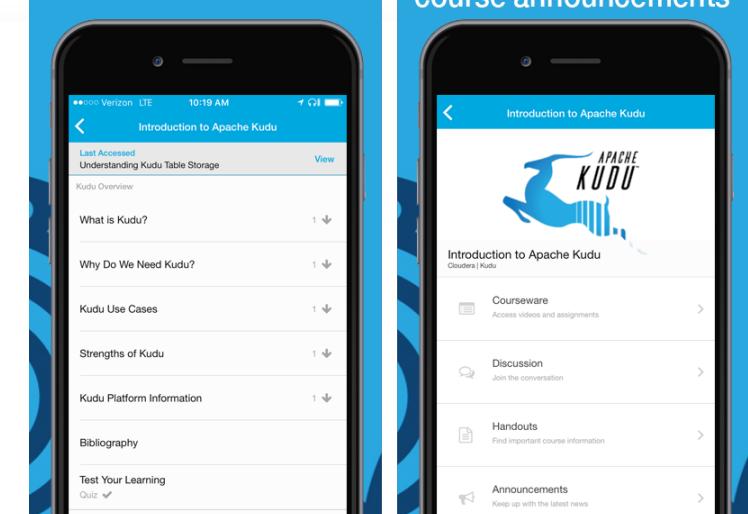
Accessing Cloudera OnDemand

- Cloudera OnDemand subscribers can access their courses online through a web browser

The screenshot shows a web browser window with the following details:

- Header:** Home, Course (selected), Discussion, Progress.
- Search Bar:** Search course content...
- Sidebar:** Introduction to Apache Hadoop and the Hadoop Ecosystem, Apache Hadoop Overview (selected), Data Storage and Ingest, Data Processing, Data Analysis and Exploration, Other Ecosystem Tools, Intro to the Hands-On Exercises, Hands-On Exercise: Accessing the Exercise Environment, Hands-On Exercise: General Notes, Hands-On Exercise: Query Hadoop Data with Apache Impala, Test Your Learning, Quiz.
- Content Area:** Introduction to Apache Hadoop and the Hadoop Ecosystem > Apache Hadoop Overview > Apache Hadoop Overview.
 - Title:** Apache Hadoop Overview
 - Description:** Apache Hadoop Overview
 - Video Player:** Common Hadoop Use Cases, Extract, Transform, and Load (ETL), Data analysis, Text mining, Index building, Graph creation and analysis, Pattern recognition, Data storage, Collaborative filtering, Prediction models, Sentiment analysis, Risk assessment, What do these workloads have in common? Nature of the data..., Volume, Velocity, Variety.
 - Transcript:** Start of transcript. Skip to the end.
 - Text:** Let's start this overview by answering the question: What is Apache Hadoop? Hadoop is an open source system for large-scale distributed data storage, processing, and analysis. It harnesses the power of relatively inexpensive servers—up to hundreds or thousands of servers—working together as a single system to process petabytes of data.

- Cloudera OnDemand is also available through an iOS app
 - Search for “Cloudera OnDemand” in the iOS App Store



Cloudera Certification

- The leader in Apache Hadoop-based certification
- Cloudera certification exams favor hands-on, performance-based problems that require execution of a set of real-world tasks against a live, working cluster
- We offer two levels of certifications
 - Cloudera Certified Associate (CCA)
 - CCA Spark and Hadoop Developer
 - CCA Data Analyst
 - CCA CDH Administrator and CCA HDP Administrator
 - Cloudera Certified Professional (CCP)
 - CCP Data Engineer

Chapter Topics

Introduction

- About This Course
- Introductions
- About Cloudera
- About Cloudera Educational Services
- Course Logistics

Logistics

- Class start and finish time
- Lunch
- Breaks
- Restrooms
- Wi-Fi access
- Virtual machines

Your instructor will give you details on how
to access the course materials for the class



Introduction to Apache Hadoop and the Hadoop Ecosystem

Chapter 2

Course Chapters

- Introduction
- **Introduction to Apache Hadoop and the Hadoop Ecosystem**
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Introduction to Apache Hadoop and the Hadoop Ecosystem

After completing this chapter, you will be able to

- List examples of use cases Hadoop is best suited for
- Describe the components of a distributed processing system
- Summarize Hadoop's key data storage and processing tools
- Use your exercise environment to complete the course exercises

Chapter Topics

Introduction to Apache Hadoop and the Hadoop Ecosystem

- **Apache Hadoop Overview**
- Data Processing
- Introduction to the Hands-On Exercises
- Essential Points
- Hands-On Exercise: Starting the Exercise Environment

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

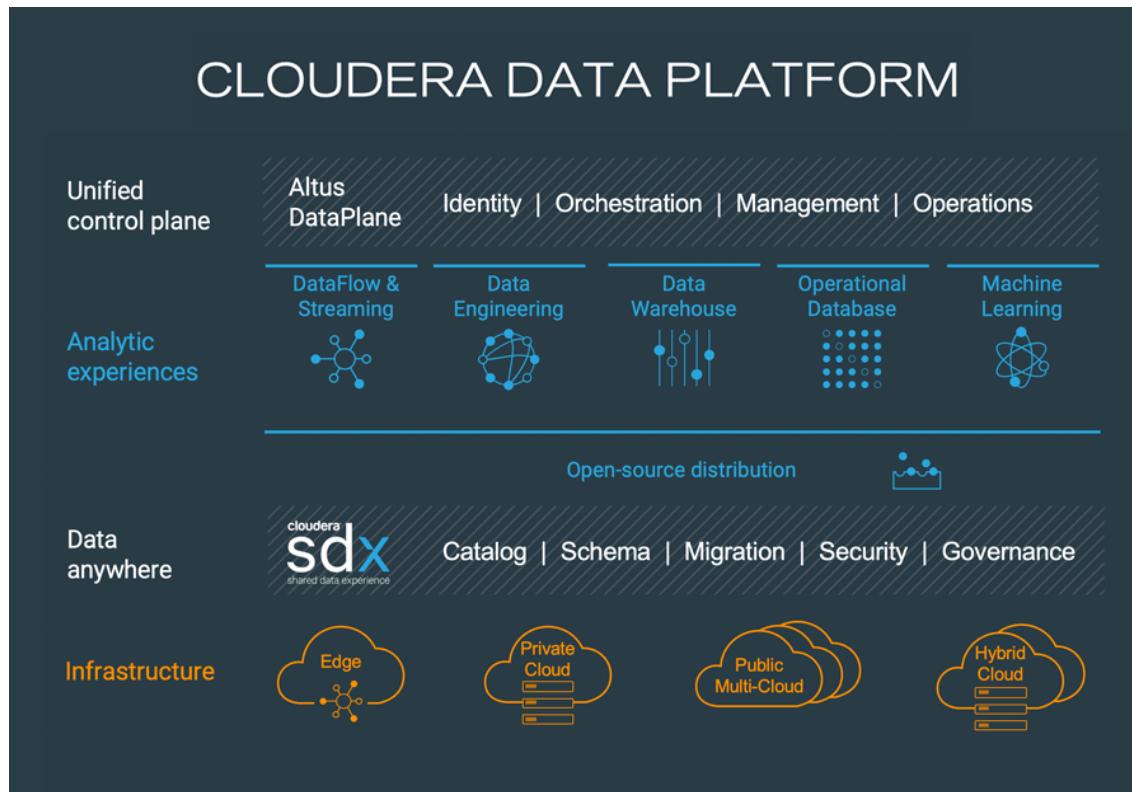
- Hadoop is ideal for applications that handle data with high
 - Volume
 - Velocity
 - Variety

- Extract, Transform, and Load (ETL)
- Data analysis
- Text mining
- Index building
- Graph creation and analysis
- Pattern recognition

- Data storage
- Collaborative filtering
- Prediction models
- Sentiment analysis
- Risk assessment

Cloudera Data Platform and Apache Hadoop

- Hadoop provides the underpinnings of Cloudera Data Platform (CDP)
- CDP combines the best of
 - Hortonworks HDP
 - Cloudera CDH

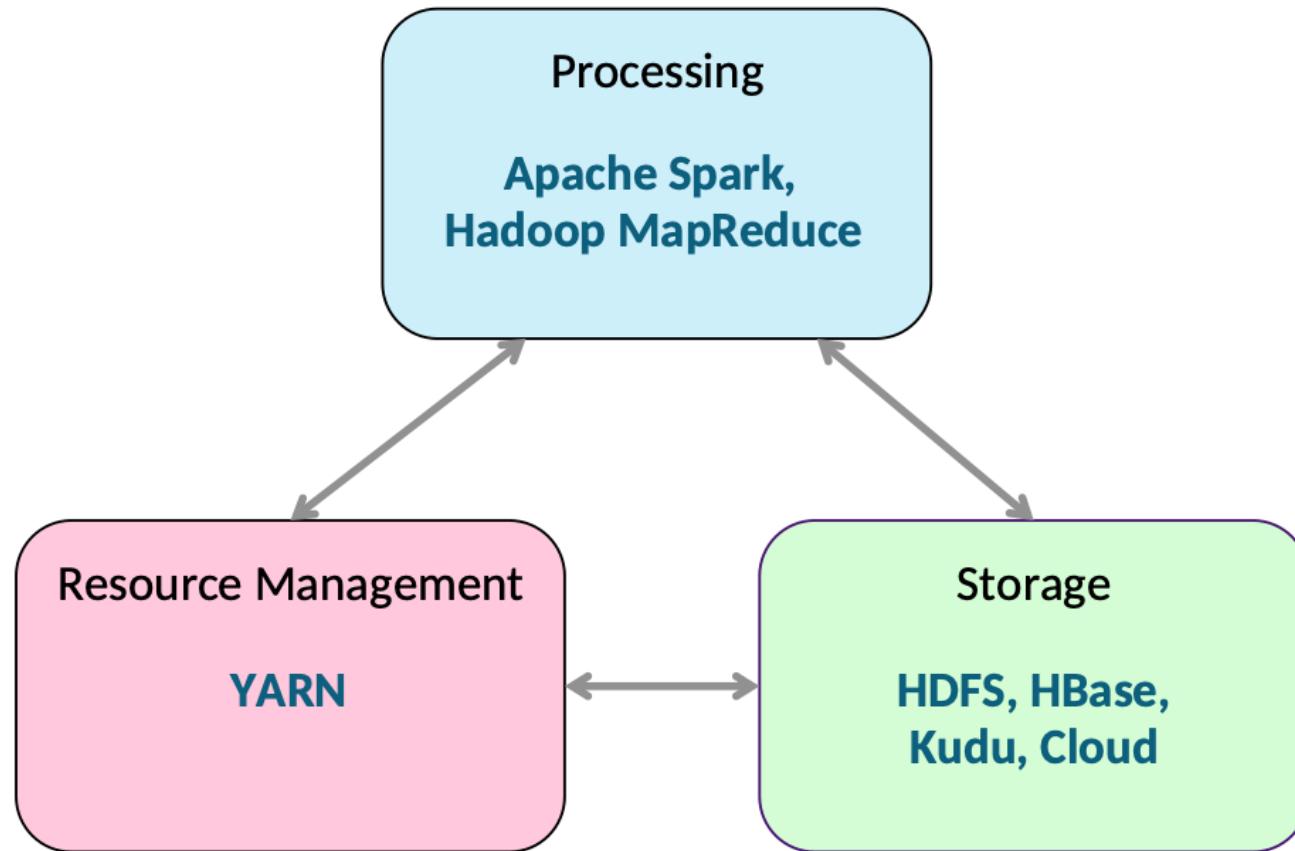


Chapter Topics

Introduction to Apache Hadoop and the Hadoop Ecosystem

- Apache Hadoop Overview
- **Data Processing**
- Introduction to the Hands-On Exercises
- Essential Points
- Hands-On Exercise: Starting the Exercise Environment

Distributed Processing with Hadoop



Data Storage (1)

- **Hadoop Distributed File System (HDFS)**
 - The main on-premises storage layer for Hadoop
 - Inexpensive reliable storage for massive amounts of data on industry-standard hardware
- **Apache HBase**
 - A NoSQL distributed database built on HDFS
 - Provides very high throughput
 - Supports tables with thousands of columns

Data Storage (2)

- **Apache Kudu**
 - Columnar key-value storage for structured data
 - High-throughput scans, low-latency random access and updates
 - Supports SQL-based analytics
 - Covered in Cloudera's online OnDemand module *Introduction to Apache Kudu*
- **Cloud storage**
 - Most Hadoop ecosystem tools support data storage in the cloud
 - Such as Amazon S3 and Microsoft ADLS

Apache Spark: An Engine for Large-Scale Data Processing

- **Spark is a general-purpose data processing engine**
 - Designed to handle very large amounts of data
 - Distributes processing across a cluster
- **Supports a wide range of workloads**
 - Machine learning
 - Business intelligence
 - Streaming
 - Batch processing
 - Querying structured data



Hadoop MapReduce: The Original Hadoop Processing Engine

- Hadoop MapReduce is the original Hadoop framework for processing big data
 - Primarily Java-based
- Based on the map-reduce programming model
- The core Hadoop processing engine before Spark was introduced
- Still in use in many production systems
 - Most new development uses Spark
- Many existing tools built using MapReduce code are still in use
- Has extensive and mature fault tolerance built into the framework



Chapter Topics

Introduction to Apache Hadoop and the Hadoop Ecosystem

- Apache Hadoop Overview
- Data Processing
- **Introduction to the Hands-On Exercises**
- Essential Points
- Hands-On Exercise: Starting the Exercise Environment

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
- 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

- **Loudacre needs to migrate their existing infrastructure to Hadoop**
 - The size and velocity of their data has exceeded their ability to process and analyze their data
- **Loudacre data sources**
 - MySQL database: customer account data (name, address, phone numbers, and 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 station files: Cell tower locations

Introduction to the Exercise Environment (1)

- You will do exercises using a provided *exercise environment*
- The environment includes a “pseudo-distributed” Hadoop cluster
 - All cluster services run on a single host
 - The cluster includes YARN and HDFS
- Access the environment by connecting to a remote desktop
 - Uses a virtual machine running Linux
 - Username training (password training)

Introduction to the Exercise Environment (2)

- Your *home directory* is `/home/training`
 - Often referred to using a tilde (~)
- The *course directory* is `~/training_materials/devsh`
- The course directory contains files used in the course
 - examples: all the example code in this course
 - exercises: starter code and solutions for the hands-on exercises
 - scripts: course setup and catch-up scripts
 - data: sample data for exercises

Chapter Topics

Introduction to Apache Hadoop and the Hadoop Ecosystem

- Apache Hadoop Overview
- Data Processing
- Introduction to the Hands-On Exercises
- **Essential Points**
- Hands-On Exercise: Starting the Exercise Environment

Essential Points

- **Hadoop is a framework for distributed data storage and processing**
- **This course focuses on creating solutions that use**
 - Apache Spark for computational processing
 - Hadoop Distributed File Storage (HDFS) for data storage
 - Hadoop YARN for cluster resource management
- **Hands-on exercises will let you practice and refine your Spark and Hadoop skills**

Chapter Topics

Introduction to Apache Hadoop and the Hadoop Ecosystem

- Apache Hadoop Overview
- Data Processing
- Introduction to the Hands-On Exercises
- Essential Points
- **Hands-On Exercise: Starting the Exercise Environment**

Hands-On Exercise: Starting the Exercise Environment

- In this exercise, you will launch your course exercise environment**
- Please refer to the Hands-On Exercise Manual for instructions**



Apache Hadoop File Storage

Chapter 3

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage**
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Apache Hadoop File Storage

After completing this chapter, you will be able to

- **Describe how HDFS stores data across a cluster**
- **Use the `hdfs dfs` command to work with distributed files**

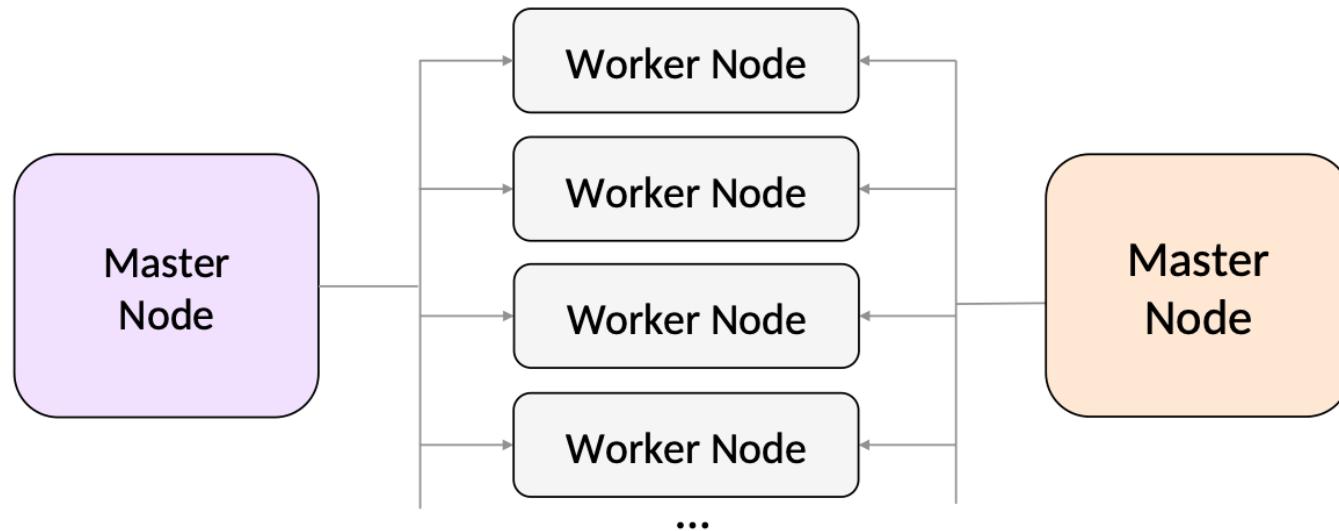
Chapter Topics

Apache Hadoop File Storage

- **Apache Hadoop Cluster Components**
- HDFS Architecture
- Using HDFS
- Essential Points
- Hands-On Exercise: Working with HDFS

Hadoop Cluster Terminology

- A **cluster** is a group of computers (hosts) working together
 - Provides data storage, computation and data processing, and resource management
 - Host serve as nodes to provide cluster services
- **Master nodes** manage distribution of work and data
- **Worker nodes** store and process data
- A **daemon** is a program running on a node to provide a service



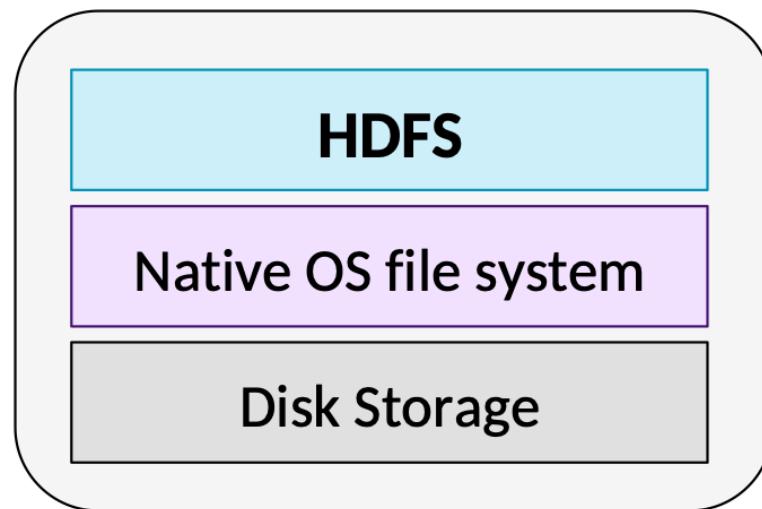
Chapter Topics

Apache Hadoop File Storage

- Apache Hadoop Cluster Components
- **HDFS Architecture**
- Using HDFS
- Essential Points
- Hands-On Exercise: Working with HDFS

HDFS Basic Concepts (1)

- **HDFS is the original Hadoop storage system**
 - Other storage options have been added since
 - Such as HBase, Kudu, or support for cloud storage
- **Sits on top of a native file system**
 - Such as ext3, ext4, or xfs
- **Provides redundant storage for massive amounts of data**
 - Using readily available, industry-standard computers



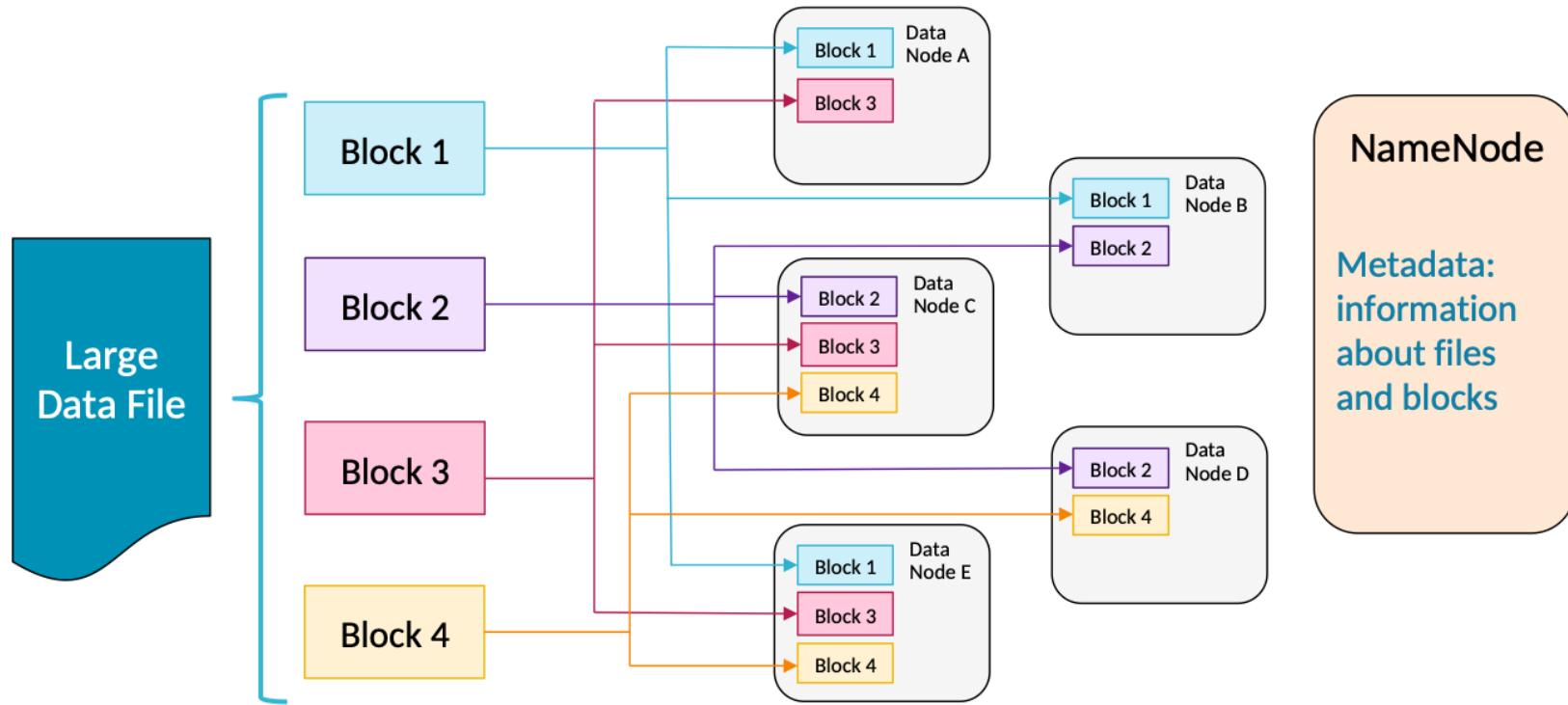
HDFS Basic Concepts (2)

- **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”**
 - Updates and random reads are not supported
- **HDFS is optimized for large, streaming reads of files**
 - Rather than random reads

How Files Are Stored (1)

- Data files are split into blocks (default 128MB) which are distributed at load time
- The actual blocks are stored on cluster worker nodes running the Hadoop HDFS Data Node service
 - Often referred to as *DataNodes*
- Each block is replicated on multiple DataNodes (default 3x)
- A cluster master node runs the HDFS Name Node service, which stores file metadata
 - Often referred to as the *NameNode*

How Files Are Stored (2)



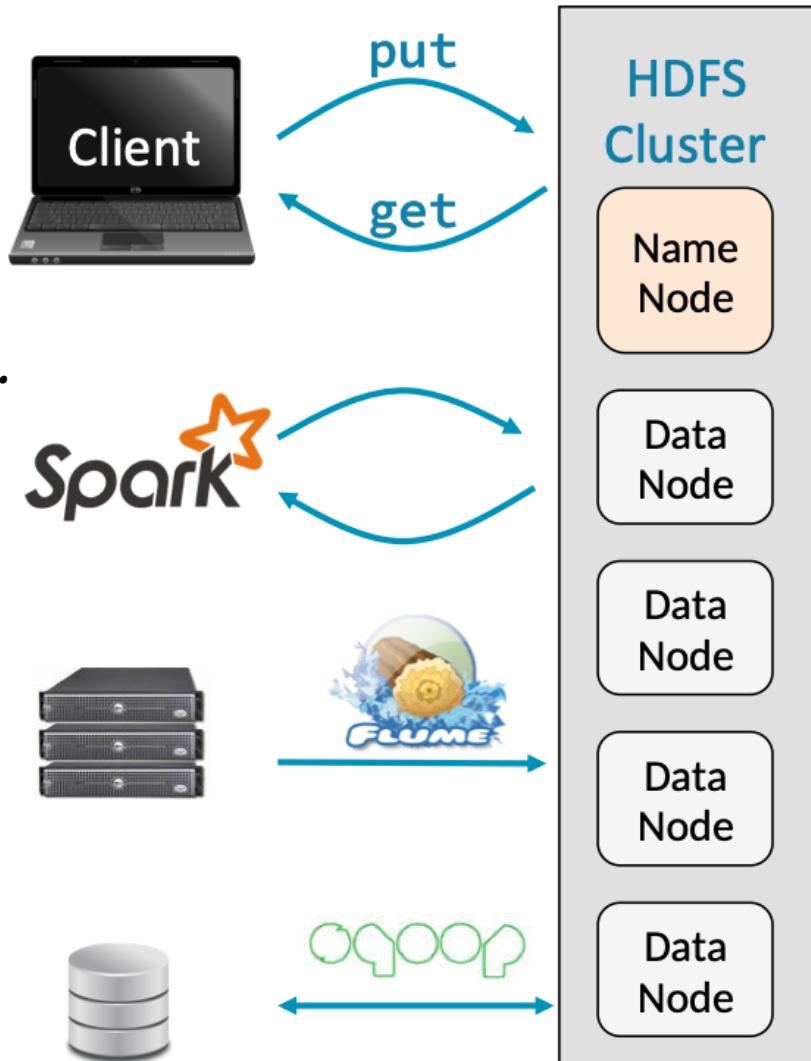
Chapter Topics

Apache Hadoop File Storage

- Apache Hadoop Cluster Components
- HDFS Architecture
- **Using HDFS**
- Essential Points
- Hands-On Exercise: Working with HDFS

Options for Accessing HDFS

- Command line interface
 - `$ hdfs dfs`
- Spark
 - By URI:
`hdfs://nnhost:port/file...`
- Other Hadoop tools such as
 - Sqoop—import from RDBMS
 - Flume—ingest real-time data
 - Hive—SQL analytics
 - HBase—no-SQL database
- Applications
 - MapReduce
 - Java API
 - 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 a file

```
$ hdfs dfs -rm input_old/myfile
```

- Delete a set of files using a wildcard

```
$ hdfs dfs -rm input_old/*
```

- Delete the directory `input_old` and all its contents

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

Chapter Topics

Apache Hadoop File Storage

- Apache Hadoop Cluster Components
- HDFS Architecture
- Using HDFS
- **Essential Points**
- Hands-On Exercise: Working with HDFS

Essential Points

- **The Hadoop Distributed File System (HDFS) is the main storage layer for Hadoop**
- **HDFS chunks data into blocks and distributes them to the cluster when data is stored**
- **HDFS clusters consist of**
 - A single NameNode to manage file metadata
 - Multiple DataNodes to store data
- **The `hdfs dfs` command allows you to use and manage files in HDFS**

Chapter Topics

Apache Hadoop File Storage

- Apache Hadoop Cluster Components
- HDFS Architecture
- Using HDFS
- Essential Points
- **Hands-On Exercise: Working with HDFS**

Hands-On Exercise: Working with HDFS

- In this exercise, you will practice managing and viewing data files
- Please refer to the Hands-On Exercise Manual for instructions



Distributed Processing on an Apache Hadoop Cluster

Chapter 4

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- **Distributed Processing on an Apache Hadoop Cluster**
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Distributed Processing on an Apache Hadoop Cluster

After completing this chapter, you will be able to

- **Describe the YARN services that run in a Hadoop cluster**
- **Summarize how YARN provides cluster resource management for distributed data processing**
- **Use the YARN web UI and the `yarn` command to monitor applications running on a cluster**

Chapter Topics

Distributed Processing on an Apache Hadoop Cluster

- **YARN Architecture**
- Working With YARN
- Essential Points
- Hands-On Exercise: Running and Monitoring 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 Daemons

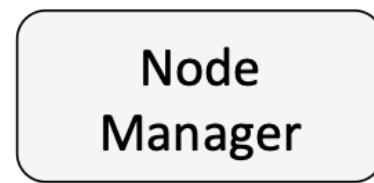
- **ResourceManager (RM)**

- Runs on master node
- Global resource scheduler
- Arbitrates system resources between competing applications
- Has a pluggable scheduler to support different algorithms, such as Capacity or Fair Scheduler



- **NodeManager (NM)**

- Runs on worker nodes
- Communicates with RM
- Manages node resources
- Launches containers



Applications on YARN (1)

■ Containers

- Containers allocate a certain amount of resources (memory, CPU cores) on a worker node
- Applications run in one or more containers
- Applications request containers from RM

Container

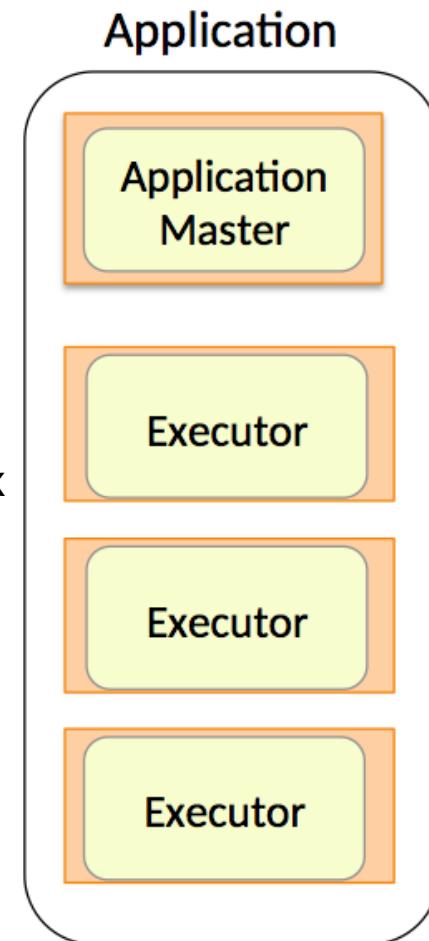
■ ApplicationMaster (AM)

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

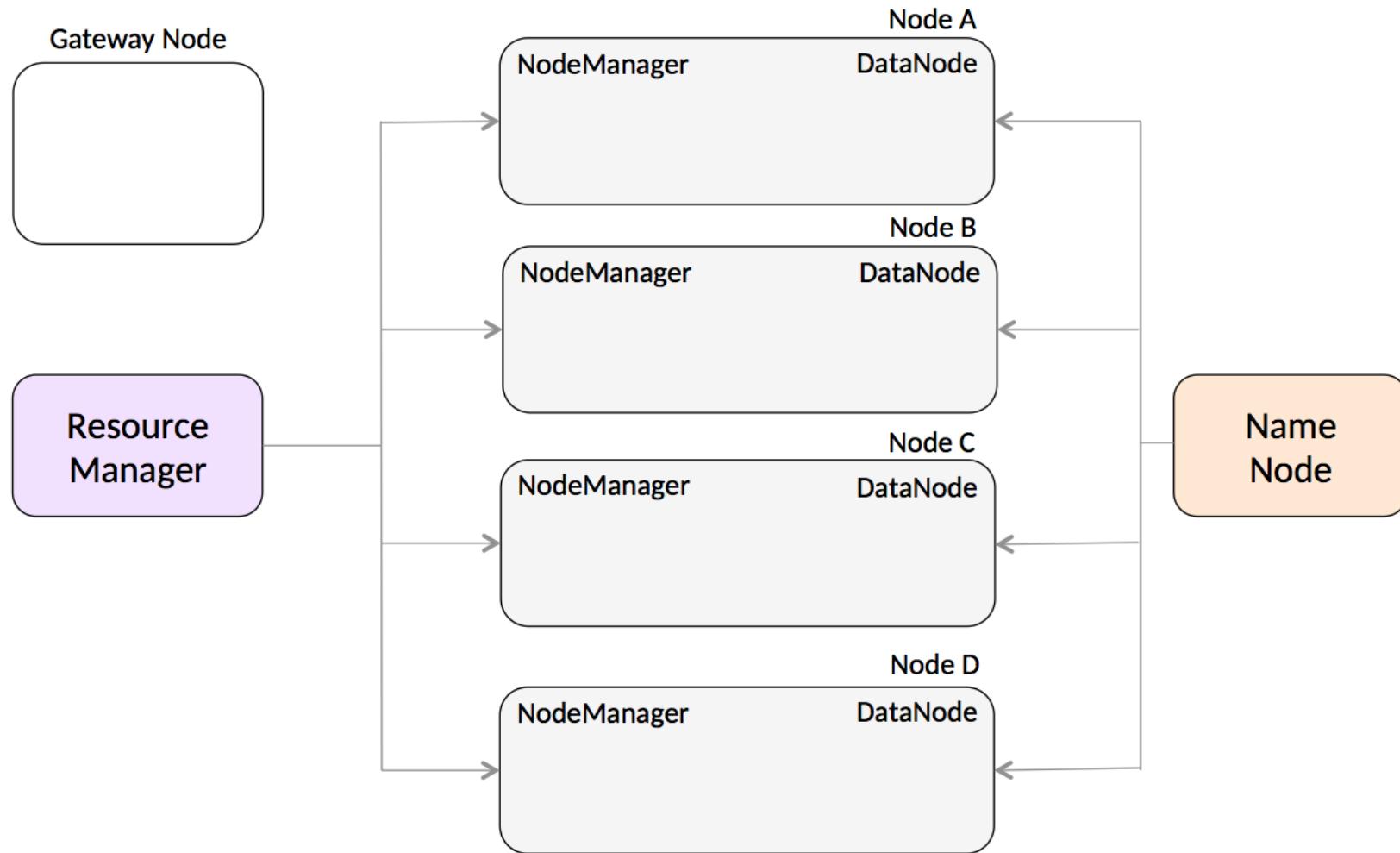
Application
Master

Applications on YARN (2)

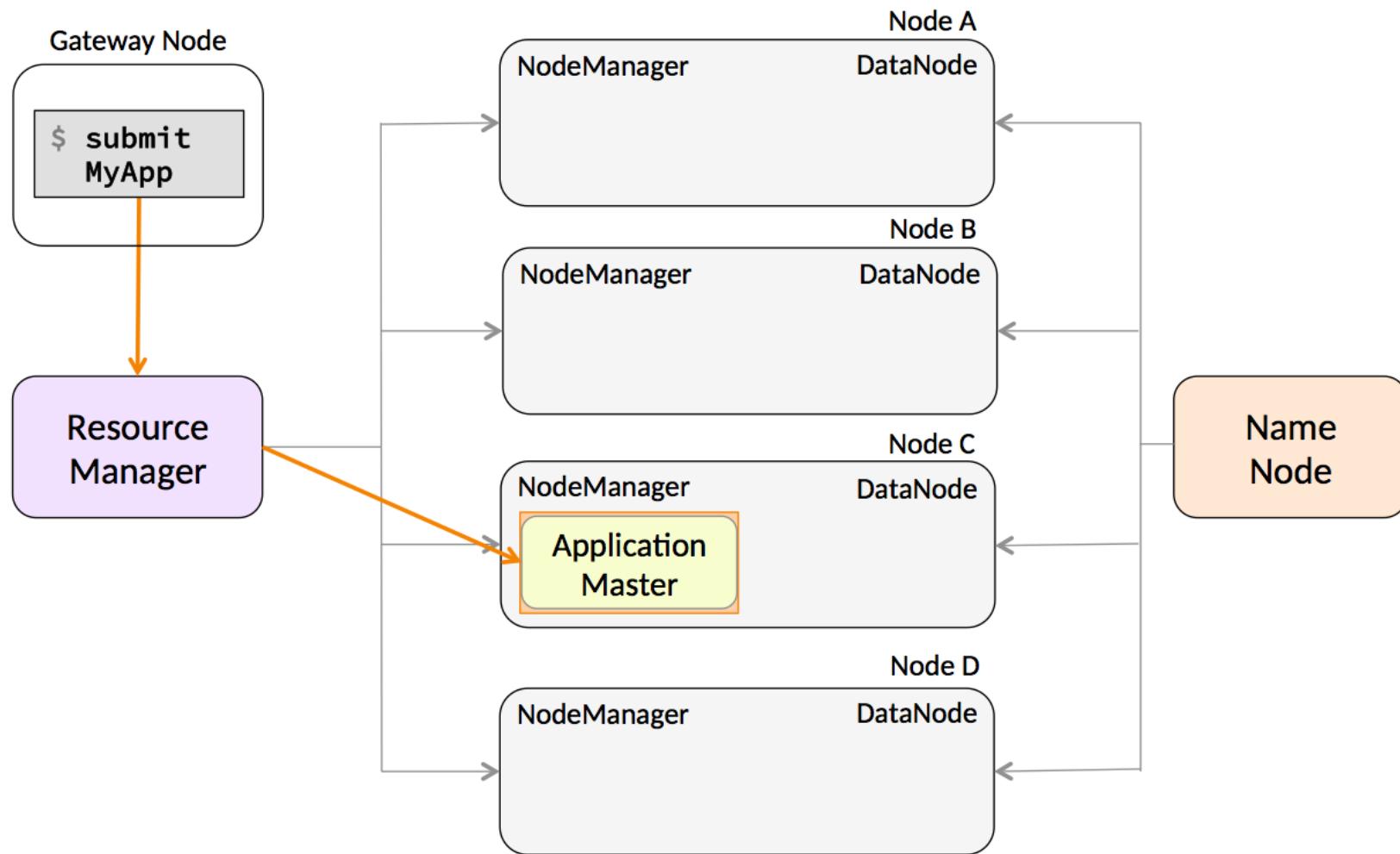
- **Each application consists of one or more containers**
 - The ApplicationMaster runs in one container
 - The application's distributed processes (JVMs) run in other containers
 - The processes run in parallel, and are managed by the AM
 - The processes are called executors in Apache Spark and tasks in Hadoop MapReduce
- **Applications are typically submitted to the cluster from an edge or gateway node**



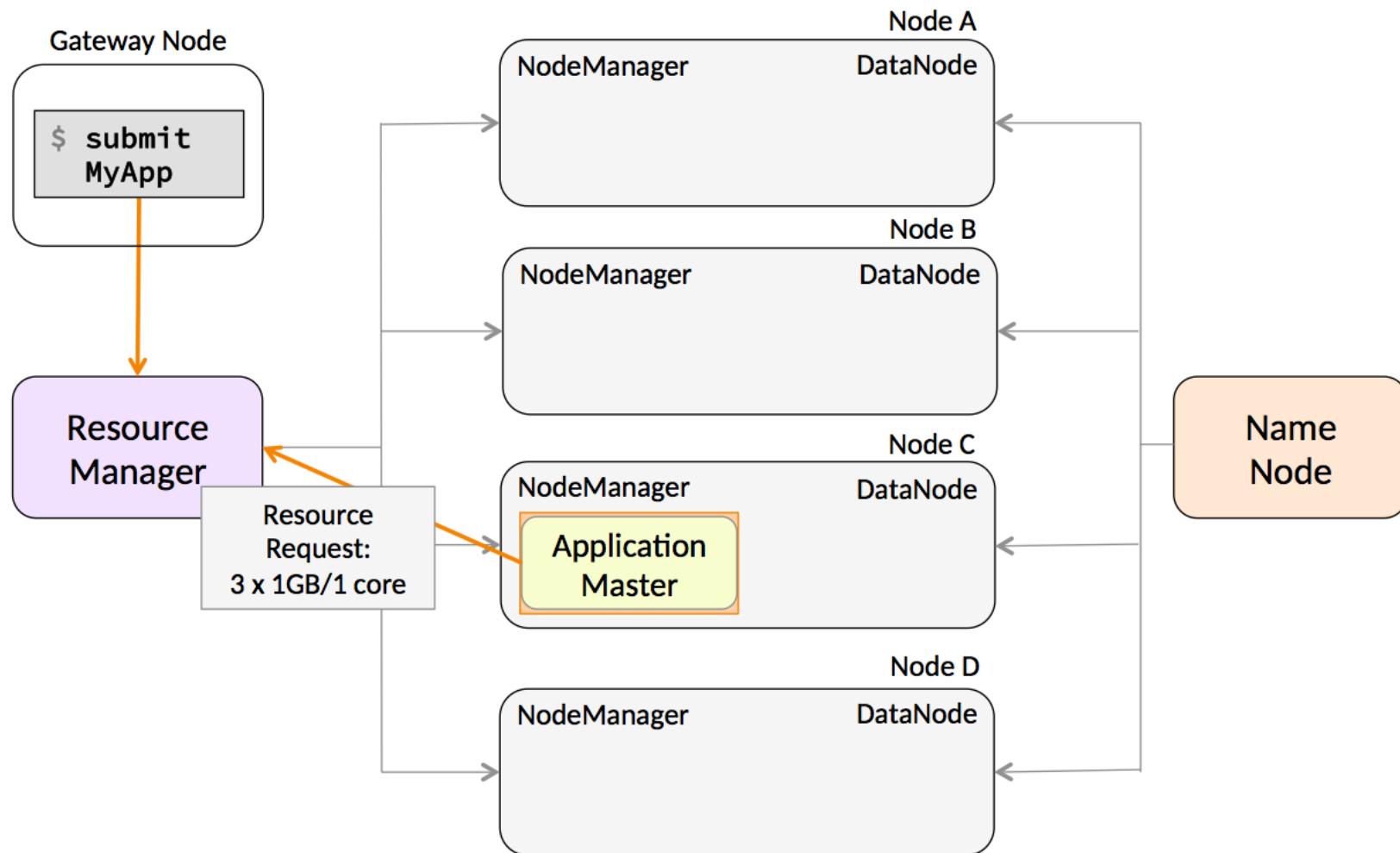
Running an Application on YARN (1)



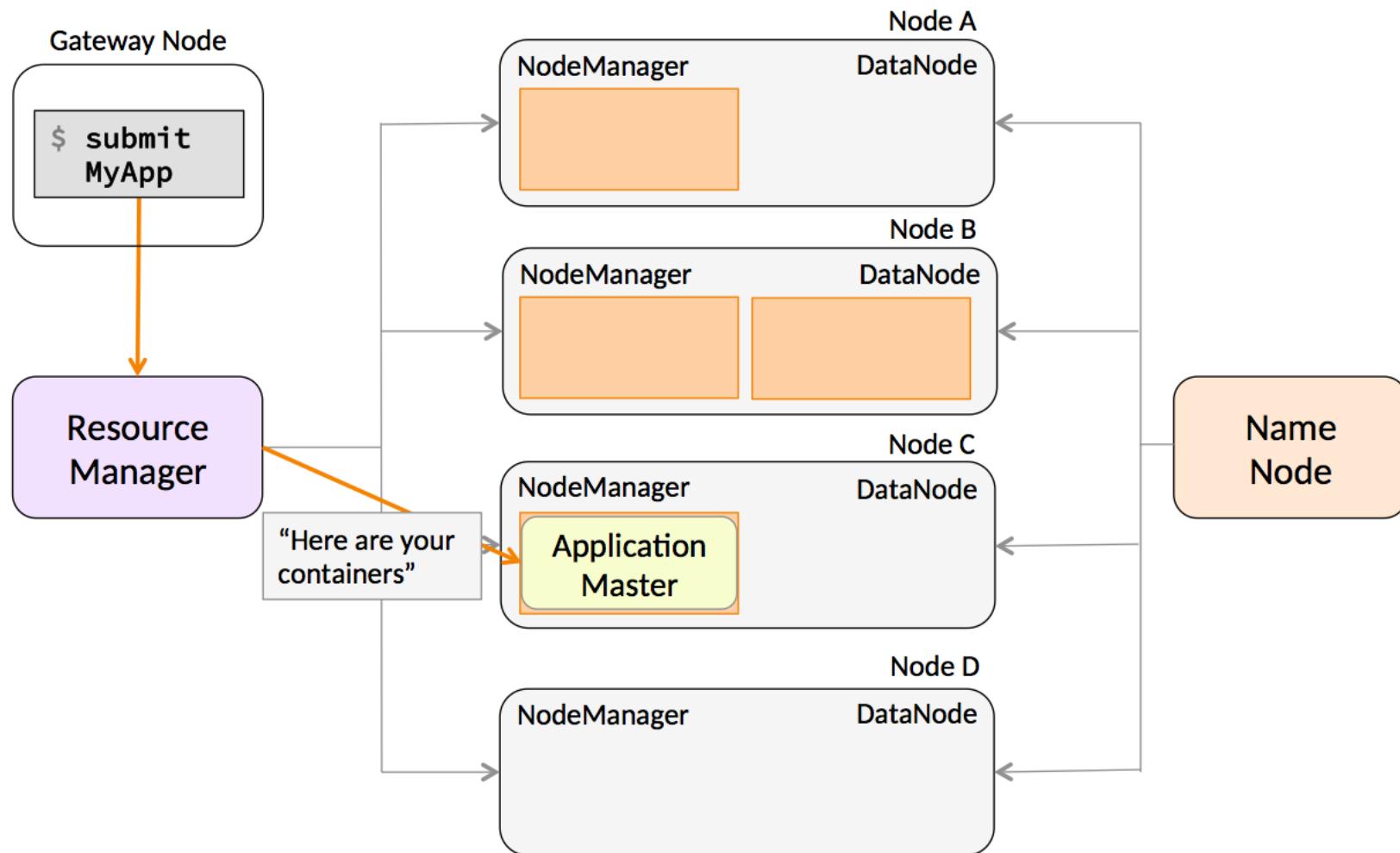
Running an Application on YARN (2)



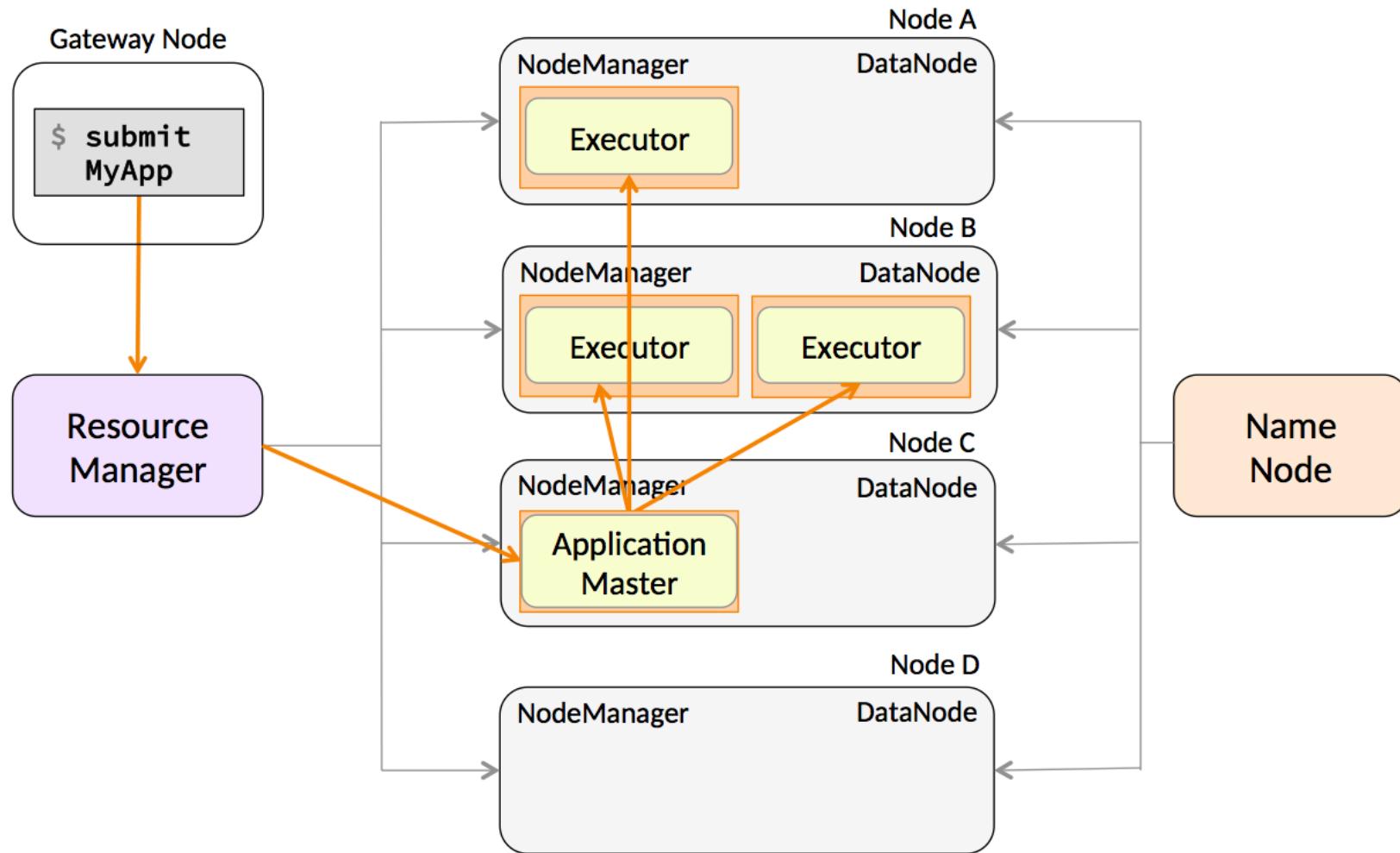
Running an Application on YARN (3)



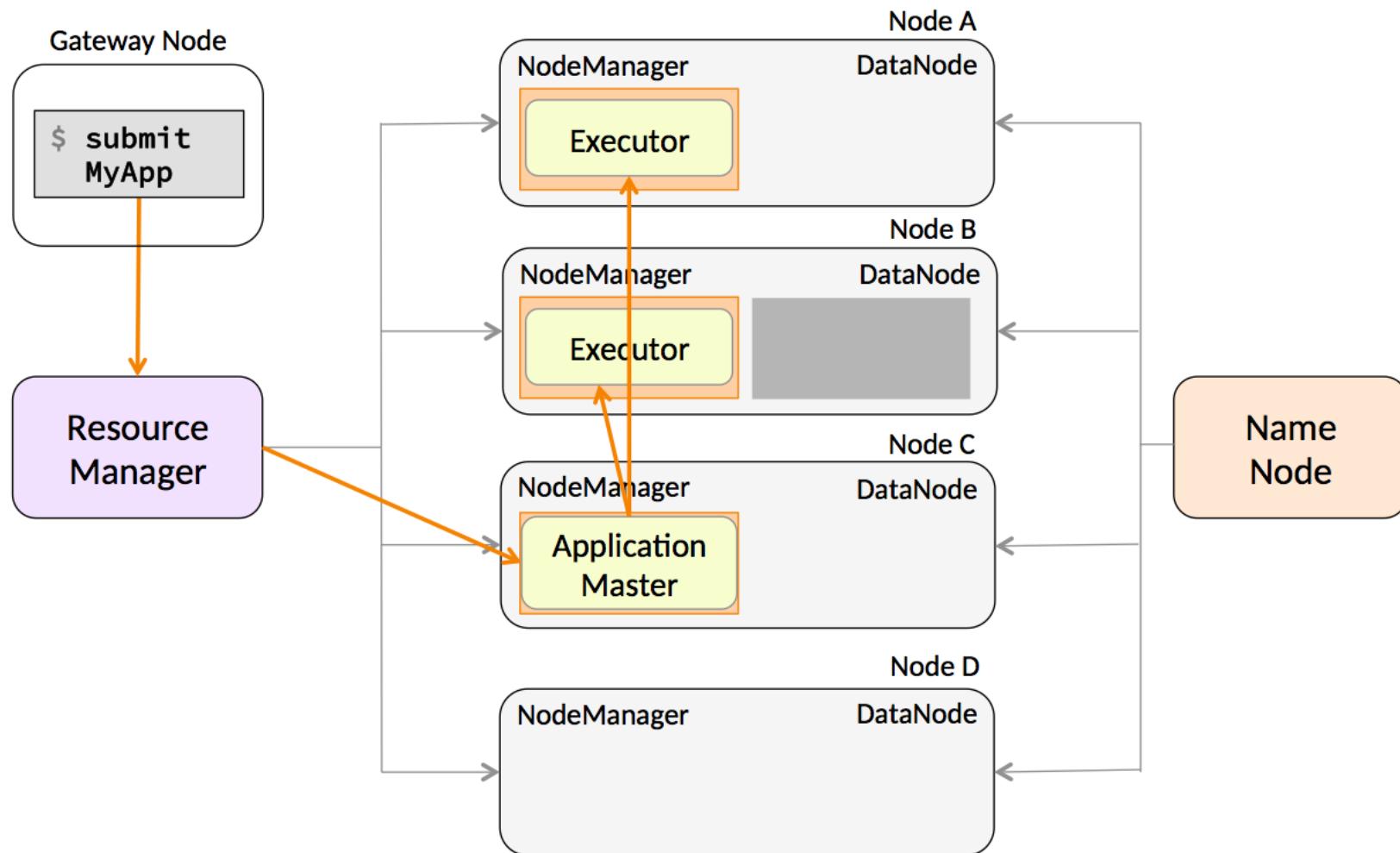
Running an Application on YARN (4)



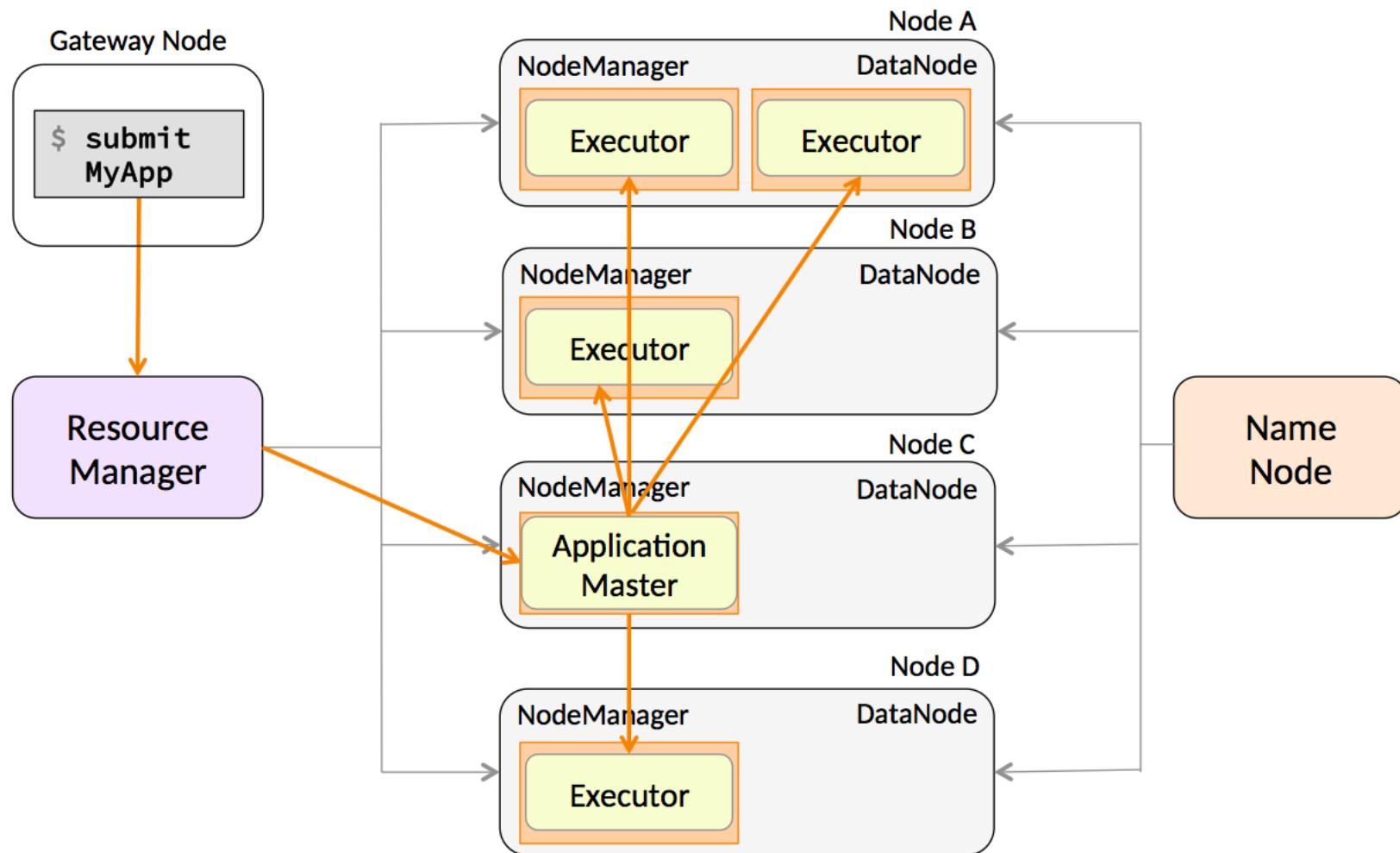
Running an Application on YARN (5)



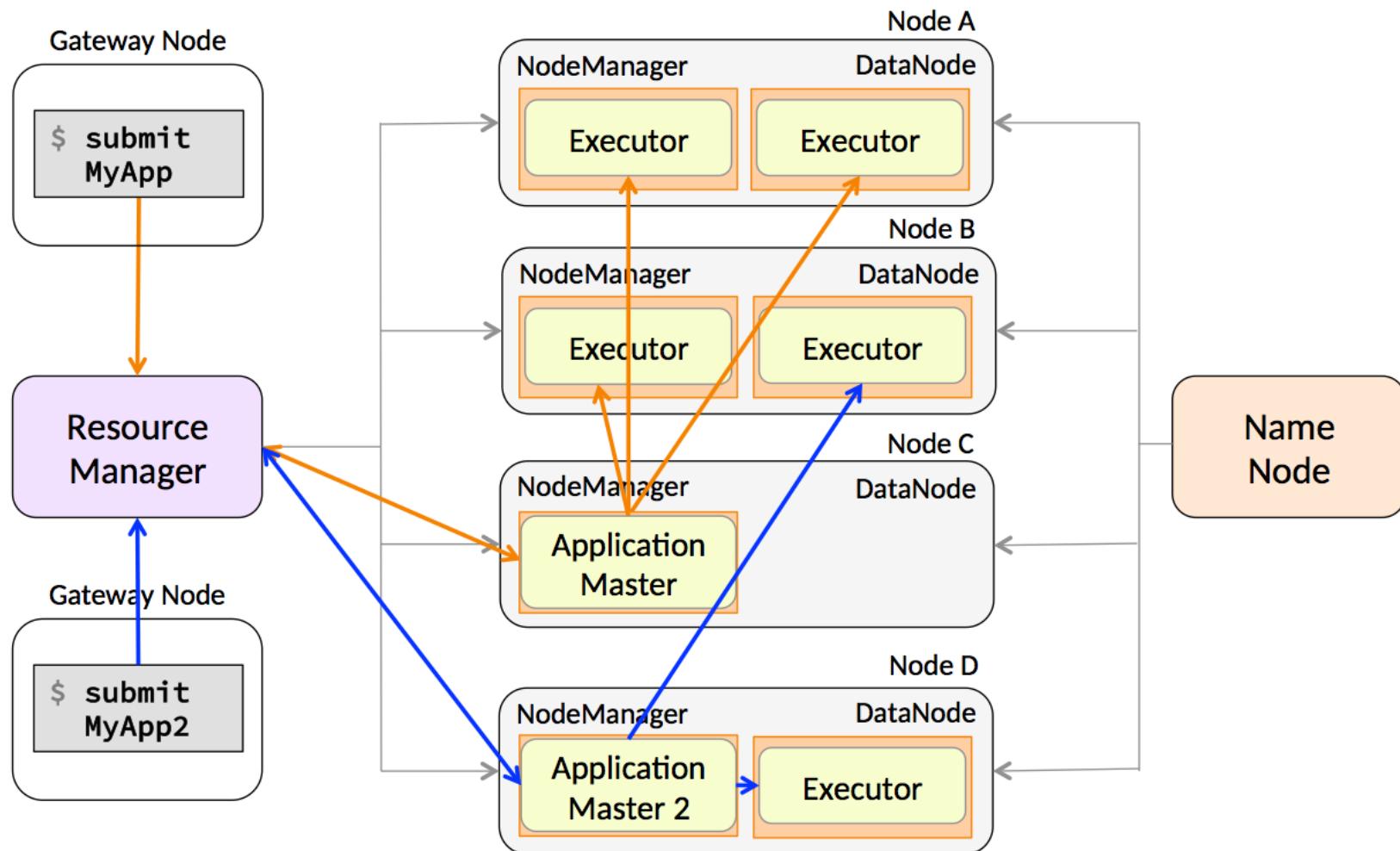
Running an Application on YARN (6)



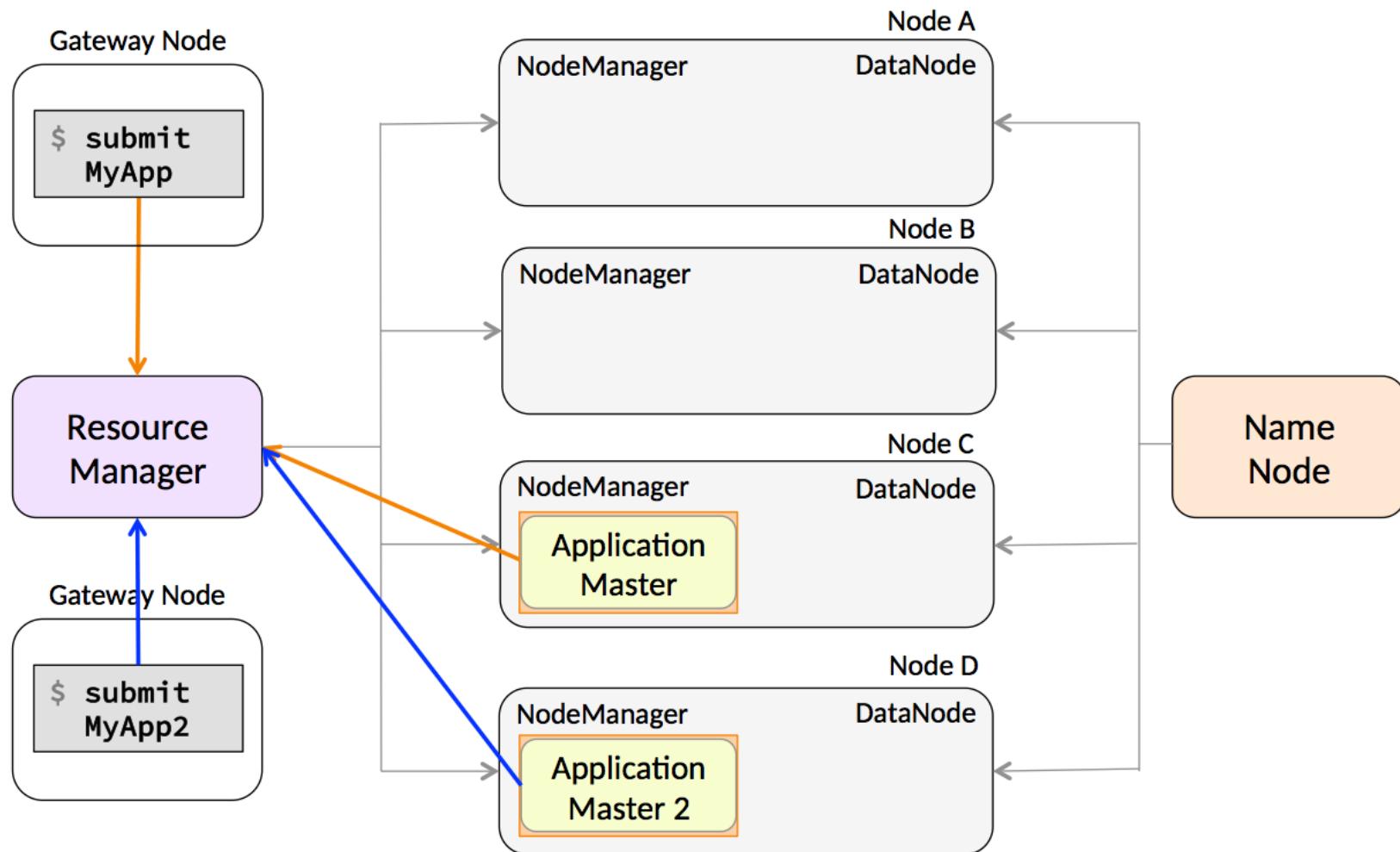
Running an Application on YARN (7)



Running an Application on YARN (8)



Running an Application on YARN (9)



Chapter Topics

Distributed Processing on an Apache Hadoop Cluster

- YARN Architecture
- **Working With YARN**
- Essential Points
- Hands-On Exercise: Running and Monitoring 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
- **YARN tools for developers**
 - The YARN web UI
 - The YARN command line interface

The YARN Web UI (1)

- The ResourceManager UI is the main entry point to the YARN UI
 - Runs on the RM host on port 8088 by default

Cluster Metrics																				
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	Vcores Used	Vcores Total	Vcores Reserved										
3	0	1	2	1	1 GB	4 GB	0 B	1	3	0										
Cluster Nodes Metrics																				
Active Nodes	Decommissioning Nodes			Decommissioned Nodes			Lost Nodes	Unhealthy Nodes			Rebooted Nodes	Shutdown Nodes								
1	0	0	0	0	0	0	0	0	0	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	0	0	0	0	0 B	0 B	0 B	0 B	0	0	0								
Scheduler Metrics																				
Scheduler Type	Scheduling Resource Type				Minimum Allocation			Maximum Allocation			Maximum Cluster Application Priority									
Fair Scheduler	{memory-mb (unit=Mi), vcores}				<memory:1024, vcores:1>			<memory:1536, vcores:3>			0									
Show 20	entries	Search:																		
ID	User	Name	Application Type	Queue	Application Priority	StartTime	LaunchTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Reserved CPU Vcores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Blacklisted Nodes
application_1556806051183_0001	training	Accounts by State	SPARK	root.users.training	0	Thu May 2 07:52:54 -0700 2019	Thu May 2 07:52:55 -0700 2019	Thu May 2 07:53:47 -0700 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0
application_1556806051183_0002	training	Accounts by State	SPARK	root.users.training	0	Thu May 2 07:54:00 -0700 2019	Thu May 2 07:54:00 -0700 2019	Thu May 2 07:54:11 -0700 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0
application_1556806051183_0003	training	PySparkShell	SPARK	root.users.training	0	Thu May 2 07:54:19 -0700 2019	Thu May 2 07:54:20 -0700 2019	N/A	RUNNING	UNDEFINED	1	1	1024	0	0	25.0	25.0		ApplicationMaster	0
Showing 1 to 3 of 3 entries															First	Previous	1	Next	Last	

The YARN Web UI (2)

- **About**
 - View status, metrics, and cluster details
- **Node Labels**
 - Groups of similarly configured nodes
- **Nodes**
 - List NodeManagers and statuses
 - Link to details such as applications and containers
- **Applications**
 - List and filter applications
 - Link to application details
- **Scheduler**
 - Details about resource distribution
- **Tools**
 - Configuration, YARN logs, server details

▼ Cluster

[About](#)
[Nodes](#)
[Node Labels](#)
[Applications](#)

[NEW](#)
[NEW_SAVING](#)
[SUBMITTED](#)
[ACCEPTED](#)
[RUNNING](#)
[FINISHED](#)
[FAILED](#)
[KILLED](#)

[Scheduler](#)

► Tools

YARN Command Line (1)

- Command to configure and view information about the YARN cluster

```
$ yarn command
```

- Most YARN commands are for administrators rather than developers
- Some helpful commands for developers
 - List running applications

```
$ yarn application -list
```

- Kill a running application

```
$ yarn application -kill app-id
```

YARN Command Line (2)

- View the logs of the specified application

```
$ yarn logs -applicationId app-id
```

- View the full list of command options

```
$ yarn --help
```

Chapter Topics

Distributed Processing on an Apache Hadoop Cluster

- YARN Architecture
- Working With YARN
- **Essential Points**
- Hands-On Exercise: Running and Monitoring a YARN Job

Essential Points

- YARN manages resources in a Hadoop cluster and schedules applications
- Worker nodes run NodeManager daemons, managed by a ResourceManager on a master node
- Applications running on YARN consist of an ApplicationMaster and one or more executors
- Use the YARN ResourceManager web UI or the `yarn` command to monitor applications

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>
- **YARN documentation**
 - <http://tiny.cloudera.com/yarndocs>
- **Cloudera Engineering Blog YARN articles**
 - <http://tiny.cloudera.com/yarnblog>

Chapter Topics

Distributed Processing on an Apache Hadoop Cluster

- YARN Architecture
- Working With YARN
- Essential Points
- **Hands-On Exercise: Running and Monitoring a YARN Job**

Hands-On Exercise: Running and Monitoring a YARN Job

- In this exercise, you will submit an application to the cluster and monitor it using the YARN command line interface and Resource Manager UI
- Please refer to the Hands-On Exercise Manual for instructions



Apache Spark Basics

Chapter 5

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- **Apache Spark Basics**
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Apache Spark Basics

After completing this chapter, you will be able to

- **Describe how Spark SQL fits into the Spark stack**
- **Start and use the Python and Scala Spark shells**
- **Create DataFrames and perform simple queries**

Chapter Topics

Apache Spark Basics

- **What is Apache Spark?**
- Starting the Spark Shell
- Using the Spark Shell
- Getting Started with Datasets and DataFrames
- DataFrame Operations
- Essential Points
- **Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell**

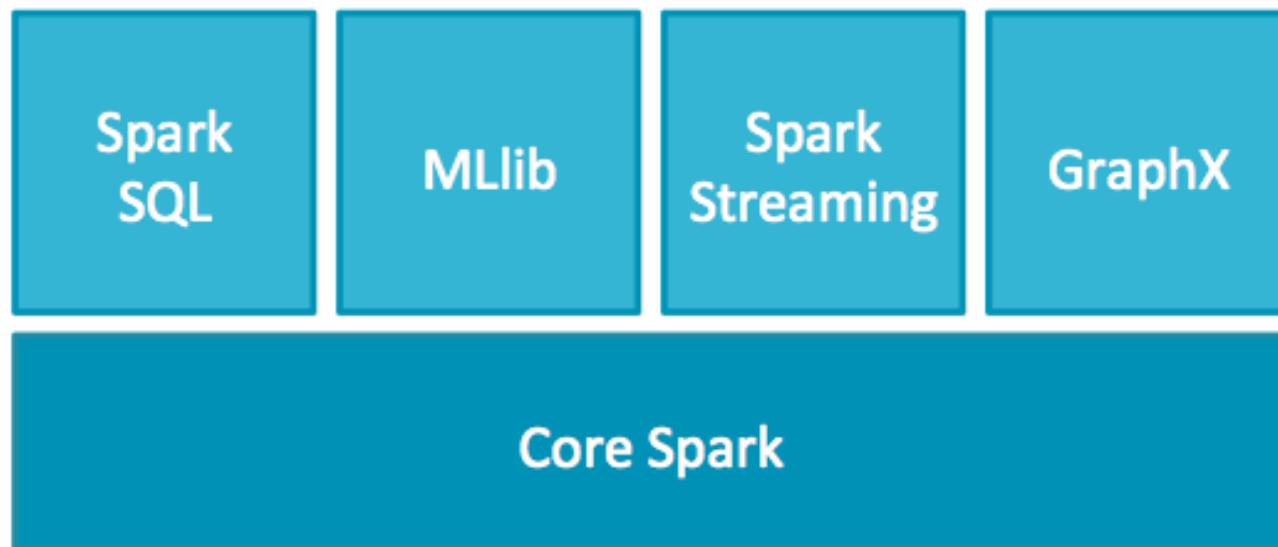
What Is Apache Spark?

- Apache Spark is a fast, general-purpose engine for large-scale data processing
- Written in Scala
 - Functional programming language that runs in a JVM
- Spark shell
 - Interactive—for learning, data exploration, or ad hoc analytics
 - Python and Scala
- Spark applications
 - For large scale data processing
 - Python, Scala, and Java



The Spark Stack

- **Spark provides a stack of libraries built on core Spark**
 - Core Spark provides the fundamental Spark abstraction: Resilient Distributed Datasets (RDDs)
 - Spark SQL works with structured data
 - MLlib supports scalable machine learning
 - Spark Streaming applications process data in real time
 - GraphX works with graphs and graph-parallel computation



Spark SQL

- **Spark SQL is a Spark library for working with structured data**
- **What does Spark SQL provide?**
 - The DataFrame and Dataset API
 - The primary entry point for developing Spark applications
 - DataFrames and Datasets are abstractions for representing structured data
 - Catalyst Optimizer—an extensible optimization framework
 - A SQL engine and command line interface

Chapter Topics

Apache Spark Basics

- What is Apache Spark?
- **Starting the Spark Shell**
- Using the Spark Shell
- Getting Started with Datasets and DataFrames
- DataFrame Operations
- Essential Points
- Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

The Spark Shell

- **The Spark shell provides an interactive Spark environment**
 - Often called a *REPL*, or Read/Evaluate/Print Loop
 - For learning, testing, data exploration, or ad hoc analytics
 - You can run the Spark shell using either Python or Scala
- **You typically run the Spark shell on a gateway node**

Starting the Spark Shell

- On a Cloudera cluster, the command to start the Spark shell is
 - pyspark for Python
 - spark-shell for Scala
- The Spark shell has a number of different start-up options, including
 - master: specify the cluster to connect to
 - jars: Additional JAR files
 - py-files: Additional Python files (Python only)
 - name: the name the Spark Application UI uses for this application
 - Defaults to PySparkShell (Python) or Spark shell (Scala)
 - help: Show all the available shell options

```
$ pyspark --name "My Application"
```

Spark Cluster Options (1)

- **Spark applications can run on these types of clusters**
 - Apache Hadoop YARN
 - Kubernetes
 - Apache Mesos
 - Spark Standalone
- **They can also run locally instead of on a cluster**
- **CDH, HDP, and CDP Data Center use YARN**
- **CDP Private Cloud and CDP Public Cloud use Kubernetes**
- **Specify the type or URL of the cluster using the `master` option**

Spark Cluster Options (2)

- Set the `master` option to specify cluster type
 - `yarn`
 - `local[*]` runs locally with as many threads as cores (default)
 - `local[n]` runs locally with n threads
 - `local` runs locally with a single thread

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

Language: Python

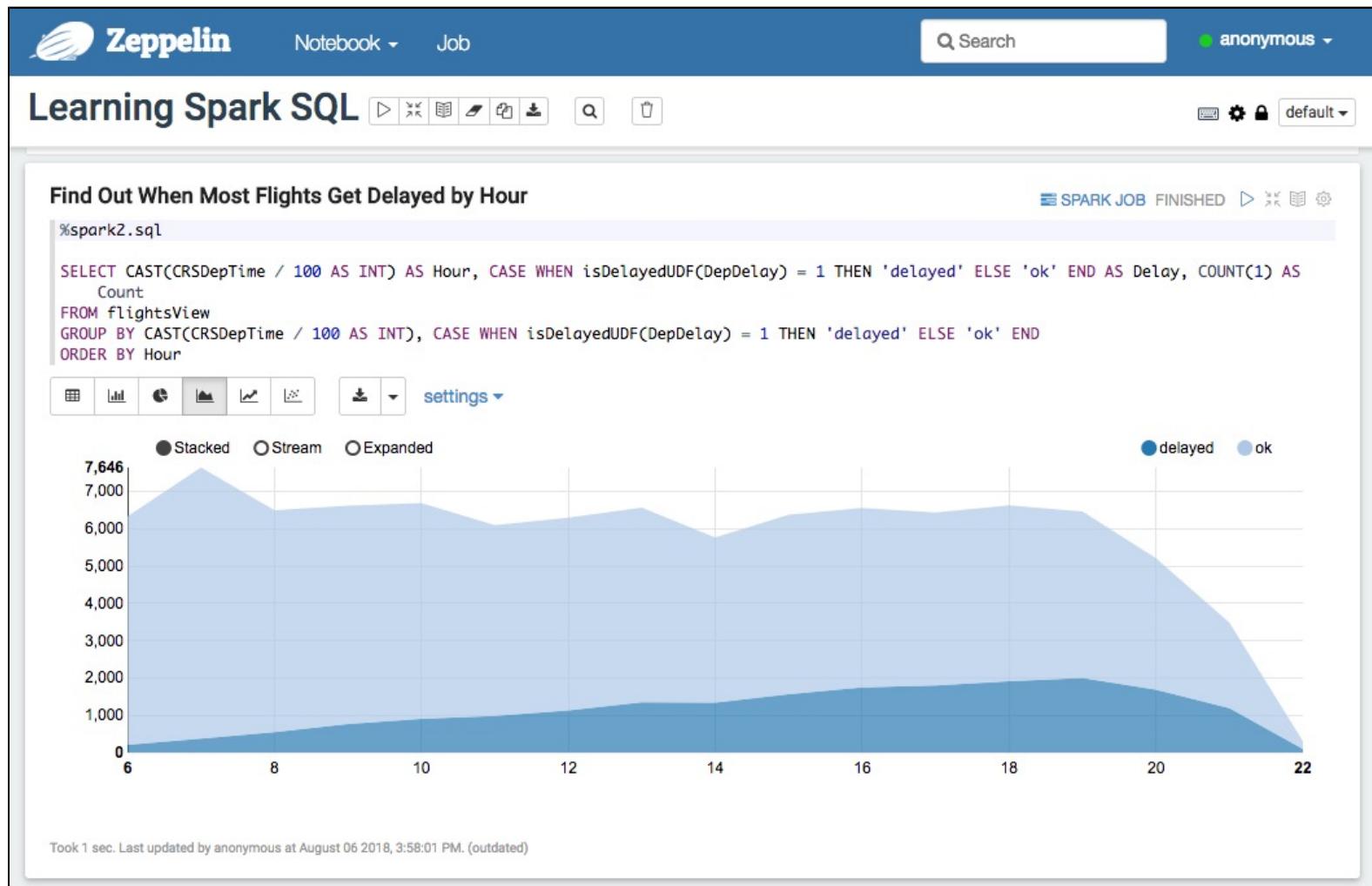
```
$ spark-shell --master yarn
```

Language: Scala

Apache Zeppelin

- Apache Zeppelin is a web-based notebook approach to interactive data analytics.

- Provides collaborative environment with Python, Scala, SQL, and more.



Chapter Topics

Apache Spark Basics

- What is Apache Spark?
- Starting the Spark Shell
- **Using the Spark Shell**
- Getting Started with Datasets and DataFrames
- DataFrame Operations
- Essential Points
- Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

Spark Session

- The main entry point for the Spark API is a Spark session
 - The Spark shell provides a preconfigured `SparkSession` object called `spark`

• • •

Welcome to

Using Python version 3.6.4 (default, Jan 16 2018 18:10:19)
SparkSession available as 'spark'.

In [1]: spark

Out [1]: <pyspark.sql.session.SparkSession at 0x1928b90>

Language: Python

Working with the Spark Session

- The `SparkSession` class provides functions and attributes to access all of Spark functionality
- Examples include
 - `sql`: execute a Spark SQL query
 - `catalog`: entry point for the Catalog API for managing tables
 - `read`: function to read data from a file or other data source
 - `conf`: object to manage Spark configuration settings
 - `sparkContext`: entry point for core Spark API

Log Levels (1)

- Spark logs messages using Apache Log4J
- Messages are tagged with their level

```
...
19/04/03 11:30:01 WARN Utils: Service 'SparkUI' ...
19/04/03 11:30:02 INFO SparkContext: Created broadcast 0 ...
19/04/03 11:30:02 INFO FileInputFormat: Total input ...
19/04/03 11:30:02 INFO SparkContext: Starting job: ...
19/04/03 11:30:02 INFO DAGScheduler: Got job 0 ...
...
...
```

Log Levels (2)

- Available log levels are
 - TRACE
 - DEBUG
 - INFO (default level in Spark applications)
 - WARN (default level in Spark shell)
 - ERROR
 - FATAL
 - OFF

Setting the Log Level

- Set the log level for the current Spark shell using the Spark context `setLogLevel` method

```
> spark.sparkContext.setLogLevel("INFO")
```

Chapter Topics

Apache Spark Basics

- What is Apache Spark?
- Starting the Spark Shell
- Using the Spark Shell
- **Getting Started with Datasets and DataFrames**
- DataFrame Operations
- Essential Points
- Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

DataFrames and Datasets

- **DataFrames and Datasets are the primary representation of data in Spark**
- **DataFrames represent structured data in a tabular form**
 - DataFrames model data similar to tables in an RDBMS
 - DataFrames consist of a collection of loosely typed Row objects
 - Rows are organized into columns described by a schema
- **Datasets represent data as a collection of objects of a specified type**
 - Datasets are strongly-typed—type checking is enforced at compile time rather than run time
 - An associated schema maps object properties to a table-like structure of rows and columns
 - Datasets are only defined in Scala and Java
 - **DataFrame** is an alias for **Dataset [Row]**—Datasets containing Row objects

DataFrames and Rows

- **DataFrames contain a collection of Row objects**
 - Rows contain an ordered collection of values
 - Row values can be basic types (such as integers, strings, and floats) or collections of those types (such as arrays and lists)
 - A schema maps column names and types to the values in a row

Example: Creating a DataFrame (1)

- The `users.json` text file contains sample data
 - Each line contains a single JSON record that can include a name, age, and postal code field

```
{"name":"Alice", "pcode":"94304"}  
{"name":"Brayden", "age":30, "pcode":"94304"}  
{"name":"Carla", "age":19, "pcode":"10036"}  
{"name":"Diana", "age":46}  
{"name":"Étienne", "pcode":"94104"}
```

Example: Creating a DataFrame (2)

- Create a DataFrame using `spark.read`
- Returns the Spark session's `DataFrameReader`
- Call `json` function to create a new DataFrame

```
> usersDF = \  
  spark.read.json("users.json")
```

Language: Python

Example: Creating a DataFrame (3)

- DataFrames always have an associated schema
- DataFrameReader can infer the schema from the data
- Use printSchema to show the DataFrame's schema

```
> usersDF = \
  spark.read.json("users.json")

> usersDF.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
 |-- pcode: string (nullable = true)
```

Language: Python

Example: Creating a DataFrame (4)

- The `show` method displays the first few rows in a tabular format

```
> usersDF = \
  spark.read.json("users.json")

> usersDF.printSchema()
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
 |-- pcode: string (nullable = true)

> usersDF.show()
+---+---+---+
| age| name|pcode|
+---+---+---+
| null| Alice|94304|
| 30| Brayden|94304|
| 19| Carla|10036|
| 46| Diana| null|
| null|Etienne|94104|
+---+---+---+
```

Language: Python

Chapter Topics

Apache Spark Basics

- What is Apache Spark?
- Starting the Spark Shell
- Using the Spark Shell
- Getting Started with Datasets and DataFrames
- **DataFrame Operations**
- Essential Points
- Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

DataFrame Operations

- **There are two main types of DataFrame operations**
 - *Transformations* create a new DataFrame based on existing one(s)
 - Transformations are executed in parallel by the application's executors
 - *Actions* output data values from the DataFrame
 - Output is typically returned from the executors to the main Spark program (called the *driver*) or saved to a file

DataFrame Operations: Actions

- Some common DataFrame actions include
 - `count`: returns the number of rows
 - `first`: returns the first row (synonym for `head()`)
 - `take(n)`: returns the first *n* rows as an array (synonym for `head(n)`)
 - `show(n)`: display the first *n* rows in tabular form (default is 20 rows)
 - `collect`: returns all the rows in the DataFrame as an array
 - `write`: save the data to a file or other data source

Example: take Action

```
> usersDF = spark.read.json("users.json")
> users = usersDF.take(3)
[Row(age=None, name=u'Alice', pcode=u'94304'),
 Row(age=30, name=u'Brayden', pcode=u'94304'),
 Row(age=19, name=u'Carla', pcode=u'10036')]
```

Language: Python

```
> val usersDF = spark.read.json("users.json")
> val users = usersDF.take(3)
usersDF: Array[org.apache.spark.sql.Row] =
  Array([null,Alice,94304],
    [30,Brayden,94304],
    [19,Carla,10036])
```

Language: Scala

DataFrame Operations: Transformations (1)

- **Transformations create a new DataFrame based on an existing one**
 - The new DataFrame may have the same schema or a different one
- **Transformations do not return any values or data to the driver**
 - Data remains distributed across the application's executors
- **DataFrames are immutable**
 - Data in a DataFrame is never modified
 - Use transformations to create a new DataFrame with the data you need

DataFrame Operations: Transformations (2)

- Common transformations include
 - `select`: only the specified columns are included
 - `where`: only rows where the specified expression is true are included (synonym for `filter`)
 - `orderBy`: rows are sorted by the specified column(s) (synonym for `sort`)
 - `join`: joins two DataFrames on the specified column(s)
 - `limit(n)`: creates a new DataFrame with only the first `n` rows

Example: select and where Transformations

```
> nameAgeDF = usersDF.select("name","age")
> nameAgeDF.show()
+-----+----+
|    name|  age|
+-----+----+
|    Alice|null|
| Brayden|   30|
|   Carla|   19|
| Diana|   46|
|Etienne|null|
+-----+----+

> over20DF = usersDF.where("age > 20")
> over20DF.show()
+-----+----+
| age|  name|pcode|
+-----+----+
| 30|Brayden|94304|
| 46| Diana| null|
+-----+----+
```

Language: Python

Defining Queries

- A sequence of transformations followed by an action is a *query*

```
> nameAgeDF = usersDF.select("name","age")
> nameAgeOver20DF = nameAgeDF.where("age > 20")
> nameAgeOver20DF.show()
+---+-----+
| age |    name |
+---+-----+
| 30 | Brayden|
| 46 | Diana   |
+---+-----+
```

Language: Python

Chaining Transformations (1)

- Transformations in a query can be chained together
- These two examples perform the same query in the same way
 - Differences are only syntactic

```
> nameAgeDF = usersDF.select("name", "age")
> nameAgeOver20DF = nameAgeDF.where("age > 20")
> nameAgeOver20DF.show()
```

Language: Python

```
> usersDF.select("name", "age").where("age > 20").show()
```

Language: Python

Chaining Transformations (2)

- This is the same example with Scala
 - The two code snippets are equivalent

```
> val nameAgeDF = usersDF.select("name", "age")
> val nameAgeOver20DF = nameAgeDF.where("age > 20")
> nameAgeOver20DF.show
```

Language: Scala

```
> usersDF.select("name", "age").where("age > 20").show
```

Language: Scala

Chapter Topics

Apache Spark Basics

- What is Apache Spark?
- Starting the Spark Shell
- Using the Spark Shell
- Getting Started with Datasets and DataFrames
- DataFrame Operations
- **Essential Points**
- Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

Essential Points

- Apache Spark is a framework for analyzing and processing big data
- The Python and Scala Spark shells are command line REPLs for executing Spark interactively
 - Spark applications run in batch mode outside the shell
- DataFrames represent structured data in tabular form by applying a schema
- Types of DataFrame operations
 - Transformations create new DataFrames by transforming data in existing ones
 - Actions collect values in a DataFrame and either save them or return them to the Spark driver
- A query consists of a sequence of transformations followed by an action

Chapter Topics

Apache Spark Basics

- What is Apache Spark?
- Starting the Spark Shell
- Using the Spark Shell
- Getting Started with Datasets and DataFrames
- DataFrame Operations
- Essential Points
- **Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell**

Introduction to Spark Exercises: Choose 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 commands that can be copied into the PySpark shell
 - **.scalaspark**: Scala commands that can be copied into the Scala Spark shell
 - **.py**: Complete Python Spark applications
 - **.scala**: Complete Scala Spark applications

Hands-On Exercise: Exploring DataFrames Using the Apache Spark Shell

- In this exercise, you will start the Python or Scala Spark shell and practice creating and querying a DataFrame
- Please refer to the Hands-On Exercise Manual for instructions



Working with DataFrames and Schemas

Chapter 6

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- **Working with DataFrames and Schemas**
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Working with DataFrames and Schemas

After completing this chapter, you will be able to

- Create DataFrames from a variety of sources
- Specify format and options to save DataFrames
- Define a DataFrame schema through inference or programmatically
- Explain the difference between lazy and eager query execution

Chapter Topics

Working with DataFrames and Schemas

- **Creating DataFrames from Data Sources**
- Saving DataFrames to Data Sources
- DataFrame Schemas
- Eager and Lazy Execution
- Essential Points
- Hands-On Exercise: Working with DataFrames and Schemas

DataFrame Data Sources

- **DataFrames read data from and write data to *data sources***
- **Spark SQL supports a wide range of data source types and formats**
 - Text files
 - CSV, JSON, plain text
 - Binary format files
 - Apache Parquet, Apache ORC, Apache Avro data format
 - Tables
 - Hive metastore, JDBC
 - Cloud
 - Such as Amazon S3 and Microsoft ADLS
- **You can also use custom or third-party data source types**

DataFrames and Apache Parquet Files

- Parquet is a very common file format for DataFrame data
- Features of Parquet
 - Optimized binary storage of structured data
 - Schema metadata is embedded in the file
 - Efficient performance and size for large amounts of data
 - Supported by many Hadoop ecosystem tools
 - Spark, Hadoop MapReduce, Hive, and others
- Use parquet-tools to view Parquet file schema and data
 - Use head to display the first few records

```
$ parquet-tools head mydatafile.parquet
```

- Use schema to view the schema

```
$ parquet-tools schema mydatafile.parquet
```

Creating a DataFrame from a Data Source

- `spark.read` returns a `DataFrameReader` object
- Use `DataFrameReader` settings to specify how to load data from the data source
 - `format` indicates the data source type, such as `csv`, `json`, or `parquet` (the default is `parquet`)
 - `option` specifies a key/value setting for the underlying data source
 - `schema` specifies a schema to use instead of inferring one from the data source
- Create the DataFrame based on the data source
 - `load` loads data from a file or files

Examples: Creating a DataFrame from a Data Source

- Example: Read a CSV text file
 - Treat the first line in the file as a header instead of data

```
myDF = spark.read. \
    format("csv"). \
    option("header","true"). \
    load("/loudacre/myFile.csv")
```

Language: Python

- Example: Read an Avro file

```
myDF = spark.read. \
    format("avro"). \
    load("/loudacre/myData.avro")
```

Language: Python

DataFrameReader Convenience Functions

- You can call a format-specific load function for some formats
 - A shortcut instead of setting the format and using `load`
- The following two code examples are equivalent

```
spark.read.format("csv").load("/loudacre/myFile.csv")
```

```
spark.read.csv("/loudacre/myFile.csv")
```

Specifying Data Source File Locations

- You must specify a location when reading from a file data source
 - The location can be a single file, a list of files, a directory, or a wildcard
 - Examples
 - `spark.read.json("myfile.json")`
 - `spark.read.json("mydata/")`
 - `spark.read.json("mydata/*.json")`
 - `spark.read.json("myfile1.json","myfile2.json")`
- Files and directories are referenced by absolute or relative URI
 - Relative URI (uses default file system)
 - `myfile.json`
 - Absolute URI
 - `hdfs://nnhost/loudacre/myfile.json`
 - `file:/home/training/myfile.json`

Creating DataFrames from Hive Tables

- Apache Hive provides database-like access to data in HDFS
 - Applies schemas to HDFS files
 - Metadata is stored in the *Hive metastore*
- Spark can read from and write to Hive tables
 - Infers the DataFrame schema from the Hive metadata
- Spark Hive support must be enabled and configured with location of the Hive warehouse in HDFS

```
usersDF = spark.read.table("users")
```

Language: Python

Creating DataFrames from Data in Memory

- You can also create DataFrames from a collection of in-memory data
 - Useful for testing and integrations

```
val mydata = List(("Josiah","Bartlet"),
                  ("Harry","Potter"))

val myDF = spark.createDataFrame(mydata)
myDF.show
+---+---+
| _1|_2|
+---+---+
| Josiah|Bartlet|
| Harry| Potter|
+---+---+
```

Language: Scala

Chapter Topics

Working with DataFrames and Schemas

- Creating DataFrames from Data Sources
- **Saving DataFrames to Data Sources**
- DataFrame Schemas
- Eager and Lazy Execution
- Essential Points
- Hands-On Exercise: Working with DataFrames and Schemas

Key DataFrameWriter Functions

- The DataFrame `write` function returns a DataFrameWriter
 - Saves data to a data source such as a table or set of files
 - Works similarly to DataFrameReader
- DataFrameWriter methods
 - `format` specifies a data source type
 - `mode` determines the behavior if the directory or table already exists
 - `error`, `overwrite`, `append`, or `ignore` (default is `error`)
 - `partitionBy` stores data in partitioned directories in the form `column=value` (as with Hive partitioning)
 - `option` specifies properties for the target data source
 - `save` saves the data as files in the specified directory
 - Or use `json`, `csv`, `parquet`, and so on
 - `saveAsTable` saves the data to a Hive metastore table
 - Data location based on Hive warehouse default
 - Set `path` option to override location

Examples: Saving a DataFrame to a Data Source

- Example: Write data to a Hive metastore table called my_table
 - Append the data if the table already exists
 - Use an alternate location

```
myDF.write. \
    mode("append"). \
    option("path","/loudacre/mydata"). \
    saveAsTable("my_table")
```

- Example: Write data as Parquet files in the mydata directory

```
myDF.write.save("mydata")
```

Saving Data to Files

- When you save data from a DataFrame, you must specify a directory
 - Spark saves the data to one or more part- files in the directory

```
devDF.write.csv("devices")
```

```
$ hdfs dfs -ls
devices
Found 4 items
-rw-r--r-- 3 training training      0 ... devices/_SUCCESS
-rw-r--r-- 3 training training  2119 ... devices/part-00000-e0fa6381....csv
-rw-r--r-- 3 training training  2202 ... devices/part-00001-e0fa6381....csv
-rw-r--r-- 3 training training  2333 ... devices/part-00002-e0fa6381....csv
```

Chapter Topics

Working with DataFrames and Schemas

- Creating DataFrames from Data Sources
- Saving DataFrames to Data Sources
- **DataFrame Schemas**
- Eager and Lazy Execution
- Essential Points
- Hands-On Exercise: Working with DataFrames and Schemas

DataFrame Schemas

- Every DataFrame has an associated schema
 - Defines the names and types of columns
 - Immutable and defined when the DataFrame is created

```
myDF.printSchema()
root
|-- lastName: string (nullable = true)
|-- firstName: string (nullable = true)
|-- age: integer (nullable = true)
```

- When creating a new DataFrame from a data source, the schema can be
 - Automatically inferred from the data source
 - Specified programmatically
- When a DataFrame is created by a transformation, Spark calculates the new schema based on the query

Inferred Schemas

- **Spark can load schemas from structured data, such as**
 - Parquet, ORC, and Avro data files—schema is embedded in the file
 - Hive tables—schema is defined in the Hive metastore
 - Parent DataFrames
- **Spark can also attempt to infer a schema from semi-structured data sources**
 - For example, JSON and CSV

Example: Inferring the Schema of a CSV File (No Header)

```
02134,Hopper,Grace,52
94020,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

```
spark.read.option("inferSchema","true").csv("people.csv"). \
    printSchema()
root
|-- _c0: integer (nullable = true)
|-- _c1: string (nullable = true)
|-- _c2: string (nullable = true)
|-- _c3: integer (nullable = true)
```

Example: Inferring the Schema of a CSV File (with Header)

```
pcode,lastName,firstName,age  
02134,Hopper,Grace,52  
94020,Turing,Alan,32  
94020,Lovelace,Ada,28  
87501,Babbage,Charles,49  
02134,Wirth,Niklaus,48
```

```
spark.read.option("inferSchema","true"). \  
  option("header","true").csv("people.csv"). \  
  printSchema()  
root  
|-- pcode: integer (nullable = true)  
|-- lastName: string (nullable = true)  
|-- firstName: string (nullable = true)  
|-- age: integer (nullable = true)
```

Inferred Schemas versus Manual Schemas

- **Drawbacks to relying on Spark's automatic schema inference**
 - Inference requires an initial file scan, which may take a long time
 - The schema may not be correct for your use case
- **You can define the schema manually instead**
 - A schema is a `StructType` object containing a list of `StructField` objects
 - Each `StructField` represents a column in the schema, specifying
 - Column name
 - Column data type
 - Whether the data can be null (optional—the default is true)

Example: Incorrect Schema Inference

- Example: This inferred schema for the CSV file is incorrect
 - The pcode column should be a string

```
spark.read.option("inferSchema","true"). \
    option("header","true").csv("people.csv"). \
    printSchema()
root
|-- pcode: integer (nullable = true)
|-- lastName: string (nullable = true)
|-- firstName: string (nullable = true)
|-- age: integer (nullable = true)
```

Example: Defining a Schema Programmatically (Python)

```
from pyspark.sql.types import *

columnsList = [
    StructField("pcode", StringType()),
    StructField("lastName", StringType()),
    StructField("firstName", StringType()),
    StructField("age", IntegerType())]

peopleSchema = StructType(columnsList)
```

Language: Python

Example: Defining a Schema Programmatically (Scala)

```
import org.apache.spark.sql.types._

val columnsList = List(
    StructField("pcode", StringType),
    StructField("lastName", StringType),
    StructField("firstName", StringType),
    StructField("age", IntegerType))

val peopleSchema = StructType(columnsList)
```

Language: Scala

Example: Applying a Schema Manually

```
spark.read.option("header","true").  
    schema(peopleSchema).csv("people.csv").printSchema()  
root  
|-- pcode: string (nullable = true)  
|-- lastName: string (nullable = true)  
|-- firstName: string (nullable = true)  
|-- age: integer (nullable = true)
```

Chapter Topics

Working with DataFrames and Schemas

- Creating DataFrames from Data Sources
- Saving DataFrames to Data Sources
- DataFrame Schemas
- **Eager and Lazy Execution**
- Essential Points
- Hands-On Exercise: Working with DataFrames and Schemas

Eager and Lazy Execution

- Operations are *eager* when they are executed as soon as the statement is reached in the code
- Operations are *lazy* when the execution occurs only when the result is referenced
- Spark queries execute both lazily and eagerly
 - DataFrame schemas are determined eagerly
 - Data transformations are executed lazily
- Lazy execution is triggered when an action is called on a series of transformations

Example: Eager and Lazy Execution (1)

```
> usersDF = \  
spark.read.json("users.json")
```

users.json

name	age	pcode

Language: Python

Example: Eager and Lazy Execution (2)

```
> usersDF = \  
    spark.read.json("users.json")  
> nameAgeDF = \  
    usersDF.select("name", "age")
```

Language: Python

users.json

name	age	pcode

name	age

Example: Eager and Lazy Execution (3)

```
> usersDF = \  
    spark.read.json("users.json")  
> nameAgeDF = \  
    usersDF.select("name", "age")  
> nameAgeDF.show()
```

Language: Python

users.json

name	age	pcode
Alice	(null)	94304
Brayden	30	94304
...

name	age
Alice	(null)
Brayden	30
...	...

Example: Eager and Lazy Execution (4)

```
> usersDF = \  
    spark.read.json("users.json")  
> nameAgeDF = \  
    usersDF.select("name", "age")  
> nameAgeDF.show()  
+-----+---+  
| name | age |  
+-----+---+  
| Alice | null |  
| Brayden | 30 |  
| Carla | 19 |  
...  
...
```

Language: Python

users.json

name	age	pcode
Alice	(null)	94304
Brayden	30	94304
...

name	age
Alice	(null)
Brayden	30
...	...

Chapter Topics

Working with DataFrames and Schemas

- Creating DataFrames from Data Sources
- Saving DataFrames to Data Sources
- DataFrame Schemas
- Eager and Lazy Execution
- **Essential Points**
- Hands-On Exercise: Working with DataFrames and Schemas

Essential Points

- **DataFrames can be loaded from and saved to several different types of data sources**
 - Semi-structured text files like CSV and JSON
 - Structured binary formats like Parquet, Avro, and ORC
 - Hive and JDBC tables
- **DataFrames can infer a schema from a data source, or you can define one manually**
- **DataFrame schemas are determined *eagerly* (at creation) but queries are executed *lazily* (when an action is called)**

Chapter Topics

Working with DataFrames and Schemas

- Creating DataFrames from Data Sources
- Saving DataFrames to Data Sources
- DataFrame Schemas
- Eager and Lazy Execution
- Essential Points
- **Hands-On Exercise: Working with DataFrames and Schemas**

Hands-On Exercise: Working with DataFrames and Schemas

- In this exercise, you will create DataFrames and define schemas based on different data sources**
- Please refer to the Hands-On Exercise Manual for instructions**



Analyzing Data with DataFrame Queries

Chapter 7

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- **Analyzing Data with DataFrame Queries**
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Analyzing Data with DataFrame Queries

After completing this chapter, you will be able to

- Use column names, column references, and column expressions when querying
- Group and aggregate values in a column
- Join two DataFrames

Chapter Topics

Analyzing Data with DataFrame Queries

- **Querying DataFrames Using Column Expressions**
- Grouping and Aggregation Queries
- Joining DataFrames
- Essential Points
- Hands-On Exercise: Analyzing Data with DataFrame Queries

Columns, Column Names, and Column Expressions

- Most DataFrame transformations require you to specify a column or columns
 - `select(column1, column2, ...)`
 - `orderBy(column1, column2, ...)`
- For many simple queries, you can just specify the column name as a string
 - `peopleDF.select("firstName", "lastName")`
- Some types of transformations use *column references* or *column expressions* instead of column name strings

```
▶      def select(col: String, cols: String*): DataFrame  
          Selects a set of columns.  


---

▶      def select(cols: Column*): DataFrame  
          Selects a set of column based expressions.
```

Example: Column References (Python)

- In Python, there are two equivalent ways to refer to a column

```
peopleDF = spark.read.option("header","true").csv("people.csv")  
  
peopleDF['age']  
Column<age>  
  
peopleDF.age  
Column<age>  
  
peopleDF.select(peopleDF.age).show()  
+---+  
| age |  
+---+  
| 52 |  
| 32 |  
| 28 |  
...  
+
```

Language: Python

Example: Column References (Scala)

- In Scala, there are two ways to refer to a column
 - The first uses the column name with the DataFrame
 - The second uses the column name only, and is not fully resolved until used in a transformation

```
val peopleDF = spark.read.  
option("header","true").csv("people.csv")  
  
peopleDF("age")  
org.apache.spark.sql.Column = age  
  
$"age"  
org.apache.spark.sql.ColumnName = age  
  
peopleDF.select(peopleDF("age")).show  
+---+  
| age |  
+---+  
| 52 |  
| 32 |  
...  
...
```

Language: Scala

Column Expressions

- Using column references instead of simple strings allows you to create *column expressions*
- Column operations include
 - Arithmetic operators such as +, -, %, /, and *
 - Comparative and logical operators such as >, <, && and ||
 - The equality comparator is === in Scala, and == in Python
 - String functions such as `contains`, `like`, and `substring`
 - Data testing functions such as `isNull`, `isNotNull`, and `NaN` (not a number)
 - Sorting functions such as `asc` and `desc`
 - Work only when used in `sort/orderBy`
- For the full list of operators and functions, see the API documentation for `Column`

Examples: Column Expressions (Python)

```
peopleDF.select("lastName", peopleDF.age * 10).show()
+-----+
| lastName| (age * 10) |
+-----+
|   Hopper|      520 |
|  Turing|      320 |
...
peopleDF.where(peopleDF.firstName.startswith("A")).show()
+-----+-----+-----+-----+
| pcode| lastName| firstName| age |
+-----+-----+-----+-----+
| 94020|  Turing|     Alan| 32 |
| 94020| Lovelace|     Ada| 28 |
+-----+-----+-----+-----+
```

Language: Python

Examples: Column Expressions (Scala)

```
peopleDF.select($"lastName", $"age" * 10).show
+-----+-----+
| lastName | (age * 10) |
+-----+-----+
| Hopper |      520 |
| Turing |      320 |
...
peopleDF.where(peopleDF("firstName").startsWith("A")).show
+-----+-----+-----+-----+
| pcode | lastName | firstName | age |
+-----+-----+-----+-----+
| 94020 | Turing | Alan | 32 |
| 94020 | Lovelace | Ada | 28 |
+-----+-----+-----+-----+
```

Language: Scala

Column Aliases (1)

- Use the column alias function to rename a column in a result set
 - name is a synonym for alias
- Example (Python): Use column name age_10 instead of (age * 10)

```
peopleDF.select("lastName",
(peopleDF.age * 10).alias("age_10")).show()
+-----+-----+
| lastName | age_10 |
+-----+-----+
| Hopper |    520 |
| Turing |    320 |
...
...
```

Language: Python

Column Aliases (2)

- Example (Scala): Use column name `age_10` instead of `(age * 10)`

```
peopleDF.select($"lastName",
  ($"age" * 10).alias("age_10")).show
+-----+-----+
|lastName|age_10|
+-----+-----+
|  Hopper|    520|
|  Turing|    320|
...
...
```

Language: Scala

Chapter Topics

Analyzing Data with DataFrame Queries

- Querying DataFrames Using Column Expressions
- **Grouping and Aggregation Queries**
- Joining DataFrames
- Essential Points
- Hands-On Exercise: Analyzing Data with DataFrame Queries

Aggregation Queries

- Aggregation queries perform a calculation on a set of values and return a single value
- To execute an aggregation on a set of grouped values, use `groupBy` combined with an aggregation function
- Example: How many people are in each postal code?

```
peopleDF.groupBy("PCODE").count().show()
+----+---+
|PCODE|COUNT|
+----+---+
|94020|    2|
|87501|    1|
|02134|    2|
+----+---+
```

The groupBy Transformation

- **groupBy takes one or more column names or references**
 - In Scala, returns a RelationalGroupedDataset object
 - In Python, returns a GroupedData object
- **Returned objects provide aggregation functions, including**
 - count
 - max and min
 - mean (and its alias avg)
 - sum
 - pivot
 - agg (aggregates using additional aggregation functions)

Additional Aggregation Functions

- The `functions` object provides several additional aggregation functions
- Aggregate functions include
 - `first/last` returns the first or last items in a group
 - `countDistinct` returns the number of unique items in a group
 - `approx_count_distinct` returns an approximate counts of unique items
 - Much faster than a full count
 - `stddev` calculates the standard deviation for a group of values
 - `var_sample/var_pop` calculates the variance for a group of values
 - `covar_samp/covar_pop` calculates the sample and population covariance of a group of values
 - `corr` returns the correlation of a group of values

Example: Using the functions Object

```
from pyspark.sql.functions import stddev

peopleDF.groupBy("PCODE").agg(stddev("age")).show()
+-----+
| pcode | stddev_samp(age) |
+-----+
| 94020 | 0.7071067811865476 |
| 87501 |           NaN |
| 02134 | 2.1213203435596424 |
+-----+
Language: Python
```

Chapter Topics

Analyzing Data with DataFrame Queries

- Querying DataFrames Using Column Expressions
- Grouping and Aggregation Queries
- **Joining DataFrames**
- Essential Points
- Hands-On Exercise: Analyzing Data with DataFrame Queries

Joining DataFrames

- Use the `join` transformation to join two DataFrames
- DataFrames support several types of joins
 - `inner` (default)
 - `outer`
 - `left_outer`
 - `right_outer`
 - `leftsemi`
- The `crossJoin` transformation joins every element of one DataFrame with every element of the other

Example: A Simple Inner Join (1)

people-no-pcode.csv

```
PCODE,lastName,firstName,age
02134,Hopper,Grace,52
,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

pCodes.csv

```
PCODE,city,state
02134,Boston,MA
94020,Palo Alto,CA
87501,Santa Fe,NM
60645,Chicago,IL
```

```
val peopleDF = spark.read.option("header","true").
  csv("people-no-pcode.csv")

val pCodesDF = spark.read.
  option("header","true").csv("pCodes.csv")
```

Language: Scala
Example continued on next slide...

Example: A Simple Inner Join (2)

- This example shows an inner join when join column is in both DataFrames

```
peopleDF.join(pcodesDF, "pcode").show()  
+-----+-----+-----+-----+  
| pcode | lastName | firstName | age | city | state |  
+-----+-----+-----+-----+  
| 02134 | Hopper | Grace | 52 | Boston | MA |  
| 94020 | Lovelace | Ada | 28 | Palo Alto | CA |  
| 87501 | Babbage | Charles | 49 | Santa Fe | NM |  
| 02134 | Wirth | Niklaus | 48 | Boston | MA |  
+-----+-----+-----+-----+  
...  
+
```

Example: A Left Outer Join (1)

- Specify type of join as `inner` (default), `outer`, `left_outer`, `right_outer`, or `leftsemi`

```
peopleDF.join(pcodesDF, "pcode", "left_outer").show()
+-----+-----+-----+-----+
|pcode|lastName|firstName|age|      city|state|
+-----+-----+-----+-----+
|02134|  Hopper|    Grace|  52|  Boston|   MA|
| null|  Turing|     Alan|  32|    null|  null|
|94020|Lovelace|      Ada|  28|Palo Alto|    CA|
|87501|Babbage|  Charles|  49|Santa Fe|    NM|
|02134|   Wirth| Niklaus|  48|  Boston|   MA|
+-----+-----+-----+-----+
```

Language: Python

Example: A Left Outer Join (2)

- Specify type of join as `inner` (default), `outer`, `left_outer`, `right_outer`, or `leftsemi`

```
peopleDF.join(pcodesDF,
  peopleDF("pcode") === pcodesDF("pcode"),
"left_outer").show
+-----+-----+-----+-----+
|pcode|lastName|firstName|age|      city|state|
+-----+-----+-----+-----+
|02134|  Hopper|    Grace|  52|  Boston|   MA|
| null|  Turing|     Alan|  32|    null|  null|
|94020|Lovelace|      Ada|  28|Palo Alto|   CA|
|87501|Babbage|  Charles|  49|Santa Fe|   NM|
|02134|   Wirth| Niklaus|  48|  Boston|   MA|
+-----+-----+-----+-----+
```

Language: Scala

Example: Joining on Columns with Different Names (1)

`people-no-pcode.csv`

```
PCODE,lastName,firstName,age
02134,Hopper,Grace,52
,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

`zcodes.csv`

```
zip,city,state
02134,Boston,MA
94020,Palo Alto,CA
87501,Santa Fe,NM
60645,Chicago,IL
```

Example: Joining on Columns with Different Names (2)

- Use column expressions when the names of the join columns are different
 - The result includes both of the join columns

```
peopleDF.join(zcodesDF, $"pcode" === $"zip").show
+-----+-----+-----+-----+-----+
|pcode|lastName|firstName|age|  zip|      city|state|
+-----+-----+-----+-----+-----+
|02134|  Hopper|    Grace| 52|02134|    Boston|   MA|
|94020|Lovelace|        Ada| 28|94020|Palo Alto|    CA|
|87501|  Babbage|   Charles| 49|87501|Santa Fe|    NM|
|02134|    Wirth|  Niklaus| 48|02134|    Boston|   MA|
+-----+-----+-----+-----+-----+
```

Language: Scala

Example: Joining on Columns with Different Names (3)

```
peopleDF.join(zcodesDF, peopleDF.pcode == zcodesDF.zip).show()
+-----+-----+-----+-----+-----+
| pcode | lastName | firstName | age | zip |      city | state |
+-----+-----+-----+-----+-----+
| 02134 | Hopper | Grace | 52 | 02134 | Boston | MA |
| 94020 | Lovelace | Ada | 28 | 94020 | Palo Alto | CA |
| 87501 | Babbage | Charles | 49 | 87501 | Santa Fe | NM |
| 02134 | Wirth | Niklaus | 48 | 02134 | Boston | MA |
+-----+-----+-----+-----+-----+
```

Language: Python

Chapter Topics

Analyzing Data with DataFrame Queries

- Querying DataFrames Using Column Expressions
- Grouping and Aggregation Queries
- Joining DataFrames
- **Essential Points**
- Hands-On Exercise: Analyzing Data with DataFrame Queries

Essential Points

- **Columns in a DataFrame can be specified by name or Column objects**
 - You can define column expressions using column operators
- **Calculate aggregate values on groups of rows using groupBy and an aggregation function**
- **Use the join operation to join two DataFrames**
 - Supports inner, left outer, right outer and semi joins

Chapter Topics

Analyzing Data with DataFrame Queries

- Querying DataFrames Using Column Expressions
- Grouping and Aggregation Queries
- Joining DataFrames
- Essential Points
- **Hands-On Exercise: Analyzing Data with DataFrame Queries**

Hands-On Exercise: Analyzing Data with DataFrame Queries

- In this exercise, you will analyze different sets of data using DataFrame queries
- Please refer to the Hands-On Exercise Manual for instructions



RDD Overview

Chapter 8

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview**
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

RDD Overview

After completing this chapter, you will be able to

- Explain what RDDs are and how they differ from DataFrames and Datasets
- Load and save RDDs with a variety of data source types
- Transform RDD data and return query results

Chapter Topics

RDD Overview

- **RDD Overview**
- RDD Data Sources
- Creating and Saving RDDs
- RDD Operations
- Essential Points
- Hands-On Exercise: Working With RDDs

Resilient Distributed Datasets (RDDs)

- **RDDs are part of core Spark**
- ***Resilient Distributed Dataset (RDD)***
 - *Resilient*: If data in memory is lost, it can be recreated
 - *Distributed*: Processed across the cluster
 - *Dataset*: Initial data can come from a source such as a file, or it can be created programmatically
- **Despite the name, RDDs are *not* Spark SQL Dataset objects**
 - RDDs predate Spark SQL and the DataFrame/Dataset API

Comparing RDDs to DataFrames and Datasets (1)

- **RDDs are unstructured**
 - No schema defining columns and rows
 - Not table-like; cannot be queried using SQL-like transformations such as `where` and `select`
 - RDD transformations use lambda functions
- **RDDs can contain any type of object**
 - DataFrames are limited to Row objects
 - Datasets are limited to Row objects, case class objects (products), and primitive types

Comparing RDDs to DataFrames and Datasets (2)

- **RDDs are used in all Spark languages (Python, Scala, Java)**
 - Strongly-typed in Scala and Java, like Datasets
- **RDDs are not optimized by the Catalyst optimizer**
 - Manually coded RDDs are typically less efficient than DataFrames
- **You can use RDDs to create DataFrames and Datasets**
 - RDDs are often used to convert unstructured or semi-structured data into structured form
 - You can also work directly with the RDDs that underlie DataFrames and Datasets

Chapter Topics

RDD Overview

- RDD Overview
- **RDD Data Sources**
- Creating and Saving RDDs
- RDD Operations
- Essential Points
- Hands-On Exercise: Working With RDDs

RDD Data Types

- **RDDs can hold any serializable type of element**
 - Primitive types such as integers, characters, and booleans
 - Collections such as strings, lists, arrays, tuples, and dictionaries (including nested collection types)
 - Scala/Java Objects (if serializable)
 - Mixed types
- **Some RDDs are specialized and have additional functionality**
 - Pair RDDs
 - RDDs consisting of key-value pairs
 - Double RDDs
 - RDDs consisting of numeric data

RDD Data Sources

- **There are several types of data sources for RDDs**
 - Files, including text files and other formats
 - Data in memory
 - Other RDDs
 - Datasets or DataFrames

Creating RDDs from Files

- **Use SparkContext object, not SparkSession**
 - `SparkContext` is part of the core Spark library
 - `SparkSession` is part of the Spark SQL library
 - One Spark context per application
 - Use `SparkSession.sparkContext` to access the Spark context
 - Called `sc` in the Spark shell
- **Create file-based RDDs using the Spark context**
 - Use `textFile` or `wholeTextFiles` to read text files
 - Use `hadoopFile` or `newAPIHadoopFile` to read other formats
 - The Hadoop “new API” was introduced in Hadoop .20
 - Spark supports both for backward compatibility

Chapter Topics

RDD Overview

- RDD Overview
- RDD Data Sources
- **Creating and Saving RDDs**
- RDD Operations
- Essential Points
- Hands-On Exercise: Working With RDDs

Creating RDDs from Text Files (1)

- **SparkContext.textFile** reads newline-terminated text files
 - Accepts a single file, a directory of files, a wildcard list of files, or a comma-separated list of files
 - Examples
 - `textFile("myfile.txt")`
 - `textFile("mydata/")`
 - `textFile("mydata/*.log")`
 - `textFile("myfile1.txt,myfile2.txt")`

```
myRDD = spark.sparkContext.textFile("mydata/")
```

Language: Python

Creating RDDs from Text Files (2)

- **textFile** maps each line in a file to a separate RDD element
 - Only supports newline-terminated text

```
myRDD = spark.\n    sparkContext.\n    textFile("purplecow.txt")
```

Language: Python

```
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
```

myRDD

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.

Multi-line Text Elements

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

user1.json

```
{  
  "firstName": "Fred",  
  "lastName": "Flintstone",  
  "userid": "123"  
}
```

user2.json

```
{  
  "firstName": "Barney",  
  "lastName": "Rubble",  
  "userid": "234"  
}
```

Example: Using wholeTextFiles

```
userRDD = spark.sparkContext. \
    wholeTextFiles("userFiles")
```

Language: Python

userRDD

```
("user1.json", {"firstName": "Fred",
  "lastName": "Flintstone", "userid": "123"} )
```

```
("user2.json", {"firstName": "Barney",
  "lastName": "Rubble", "userid": "234"} )
```

```
("user3.json", ... )
```

```
("user4.json", ... )
```

Creating RDDs from Collections

- You can create RDDs from collections instead of files
 - `SparkContext.parallelize(collection)`
- Useful when
 - Testing
 - Generating data programmatically
 - Integrating with other systems or libraries
 - Learning

```
myData = ["Alice","Carlos","Frank","Barbara"]  
myRDD = sc.parallelize(myData)
```

Language: Python

Saving RDDs

- You can save RDDs to the same data source types supported for reading RDDs
 - Use `RDD.saveAsTextFile` to save as plain text files in the specified directory
 - Use `RDD.saveAsHadoopFile` or `saveAsNewAPIHadoopFile` with a specified Hadoop `OutputFormat` to save using other formats
- The specified save directory cannot already exist

```
myRDD.saveAsTextFile("mydata/")
```

Chapter Topics

RDD Overview

- RDD Overview
- RDD Data Sources
- Creating and Saving RDDs
- **RDD Operations**
- Essential Points
- Hands-On Exercise: Working With RDDs

RDD Operations

- **Two general types of RDD operations**
 - Actions return a value to the Spark driver or save data to a data source
 - *Transformations* define a new RDD based on the current one(s)
- **RDDs operations are performed lazily**
 - Actions trigger execution of the base RDD transformations

RDD Action Operations

■ Some common actions

- `count` returns the number of elements
- `first` returns the first element
- `take(n)` returns an array (Scala) or list (Python) of the first *n* elements
- `collect` returns an array (Scala) or list (Python) of all elements
- `saveAsTextFile(dir)` saves to text files

```
myRDD = sc. \
    textFile("purplecow.txt")

for line in myRDD.take(2):
    print(line)
I've never seen a purple cow.
I never hope to see one;
```

Language: Python

```
val myRDD = sc.
    textFile("purplecow.txt")

for (line <- myRDD.take(2))
    println(line)
I've never seen a purple cow.
I never hope to see one;
```

Language: Scala

RDD Transformation Operations (1)

- **Transformations create a new RDD from an existing one**
- **RDDs are immutable**
 - Data in an RDD is never changed
 - Transform data to create a new RDD
- **A transformation operation executes a *transformation function***
 - The function transforms elements of an RDD into new elements
 - Some transformations implement their own transformation logic
 - For many, you must provide the function to perform the transformation

RDD Transformation Operations (2)

- Transformation operations include
 - `distinct` creates a new RDD with duplicate elements in the base RDD removed
 - `union(rdd)` creates a new RDD by appending the data in one RDD to another
 - `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: distinct and union Transformations

cities1.csv

```
Boston,MA  
Palo Alto,CA  
Santa Fe,NM  
Palo Alto,CA
```

cities2.csv

```
Calgary,AB  
Chicago,IL  
Palo Alto,CA
```

```
distinctRDD = sc.\  
    textFile("cities1.csv").distinct()  
for city in distinctRDD.collect(): \  
    print(city)  
Boston,MA  
Palo Alto,CA  
Santa Fe,NM
```

```
unionRDD = sc.textFile("cities2.csv"). \  
    union(distinctRDD)  
for city in unionRDD.collect(): \  
    print(city)  
Calgary,AB  
Chicago,IL  
Palo Alto,CA  
Boston,MA  
Palo Alto,CA  
Santa Fe,NM
```

Language: Python

Chapter Topics

RDD Overview

- RDD Overview
- RDD Data Sources
- Creating and Saving RDDs
- RDD Operations
- **Essential Points**
- Hands-On Exercise: Working With RDDs

Essential Points

- **RDDs (Resilient Distributed Datasets) are a key concept in Spark**
 - Represent a distributed collection of elements
 - Elements can be of any type
- **RDDs are created from data sources**
 - Text files and other data file formats
 - Data in other RDDs
 - Data in memory
 - DataFrames and Datasets
- **RDDs contain unstructured data**
 - No associated schema like DataFrames and Datasets
- **RDD Operations**
 - Transformations create a new RDD based on an existing one
 - Actions return a value from an RDD

Chapter Topics

RDD Overview

- RDD Overview
- RDD Data Sources
- Creating and Saving RDDs
- RDD Operations
- Essential Points
- **Hands-On Exercise: Working With RDDs**

Hands-On Exercise: Working With RDDs

- In this exercise, you will load, transform, display, and save data using RDDs
- Please refer to the Hands-On Exercise Manual for instructions



Transforming Data with RDDs

Chapter 9

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- **Transforming Data with RDDs**
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Apache Spark Basics

After completing this chapter, you will be able to

- **Use functional programming with RDD transformations**
- **Describe RDD transformations are executed**
- **Create DataFrames from RDD data**

Chapter Topics

Transforming Data with RDDs

- **Writing and Passing Transformation Functions**
- Transformation Execution
- Converting Between RDDs and DataFrames
- Essential Points
- Hands-On Exercise: Transforming Data Using RDDs

Functional Programming in Spark

- Key concepts of *functional programming*
 - Functions are the fundamental unit of programming
 - Functions have input and output only
 - No state or side effects
 - Functions can be passed as arguments to other functions
 - Called *procedural parameters*
- Spark's architecture is based on functional programming
 - Passed functions can be executed by multiple executors in parallel

RDD Transformation Procedures

- **RDD transformations execute a *transformation procedure***
 - Transforms elements of an RDD into new elements
 - Runs on executors
- **A few transformation operations implement their own transformation logic**
 - Examples: `distinct` and `union`
- **Most transformation operations require you to pass a function**
 - Function implements your own transformation procedure
 - Examples: `map` and `filter`
- **This is a key difference between RDDs and DataFrame/Datasets**

Passing Functions

- Passed functions can be named or anonymous
- Anonymous functions are defined inline without an identifier
 - Best for short, one-off functions
 - Supported in many programming languages
 - Python: `lambda x: ...`
 - Scala: `x => ...`
 - Java 8: `x -> ...`

Example: Passing Named Functions (Python)

```
def toUpper(s):
    return s.upper()

myRDD = sc. \
    textFile("purplecow.txt")

myUpperRDD = myRDD.map(toUpper)

for line in myUpperRDD.take(2):
    print(line)
I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
```

myRDD

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.



myUpperRDD

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.

Example: Passing Named Functions (Scala)

```
def toUpper(s: String):  
    String = { s.toUpperCase }  
  
val myRDD =  
    sc.textFile("purplecow.txt")  
  
val myUpperRDD =  
    myRDD.map(toUpper)  
  
myUpperRDD.take(2).  
    foreach(println)  
I'VE NEVER SEEN A PURPLE COW.  
I NEVER HOPE TO SEE ONE;
```

myRDD

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.



myUpperRDD

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.

Example: Passing Anonymous Functions

- Python: Use the `lambda` keyword to specify the name of the input parameter(s) and the function that returns the output

```
myUpperRDD = myRDD.map(lambda line: line.upper())
```

Language: Python

- Scala: Use the `=>` operator to specify the name of the input parameter(s) and the function that returns the output

```
val myUpperRDD = myRDD.map(line => line.toUpperCase)
```

Language: Scala

- Scala shortcut: Use underscore (`_`) to stand for anonymous input parameters

```
val myUpperRDD = myRDD.map(_.toUpperCase)
```

Language: Scala

Example: map and filter Transformations

```
myFilteredRDD = myRDD. \
    map(lambda line: line.upper()). \
    filter(lambda line: \
        line.startswith('I'))
```

Language: Python

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.



```
val myFilteredRDD = myRDD. \
    map(line => line.toUpperCase()). \
    filter(line =>
        line.startsWith("I"))
```

Language: Scala

↓
myFilteredRDD

I'VE NEVER SEEN A PURPLE COW.
I NEVER HOPE TO SEE ONE;
I'D RATHER SEE THAN BE ONE.

Chapter Topics

Transforming Data with RDDs

- Writing and Passing Transformation Functions
- **Transformation Execution**
- Converting Between RDDs and DataFrames
- Essential Points
- Hands-On Exercise: Transforming Data Using RDDs

RDD Execution

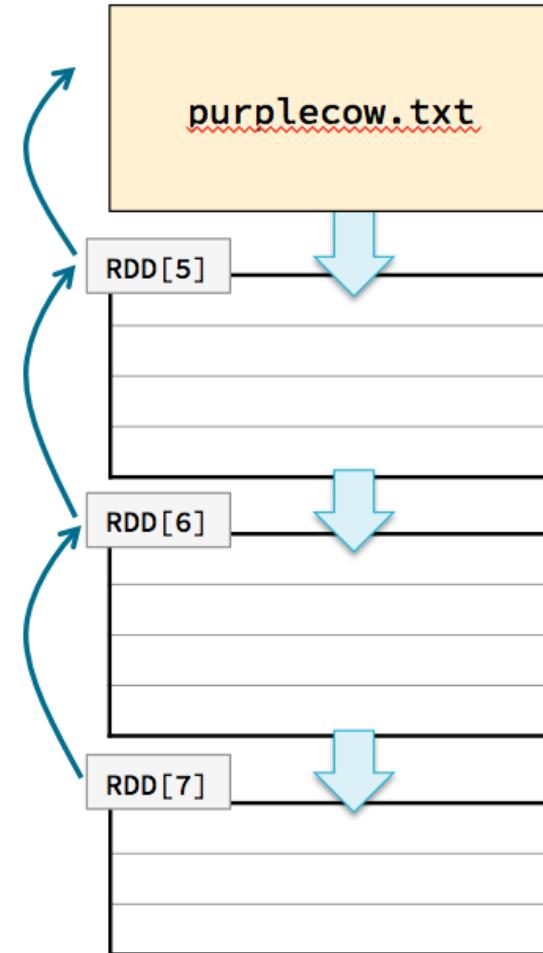
- An RDD query consists of a sequence of one or more transformations completed by an action
- RDD queries are executed *lazily*
 - When the action is called
- RDD queries are executed differently than DataFrame and Dataset queries
 - DataFrames and Datasets scan their sources to determine the schema *eagerly* (when created)
 - RDDs do not have schemas and do not scan their sources before loading

RDD Lineage

- **Transformations create a new RDD based on one or more existing RDDs**
 - Result RDDs are considered *children* of the base (*parent*) RDD
 - Child RDDs depend on their parent RDD
- **An RDD's *lineage* is the sequence of ancestor RDDs that it depends on**
 - When an RDD executes, it executes its lineage starting from the source
- **Spark maintains each RDD's lineage**
 - Use `toDebugString` to view the lineage

RDD Lineage and `toDebugString` (Scala)

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



RDD Lineage and `toDebugString` (Python)

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

```
myFilteredRDD.toDebugString()
(2) PythonRDD[7] at RDD at PythonRDD.scala:48 []
| purplecow.txt MapPartitionsRDD[6] ... []
| purplecow.txt HadoopRDD[5] at textFile ... []
```

- Use `print` for prettier output

```
print myFilteredRDD.toDebugString()
(2) PythonRDD[7] at RDD at PythonRDD.scala:48 []
| purplecow.txt MapPartitionsRDD[6] at textFile ...
| purplecow.txt HadoopRDD[5] at textFile ...
```

Pipelining (1)

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

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

Language: Scala

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.

Pipelining (2)

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

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

Language: Scala

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.

Pipelining (3)

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

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

Language: Scala

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.

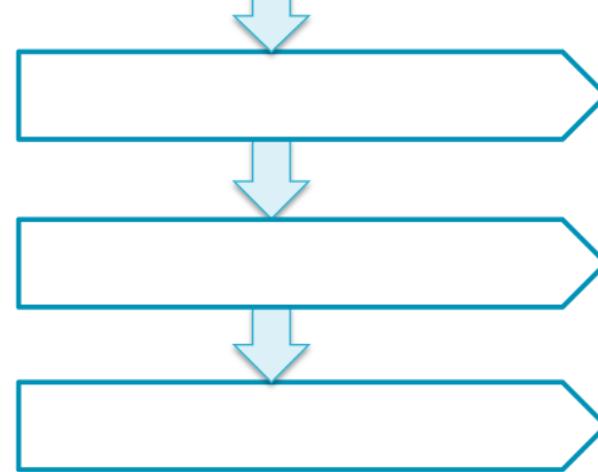
Pipelining (4)

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

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

Language: Scala

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.



Pipelining (5)

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

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

Language: Scala

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 never hope to see one;



Pipelining (6)

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

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

Language: Scala

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 NEVER HOPE TO SEE ONE;

Pipelining (7)

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

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

Language: Scala

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 NEVER HOPE TO SEE ONE;

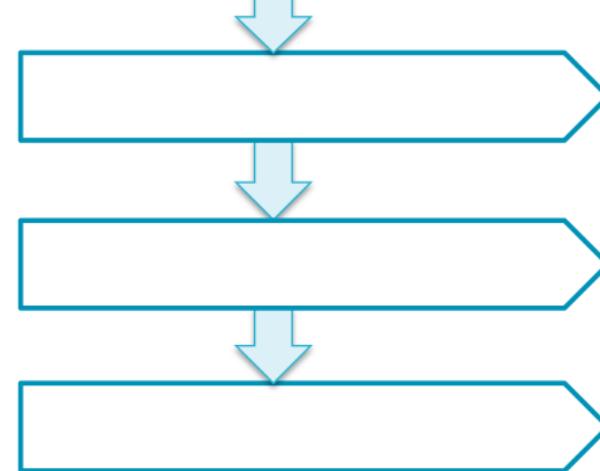
Pipelining (8)

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

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

Language: Scala

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

Transforming Data with RDDs

- Writing and Passing Transformation Functions
- Transformation Execution
- **Converting Between RDDs and DataFrames**
- Essential Points
- Hands-On Exercise: Transforming Data Using RDDs

Converting RDDs to DataFrames

- You can create a DataFrame from an RDD
 - Useful with unstructured or semi-structured data such as text
 - Define a schema
 - Transform the base RDD to an RDD of Row objects (Scala) or lists (Python)
 - Use `SparkSession.createDataFrame`
- You can also return the underlying RDD of a DataFrame
 - Use the `DataFrame.rdd` attribute to return an RDD of Row objects

Example: Create a DataFrame from an RDD

- Example data: semi-structured text data source

```
02134,Hopper,Grace,52
94020,Turing,Alan,32
94020,Lovelace,Ada,28
87501,Babbage,Charles,49
02134,Wirth,Niklaus,48
```

Scala Example: Create a DataFrame from an RDD (1)

```
import org.apache.spark.sql.types._  
import org.apache.spark.sql.Row  
  
val mySchema = StructType(Array(  
    StructField("pcode", StringType),  
    StructField("lastName", StringType),  
    StructField("firstName", StringType),  
    StructField("age", IntegerType)  
))
```

Language: Scala
Continued on next slide...

Scala Example: Create a DataFrame from an RDD (2)

```
val rowRDD = sc.textFile("people.txt").  
  map(line => line.split(",")).  
  map(values =>  
    Row(values(0),values(1),values(2),values(3).toInt))  
  
val myDF = spark.createDataFrame(rowRDD,mySchema)  
  
myDF.show(2)  
+-----+-----+-----+  
|pcode|lastName|firstName|age|  
+-----+-----+-----+  
|02134| Hopper| Grace| 52|  
|94020| Turing| Alan| 32|  
+-----+-----+-----+
```

Language: Scala

Python Example: Create a DataFrame from an RDD (1)

```
from pyspark.sql.types import *
mySchema = \
StructType([
    StructField("pcode", StringType()),
    StructField("lastName", StringType()),
    StructField("firstName", StringType()),
    StructField("age", IntegerType())])
```

Language: Python
Continued on next slide...

Python Example: Create a DataFrame from an RDD (2)

```
myRDD = sc.textFile("people.txt"). \
    map(lambda line: line.split(",")). \
    map(lambda values:
        [values[0],values[1],values[2],int(values[3])])

myDF = spark.createDataFrame(myRDD,mySchema)

myDF.show(2)
+-----+-----+-----+
|pcode|lastName|firstName|age|
+-----+-----+-----+
|02134| Hopper| Grace| 52|
|94020| Turing| Alan| 32|
+-----+-----+-----+
```

Language: Python

Example: Return a DataFrame's Underlying RDD

```
myRDD2 = myDF.rdd

for row in myRDD2.take(2): print(row)
Row(pcode=u'02134', lastName=u'Hopper', firstName=u'Grace',
    age=52)
Row(pcode=u'94020', lastName=u'Turing', firstName=u'Alan',
    age=32)
```

Language: Python

```
val myRDD2 = myDF.rdd

myRDD2.take(2).foreach(println)
[02134,Hopper,Grace,52]
[94020,Turing,Alan,32]
```

Language: Scala

Chapter Topics

Transforming Data with RDDs

- Writing and Passing Transformation Functions
- Transformation Execution
- Converting Between RDDs and DataFrames
- **Essential Points**
- Hands-On Exercise: Transforming Data Using RDDs

Essential Points

- **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
- **RDD Transformations use functional programming**
 - Take named or anonymous functions as parameters
- **RDD query execution is *lazy*—transformation are not executed until triggered by an action**
- **Operations on the same RDD element are pipelined together if possible**

Chapter Topics

Transforming Data with RDDs

- Writing and Passing Transformation Functions
- Transformation Execution
- Converting Between RDDs and DataFrames
- Essential Points
- **Hands-On Exercise: Transforming Data Using RDDs**

Hands-On Exercise: Transforming Data Using RDDs

- In this exercise, you will transform, display, and save data using RDDs
- Please refer to the Hands-On Exercise Manual for instructions



Aggregating Data with Pair RDDs

Chapter 10

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- **Aggregating Data with Pair RDDs**
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Aggregating Data with Pair RDDs

After completing this chapter, you will be able to

- Create pair RDDs of key-value pairs from generic RDDs
- Summarize the special operations available on pair RDDs
- Describe how map-reduce algorithms are implemented in Spark

Chapter Topics

Aggregating Data with Pair RDDs

- **Key-Value Pair RDDs**
- Map-Reduce
- Other Pair RDD Operations
- Essential Points
- Hands-On Exercise: Joining Data Using Pair RDDs

Pair RDDs

Pair RDD

- 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
 - Such as sorting, joining, grouping, and counting

(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 (Python)

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

```
usersRDD = sc.textFile("userlist.tsv"). \
    map(lambda line: line.split('\t')). \
    map(lambda fields: (fields[0], fields[1]))
```

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



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

Example: A Simple Pair RDD (Scala)

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

```
val usersRDD = sc.textFile("userlist.tsv").  
  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("weblogs/"). \  
  keyBy(lambda line: line.split(' ')[2])
```

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...")

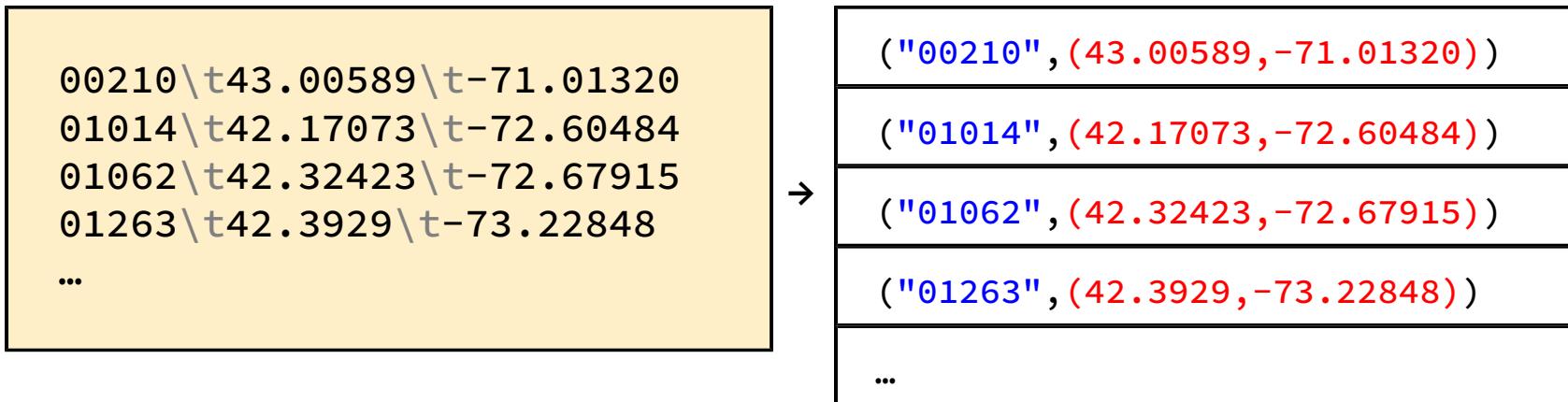
(25254,203.146.17.59 - 25254 "GET /KBDOC-00230.html...")

...

Question 1: Pairs with Complex Values

- How would you do this?

- Input: a tab-delimited 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("latlon.tsv"). \  
  map(lambda line: line.split('\t')). \  
  map(lambda fields:  
    (fields[0],(float(fields[1]),float(fields[2]))))
```

Language: Python

00210\t43.00589\t-71.01320
01014\t42.17073\t-72.60484
01062\t42.32423\t-72.67915
01263\t42.3929\t-73.22848
...



("00210", (43.00589, -71.01320))
("01014", (42.17073, -72.60484))
("01062", (42.32423, -72.67915))
("01263", (42.3929, -73.22848))
...

Question 2: Mapping Single Elements to Multiple Pairs

- How would you do this?

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

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

The diagram illustrates a mapping process. On the left, a yellow box contains four input rows. An arrow points from this box to a vertical stack of six white boxes, each representing an output pair. The first three pairs correspond to the first three input rows, while the last three pairs are represented by ellipses.

"00001", "sku010")
"00001", "sku933")
"00001", "sku022")
"00002", "sku912")
"00002", "sku331")
"00003", "sku888") ...

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

```
val ordersRDD = sc.textFile("orderskus.txt")
```

Language: Scala

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

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

```
val ordersRDD = sc.textFile("orderskus.txt").  
map(line => line.split(' '))
```

Language: Scala

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

split returns two-element arrays

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

```
val ordersRDD = sc.textFile("orderskus.txt").  
  map(line => line.split(' ')).  
  map(fields => (fields(0),fields(1)))
```

Language: Scala

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

Map array elements to tuples to produce a pair RDD

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

```
val ordersRDD = sc.textFile("orderskus.txt").  
  map(line => line.split(' ')).  
  map(fields => (fields(0), fields(1))).  
  flatMapValues(skus => skus.split(':'))
```

Language: Scala

("00001" , "sku010")
("00001" , "sku933")
("00001" , "sku022")
("00002" , "sku912")
("00002" , "sku331")
("00003" , "sku888")
...

flatMapValues splits a single value (a colon-separated string of SKUs) into multiple elements

Chapter Topics

Aggregating Data with Pair RDDs

- Key-Value Pair RDDs
- **Map-Reduce**
- Other Pair RDD Operations
- Essential Points
- Hands-On Exercise: Joining Data Using Pair RDDs

Map-Reduce

- **Map-reduce is a common programming model**
 - Easily applicable to distributed processing of large data sets
- **Hadoop MapReduce was the first major distributed 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 zero or more new records
 - Examples: `map`, `flatMap`, `filter`, `keyBy`
- **Reduce phase**
 - Works on map output
 - Consolidates multiple records
 - Examples: `reduceByKey`, `sortByKey`, `mean`

Map-Reduce Example: Word Count

Input Data

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

Result

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

?
→

Example: Word Count (1)

```
countsRDD = sc.textFile("catsat.txt"). \  
    flatMap(lambda line: line.split(' '))
```

Language: Python

the
cat
sat
on
the
mat
the
aardvark
...

Example: Word Count (2)

```
countsRDD = sc.textFile("catsat.txt"). \  
    flatMap(lambda line: line.split(' ')). \  
    map(lambda word: (word,1))
```

Language: Python

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 (3)

```
countsRDD = sc.textFile("catsat.txt"). \
    flatMap(lambda line: line.split(' ')). \
    map(lambda word: (word,1)). \
    reduceByKey(lambda v1,v2: v1+v2)
```

Language: Python

the
cat
sat
on
the
mat
the
aardvark
...

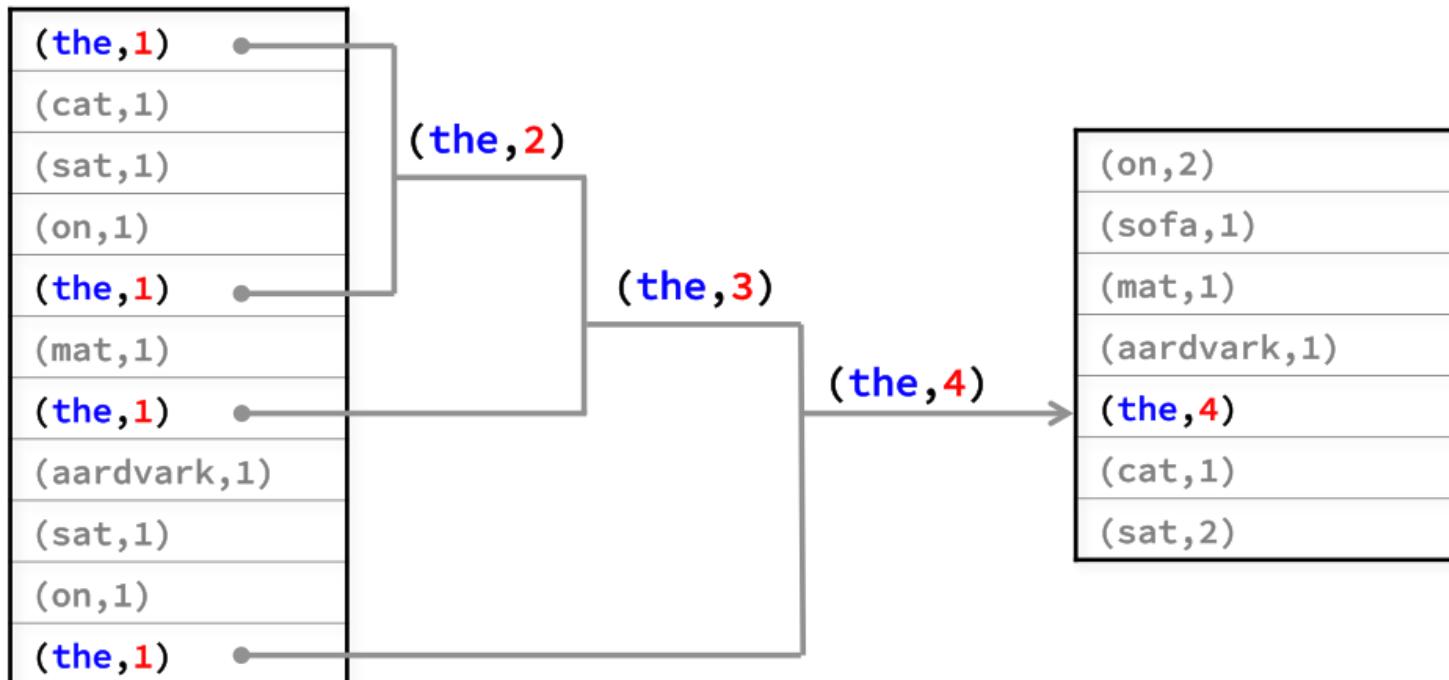
(the, 1)
(cat, 1)
(sat, 1)
(on, 1)
(the, 1)
(mat, 1)
(the, 1)
(aardvark, 1)
...

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

reduceByKey (1)

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

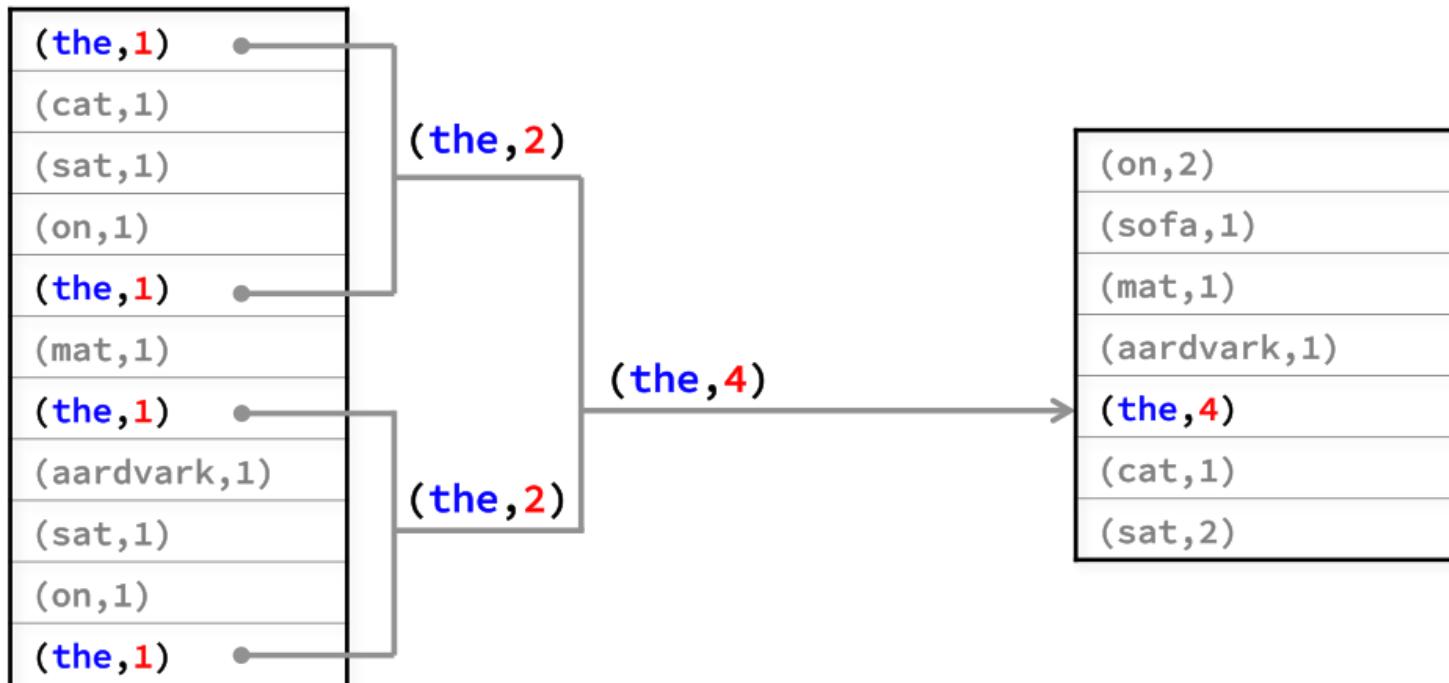
```
reduceByKey(lambda v1,v2: v1+v2)
```



reduceByKey (2)

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

```
reduceByKey(lambda v1,v2: v1+v2)
```



Word Count Recap (The Scala Version)

```
val counts = sc.textFile("catsat.txt").  
  flatMap(line => line.split(' ')).  
  map(word => (word,1)).  
  reduceByKey((v1,v2) => v1+v2)
```

OR

```
val counts = sc.textFile("catsat.txt").  
  flatMap(_.split(' ')).  
  map((_,1)).  
  reduceByKey(_+_)
```

Why Do We Care about Counting Words?

- **Word count is challenging with massive amounts of data**
 - Using a single compute node would be too time-consuming
- **Statistics are often simple aggregate functions**
 - Distributive in nature
 - For example: max, min, sum, and count
- **Map-reduce breaks complex tasks down into smaller elements which can be executed in parallel**
 - RDD transformations are implemented using the map-reduce paradigm
- **Many common tasks are very similar to word count**
 - Such as log file analysis

Chapter Topics

Aggregating Data with Pair RDDs

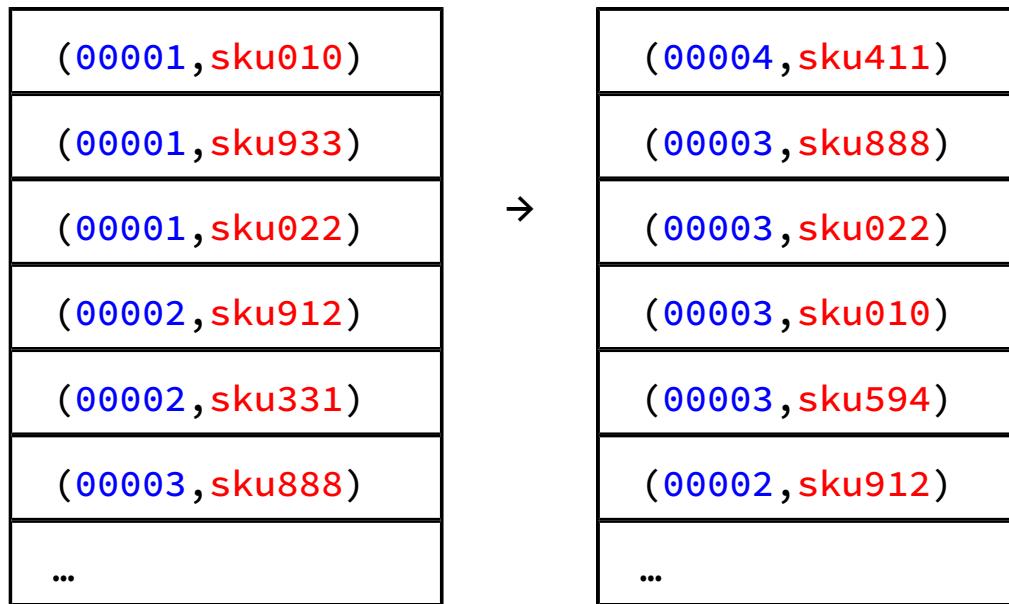
- Key-Value Pair RDDs
- Map-Reduce
- **Other Pair RDD Operations**
- Essential Points
- Hands-On Exercise: Joining Data Using Pair RDDs

Pair RDD Operations

- In addition to `map` and `reduceByKey` operations, Spark has several operations specific to pair RDDs
- Examples
 - `countByKey` returns a map with the count of occurrences of each key
 - `groupByKey` groups all the values for each key in an RDD
 - `sortByKey` sorts in ascending or descending order
 - `join` returns an RDD containing all pairs with matching keys from two RDDs
 - `leftOuterJoin`, `rightOuterJoin`, `fullOuterJoin` join two RDDs, including keys defined in the left, right, or both RDDs respectively
 - `mapValues`, `flatMapValues` execute a function on just the values, keeping the key the same
 - `lookup(key)` returns the value(s) for a key as a list

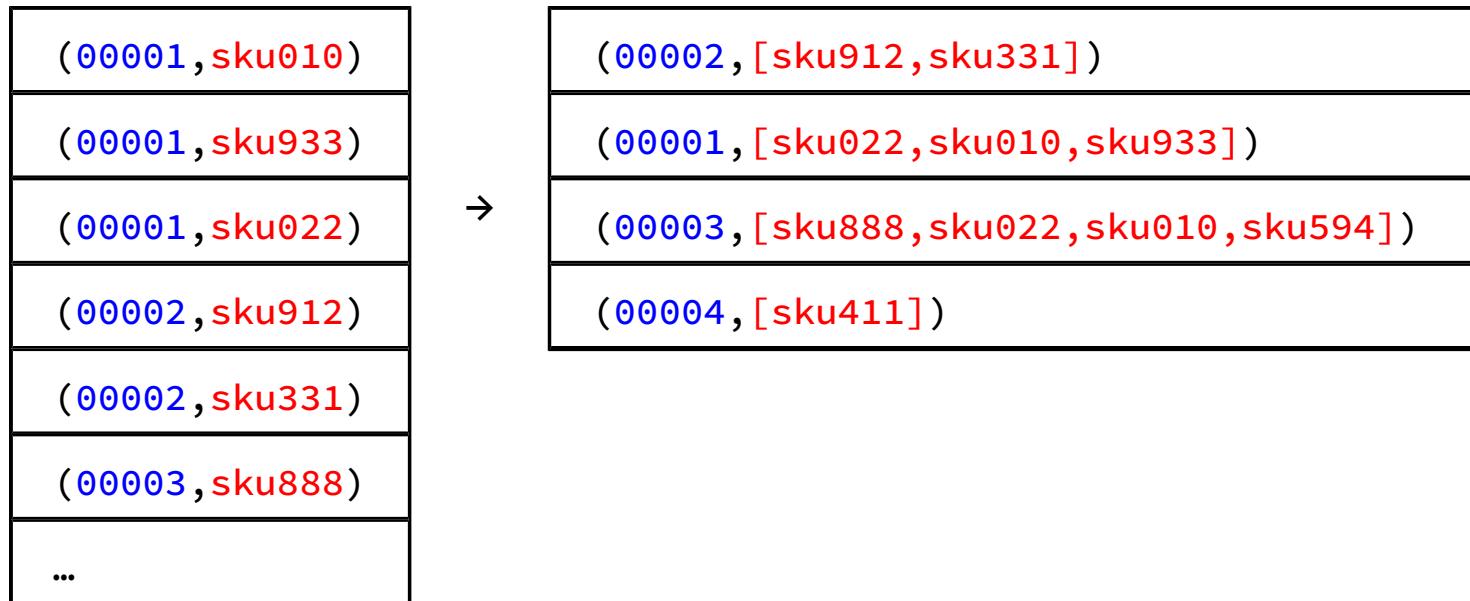
Example: sortByKey Transformation

```
ordersRDD.sortByKey(ascending=false)
```



Example: groupByKey Transformation

```
ordersRDD.groupByKey()
```



*Use `reduceByKey` in place of `groupByKey` where possible, as it can provide better performance.

Example: Joining by Key

movieGrossRDD

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

movieYearRDD

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



joinedRDD

```
joinedRDD = movieGrossRDD.  
join(movieYearRDD)
```



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

Chapter Topics

Aggregating Data with Pair RDDs

- Key-Value Pair RDDs
- Map-Reduce
- Other Pair RDD Operations
- **Essential Points**
- Hands-On Exercise: Joining Data Using Pair RDDs

Essential Points

- **Pair RDDs are a special form of RDD consisting of key-value pairs (tuples)**
- **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 several operations for working with pair RDDs**

Chapter Topics

Aggregating Data with Pair RDDs

- Key-Value Pair RDDs
- Map-Reduce
- Other Pair RDD Operations
- Essential Points
- **Hands-On Exercise: Joining Data Using Pair RDDs**

Hands-On Exercise: Joining Data Using Pair RDDs

- In this exercise, you will join account data with web log data
- Please refer to the Hands-On Exercise Manual for instructions



Querying Tables and Views with SQL

Chapter 11

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- **Querying Tables and Views with SQL**
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Querying Tables and Views with SQL

After completing this chapter, you will be able to

- Create DataFrames based on SQL queries
- Query and manage tables and views using Spark SQL and the Catalog API
- Summarize how Spark SQL compares to other SQL-on-Hadoop tools

Chapter Topics

Querying Tables and Views with SQL

- **Querying Tables in Spark Using SQL**
- Querying Files and Views
- The Catalog API
- Essential Points
- Hands-On Exercise: Querying Tables and Views with SQL

Spark SQL Queries

- You can query data in Spark SQL using SQL commands
 - Similar to queries in a relational database
 - Spark SQL includes a native SQL 2003 parser
- You can query Hive tables or DataFrame/Dataset views
- Spark SQL queries are particularly useful for
 - Developers or analysts who are comfortable with SQL
 - Doing ad hoc analysis
- Use the `SparkSession.sql` function to execute a SQL query on a table
 - Returns a DataFrame

Example: Spark SQL Query (1)

- For Spark installations integrated with Hive, Spark can query tables defined in the Hive metastore

Hive Table: people			
age	first_name	last_name	pcode
52	Grace	Hopper	02134
null	Alan	Turing	94020
28	Ada	Lovelace	94020
...

Example: Spark SQL Query (2)

```
myDF = spark.sql("SELECT * FROM people WHERE pcode = 94020")

myDF.printSchema()
root
 |-- age: long (nullable = true)
 |-- first_name: string (nullable = true)
 |-- last_name: string (nullable = true)
 |-- pcode: string (nullable = true)

myDF.show()
+-----+-----+-----+
| age|first_name|last_name|pcode|
+-----+-----+-----+
|null|      Alan|    Turing|94020|
|  28|        Ada| Lovelace|94020|
+-----+-----+-----+
```

Language: Python

Example: A More Complex Query

```
maAgeDF = spark.  
    sql("SELECT MEAN(age) AS mean_age,STDDEV(age)  
        AS sdev_age FROM people WHERE pcode IN  
        (SELECT pcode FROM pcodes WHERE state='MA'))  
  
maAgeDF.printSchema()  
root  
|-- mean_age: double (nullable = true)  
|-- sdev_age: double (nullable = true)  
  
maAgeDF.show()  
+-----+-----+  
| mean_age | sdev_age |  
+-----+-----+  
| 50.0 | 2.8284271247461903 |  
+-----+-----+
```

Language: Python

SQL Queries and DataFrame Queries

- SQL queries and DataFrame transformations provide equivalent functionality
- Both are executed as series of transformations
 - Optimized by the Catalyst optimizer
- The following Python examples are equivalent

```
myDF = spark.sql("SELECT * FROM people WHERE pcode = 94020")
```

```
myDF = spark.read.table("people").where("pcode=94020")
```

Chapter Topics

Querying Tables and Views with SQL

- Querying Tables in Spark Using SQL
- **Querying Files and Views**
- The Catalog API
- Essential Points
- Hands-On Exercise: Querying Tables and Views with SQL

SQL Queries on Files

- You can query directly from Parquet or JSON files that are not Hive tables

```
spark. \
    sql("SELECT * FROM parquet.`/loudacre/people.parquet` \
        WHERE firstName LIKE 'A%'"). \
    show()
+-----+-----+-----+
|pcode|lastName|firstName|age|
+-----+-----+-----+
|94020| Turing|      Alan| 32|
|94020| Lovelace|       Ada| 28|
+-----+-----+-----+
```

SQL Queries on Views

- You can also query a view
 - Views provide the ability to perform SQL queries on a DataFrame or Dataset
- Views are temporary
 - Regular views can only be used within a single Spark session
 - Global views can be shared between multiple Spark sessions within a single Spark application
- Creating a view
 - `DataFrame.createTempView(view-name)`
 - `DataFrame.createOrReplaceTempView(view-name)`
 - `DataFrame.createGlobalTempView(view-name)`
 - `DataFrame.createOrReplaceGlobalTempView(view-name)`

Example: Creating and Querying a View

- After defining a DataFrame view, you can query with SQL just as with a table

```
spark.read.load("/loudacre/people.parquet"). \
    select("firstName", "lastName"). \
    createTempView("user_names")

spark.sql( \
    "SELECT * FROM user_names WHERE firstName LIKE 'A%'"). \
    show()
+-----+-----+
|firstName|lastName|
+-----+-----+
|      Alan|   Turing|
|      Ada|Lovelace|
+-----+-----+
```

Chapter Topics

Querying Tables and Views with SQL

- Querying Tables in Spark Using SQL
- Querying Files and Views
- **The Catalog API**
- Essential Points
- Hands-On Exercise: Querying Tables and Views with SQL

The Catalog API (1)

- Use the Catalog API to explore tables and manage views
- The entry point for the Catalog API is `spark.catalog`
- Functions include
 - `listDatabases` returns a Dataset (Scala) or list (Python) of existing databases
 - `setCurrentDatabase(dbname)` sets the current database for the session
 - Default database is `default`
 - Subsequent commands refer to current database unless otherwise specified
 - Equivalent to the USE statement in SQL

The Catalog API (2)

- Functions include (continued)

- `listTables(database)` returns a Dataset (Scala) or list (Python) of tables and views
 - Specified database (or current database if not specified)
 - `listColumns(tablename)` returns a Dataset (Scala) or list (Python) of the columns in the specified table or view in the current database
 - Use `listColumns(tablename, database)` in Python or `listColumns(database, tablename)` in Scala to specify a database
 - `dropTempView(viewname)` removes a temporary view

Example: Listing Tables and Views (Scala)

```
spark.catalog.listTables.show
+-----+-----+-----+-----+
|     name|database|description|tableType|isTemporary|
+-----+-----+-----+-----+
|   people| default|      null| EXTERNAL|    false|
| user_names| default|      null|TEMPORARY|     true|
+-----+-----+-----+-----+
```

Example: Listing Tables and Views (Python)

```
for table in spark.catalog.listTables(): print(table)
Table(name=u'people', database=u'default',
      description=None, tableType=u'EXTERNAL',
      isTemporary=False)
Table(name=u'user_names', database=u'default',
      description=None,
      tableType=u'TEMPORARY',
      isTemporary=True)
```

Chapter Topics

Querying Tables and Views with SQL

- Querying Tables in Spark Using SQL
- Querying Files and Views
- The Catalog API
- **Essential Points**
- Hands-On Exercise: Querying Tables and Views with SQL

Essential Points

- You can query using SQL in addition to DataFrame and Dataset operations
 - Incorporates SQL queries into procedural development
- You can use SQL with Hive tables and temporary views
 - Temporary views let you use SQL on data in DataFrames and Datasets
- SQL queries and DataFrame/Dataset queries are equivalent
 - Both are optimized by Catalyst
- The Catalog API lets you list and describe tables, views, and columns, choose a database, or delete a view

Chapter Topics

Querying Tables and Views with SQL

- Querying Tables in Spark Using SQL
- Querying Files and Views
- The Catalog API
- Essential Points
- **Hands-On Exercise: Querying Tables and Views with SQL**

Hands-On Exercise: Querying Tables and Views with SQL

- In this exercise, you will query tables and views with SQL
- Please refer to the Hands-On Exercise Manual for instructions



Working with Datasets in Scala

Chapter 12

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- **Working with Datasets in Scala**
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Working with Datasets in Scala

After completing this chapter, you will be able to

- Explain what Datasets are and how they differ from DataFrames
- Create Datasets in Scala from data sources and in-memory data
- Query Datasets using typed and untyped transformations

Chapter Topics

Working with Datasets in Scala

- **Datasets and DataFrames**
- Creating Datasets
- Loading and Saving Datasets
- Dataset Operations
- Essential Points
- Hands-On Exercise: Using Datasets in Scala

What Is a Dataset?

- A distributed collection of *strongly-typed* objects
 - Primitive types such as `Int` or `String`
 - Complex types such as arrays and lists containing supported types
 - Product objects based on Scala case classes (or JavaBean objects in Java)
 - Row objects
- Mapped to a relational schema
 - The schema is defined by an encoder
 - The schema maps object properties to typed columns
- Implemented only in Scala and Java
 - Python is not a statically-typed language—no benefit from Dataset strong typing

Datasets and DataFrames

- In Scala, DataFrame is an alias for a Dataset containing Row objects
 - There is no distinct class for DataFrame
- DataFrames and Datasets represent different types of data
 - DataFrames (Datasets of Row objects) represent tabular data
 - Datasets represent typed, object-oriented data
- DataFrame transformations are referred to as *untyped*
 - Rows can hold elements of any type
 - Schemas defining column types are not applied until runtime
- Dataset transformations are *typed*
 - Object properties are inherently typed at compile time

Chapter Topics

Working with Datasets in Scala

- Datasets and DataFrames
- **Creating Datasets**
- Loading and Saving Datasets
- Dataset Operations
- Essential Points
- Hands-On Exercise: Using Datasets in Scala

Creating Datasets: A Simple Example

- Use `SparkSession.createDataset(Seq)` to create a Dataset from in-memory data (experimental)
- Example: Create a Dataset of strings (`Dataset[String]`)

```
val strings = Seq("a string","another string")
val stringDS = spark.createDataset(strings)
stringDS.show
+-----+
|      value|
+-----+
|    a string|
|another string|
+-----+
```

Datasets and Case Classes (1)

- Scala case classes are a useful way to represent data in a Dataset
 - They are often used to create simple data-holding objects in Scala
 - Instances of case classes are called *products*

```
case class Name(firstName: String, lastName: String)

val names = Seq(Name("Fred", "Flintstone"),
                Name("Barney", "Rubble"))
names.foreach(name => println(name.firstName))
Fred
Barney
```

Continues on next slide...

Datasets and Case Classes (2)

- Encoders define a Dataset's schema using reflection on the object type
 - Case class arguments are treated as columns

```
import spark.implicits._ // required if not running in shell

val namesDS = spark.createDataset(names)
namesDS.show
+-----+
|firstName| lastName|
+-----+
|      Fred| Flintstone|
|    Barney|      Rubble|
+-----+
```

Type Safety in Datasets and DataFrames

- Type safety means that type errors are found at compile time rather than runtime
- Example: Assigning a String value to an Int variable

```
val i:Int = namesDS.first.lastName // Name(Fred,Flintstone)
Compilation: error: type mismatch;
              found: String / required: Int
```

```
val row = namesDF.first // Row(Fred,Flintstone)
val i:Int = row.getInt(row.fieldIndex("lastName"))
Run time: java.lang.ClassCastException: java.lang.String
          cannot be cast to java.lang.Integer
```

Chapter Topics

Working with Datasets in Scala

- Datasets and DataFrames
- Creating Datasets
- **Loading and Saving Datasets**
- Dataset Operations
- Essential Points
- Hands-On Exercise: Using Datasets in Scala

Loading and Saving Datasets

- You cannot load a Dataset directly from a structured source
 - Create a Dataset by loading a DataFrame or RDD and converting to a Dataset
- Datasets are saved as DataFrames
 - Save using Dataset.write (returns a DataFrameWriter)
 - The type of object in the Dataset is not saved

Example: Creating a Dataset from a DataFrame (1)

- Use `Dataset.as [type]` to create a Dataset from a DataFrame
 - Encoders convert Row elements to the Dataset's type
 - The `Dataset.as` function is experimental
- Example: a Dataset of type Name based a JSON file

Data File: `names.json`

```
{"firstName":"Grace","lastName":"Hopper"}  
 {"firstName":"Alan","lastName":"Turing"}  
 {"firstName":"Ada","lastName":"Lovelace"}  
 {"firstName":"Charles","lastName":"Babbage"}
```

Example: Creating a Dataset from a DataFrame (2)

```
val namesDF = spark.read.json("names.json")
namesDF: org.apache.spark.sql.DataFrame =
  [firstName: string, lastName: string]

namesDF.show
+-----+-----+
|firstName|lastName|
+-----+-----+
|  Grace|  Hopper|
|   Alan|  Turing|
|   Ada|Lovelace|
|Charles| Babbage|
+-----+-----+
```

Example: Creating a Dataset from a DataFrame (3)

```
case class Name(firstName: String, lastName: String)

val namesDS = namesDF.as[Name]
namesDS: org.apache.spark.sql.Dataset[Name] =
  [firstName: string, lastName: string]

namesDS.show
+-----+-----+
|firstName|lastName|
+-----+-----+
|  Grace |  Hopper |
|   Alan |  Turing |
|   Ada | Lovelace |
| Charles | Babbage |
+-----+-----+
```

Example: Creating Datasets from RDDs (1)

- Datasets can be created based on RDDs
 - Useful with unstructured or semi-structured data such as text
- Example:
 - Tab-separated text file with postal code, latitude, and longitude

```
00210\t43.005895\t71.013202  
01014\t42.170731\t-72.604842  
01062\t42.324232\t-72.67915  
...
```

- Output: Dataset with `PcodeLatLon(pcode, (lat, lon))` objects

```
case class PcodeLatLon(pcode: String,  
                      latlon: Tuple2[Double, Double])
```

Continues on next slide...

Example: Creating Datasets from RDDs (2)

- Parse input to structure the data
- Create RDD of PcodeLatLon case class instances

```
val pLatLonRDD = sc.textFile("latlon.tsv").  
  map(line => line.split('\t')).  
  map(fields =>  
    (PcodeLatLon(fields(0),  
      (fields(1).toFloat, fields(2).toFloat))))
```

Continues on next slide...

PcodeLatLon("00210", (43.005895, -71.013202))

PcodeLatLon("01014", (42.170731, -72.604842))

PcodeLatLon("01062", (42.324232, -72.67915))

...

Example: Creating Datasets from RDDs (3)

- Convert RDD to a Dataset of PcodeLatLon objects

```
val pLatLonDS = spark.createDataset(pLatLonRDD)
pLatLonDS: org.apache.spark.sql.Dataset[PcodeLatLon] = [PCODE: string, LATLON: struct<_1: double, _2: double>]

pLatLonDS.printSchema
root
|-- PCODE: string (nullable = true)
|-- LATLON: struct (nullable = true)
|   |-- _1: double (nullable = true)
|   |-- _2: double (nullable = true)

println(pLatLonDS.first)
PcodeLatLon(00210,(43.00589370727539,-71.01319885253906))
```

Chapter Topics

Working with Datasets in Scala

- Datasets and DataFrames
- Creating Datasets
- Loading and Saving Datasets
- **Dataset Operations**
- Essential Points
- Hands-On Exercise: Using Datasets in Scala

Typed and Untyped Transformations (1)

- **Typed transformations create a new Dataset based on an existing Dataset**
 - Typed transformations can be used on Datasets of any type (including Row)
- **Untyped transformations return DataFrames (Datasets containing Row objects) or untyped Columns**
 - Do not preserve type of the data in the parent Dataset

Typed and Untyped Transformations (2)

- Untyped operations (those that return Row Datasets) include
 - join
 - groupBy (with aggregation function)
 - drop
 - select
 - withColumn
- Typed operations (operations that return typed Datasets) include
 - filter (and its alias, where)
 - distinct
 - limit
 - sort (and its alias, orderBy)
 - union

Example: Typed and Untyped Transformations (1)

```
case class Person(pcode:String, lastName:String,  
                  firstName:String, age:Int)  
  
val people = Seq(Person("02134","Hopper","Grace",48),...)  
  
val peopleDS = spark.createDataset(people)  
peopleDS: org.apache.spark.sql.Dataset[Person] =  
  [pcode: string, firstName: string ... 2 more fields]
```

Continues on next slide...

Example: Typed and Untyped Transformations (2)

- Typed operations return Datasets based on the starting Dataset
- Untyped operations return DataFrames (Datasets of Rows)

```
val sortedDS = peopleDS.sort("age")
sortedDS: org.apache.spark.sql.Dataset[Person] =
  [pcode: string, lastName: string ... 2 more fields]

val firstLastDF = peopleDS.select("firstName","lastName")
firstLastDF: org.apache.spark.sql.DataFrame =
  [firstName: string, lastName: string]
```

Example: Combining Typed and Untyped Operations

```
val combineDF = peopleDS.sort("lastName").  
  where("age > 40").select("firstName","lastName")  
combineDF: org.apache.spark.sql.DataFrame =  
  [firstName: string, lastName: string]  
  
combineDF.show  
+-----+-----+  
|firstName|lastName|  
+-----+-----+  
|  Charles|  Babbage|  
|   Grace|   Hopper|  
+-----+-----+
```

Chapter Topics

Working with Datasets in Scala

- Datasets and DataFrames
- Creating Datasets
- Loading and Saving Datasets
- Dataset Operations
- **Essential Points**
- Hands-On Exercise: Using Datasets in Scala

Essential Points

- **Datasets represent data consisting of strongly-typed objects**
 - Primitive types, complex types, and `Product` and `Row` objects
 - Encoders map the Dataset's data type to a table-like schema
- **Datasets are defined in Scala and Java**
 - Python is a dynamically-typed language, no need for strongly-typed data representation
- **In Scala and Java, DataFrame is just an alias for Dataset[Row]**
- **Datasets can be created from in-memory data, DataFrames, and RDDs**
- **Datasets have typed and untyped operations**
 - Typed operations return Datasets based on the original type
 - Untyped operations return DataFrames (Datasets of rows)

Chapter Topics

Working with Datasets in Scala

- Datasets and DataFrames
- Creating Datasets
- Loading and Saving Datasets
- Dataset Operations
- Essential Points
- **Hands-On Exercise: Using Datasets in Scala**

Hands-On Exercise: Using Datasets in Scala

- In this exercise, you will create, query, and save Datasets in Scala
- Please refer to the Hands-On Exercise Manual for instructions



Writing, Configuring, and Running Spark Applications

Chapter 13

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- **Writing, Configuring, and Running Spark Applications**
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Writing, Configuring, and Running Spark Applications

After completing this chapter, you will be able to

- Explain the difference between a Spark application and the Spark shell
- Write a Spark application
- Build a Scala or Java Spark application
- Submit and run a Spark application
- View the Spark application web UI
- Configure Spark application properties

Chapter Topics

Writing, Configuring, and Running Spark Applications

- **Writing a Spark Application**
- Building and Running an Application
- Application Deployment Mode
- The Spark Application Web UI
- Configuring Application Properties
- Essential Points
- **Hands-On Exercise: Writing, Configuring, and Running a Spark Application**

The Spark Shell and Spark Applications

- **The Spark shell allows interactive exploration and manipulation of data**
 - REPL using Python or Scala
- **Spark applications run as independent programs**
 - For jobs such as ETL processing, streaming, and so on
 - Python, Scala, or Java

The Spark Session and Spark Context

- **Every Spark program needs**
 - One `SparkContext` object
 - One or more `SparkSession` objects
 - If you are using Spark SQL
- **The interactive shell creates these for you**
 - A `SparkSession` object called `spark`
 - A `SparkContext` object called `sc`
- **In a standalone Spark application you must create these yourself**
 - Use a Spark session builder to create a new session
 - The builder automatically creates a new Spark context as well
 - Call `stop` on the session or context when program terminates

Creating a SparkSession Object

- **SparkSession.builder points to a Builder object**
 - Use the builder to create and configure a SparkSession object
- **The getOrCreate builder function returns the existing SparkSession object if it exists**
 - Creates a new Spark session if none exists
 - Automatically creates a new SparkContext object as sparkContext on the SparkSession object

Python Example: Name List

```
import sys

from pyspark.sql import SparkSession

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print(sys.stderr,
              "Usage: spark-submit NameList.py <input-file> <output-
file>")
        sys.exit()

    spark = SparkSession.builder.getOrCreate()
    spark.sparkContext.setLogLevel("WARN")

    peopleDF = spark.read.json(sys.argv[1])
    namesDF = peopleDF.select("firstName", "lastName")
    namesDF.write.option("header", "true").csv(sys.argv[2])

    spark.stop()
```

Language: Python

Scala Example: Name List

```
import org.apache.spark.sql.SparkSession

object NameList {
    def main(args: Array[String]) {
        if (args.length < 2) {
            System.err.println(
                "Usage: NameList <input-file> <output-file>")
            System.exit(1)
        }

        val spark = SparkSession.builder.getOrCreate()
        spark.sparkContext.setLogLevel("WARN")
        val peopleDF = spark.read.json(args(0))
        val namesDF = peopleDF.select("firstName", "lastName")
        namesDF.write.option("header", "true").csv(args(1))

        spark.stop
    }
}
```

Language: Scala

Chapter Topics

Writing, Configuring, and Running Spark Applications

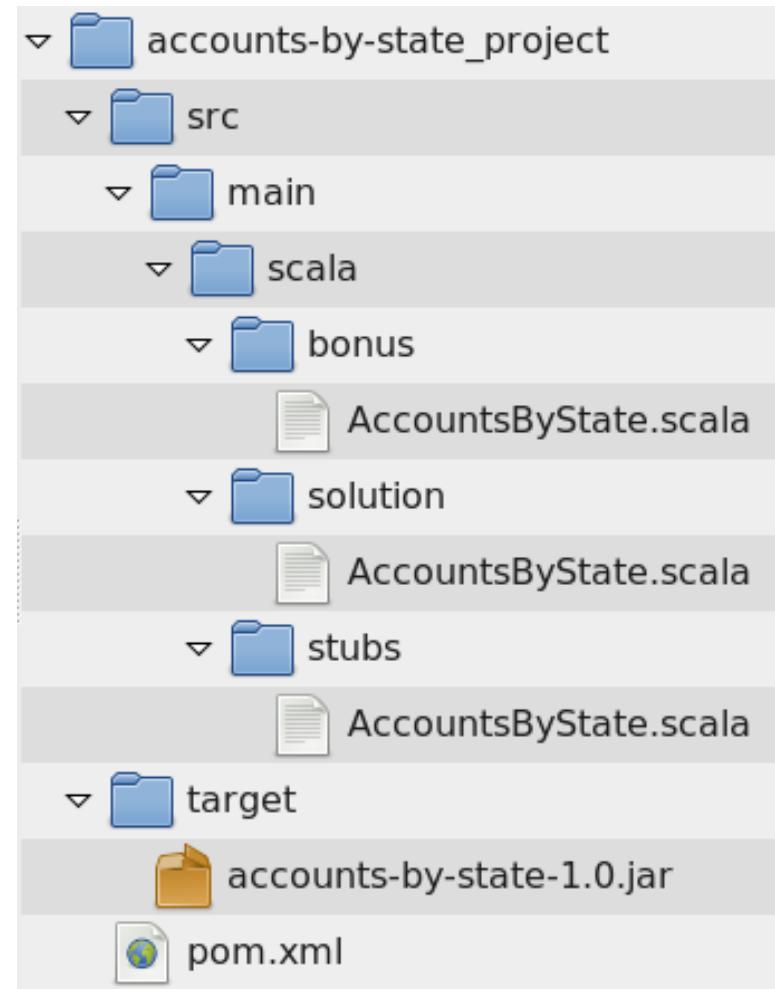
- Writing a Spark Application
- **Building and Running an Application**
- Application Deployment Mode
- The Spark Application Web UI
- Configuring Application Properties
- Essential Points
- Hands-On Exercise: Writing, Configuring, and Running a Spark Application

Building an 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 the [Spark Programming Guide](#)
- **Build details will differ depending on**
 - Version of Hadoop and CDP
 - Deployment platform (YARN, Mesos, Spark Standalone)
- **Consider using an Integrated Development Environment (IDE)**
 - IntelliJ or Eclipse are two popular examples
 - Can run Spark locally in a debugger

Building Scala Applications in the Hands-On Exercises

- Basic Apache Maven projects are provided in the exercise directory
 - **stubs**: starter Scala files—do exercises here
 - **solution**: exercise solutions
 - **bonus**: bonus solutions
- Build command: `mvn package`



Running a Spark Application

- The easiest way to run a Spark application is to use the submit script
 - Python

```
$ spark-submit NameList.py people.json namelist/
```

- Scala or Java

```
$ spark-submit --class NameList MyJarFile.jar \  
people.json namelist/
```

Submit Script Options

- The Spark submit script provides many options to specify how the application should run
 - Most are the same as for `pyspark` and `spark-shell`
- General submit flags include
 - `master`: local, yarn, or a Mesos or Spark Standalone cluster manager URI
 - `jars`: Additional JAR files
 - `pyfiles`: Additional Python files (Python only)
 - `driver-java-options`: Parameters to pass to the driver JVM
- YARN-specific flags include
 - `num-executors`: Number of executors to start application with
 - `driver-cores`: Number cores to allocate for the Spark driver
 - `queue`: YARN queue to run in
- Show all available options
 - `help`

Chapter Topics

Writing, Configuring, and Running Spark Applications

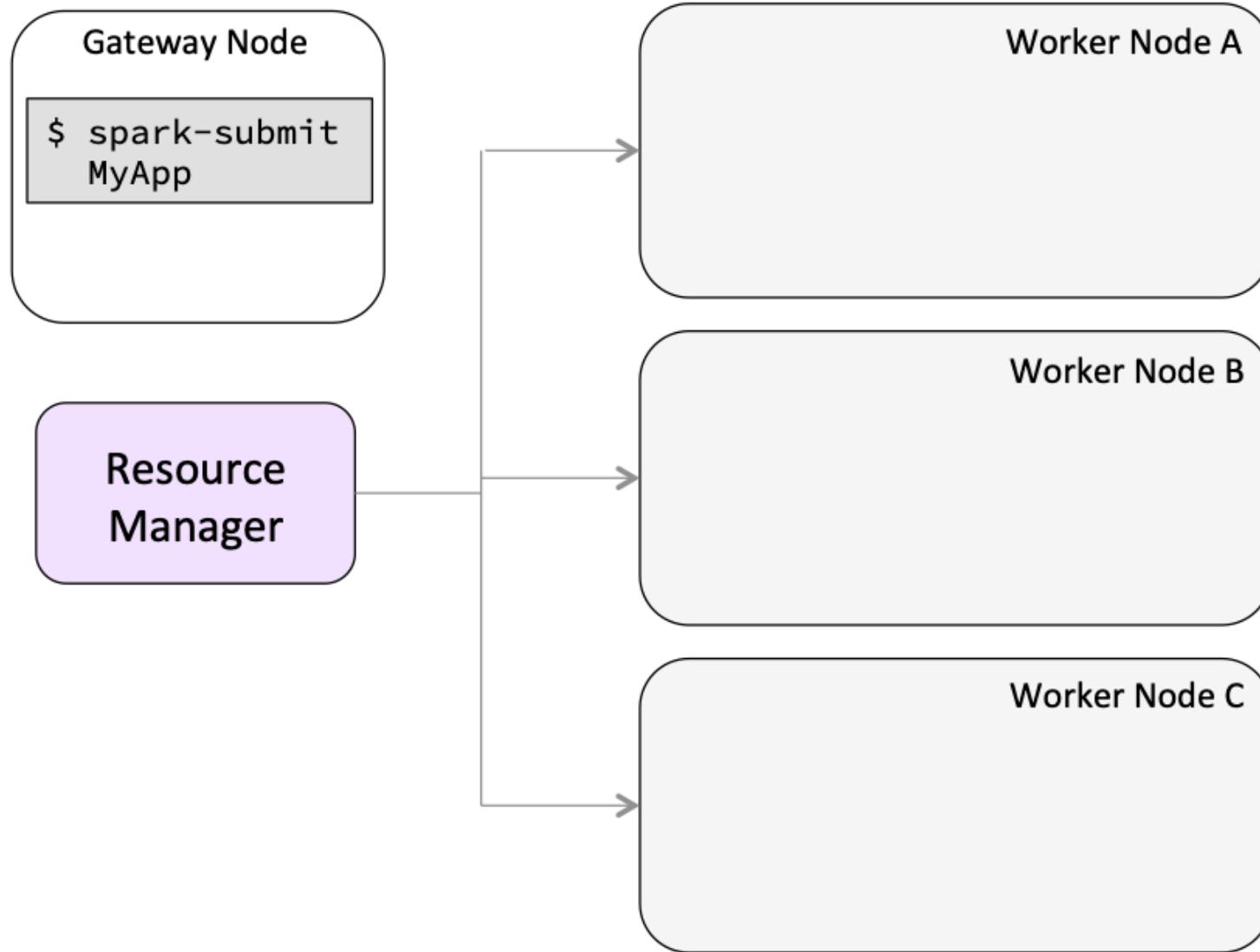
- Writing a Spark Application
- Building and Running an Application
- **Application Deployment Mode**
- The Spark Application Web UI
- Configuring Application Properties
- Essential Points
- Hands-On Exercise: Writing, Configuring, and Running a Spark Application

Application Deployment Mode

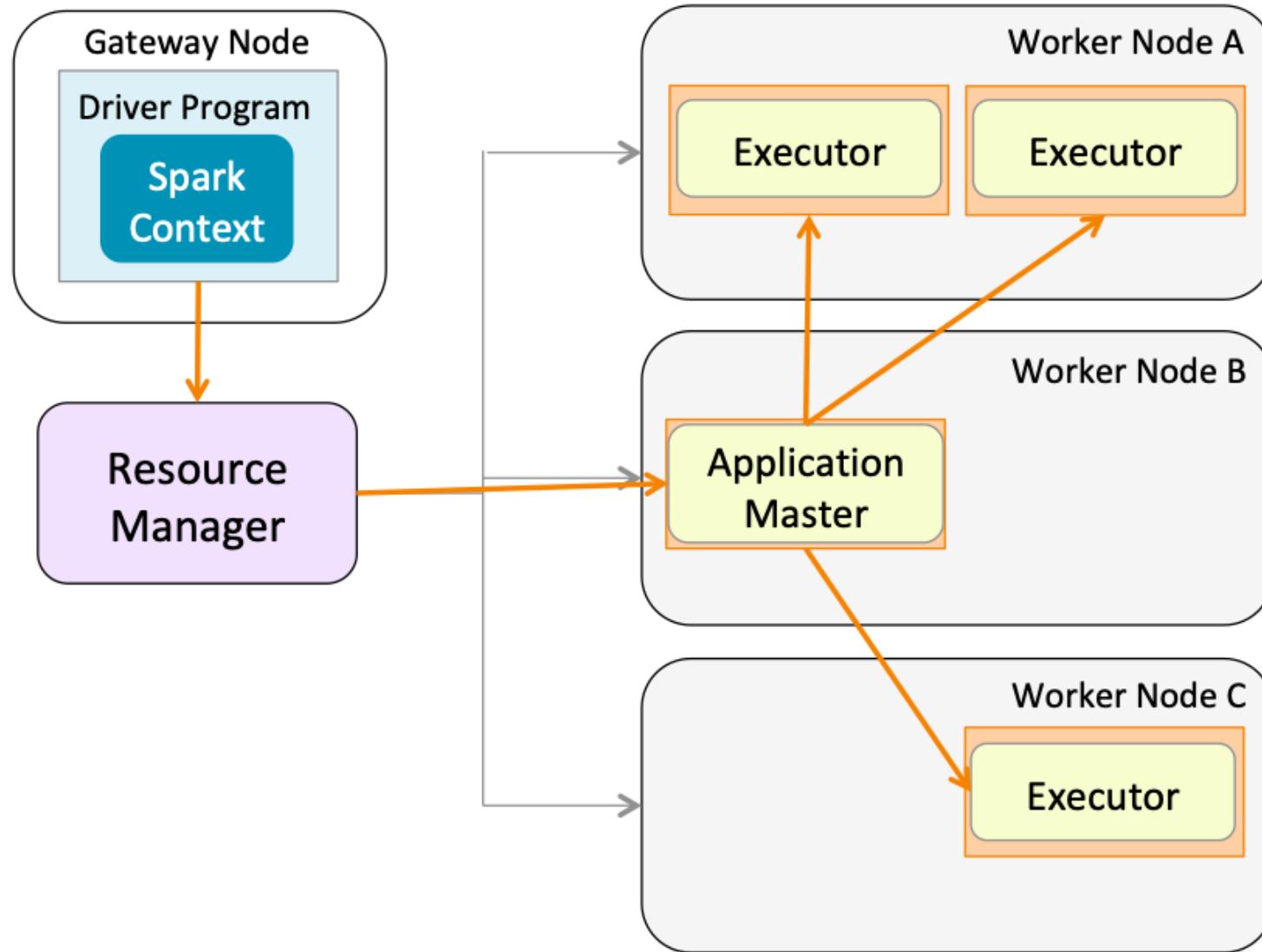
- **Spark applications can run**
 - Locally with one or more threads
 - On a cluster
 - In **client** mode (default), the driver runs locally on a gateway node
 - Requires direct communication between driver and cluster worker nodes
 - In **cluster** mode, the driver runs in the application master on the cluster
 - Common in production systems
- **Specify the deployment mode when submitting the application**

```
$ spark-submit --master yarn --deploy-mode cluster \
  NameList.py people.json namelist/
```

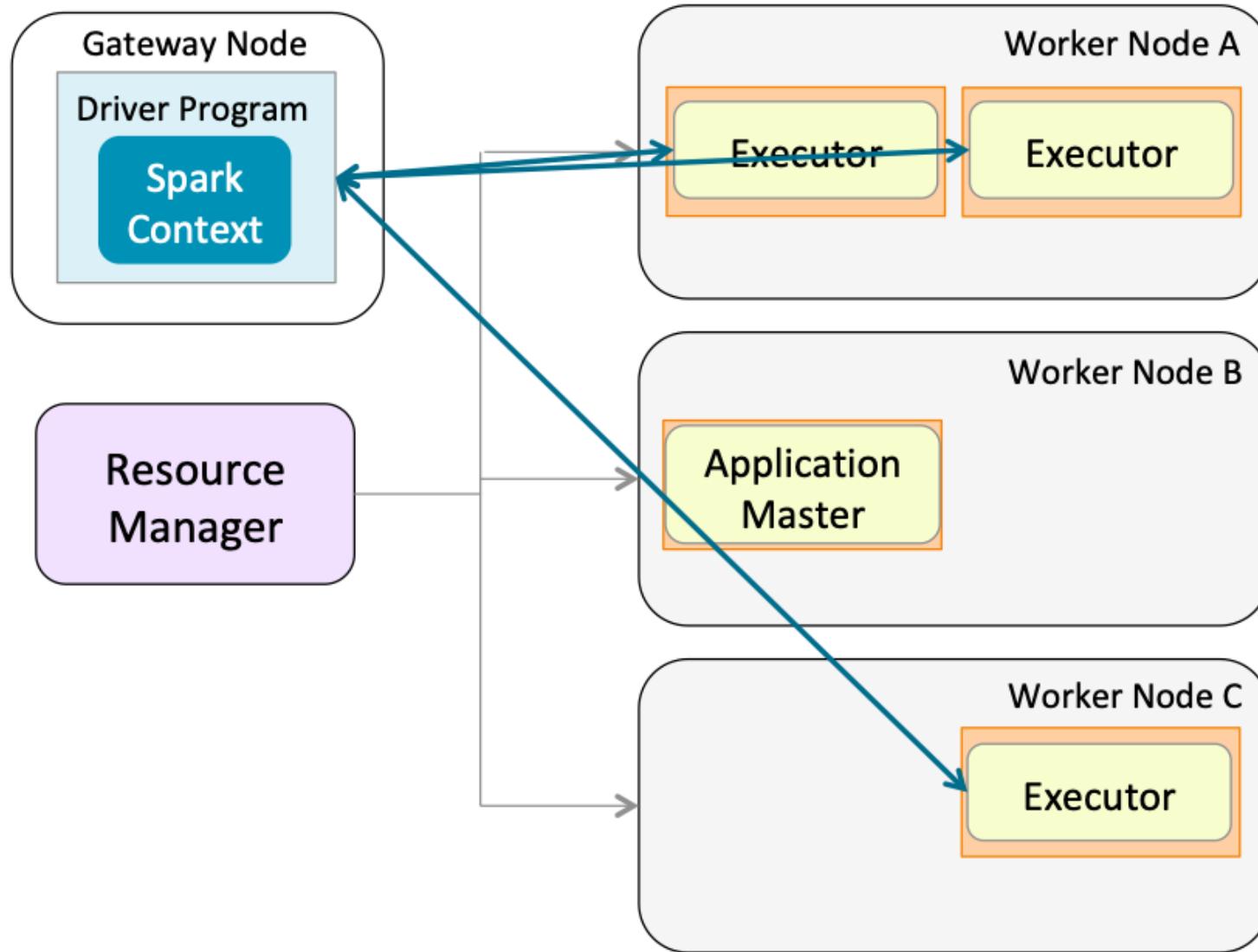
Spark Deployment Mode on YARN: Client Mode (1)



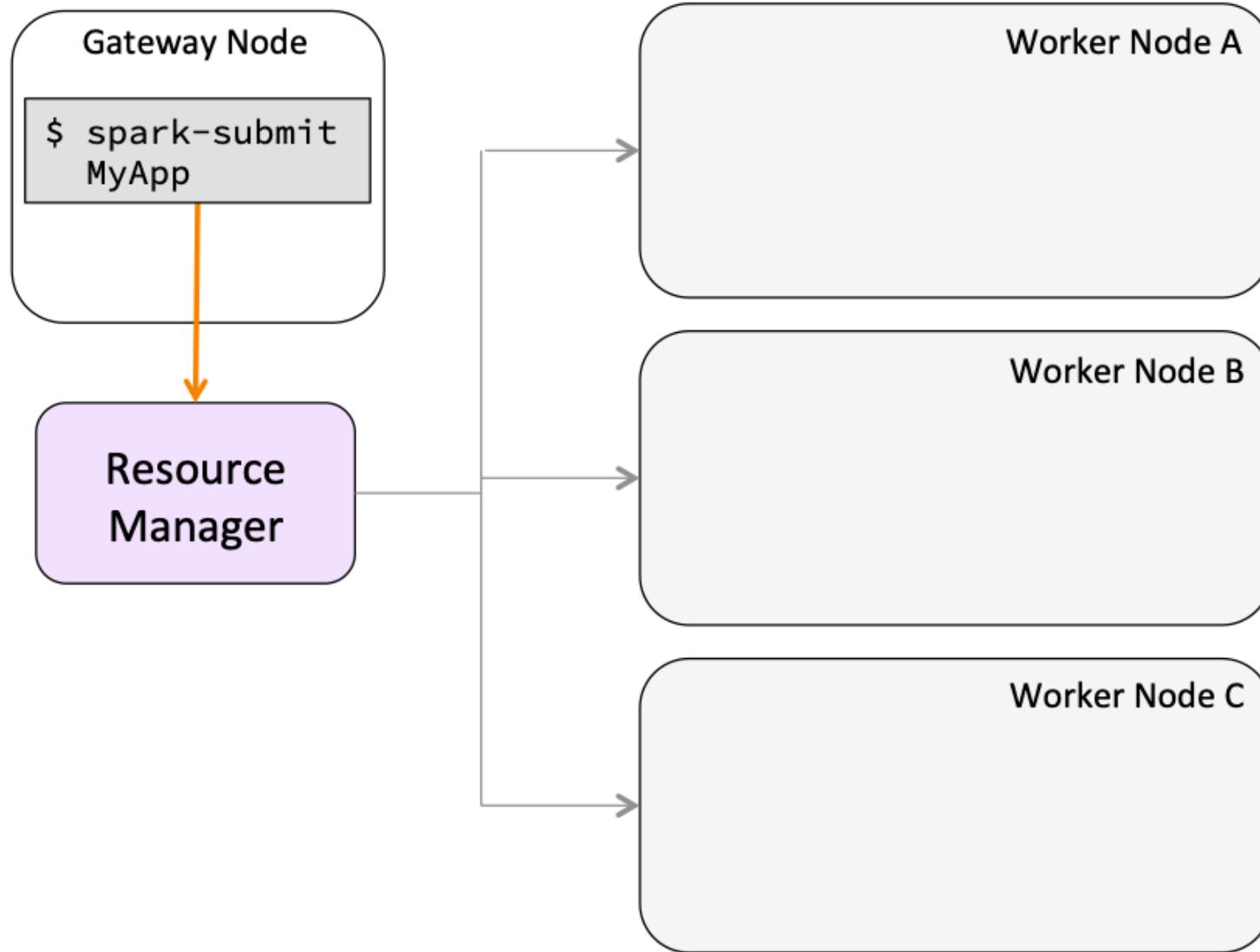
Spark Deployment Mode on YARN: Client Mode (2)



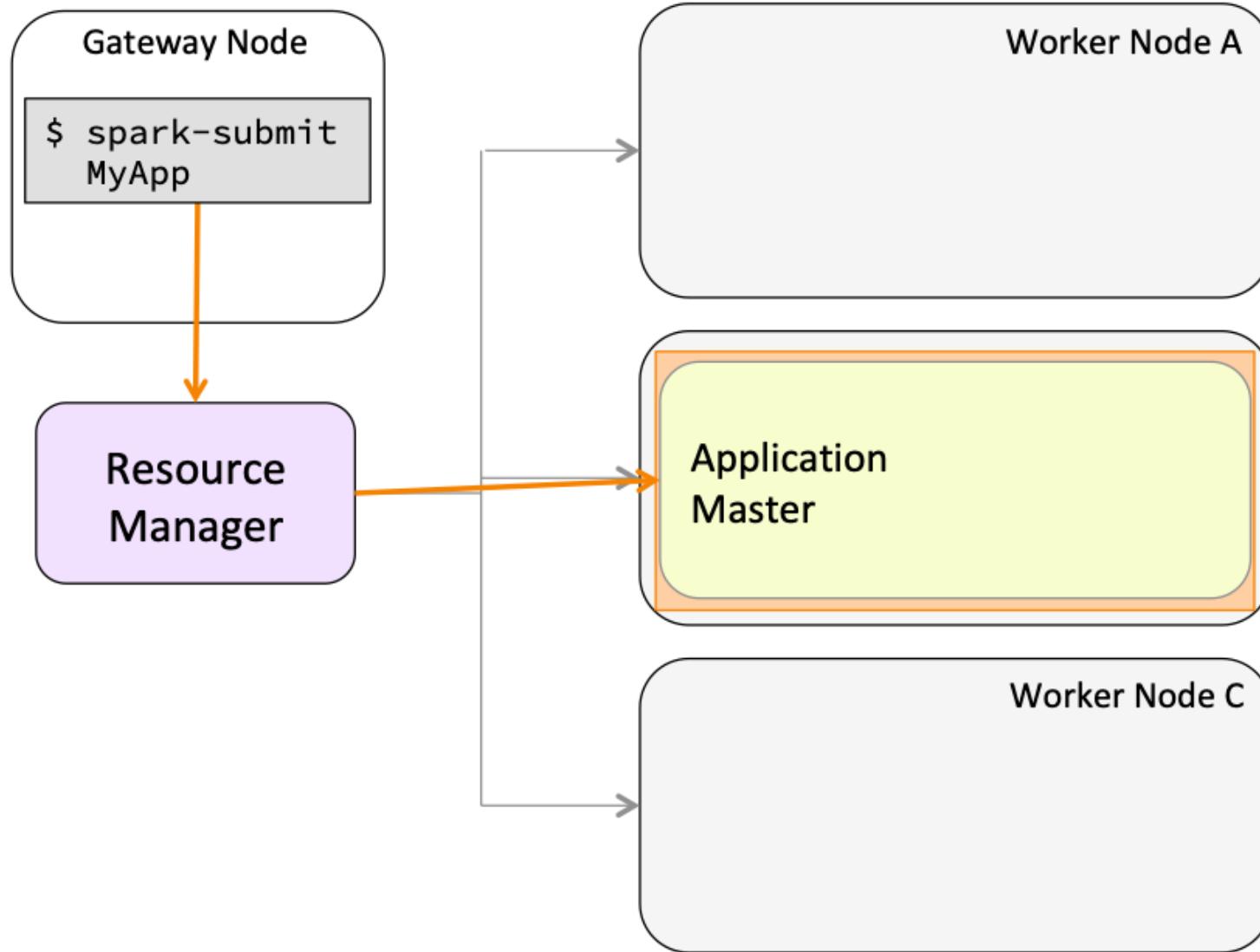
Spark Deployment Mode on YARN: Client Mode (3)



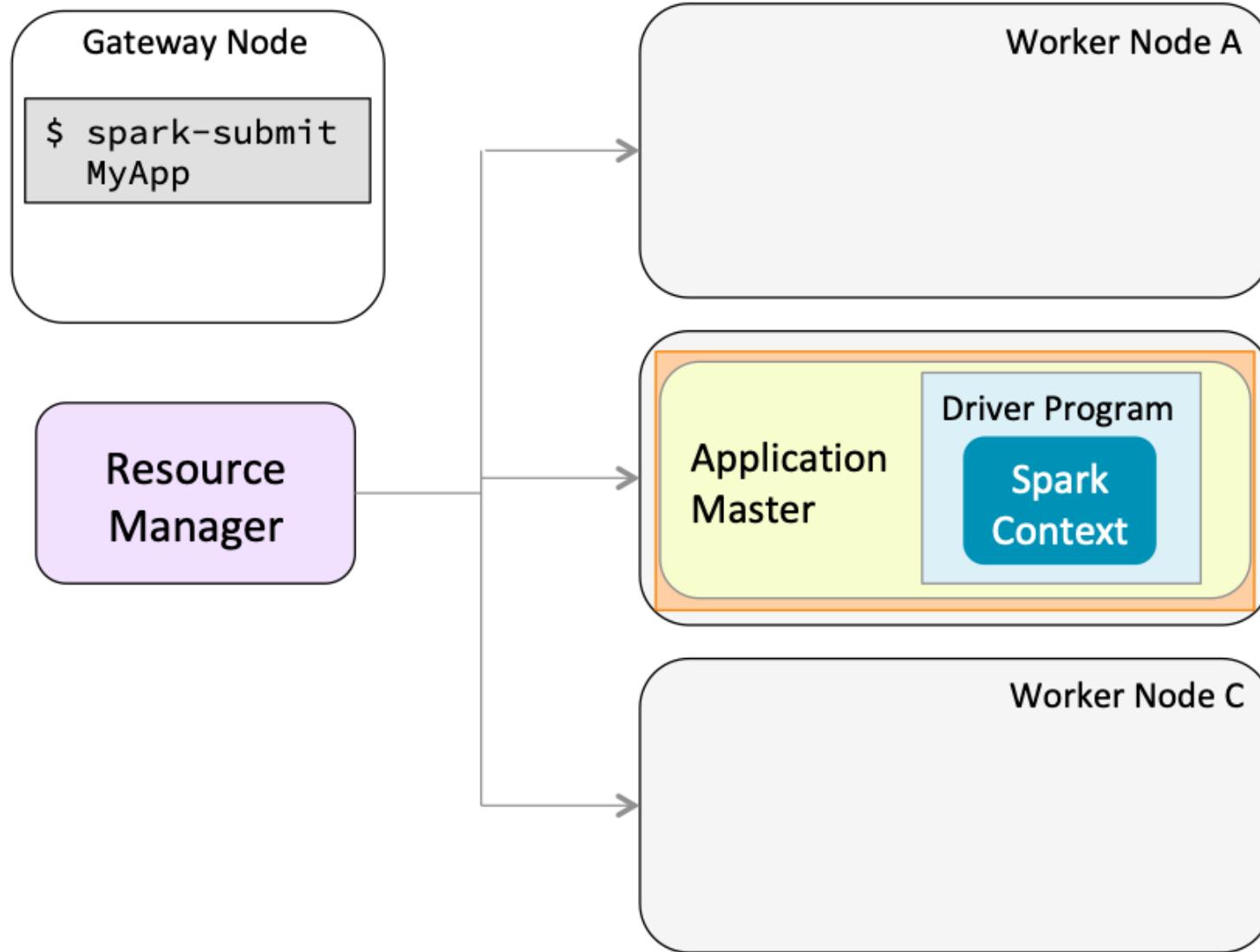
Spark Deployment Mode on YARN: Cluster Mode (1)



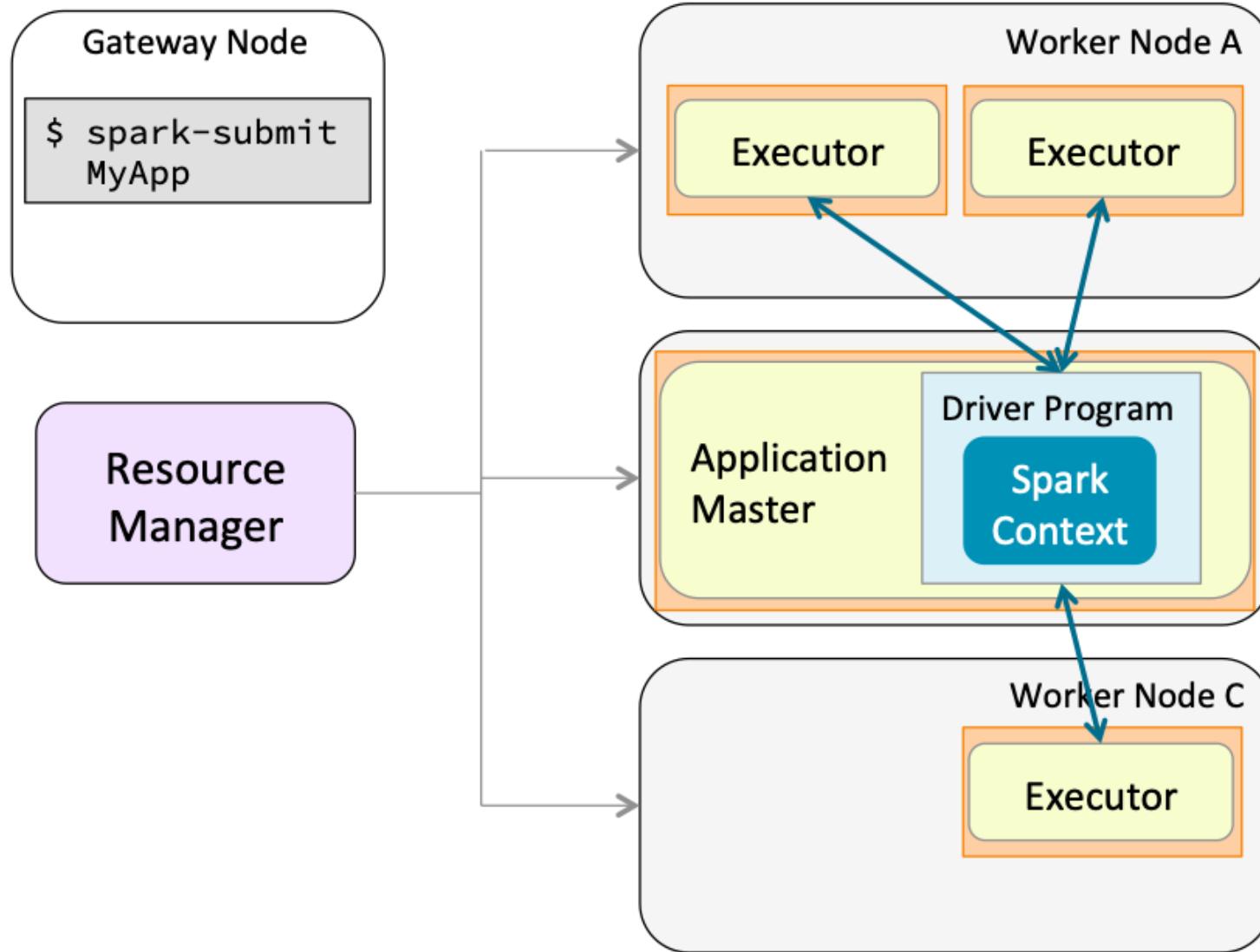
Spark Deployment Mode on YARN: Cluster Mode (2)



Spark Deployment Mode on YARN: Cluster Mode (3)



Spark Deployment Mode on YARN: Cluster Mode (4)



Chapter Topics

Writing, Configuring, and Running Spark Applications

- Writing a Spark Application
- Building and Running an Application
- Application Deployment Mode
- **The Spark Application Web UI**
- Configuring Application Properties
- Essential Points
- Hands-On Exercise: Writing, Configuring, and Running a Spark Application

The Spark Application Web UI

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

The screenshot displays the Apache Spark Application Web UI interface. It includes two main sections: "Executors" and "Jobs".

Executors Section:

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Total(4)	13	267.6 KB / 1.5 GB	0.0 B	3	1	0	60	61	12 s (0.4 s)	35.6 MB	0.0 B	2.2 MB	0
Dead(2)	4	87.8 KB / 768.2 MB	0.0 B	2	0	0	60	60	12 s (0.4 s)	35.6 MB	0.0 B	2.2 MB	0
Active(2)	9	179.8 KB / 768.2 MB	0.0 B	1	1	0	0	1	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0

Jobs Section:

Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
driver	10.0.6.229:40885	Active	7	148.5 KB / 384.1 MB	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	Thread Dump	
1	worker-2:46762	Dead	2	30.6 KB / 384.1 MB	0.0 B	1	0	0	1	1	2 s (49 ms)	65.5 KB	0.0 B	0.0 B	stdout	Thread Dump
2	worker-2:36897	Dead	2	57.2 KB / 384.1 MB	0.0 B	1	0	0	59	59	10 s (0.3 s)	35.5 MB	0.0 B	2.2 MB	stdout	Thread Dump
3	worker-2:42528	Active	2	31.3 KB / 384.1 MB	0.0 B	1	1	0	0	1	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	stdout	Thread Dump

Showing 1 to 4 of 4 entries Previous 1 Next

Completed Jobs Section:

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
3	saveAsTextFile at <console>:33	2017/05/26 06:52:57	12 s	3/3	41/41
2	saveAsTextFile at <console>:33	2017/05/26 06:50:38	2 s	1/1 (2 skipped)	18/18 (23 skipped)
1	saveAsTextFile at <console>:33	2017/05/26 06:50:19	13 s	3/3	41/41
0	first at <console>:27	2017/05/26 06:46:47	15 s	1/1	1/1

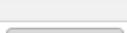
Accessing the Spark UI

- The web UI is run by the Spark driver
 - When running locally: `http://localhost:4040`
 - When running in client mode: `http://gateway:4040`
 - When running in cluster mode, access via the YARN UI

Search:												
FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Reserved CPU Vcores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Bl
Thu May 2 07:53:47 -0700 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0
Thu May 2 07:54:11 -0700 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0		History	0
N/A	RUNNING	UNDEFINED	1	1	1024	0	0	25.0	25.0		ApplicationMaster	0

Spark Application History UI

- The Spark UI is only available while the application is running
- Use the Spark application history server to view metrics for a completed application
 - Optional Spark component

Search:												
FinishTime	State	FinalStatus	Running Containers	Allocated CPU Vcores	Allocated Memory MB	Reserved CPU Vcores	Reserved Memory MB	% of Queue	% of Cluster	Progress	Tracking UI	Bl
Thu May 2 07:53:47 -0700 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0	 History	0	
Thu May 2 07:54:11 -0700 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0	 History	0	
N/A	RUNNING	UNDEFINED	1	1	1024	0	0	25.0	25.0	 ApplicationMaster	0	

Viewing the Application History UI

- You can access the history server UI by
 - Using a URL with host and port configured by a system administrator
 - Following the **History** link in the YARN UI

The screenshot shows the Apache Spark History Server interface. At the top left is the Apache Spark logo with the text "2.1.0.cloudera1". To the right is the title "History Server". Below the title is the text "Event log directory: hdfs://master-1:8020/user/spark/spark2ApplicationHistory". Underneath that is the text "Last updated: 5/26/2017, 7:11:04 AM". On the right side, there is a search bar labeled "Search:" with a placeholder "Search". Below the search bar is a table with the following data:

App ID	App Name	Started	Completed	Duration	Spark User	Last Updated	Event Log
application_1495803008771_0004	PythonWordCount	2017-05-26 14:09:48	2017-05-26 14:10:41	53 s	training	2017-05-26 14:10:41	Download
application_1495803008771_0003	PythonWordCount	2017-05-26 14:05:40	2017-05-26 14:09:43	4.1 min	training	2017-05-26 14:09:43	Download
application_1495803008771_0001	Spark shell	2017-05-26 13:07:56	2017-05-26 14:02:39	55 min	training	2017-05-26 14:02:39	Download

Chapter Topics

Writing, Configuring, and Running Spark Applications

- Writing a Spark Application
- Building and Running an Application
- Application Deployment Mode
- The Spark Application Web UI
- **Configuring Application Properties**
- Essential Points
- Hands-On Exercise: Writing, Configuring, and Running a Spark Application

Spark Application Configuration Properties

- **Spark provides numerous properties to configure your application**
- **Some example properties**
 - `spark.master`: Cluster type or URI to submit application to
 - `spark.app.name`: Application name displayed in the Spark UI
 - `spark.submit.deployMode`: Whether to run application in client or cluster mode (default: `client`)
 - `spark.ui.port`: Port to run the Spark Application UI (default `4040`)
 - `spark.executor.memory`: How much memory to allocate to each Executor (default `1g`)
 - `spark.pyspark.python`: Which Python executable to use for Pyspark applications
 - And many more...
 - See the [Spark Configuration page](#) in the Spark documentation for more details

Setting Configuration Properties

- **Most properties are set by system administrators**
 - Managed manually or using Cloudera Manager
 - Stored in a *properties file*
- **Developers can override system settings when submitting applications by**
 - Using submit script flags
 - Loading settings from a custom properties file instead of the system file
 - Setting properties programmatically in the application
- **Properties that are not set explicitly use Spark default values**

Overriding Properties Using Submit Script

- Some Spark submit script flags set application properties
 - For example
 - Use --master to set spark.master
 - Use --name to set spark.app.name
 - Use --deploy-mode to set spark.submit.deployMode
- Not every property has a corresponding script flag
 - Use --conf to set any property

```
$ spark-submit \
--conf spark.pyspark.python=/alt/path/to/python
```

Setting Properties in a Properties File

- System administrators set system properties in properties files
 - You can use your own custom properties file instead

```
spark.master          local[*]
spark.executor.memory 512k
spark.pyspark.python /alt/path/to/python
```

- Specify your properties file using the `properties-file` option

```
$ spark-submit \
--properties-file=dir/my-properties.conf
```

- Note that Spark will load **only** your custom properties file
 - System properties file is ignored
 - Copy important system settings into your custom properties file
 - Custom file will not reflect future changes to system settings

Setting Configuration Properties Programmatically

- Spark configuration settings are part of the Spark session or Spark context
- Set using the Spark session builder functions
 - appName sets spark.app.name
 - master sets spark.master
 - config can set any property

```
import org.apache.spark.sql.SparkSession  
...  
val spark = SparkSession.builder.  
  appName("my-spark-app").  
  config("spark.ui.port","5050").  
  getOrCreate()  
...
```

Language: Scala

Priority of Spark Property Settings

- **Properties set with higher priority methods override lower priority methods**
 1. Programmatic settings
 2. Submit script (command line) settings
 3. Properties file settings
 - Either administrator site-wide file or custom properties file
 4. Spark default settings
 - See the [Spark Configuration guide](#)

Viewing Spark Properties

- You can view the Spark property settings two ways
 - Using --verbose with the submit script
 - In the Spark Application UI **Environment** tab

The screenshot shows the Apache Spark Application UI interface. At the top, there's a navigation bar with tabs: Jobs, Stages, Storage, Environment (which is highlighted in grey), Executors, SQL, and PySparkShell application UI. Below the tabs, the page title is "Environment". Under "Runtime Information", there's a table with four rows:

Name	Value
Java Home	/usr/java/jdk1.7.0_67-cloudera/jre
Java Version	1.7.0_67 (Oracle Corporation)
Scala Version	version 2.11.8

Below that, under "Spark Properties", is another table with eight rows:

Name	Value
spark.app.id	application_1495803008771_0005
spark.app.name	PySparkShell
spark.authenticate	false
spark.driver.appUIAddress	http://10.0.6.229:4040
spark.driver.extraLibraryPath	/opt/cloudera/parcels/CDH-5.11.0-1.cdh5.11.0.p0.34 /lib/hadoop/lib/native
spark.driver.host	10.0.6.229
spark.driver.port	50884

Chapter Topics

Writing, Configuring, and Running Spark Applications

- Writing a Spark Application
- Building and Running an Application
- Application Deployment Mode
- The Spark Application Web UI
- Configuring Application Properties
- **Essential Points**
- Hands-On Exercise: Writing, Configuring, and Running a Spark Application

Essential Points

- Use the Spark shell for interactive data exploration
- Write a Spark application to run independently
- Spark applications require a `SparkContext` object and usually a `SparkSession` object
- Use Maven or a similar build tool to compile and package Scala and Java applications
 - Not required for Python
- Deployment mode determines where the application driver runs—on the gateway or on a worker node
- Use the `spark-submit` script to run Spark applications locally or on a cluster
- Application properties can be set on the command line, in a properties file, or in the application code

Chapter Topics

Writing, Configuring, and Running Spark Applications

- Writing a Spark Application
- Building and Running an Application
- Application Deployment Mode
- The Spark Application Web UI
- Configuring Application Properties
- Essential Points
- **Hands-On Exercise: Writing, Configuring, and Running a Spark Application**

Hands-On Exercise: Writing, Configuring, and Running a Spark Application

- In this exercise, you will write, configure, and run a standalone Spark application
- Please refer to the Hands-On Exercise Manual for instructions



Spark Distributed Processing

Chapter 14

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- **Spark Distributed Processing**
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Spark Distributed Processing

After completing this chapter, you will be able to

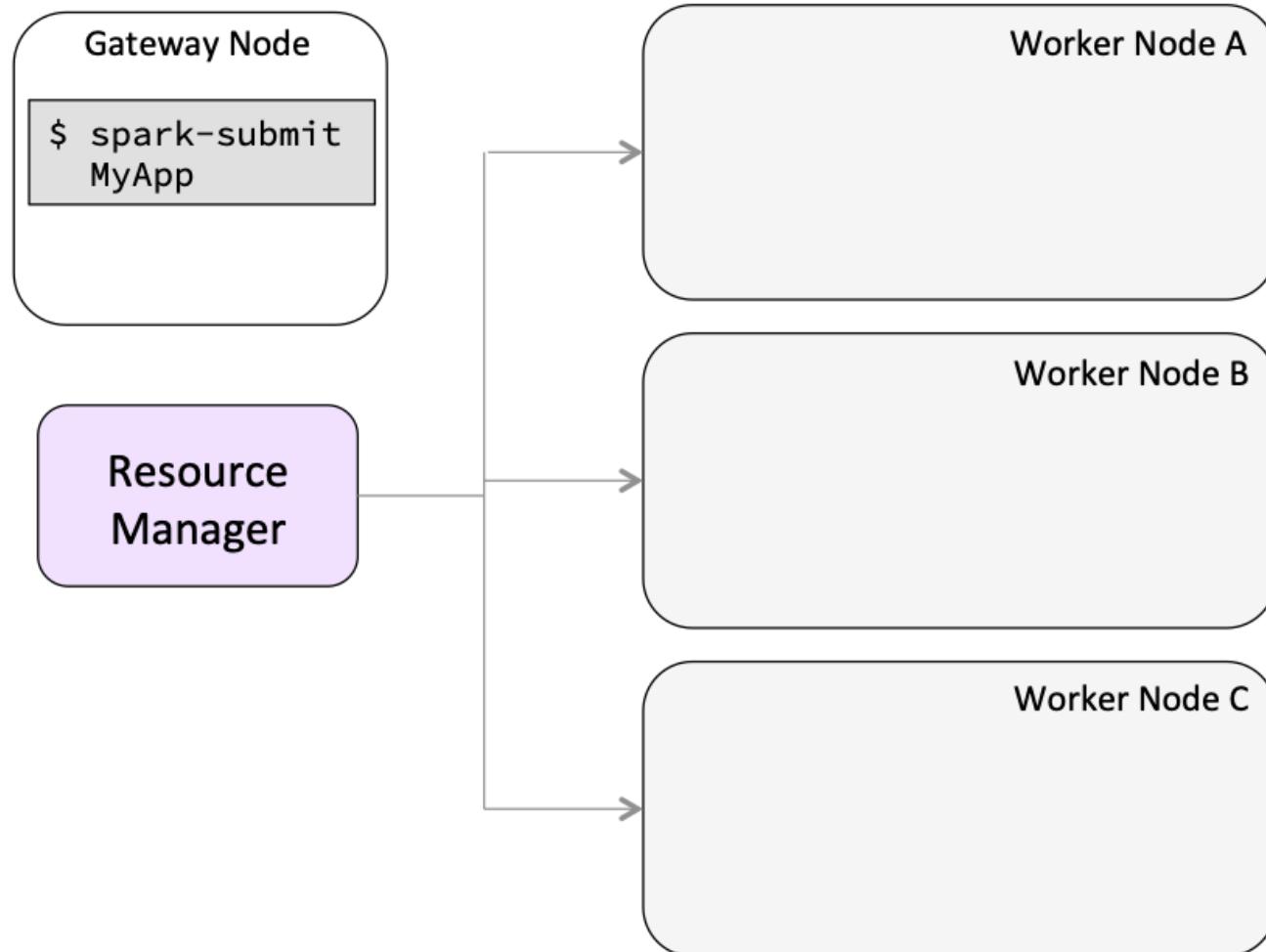
- **Describe how partitions distribute data in RDDs, DataFrames, and Datasets across a cluster**
- **Explain how Spark executes queries in parallel**
- **Control parallelization through partitioning**
- **View query execution plans and RDD lineages**
- **View and monitor tasks and stages**

Chapter Topics

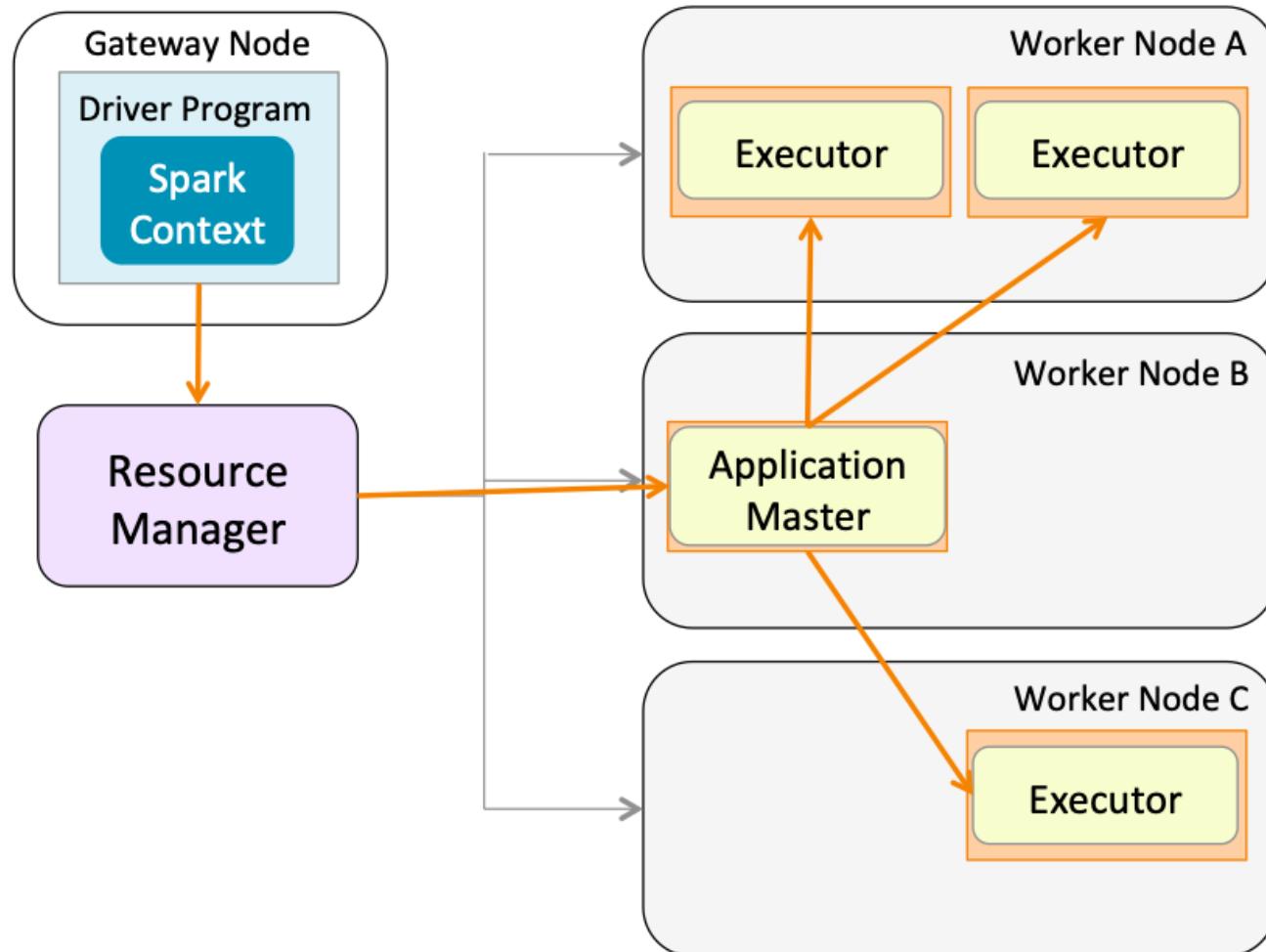
Spark Distributed Processing

- **Review: Apache Spark on a Cluster**
- RDD Partitions
- Example: Partitioning in Queries
- Stages and Tasks
- Job Execution Planning
- Example: Catalyst Execution Plan
- Example: RDD Execution Plan
- Essential Points
- Hands-On Exercise: Exploring Query Execution

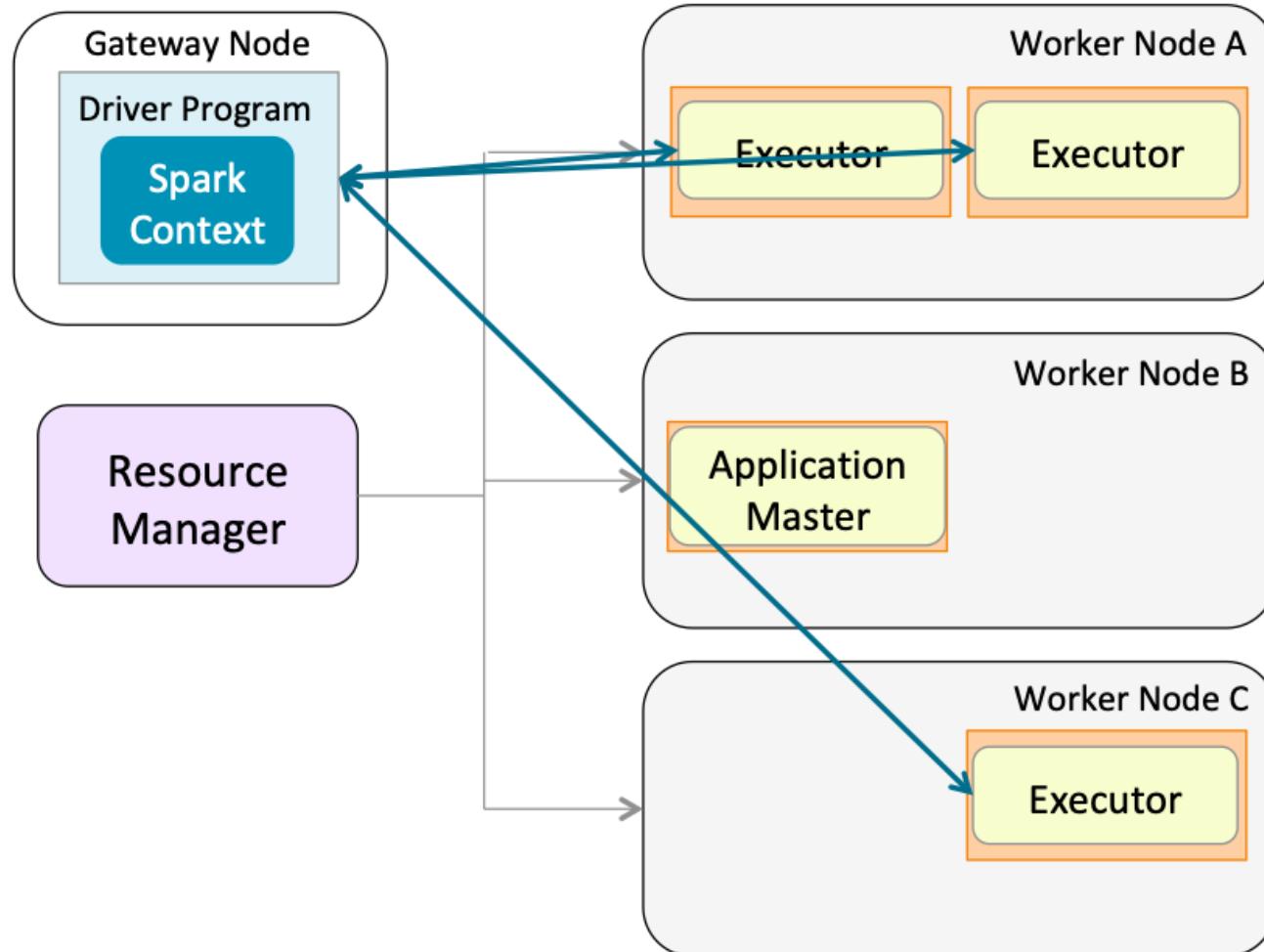
Review of Spark on YARN (1)



Review of Spark on YARN (2)



Review of Spark on YARN (3)



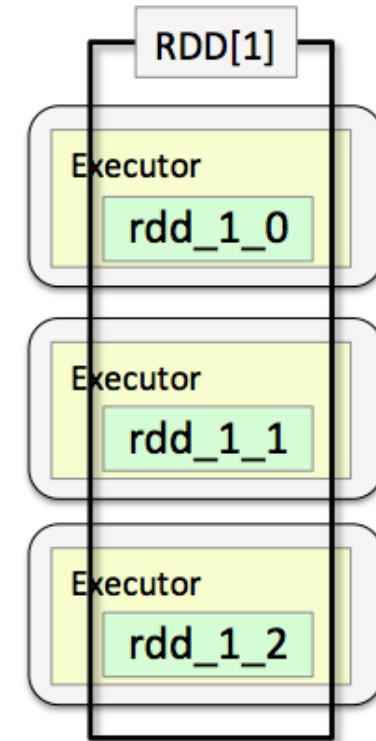
Chapter Topics

Spark Distributed Processing

- Review: Apache Spark on a Cluster
- **RDD Partitions**
- Example: Partitioning in Queries
- Stages and Tasks
- Job Execution Planning
- Example: Catalyst Execution Plan
- Example: RDD Execution Plan
- Essential Points
- Hands-On Exercise: Exploring Query Execution

Data Partitioning (1)

- Data in Datasets and DataFrames is managed by underlying RDDs
- Data in an RDD is *partitioned* across executors
 - This is what makes RDDs *distributed*
 - Spark assigns tasks to process a partition to the executor managing that partition
- Data Partitioning is done automatically by Spark
 - In some cases, you can control how many partitions are created
 - More partitions = more parallelism



Data Partitioning (2)

- **Spark determines how to partition data in an RDD, Dataset, or DataFrame when**
 - The data source is read
 - An operation is performed on a DataFrame, Dataset, or RDD
 - Spark optimizes a query
 - You call `repartition` or `coalesce`

Partitioning from Data in Files

- **Partitions are determined when files are read**
 - Core Spark determines RDD partitioning based on location, number, and size of files
 - Usually each file is loaded into a single partition
 - Very large files are split across multiple partitions
- Catalyst optimizer manages partitioning of RDDs that implement DataFrames and Datasets

Finding the Number of Partitions in an RDD

- You can view the number of partitions in an RDD by calling the function `getNumPartitions`

```
myRDD.getNumPartitions
```

Language: Scala

```
myRDD.getNumPartitions()
```

Language: Python

Chapter Topics

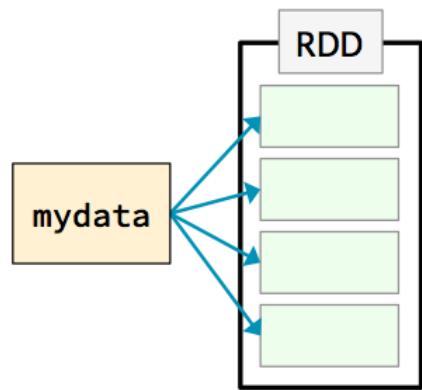
Spark Distributed Processing

- Review: Apache Spark on a Cluster
- RDD Partitions
- **Example: Partitioning in Queries**
- Stages and Tasks
- Job Execution Planning
- Example: Catalyst Execution Plan
- Example: RDD Execution Plan
- Essential Points
- Hands-On Exercise: Exploring Query Execution

Example: Average Word Length by Letter (1)

```
avglens = sc.textFile(mydata)
```

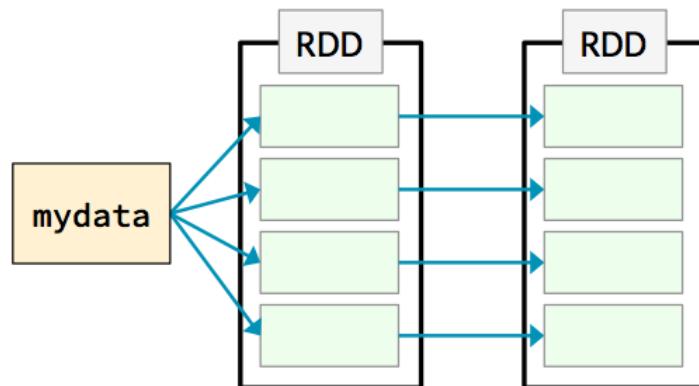
Language: Python



Example: Average Word Length by Letter (2)

```
avglens = sc.textFile(mydata) \  
.flatMap(lambda line: line.split(' '))
```

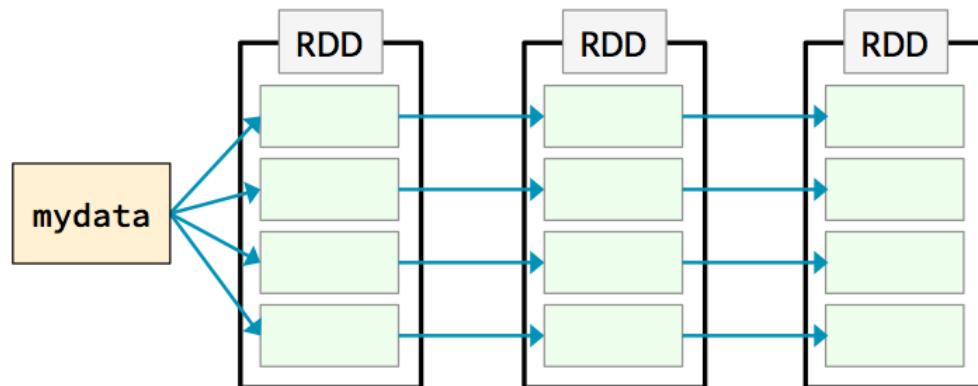
Language: Python



Example: Average Word Length by Letter (3)

```
avglens = sc.textFile(mydata) \
.flatMap(lambda line: line.split(' ')) \
.map(lambda word: (word[0],len(word)))
```

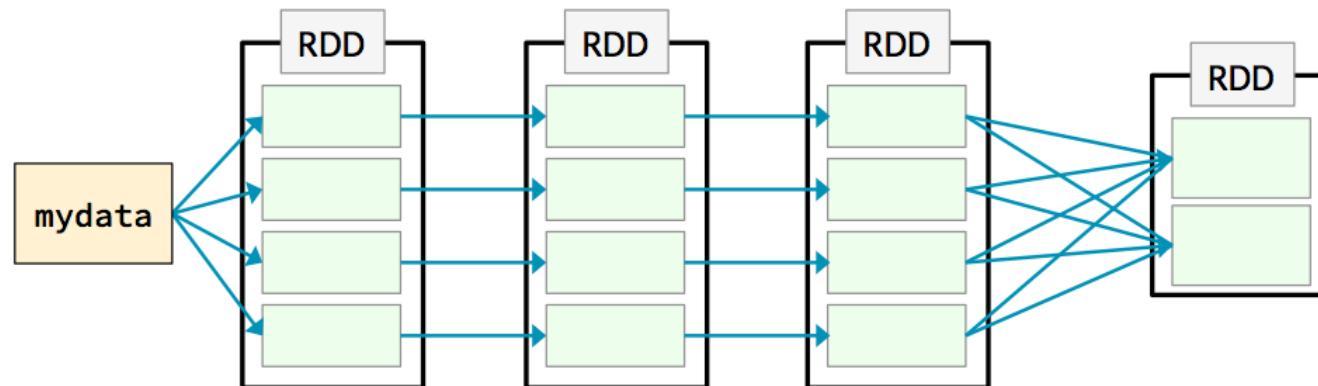
Language: Python



Example: Average Word Length by Letter (4)

```
avglens = sc.textFile(mydata) \  
.flatMap(lambda line: line.split(' ')) \  
.map(lambda word: (word[0],len(word))) \  
.groupByKey()
```

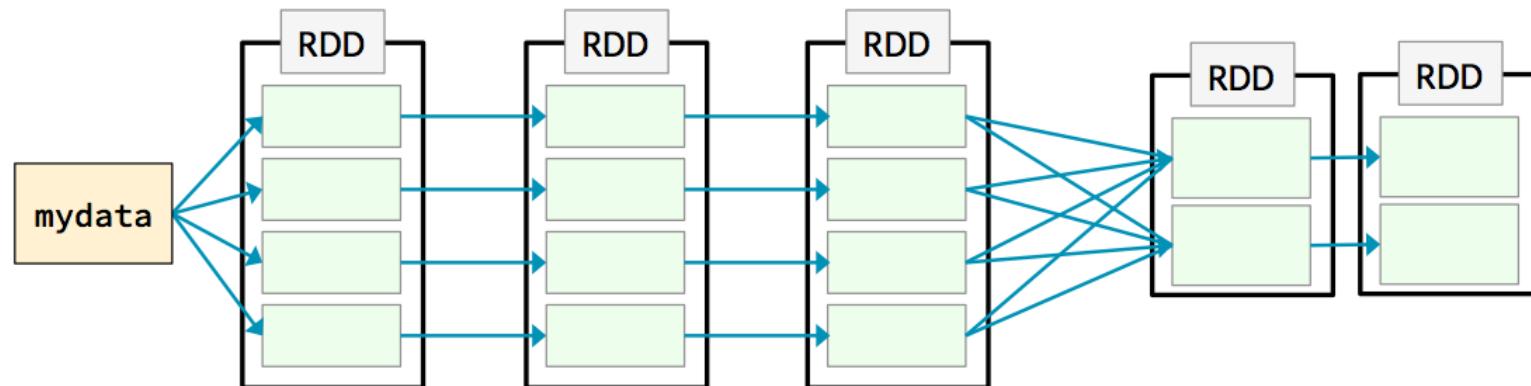
Language: Python



Example: Average Word Length by Letter (5)

```
avglens = sc.textFile(mydata) \  
.flatMap(lambda line: line.split(' ')) \  
.map(lambda word: (word[0],len(word))) \  
.groupByKey() \  
.map(lambda (k, values): \  
(k, sum(values)/len(values)))
```

Language: Python



Chapter Topics

Spark Distributed Processing

- Review: Apache Spark on a Cluster
- RDD Partitions
- Example: Partitioning in Queries
- **Stages and Tasks**
- Job Execution Planning
- Example: Catalyst Execution Plan
- Example: RDD Execution Plan
- Essential Points
- Hands-On Exercise: Exploring Query Execution

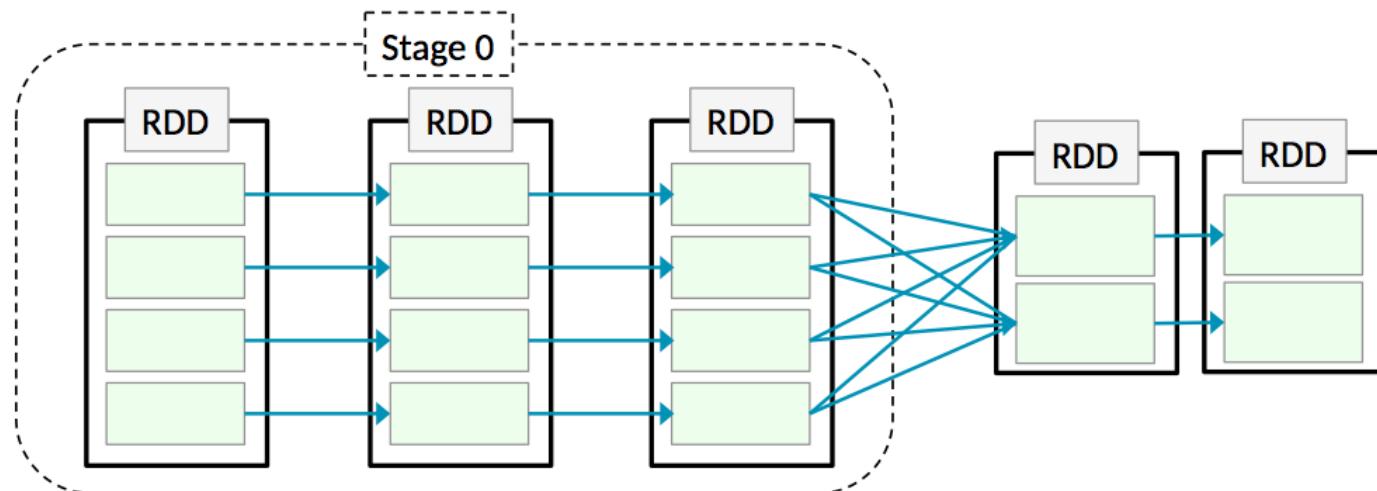
Stages and Tasks

- A **task** is a series of operations that work on the same partition and are pipelined together
- **Stages** group together tasks that can run in parallel on different partitions of the same RDD
- **Jobs** consist of all the stages that make up a query
- **Catalyst optimizes partitions and stages when using DataFrames and Datasets**
 - Core Spark provides limited optimizations when you work directly with RDDs
 - You need to code most RDD optimizations manually
 - To improve performance, be aware of how tasks and stages are executed when working with RDDs

Example: Query Stages and Tasks (1)

```
avglens = sc.textFile(mydata) \
.flatMap(lambda line: line.split(' ')) \
.map(lambda word: (word[0],len(word))) \
.groupByKey() \
.map(lambda (k, values): \
(k, sum(values)/len(values)))\n\navglens.saveAsTextFile("avglen-output")
```

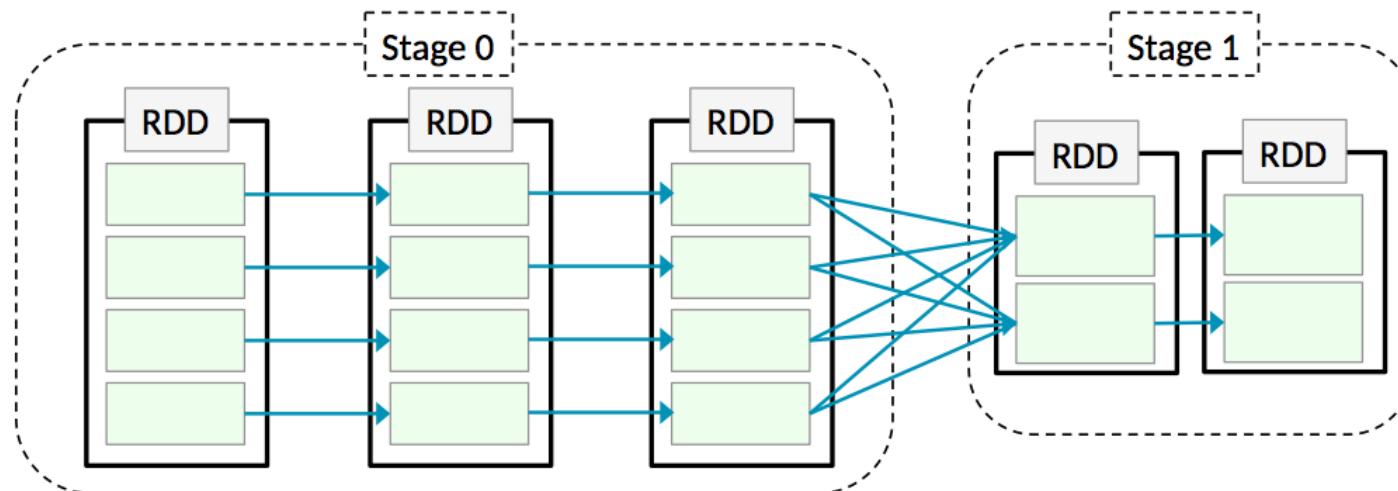
Language: Python



Example: Query Stages and Tasks (2)

```
avglens = sc.textFile(mydata) \
.flatMap(lambda line: line.split(' ')) \
.map(lambda word: (word[0],len(word))) \
.groupByKey() \
.map(lambda (k, values): \
(k, sum(values)/len(values)))\n\navglens.saveAsTextFile("avglen-output")
```

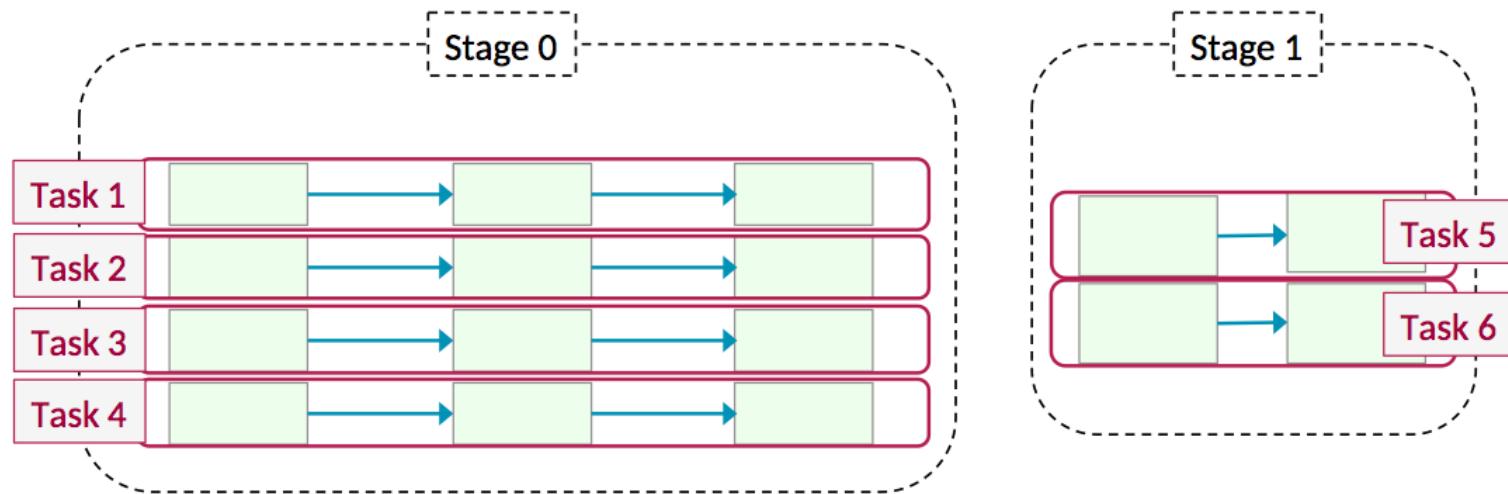
Language: Python



Example: Query Stages and Tasks (3)

```
avglens = sc.textFile(mydata) \
.flatMap(lambda line: line.split(' ')) \
.map(lambda word: (word[0],len(word))) \
.groupByKey() \
.map(lambda (k, values): \
(k, sum(values)/len(values)))\n\navglens.saveAsTextFile("avglen-output")
```

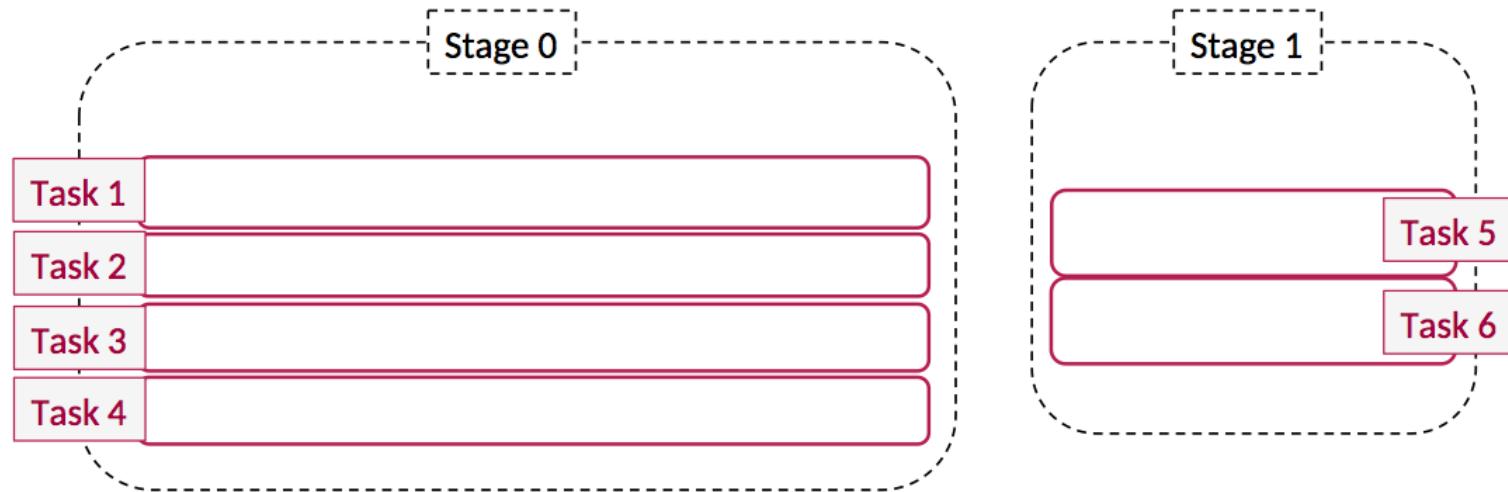
Language: Python



Example: Query Stages and Tasks (4)

```
avglens = sc.textFile(mydata) \
.flatMap(lambda line: line.split(' ')) \
.map(lambda word: (word[0],len(word))) \
.groupByKey() \
.map(lambda (k, values): \
(k, sum(values)/len(values)))\n\navglens.saveAsTextFile("avglen-output")
```

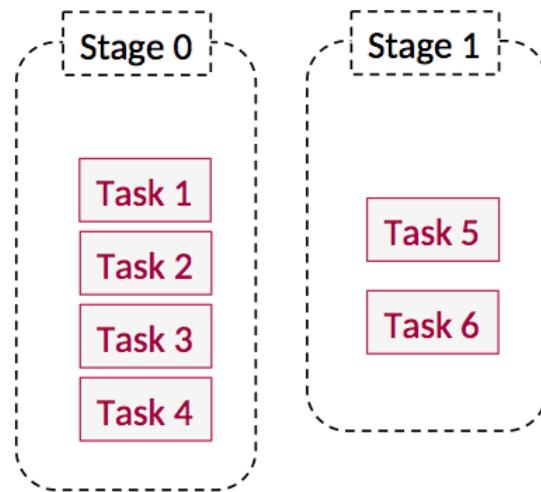
Language: Python



Example: Query Stages and Tasks (5)

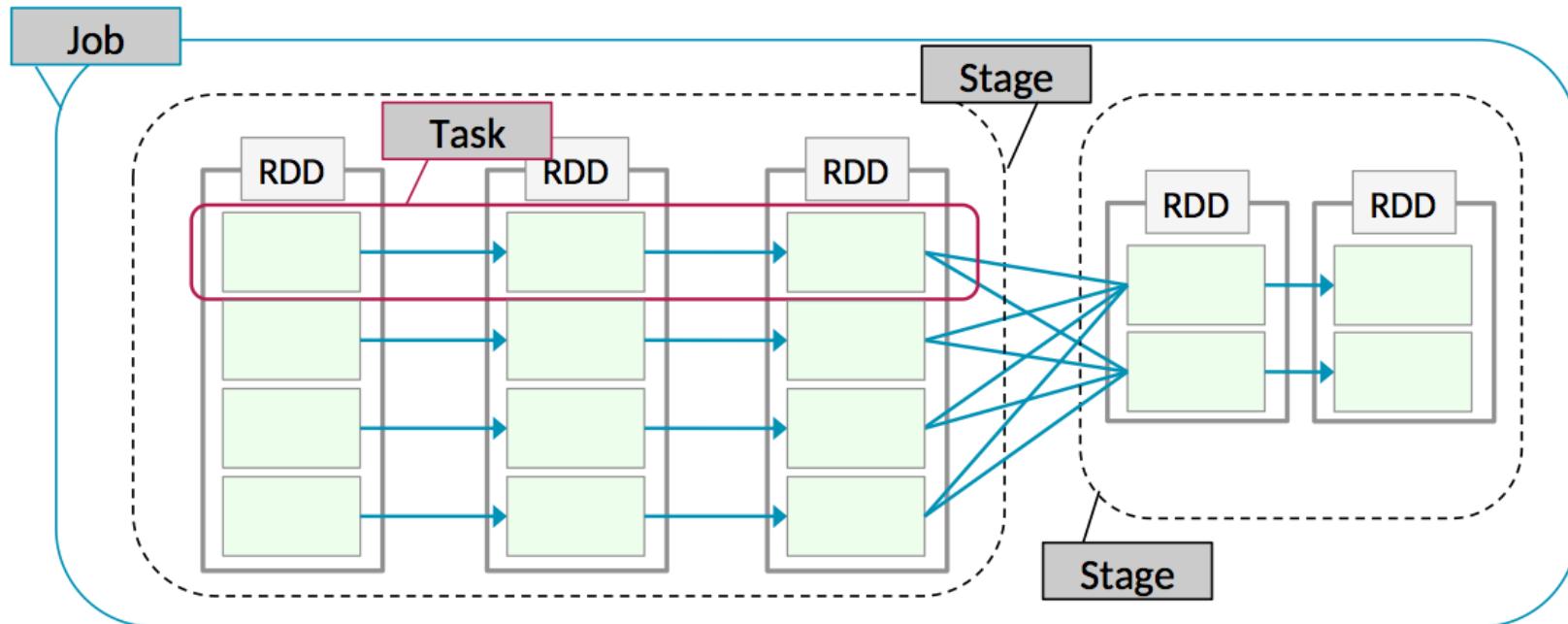
```
avglens = sc.textFile(mydata) \
.flatMap(lambda line: line.split(' ')) \
.map(lambda word: (word[0],len(word))) \
.groupByKey() \
.map(lambda (k, values): \
(k, sum(values)/len(values)))\n\navglens.saveAsTextFile("avglen-output")
```

Language: Python



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—the set of jobs managed by a single driver



Chapter Topics

Spark Distributed Processing

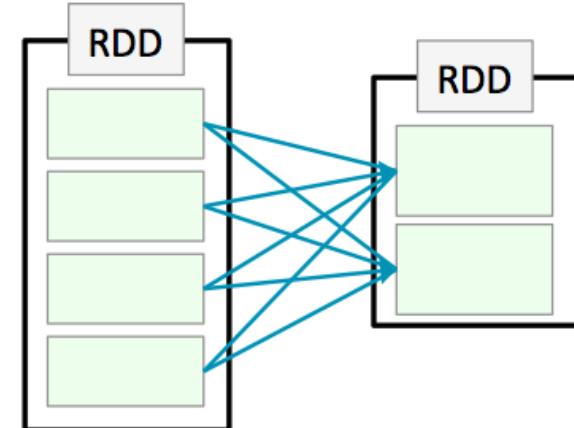
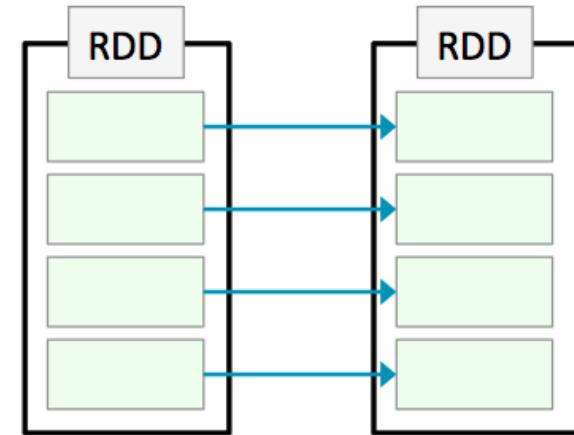
- Review: Apache Spark on a Cluster
- RDD Partitions
- Example: Partitioning in Queries
- Stages and Tasks
- **Job Execution Planning**
- Example: Catalyst Execution Plan
- Example: RDD Execution Plan
- Essential Points
- Hands-On Exercise: Exploring Query Execution

Execution Plans

- **Spark creates an execution plan for each job in an application**
- **Catalyst creates SQL, Dataset, and DataFrame execution plans**
 - Highly optimized
- **Core Spark creates execution plans for RDDs**
 - Based on RDD lineage
 - Limited optimization

How Execution Plans are Created

- Spark constructs a DAG (Directed Acyclic Graph) based on RDD dependencies
- **Narrow dependencies**
 - Each partition in the child RDD depends on just one partition of the parent RDD
 - No shuffle required between executors
 - Can be pipelined into a single stage
 - Examples: `map`, `filter`, and `union`
- **Wide (or shuffle) dependencies**
 - Child partitions depend on multiple partitions in the parent RDD
 - Defines a new stage
 - Examples: `reduceByKey`, `join`, and `groupByKey`



Controlling the Number of Partitions in RDDs (1)

- **Partitioning determines how queries execute on a cluster**
 - More partitions = more parallel tasks
 - Cluster will be under-utilized if there are too few partitions
 - But too many partitions will increase overhead without an offsetting increase in performance
- **Catalyst controls partitioning for SQL, DataFrame, and Dataset queries**
- **You can control how many partitions are created for RDD queries**

Controlling the Number of Partitions in RDDs (2)

- Specify the number of partitions when data is read
 - Default partitioning is based on size and number of the files (minimum is two)
 - Specify a different minimum number when reading a file

```
myRDD = sc.textFile(myfile, 5)
```

- Manually repartition
 - Create a new RDD with a specified number of partitions using `repartition` or `coalesce`
 - `coalesce` reduces the number of partitions without requiring a shuffle
 - `repartition` shuffles the data into more or fewer partitions

```
newRDD = myRDD.repartition(15)
```

Controlling the Number of Partitions in RDDs (3)

- Specify the number of partitions created by transformations
 - Wide (shuffle) operations such as `reduceByKey` and `join` repartition data
 - By default, the number of partitions created is based on the number of partitions of the parent RDD(s)
 - Choose a different default by configuring the `spark.default.parallelism` property

```
spark.default.parallelism 15
```

- Override the default with the optional `numPartitions` operation parameter

```
countRDD = wordsRDD. \
    reduceByKey(lambda v1, v2: v1 + v2, 15)
```

Catalyst Optimizer

- Catalyst can improve SQL, DataFrame, and Dataset query performance by optimizing the DAG to
 - Minimize data transfer between executors
 - Such as *broadcast* joins—small data sets are pushed to the executors where the larger data sets reside
 - Minimize wide (shuffle) operations
 - Such as unioning two RDDs—grouping, sorting, and joining do not require shuffling
 - Pipeline as many operations into a single stage as possible
 - Generate code for a whole stage at run time
 - Break a query job into multiple jobs, executed in a series

Catalyst Execution Plans

- Execution plans for DataFrame, Dataset, and SQL queries include the following phases
 - Parsed logical plan—calculated directly from the sequence of operations specified in the query
 - Analyzed logical plan—resolves relationships between data sources and columns
 - Optimized logical plan—applies rule-based optimizations
 - Physical plan—describes the actual sequence of operations
 - Code generation—generates bytecode to run on each node, based on a cost model

Chapter Topics

Spark Distributed Processing

- Review: Apache Spark on a Cluster
- RDD Partitions
- Example: Partitioning in Queries
- Stages and Tasks
- Job Execution Planning
- **Example: Catalyst Execution Plan**
- Example: RDD Execution Plan
- Essential Points
- Hands-On Exercise: Exploring Query Execution

Viewing Catalyst Execution Plans

- You can view SQL, DataFrame, and Dataset (Catalyst) execution plans
 - Use DataFrame/Dataset `explain`
 - Shows only the physical execution plan by default
 - Pass `true` to see the full execution plan
 - Use **SQL** tab in the Spark UI or history server
 - Shows details of execution after job runs

Example: Catalyst Execution Plan (1)

```
peopleDF = spark.read. \
option("header","true").csv("people.csv")
pcodesDF = spark.read. \
option("header","true").csv("pcodes.csv")
joinedDF = peopleDF.join(pcodesDF, "pcode")
joinedDF.explain(True)

== Parsed Logical Plan ==
'Join UsingJoin(Inner,Buffer(pcode))
:- Relation[pcode#10,lastName#11,firstName#12,age#13] csv
+- Relation[pcode#28,city#29,state#30] csv
```

Language: Python
continued on next slide...

Example: Catalyst Execution Plan (2)

```
== Analyzed Logical Plan ==
pcode: string, lastName: string, firstName: string, age: string,
city: string, state: string
Project [pcode#10, lastName#11, firstName#12, age#13, city#29, state#30]
+- Join Inner, (pcode#10 = pcode#28)
  :- Relation[pcode#10,lastName#11,firstName#12,age#13] csv
  +- Relation[pcode#28,city#29,state#30] csv

== Optimized Logical Plan ==
Project [pcode#10, lastName#11, firstName#12, age#13, city#29, state#30]
+- Join Inner, (pcode#10 = pcode#28)
  :- Filter isnotnull(pcode#10)
  :  +- Relation[pcode#10,lastName#11,firstName#12,age#13] csv
  +- Filter isnotnull(pcode#28)
    +- Relation[pcode#28,city#29,state#30] csv
```

Language: Python
continued on next slide...

Example: Catalyst Execution Plan (3)

```
== Physical Plan ==
(2) Project [PCODE#10, lastName#11, firstName#12, age#13, city#29, state#30]
+- *(2) BroadcastHashJoin [PCODE#10], [PCODE#28], Inner, BuildRight
  :- *(2) Project [PCODE#10, lastName#11, firstName#12, age#13]
  :  +- *(2) Filter isnotnull(PCODE#10)
  :     +- *(2) FileScan csv [PCODE#10,lastName#11,firstName#12,age#13]
          Batched: false, Format: CSV, Location:
          InMemoryFileIndex[...people.csv], PartitionFilters: [],
          PushedFilters: [IsNotNull(PCODE)], ReadSchema:
          struct<PCODE:string,lastName:string,firstName:string,age:string>
+- BroadcastExchange HashedRelationBroadcastMode(List(input[0, string, true]))
  +- *(1) Project [PCODE#28, city#29, state#30]
    +- *(1) Filter isnotnull(PCODE#28)
      +- *(1) FileScan csv [PCODE#28,city#29,state#30] Batched: false,
        Format: CSV, Location: InMemoryFileIndex[...pcodes.csv],
        PartitionFilters: [], PushedFilters: [IsNotNull(PCODE)],
        ReadSchema: struct<PCODE:string,city:string,state:string>
```

Language: Python

Example: Catalyst Execution Plan (4)

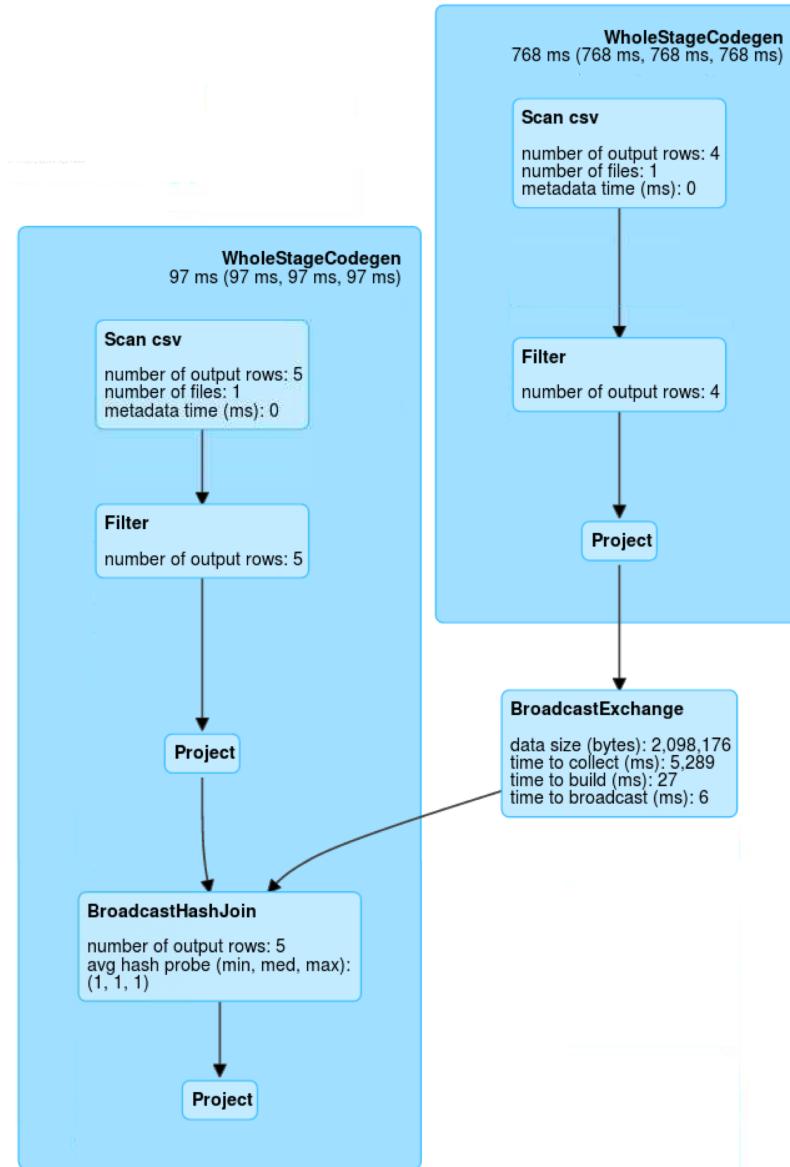
SQL

Completed Queries: 3

Completed Queries (3)

ID	Description		Submitted	Duration	Job IDs
2	collect at <ipython-input-2-31b1b37d0504>:1	+details	2019/05/06 05:25:27	6 s	[2][3]
1	csv at NativeMethodAccessorImpl.java:0	+details	2019/05/06 05:14:47	0.1 s	[1]
0	csv at NativeMethodAccessorImpl.java:0	+details	2019/05/06 05:14:37	7 s	[0]

Example: Catalyst Execution Plan (5)



Chapter Topics

Spark Distributed Processing

- Review: Apache Spark on a Cluster
- RDD Partitions
- Example: Partitioning in Queries
- Stages and Tasks
- Job Execution Planning
- Example: Catalyst Execution Plan
- **Example: RDD Execution Plan**
- Essential Points
- Hands-On Exercise: Exploring Query Execution

Viewing RDD Execution Plans

- You can view RDD (lineage-based) execution plans
 - Use the RDD `toDebugString` function
 - Use **Jobs** and **Stages** tabs in the Spark UI or history server
 - Shows details of execution after job runs
- Note that plans may be different depending on programming language
 - Plan optimization rules vary

Example: RDD Execution Plan (1)

```
val peopleRDD = sc.textFile("people2.csv").keyBy(s => s.split(',')(0))
val pcodesRDD = sc.textFile("pcodes2.csv").keyBy(s => s.split(',')(0))
val joinedRDD = peopleRDD.join(pcodesRDD)
joinedRDD.toDebugString
(2) MapPartitionsRDD[8] at join at ①
|  MapPartitionsRDD[7] at join at <console>27 []
|  CoGroupedRDD[6] at join at <console>27 []
+-(2) MapPartitionsRDD[2] at keyBy at <console>24 [] ②
|  |  people2.csv MapPartitionsRDD[1] at textFile at <console>27 []
|  |  people2.csv HadoopRDD[0] at <console>24 []
+-(2) MapPartitionsRDD[5] at keyBy at at <console>24 [] ③
|  |  pcodes2.csv MapPartitionsRDD[4] at textFile at at <console>24 []
④  |  pcodes2.csv HadoopRDD[3] at textFile at at <console>24 []
```

Language: Scala

- ① Stage 2
- ② Stage 1
- ③ Stage 0
- ④ Indents indicate stages (shuffle boundaries)

Example: RDD Execution Plan (2)

Spark Jobs [\(?\)](#)

User: training

Total Uptime: 38 s

Scheduling Mode: FIFO

Completed Jobs: 1

▶ Event Timeline

▼ Completed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	collect at <console>:26 collect at <console>:26	2019/07/18 01:10:52	2 s	3/3	6/6

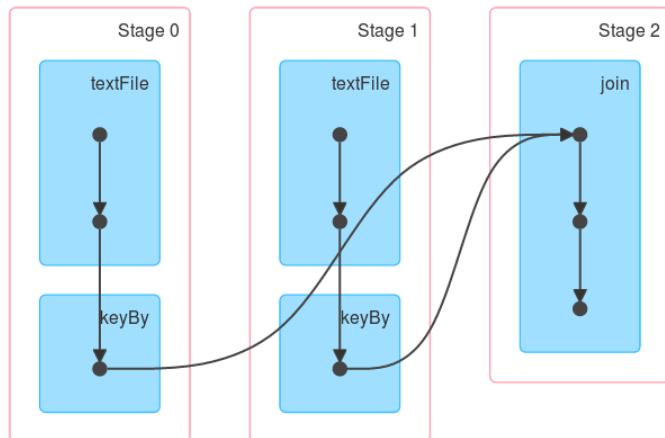
Example: RDD Execution Plan (3)

Details for Job 0

Status: SUCCEEDED

Completed Stages: 3

- ▶ Event Timeline
- ▼ DAG Visualization



▼ Completed Stages (3)

Stage Id ▼	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	collect at <console>:26 +details	2019/05/06 06:13:14	0.1 s	2/2			529.0 B	
1	keyBy at <console>:24 +details	2019/05/06 06:13:08	0.1 s	2/2	105.0 B			220.0 B
0	keyBy at <console>:24 +details	2019/05/06 06:13:08	2 s	2/2	170.0 B			309.0 B

Chapter Topics

Spark Distributed Processing

- Review: Apache Spark on a Cluster
- RDD Partitions
- Example: Partitioning in Queries
- Stages and Tasks
- Job Execution Planning
- Example: Catalyst Execution Plan
- Example: RDD Execution Plan
- **Essential Points**
- Hands-On Exercise: Exploring Query Execution

Essential Points

- **Spark partitions split data across different executors in an application**
- **Executors execute query tasks that process the data in their partitions**
- **Narrow operations like map and filter are pipelined within a single stage**
 - Wide operations like groupByKey and join shuffle and repartition data between stages
- **Jobs consist of a sequence of stages triggered by a single action**
- **Jobs execute according to execution plans**
 - Core Spark creates RDD execution plans based on RDD lineages
 - Catalyst builds optimized query execution plans
- **You can explore how Spark executes queries in the Spark Application UI**

Chapter Topics

Spark Distributed Processing

- Review: Apache Spark on a Cluster
- RDD Partitions
- Example: Partitioning in Queries
- Stages and Tasks
- Job Execution Planning
- Example: Catalyst Execution Plan
- Example: RDD Execution Plan
- Essential Points
- **Hands-On Exercise: Exploring Query Execution**

Hands-On Exercise: Exploring Query Execution

- In this exercise, you will explore how Spark plans and executes RDD and DataFrame/Dataset queries
- Please refer to the Hands-On Exercise Manual for instructions



Distributed Data Persistence

Chapter 15

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence**
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Distributed Data Persistence

After completing this chapter, you will be able to

- Improve performance and fault-tolerance using persistence
- Explain when to use different storage levels
- Use the Spark UI to view details about persisted data

Chapter Topics

Distributed Data Persistence

- **DataFrame and Dataset Persistence**
- Persistence Storage Levels
- Viewing Persisted RDDs
- Essential Points
- Hands-On Exercise: Persisting DataFrames

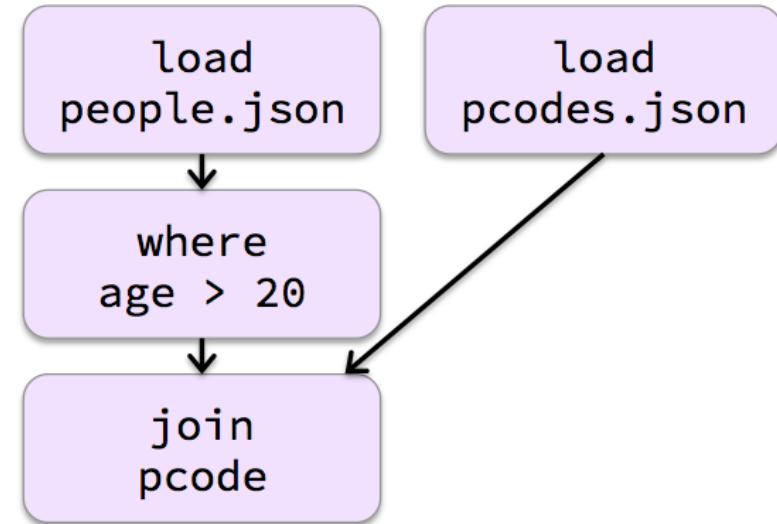
Persistence

- You can *persist* a DataFrame, Dataset, or RDD
 - Also called *caching*
 - Data is temporarily saved to memory and/or disk
- Persistence can improve performance and fault-tolerance
- Use persistence when
 - Query results will be used repeatedly
 - Executing the query again in case of failure would be very expensive
- Persisted data cannot be shared between applications

Example: DataFrame Persistence (1)

```
over20DF = spark.read. \
    json("people.json"). \
    where("age > 20")
pcodesDF = spark.read. \
    json("pcodes.json")
joinedDF = over20DF. \
    join(pcodesDF, "pcode")
```

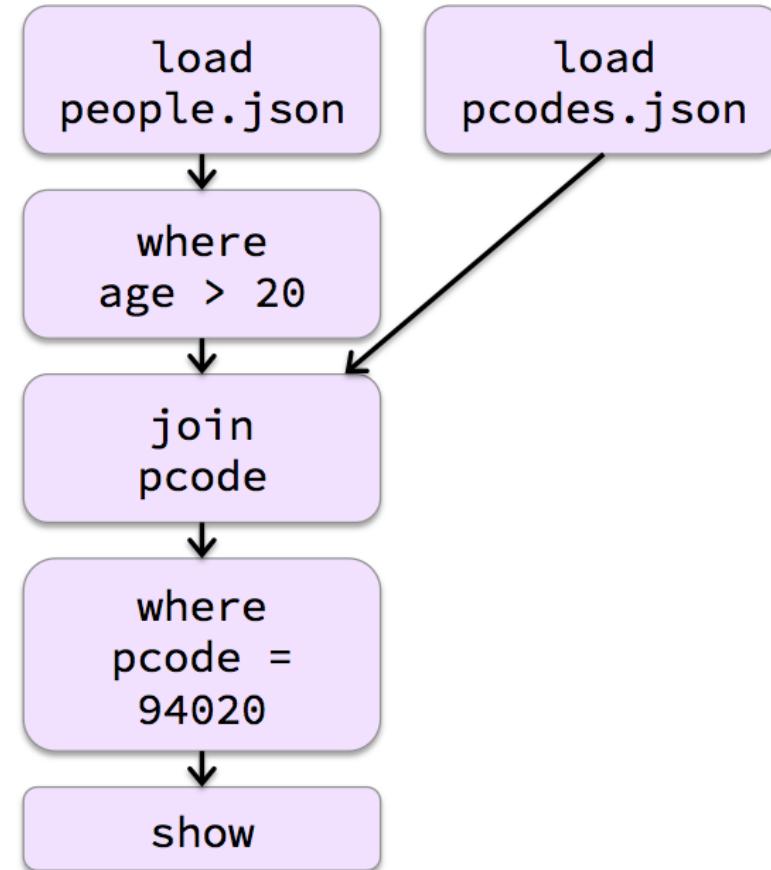
Language: Python



Example: DataFrame Persistence (2)

```
over20DF = spark.read. \
    json("people.json"). \
    where("age > 20")
pcodesDF = spark.read. \
    json("pcodes.json")
joinedDF = over20DF. \
    join(pcodesDF, "PCODE")
joinedDF. \
    where("PCODE = 94020"). \
    show()
```

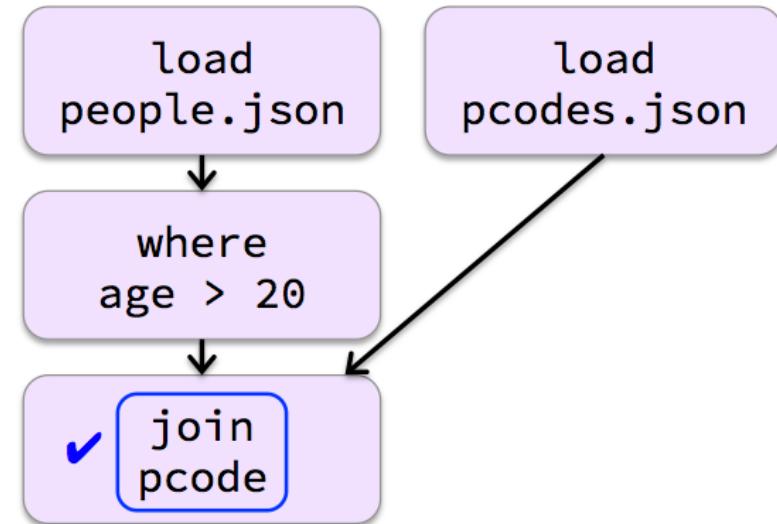
Language: Python



Example: DataFrame Persistence (3)

```
over20DF = spark.read. \
    json("people.json"). \
    where("age > 20")
pcodesDF = spark.read. \
    json("pcodes.json")
joinedDF = over20DF. \
    join(pcodesDF, "PCODE"). \
    persist()
```

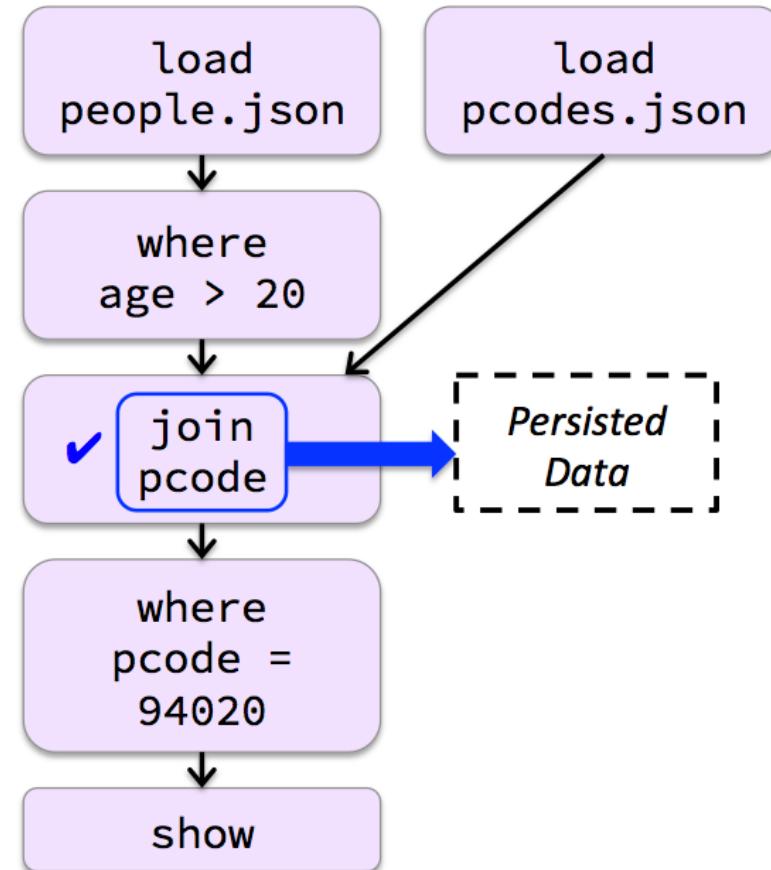
Language: Python



Example: DataFrame Persistence (4)

```
over20DF = spark.read. \
    json("people.json"). \
    where("age > 20")
pcodesDF = spark.read. \
    json("pcodes.json")
joinedDF = over20DF. \
    join(pcodesDF, "PCODE"). \
    persist()
joinedDF. \
    where("PCODE = 94020"). \
    show()
```

Language: Python



Example: DataFrame Persistence (5)

```
over20DF = spark.read. \
    json("people.json"). \
    where("age > 20")
pcodesDF = spark.read. \
    json("pcodes.json")
joinedDF = over20DF. \
    join(pcodesDF, "PCODE"). \
    persist()
joinedDF. \
    where("PCODE = 94020"). \
    show()
joinedDF. \
    where("PCODE = 87501"). \
    show()
```

Language: Python

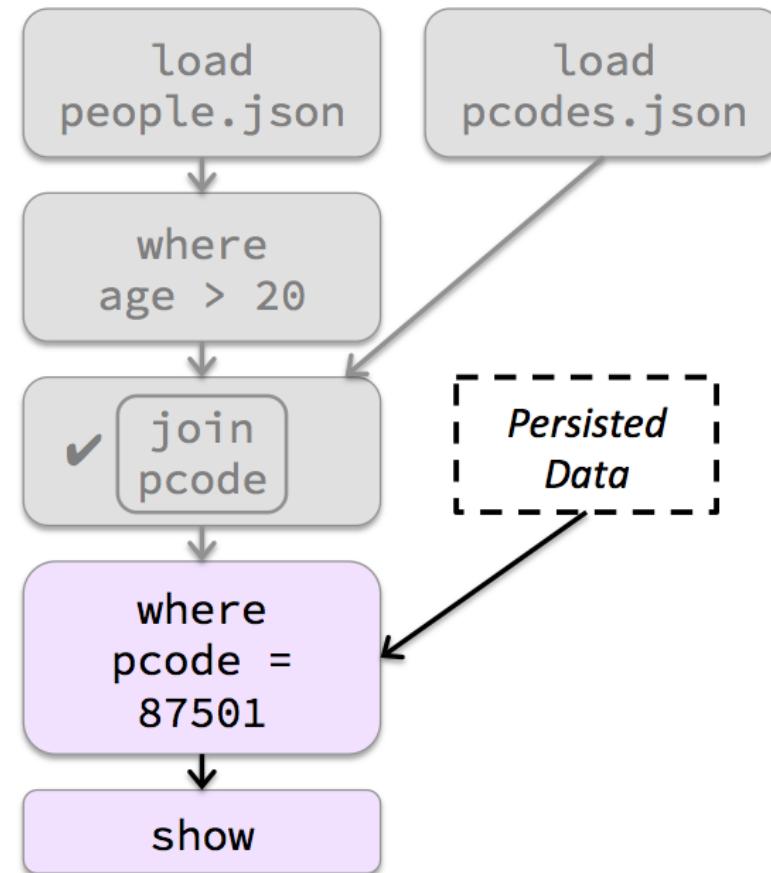


Table and View Persistence

- Tables and views can be persisted in memory using `CACHE TABLE`

```
spark.sql("CACHE TABLE people")
```

- `CACHE TABLE` can create a view based on a SQL query and cache it at the same time

```
spark.sql("CACHE TABLE over_20 AS SELECT *  
        FROM people WHERE age > 20")
```

- Queries on cached tables work the same as on persisted DataFrames, Datasets, and RDDs
 - The first query caches the data
 - Subsequent queries use the cached data

Chapter Topics

Distributed Data Persistence

- DataFrame and Dataset Persistence
- **Persistence Storage Levels**
- Viewing Persisted RDDs
- Essential Points
- Hands-On Exercise: Persisting DataFrames

Storage Levels

- ***Storage levels provide several options to manage how data is persisted***
 - Storage location (memory and/or disk)
 - Serialization of data in memory
 - Replication
- **Specify storage level when persisting a DataFrame, Dataset, or RDD**
 - Tables and views do not use storage levels
 - Always persisted in memory
- **Data is persisted based on partitions of the underlying RDDs**
 - Executors persist partitions in JVM memory or temporary local files
 - The application driver keeps track of the location of each persisted partition's data

Storage Levels: Location

- Storage location—where is the data stored?
 - MEMORY_ONLY: Store data in memory if it fits
 - DISK_ONLY: Store all partitions on disk
 - MEMORY_AND_DISK: Store any partition that does not fit in memory on disk
 - Called *spilling*

```
from pyspark import StorageLevel  
myDF.persist(StorageLevel.DISK_ONLY)
```

Language: Python

```
import org.apache.spark.storage.StorageLevel  
myDF.persist(StorageLevel.DISK_ONLY)
```

Language: Scala

Storage Levels: Memory Serialization

- In Python, data in memory is always serialized
- In Scala, you can choose to serialize data in memory
 - By default, in Scala and Java, data in memory is stored objects
 - Use MEMORY_ONLY_SER and MEMORY_AND_DISK_SER to serialize the objects into a sequence of bytes instead
 - Much more space efficient but less time efficient
- Datasets are serialized by Spark SQL encoders, which are very efficient
 - Plain RDDs use native Java/Scala serialization by default
 - Use Kryo instead for better performance
- Serialization options do not apply to disk persistence
 - Files are always in serialized form by definition

Storage Levels: Partition Replication

- **Replication—store partitions on two nodes**
 - DISK_ONLY_2
 - MEMORY_AND_DISK_2
 - MEMORY_ONLY_2
 - MEMORY_AND_DISK_SER_2 (Scala and Java only)
 - MEMORY_ONLY_SER_2 (Scala and Java only)
 - You can also define custom storage levels for additional replication

Default Storage Levels

- The **storageLevel** parameter for the DataFrame, Dataset, or RDD **persist** operation is optional
 - The default for DataFrames and Datasets is **MEMORY_AND_DISK**
 - The default for RDDs is **MEMORY_ONLY**
- **persist** with no storage level specified is a synonym for **cache**

```
myDF.persist()
```

is equivalent to

```
myDF.cache()
```

- Table and view storage level is always **MEMORY_ONLY**

When and Where to Persist

- **When should you persist a DataFrame, Dataset, or RDD?**
 - When the data is likely to be reused
 - Such as in iterative algorithms and machine learning
 - When it would be very expensive to recreate the data if a job or node fails
- **How to choose a storage level**
 - **Memory**—use when possible for best performance
 - Save space by serializing the data if necessary
 - **Disk**—use when re-executing the query is more expensive than disk read
 - Such as expensive functions or filtering large datasets
 - **Replication**—use when re-execution is more expensive than bandwidth

Changing Storage Levels

- You can remove persisted data from memory and disk
 - Use `unpersist` for Datasets, DataFrames, and RDDs
 - Use `Catalog.uncacheTable(table-name)` for tables and views
 - Call with no parameter to uncache all tables and views
- Unpersist before changing to a different storage level
 - Re-persisting already-persisted data results in an exception

```
myDF.unpersist()  
myDF.persist(new-level)
```

Chapter Topics

Distributed Data Persistence

- DataFrame and Dataset Persistence
- Persistence Storage Levels
- **Viewing Persisted RDDs**
- Essential Points
- Hands-On Exercise: Persisting DataFrames

Viewing Persisted RDDs (1)

- The Storage tab in the Spark UI shows persisted RDDs

Jobs	Stages	Storage	Environment	Executors	SQL
Storage					
RDDs					
RDD Name	DataFrame	Storage Level	Cached Partitions	Fraction Cached	Size in Memory
*Project [acct_num#0] +- *Filter isnotnull(acct_close_dt#2) +- HiveTableScan [acct_num#0, acct_close_dt#2], MetastoreRelation default, accounts		Memory Deserialized 1x Replicated	5	100%	43.4 KB 0.0 B
ShuffledRDD RDD		Disk Serialized 1x Replicated	5	100%	0.0 B 23.4 MB

Viewing Persisted RDDs (2)

RDD Storage Info for ShuffledRDD

Storage Level: Memory Deserialized 1x Replicated

Cached Partitions: 5

Total Partitions: 5

Memory Size: 42.0 MB

Disk Size: 0.0 B

Data Distribution on 3 Executors

Host	Memory Usage	Disk Usage
worker-1:57827	8.4 MB (357.8 MB Remaining)	0.0 B
10.0.8.135:36865	0.0 B (366.2 MB Remaining)	0.0 B
worker-2:58504	33.6 MB (332.6 MB Remaining)	0.0 B

5 Partitions

Block Name ▲	Storage Level	Size in Memory	Size on Disk	Executors
rdd_8_0	Memory Deserialized 1x Replicated	8.4 MB	0.0 B	worker-1:57827
rdd_8_1	Memory Deserialized 1x Replicated	8.3 MB	0.0 B	worker-2:58504
rdd_8_2	Memory Deserialized 1x Replicated	8.4 MB	0.0 B	worker-2:58504
rdd_8_3	Memory Deserialized 1x Replicated	8.4 MB	0.0 B	worker-2:58504
rdd_8_4	Memory Deserialized 1x Replicated	8.5 MB	0.0 B	worker-2:58504

Chapter Topics

Distributed Data Persistence

- DataFrame and Dataset Persistence
- Persistence Storage Levels
- Viewing Persisted RDDs
- **Essential Points**
- Hands-On Exercise: Persisting DataFrames

Essential Points

- Persisting data means temporarily storing data in Datasets, DataFrames, RDDs, tables, and views to improve performance and resilience
- Persisted data is stored in executor memory and/or disk files on worker nodes
- Replication can improve performance when recreating partitions after executor failure
- Replication is most useful in iterative applications or when executing a complicated query is very expensive

Chapter Topics

Distributed Data Persistence

- DataFrame and Dataset Persistence
- Persistence Storage Levels
- Viewing Persisted RDDs
- Essential Points
- **Hands-On Exercise: Persisting DataFrames**

Hands-On Exercise: Persisting DataFrames

- In this exercise, you will explore DataFrame persistence
- Please refer to the Hands-On Exercise Manual for instructions



Common Patterns in Spark Data Processing

Chapter 16

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- **Common Patterns in Spark Data Processing**
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Distributed Data Persistence

After completing this chapter, you will be able to

- Summarize what kinds of processing and analysis Apache Spark is best suited for
- Implement an iterative algorithm in Spark
- List major features and benefits that are provided by Spark's machine learning libraries

Chapter Topics

Common Patterns in Spark Data Processing

- **Common Apache Spark Use Cases**
- Iterative Algorithms in Apache Spark
- Machine Learning
- Example: k-means
- Essential Points
- Hands-On Exercise: Implementing an Iterative Algorithm

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)

- **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
 - Calculating pi
 - Alternating least squares (ALS)
 - Querying Apache web logs
 - Processing Twitter feeds
- Examples
 - *SPARK_HOME/lib*
 - *spark-examples.jar*: Java and Scala examples
 - *python.tar.gz*: Pyspark examples

Chapter Topics

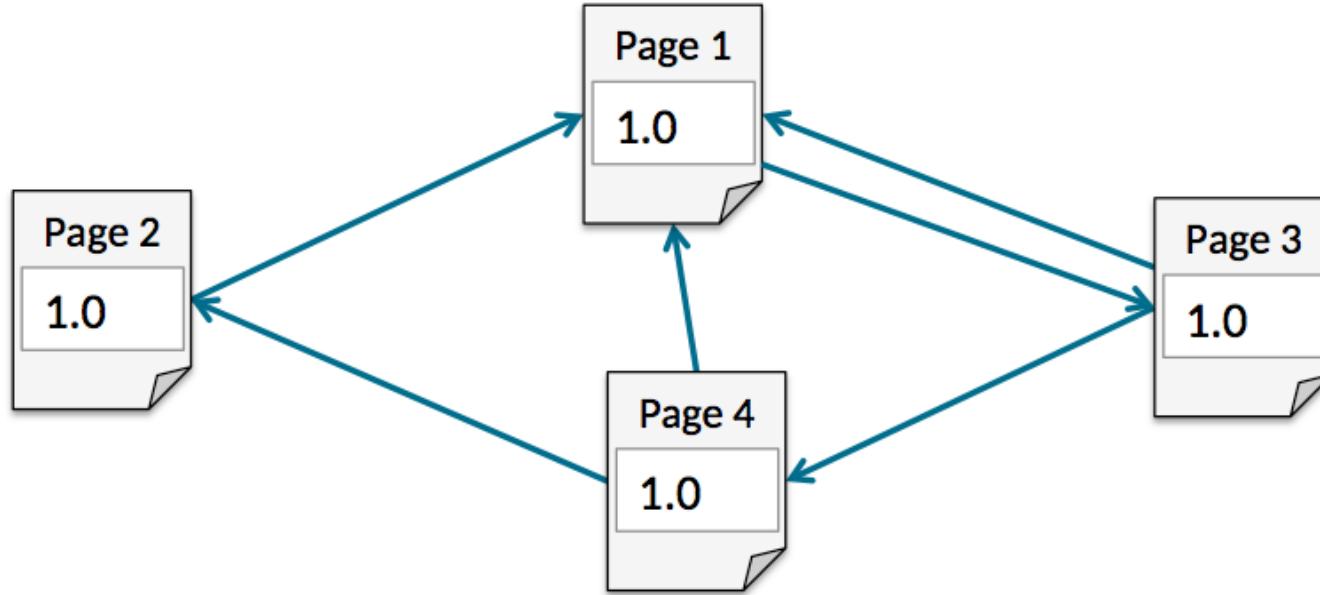
Common Patterns in Spark Data Processing

- Common Apache Spark Use Cases
- **Iterative Algorithms in Apache Spark**
- Machine Learning
- Example: k-means
- Essential Points
- Hands-On Exercise: Implementing an Iterative Algorithm

Example: PageRank

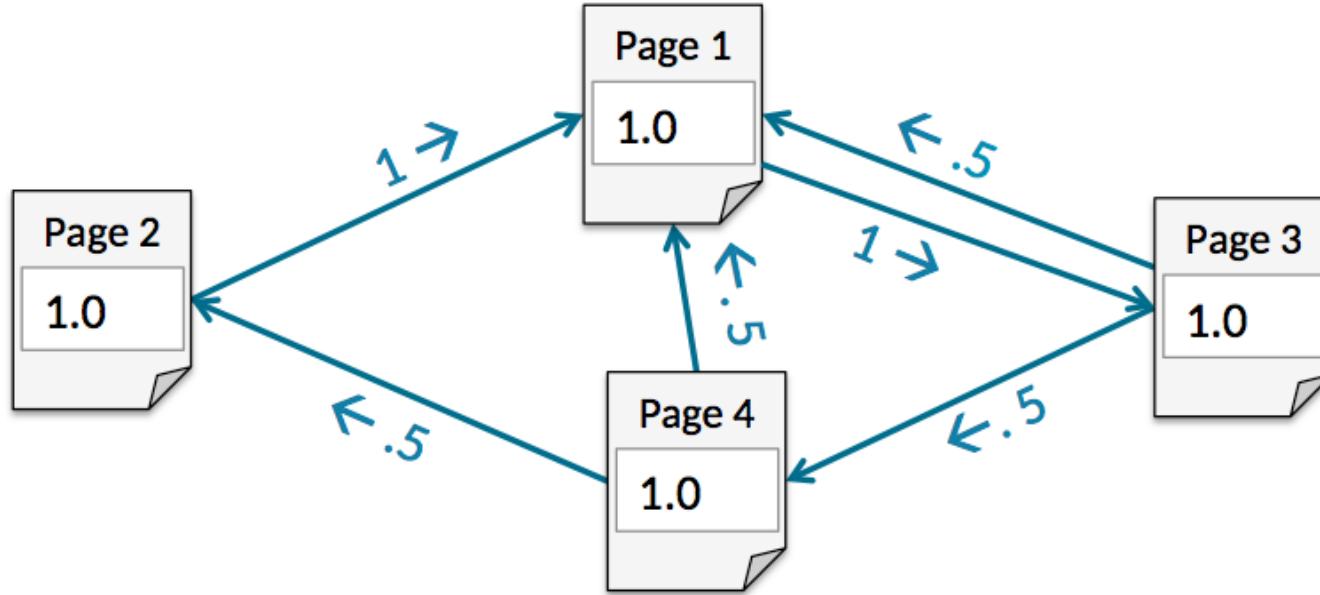
- **PageRank gives web pages a ranking score based on links from other pages**
 - Higher scores given for more links, and links from other high ranking pages
- **PageRank is a classic example of big data analysis (like word count)**
 - Lots of data: Needs an algorithm that is distributable and scalable
 - Iterative: The more iterations, the better than answer

PageRank Algorithm (1)



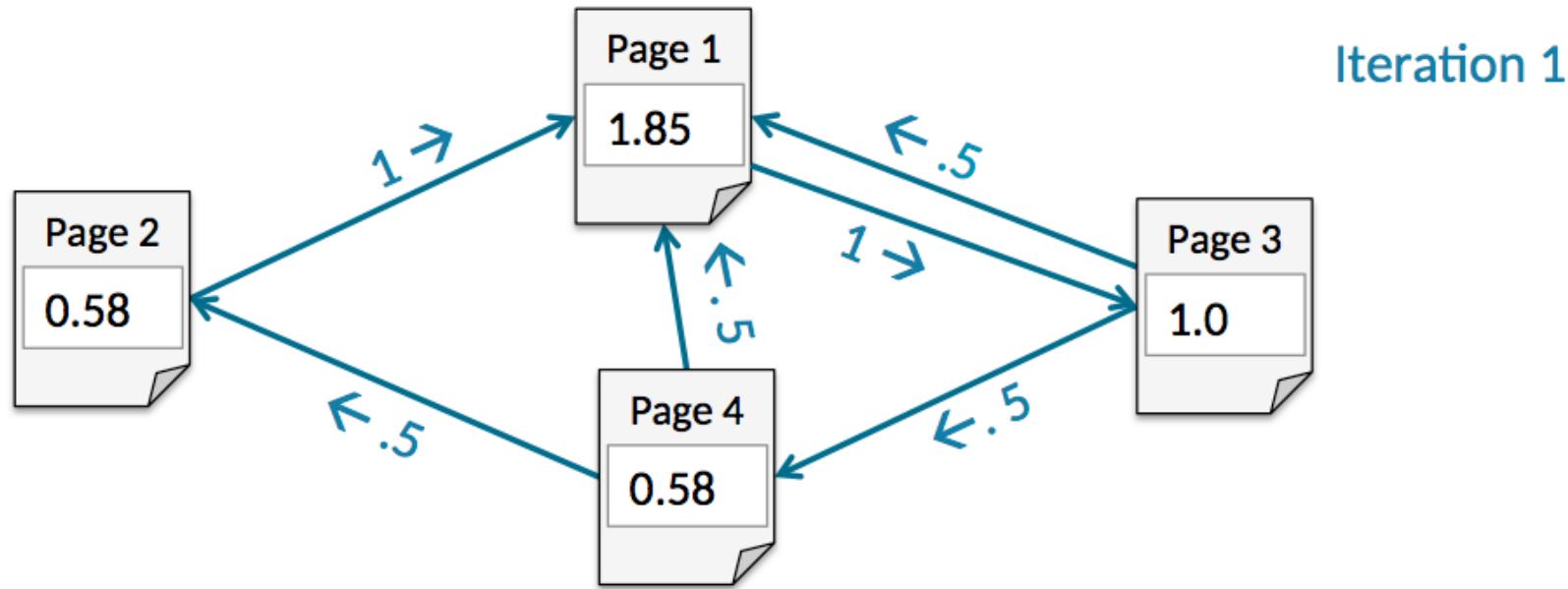
1. Start each page with a rank of 1.0

PageRank Algorithm (2)



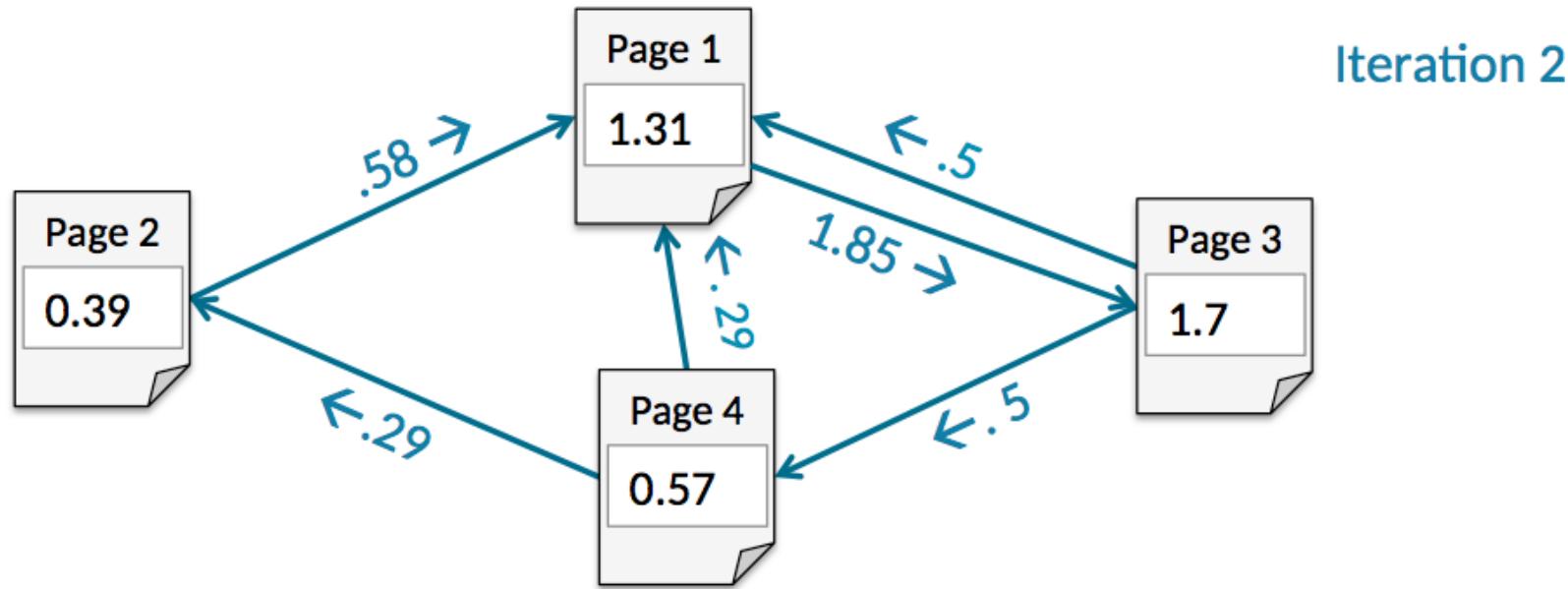
1. Start each page with a rank of 1.0
2. On each iteration:
 - a. Each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$

PageRank Algorithm (3)



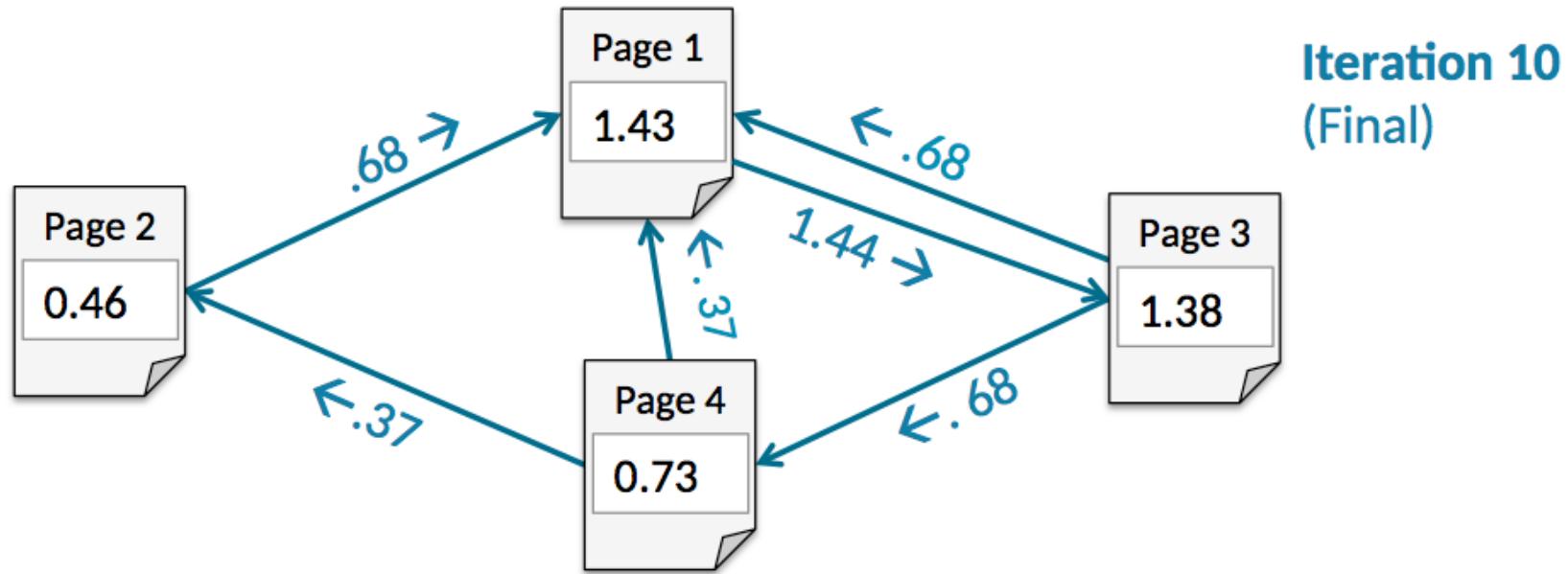
1. Start each page with a rank of 1.0
2. On each iteration:
 - a. Each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$
 - b. Set each page's new rank based on the sum of its neighbors contribution: $\text{new_rank} = \sum \text{contrib} * .85 + .15$

PageRank Algorithm (4)



1. Start each page with a rank of 1.0
2. On each iteration:
 - a. Each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$
 - b. Set each page's new rank based on the sum of its neighbors contribution: $\text{new_rank} = \sum \text{contrib} * .85 + .15$
3. Each iteration incrementally improves the page ranking

PageRank Algorithm (5)

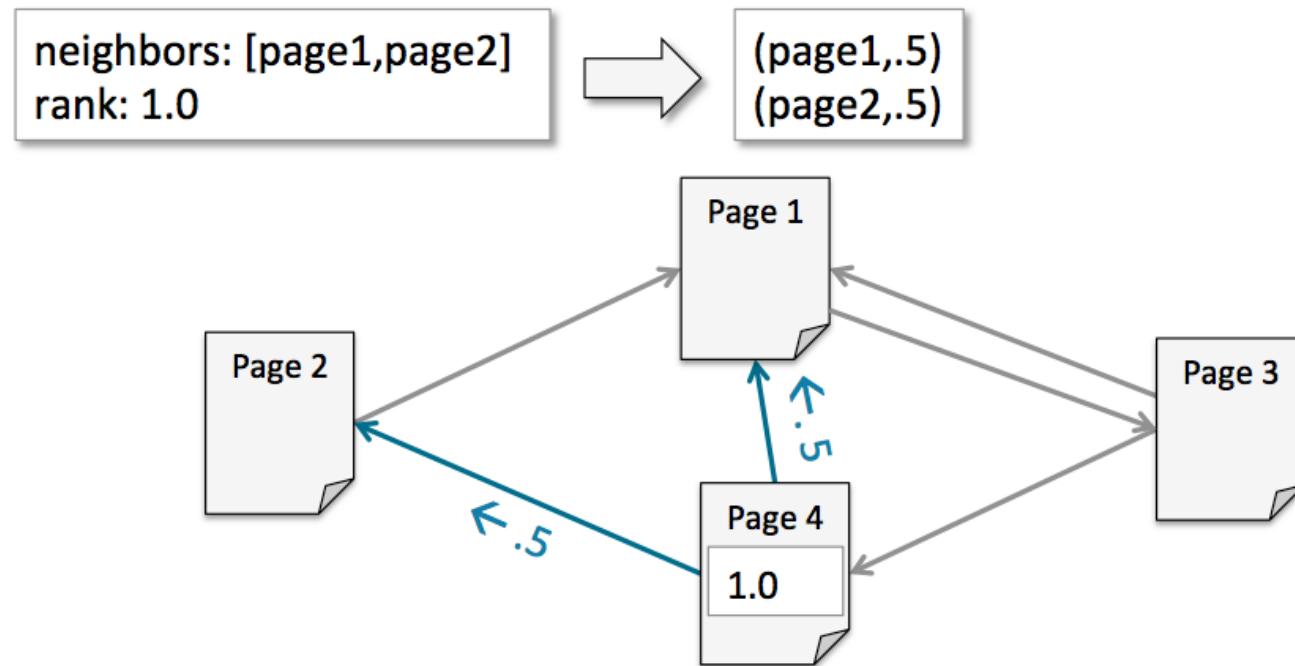


1. Start each page with a rank of 1.0
2. On each iteration:
 - a. Each page contributes to its neighbors its own rank divided by the number of its neighbors: $\text{contrib}_p = \text{rank}_p / \text{neighbors}_p$
 - b. Set each page's new rank based on the sum of its neighbors contribution: $\text{new_rank} = \sum \text{contrib} * .85 + .15$
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))
```

Language: Python



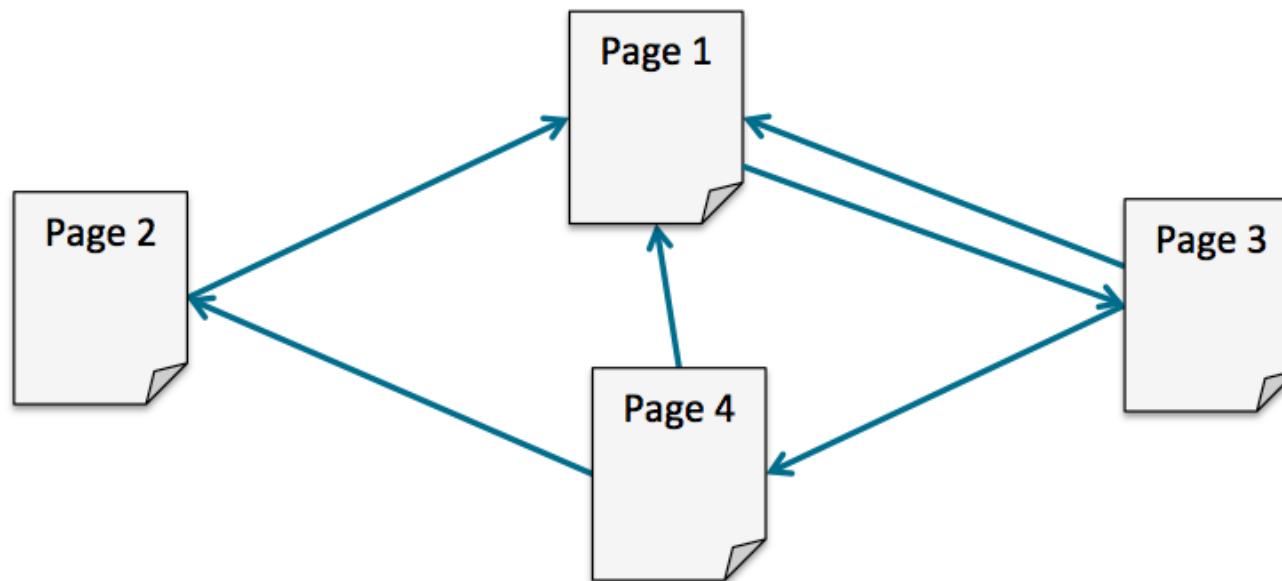
PageRank in Spark: Example Data

Data Format:

source-page destination-page

...

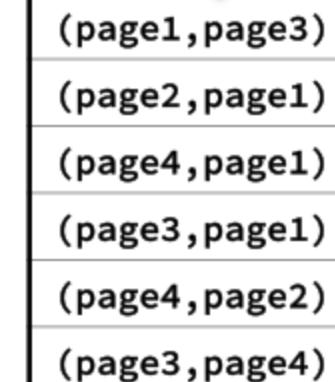
```
page1 page3  
page2 page1  
page4 page1  
page3 page1  
page4 page2  
page3 page4
```



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



The diagram illustrates the execution of the provided Python code. On the left, a light blue box contains the code. On the right, a vertical stack of boxes shows the data flow. The top box, labeled 'Input Data' in orange, contains the list of page links: 'page1 page3', 'page2 page1', 'page4 page1', 'page3 page1', 'page4 page2', and 'page3 page4'. A blue arrow points down to the second box, labeled '(page1,page3)' in black. This is followed by five more boxes, each containing a pair of page links: '(page2,page1)', '(page4,page1)', '(page3,page1)', '(page4,page2)', and '(page3,page4)'. This visualizes how the map operation splits each line into individual pages and then pairs them together.

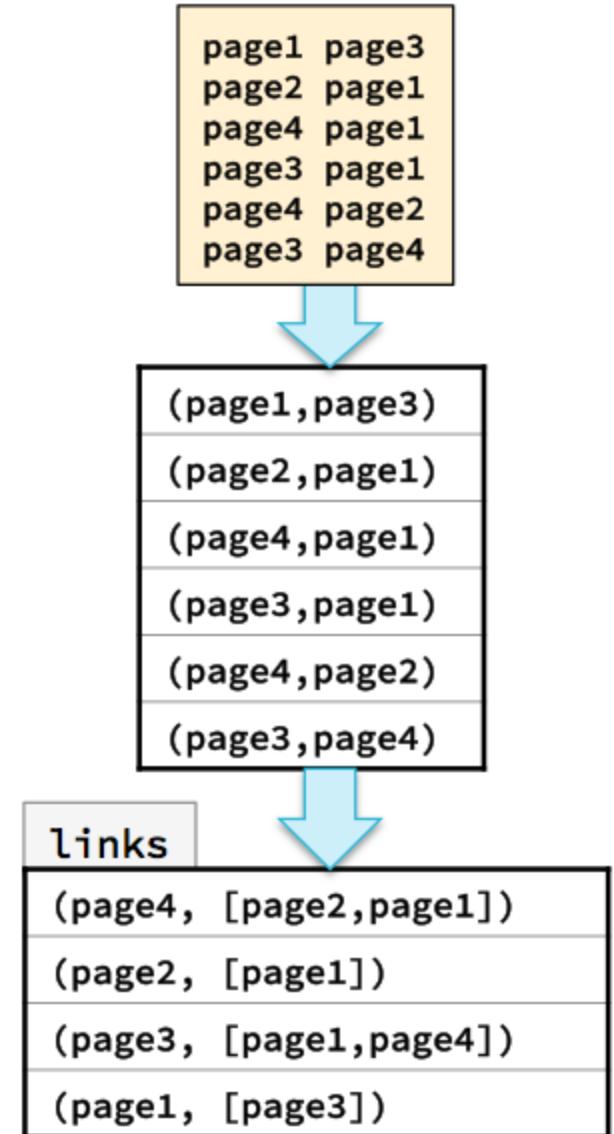
(page1,page3)
(page2,page1)
(page4,page1)
(page3,page1)
(page4,page2)
(page3,page4)

Language: Python

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()
```

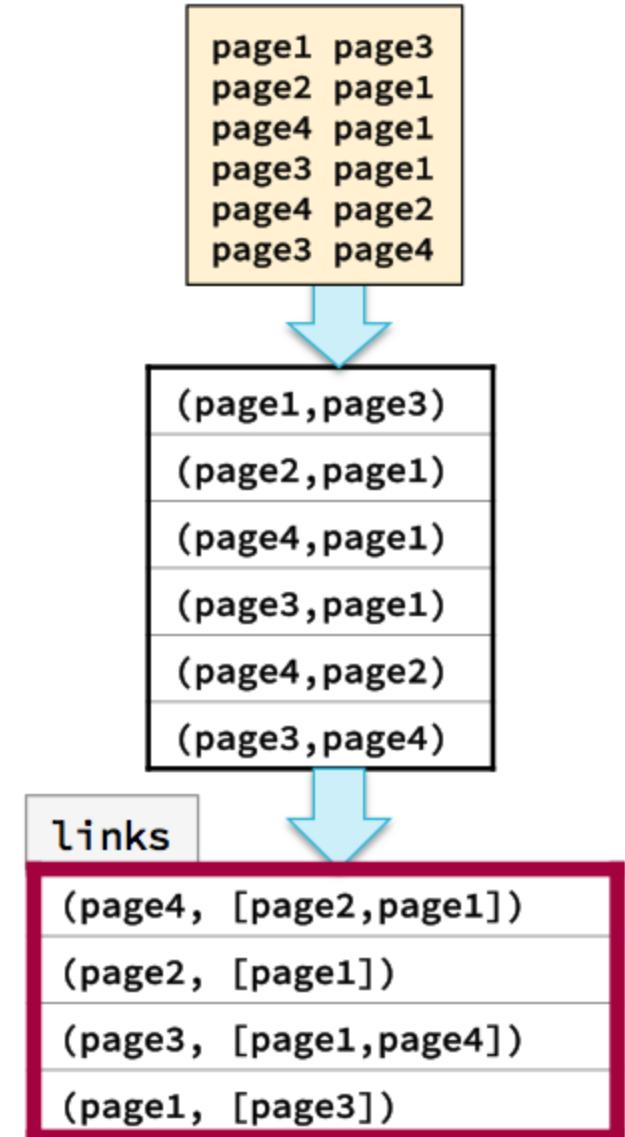
Language: Python



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()
```

Language: Python



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))
```

links

(page4, [page2,page1])
(page2, [page1])
(page3, [page1,page4])
(page1, [page3])

ranks

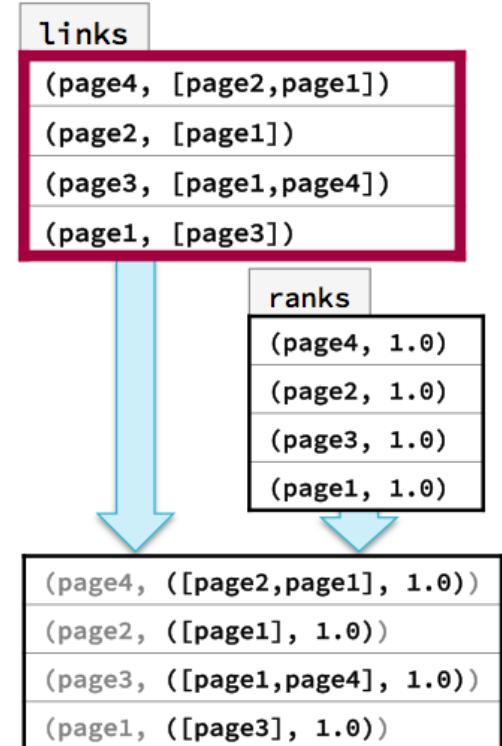
(page4, 1.0)
(page2, 1.0)
(page3, 1.0)
(page1, 1.0)

Language: Python

PageRank in Spark: First Iteration (1)

```
def computeContribs(neighbors, rank):...  
links = ...  
ranks = ...  
  
for x in xrange(10):  
    contribs=links. \  
        join(ranks)
```

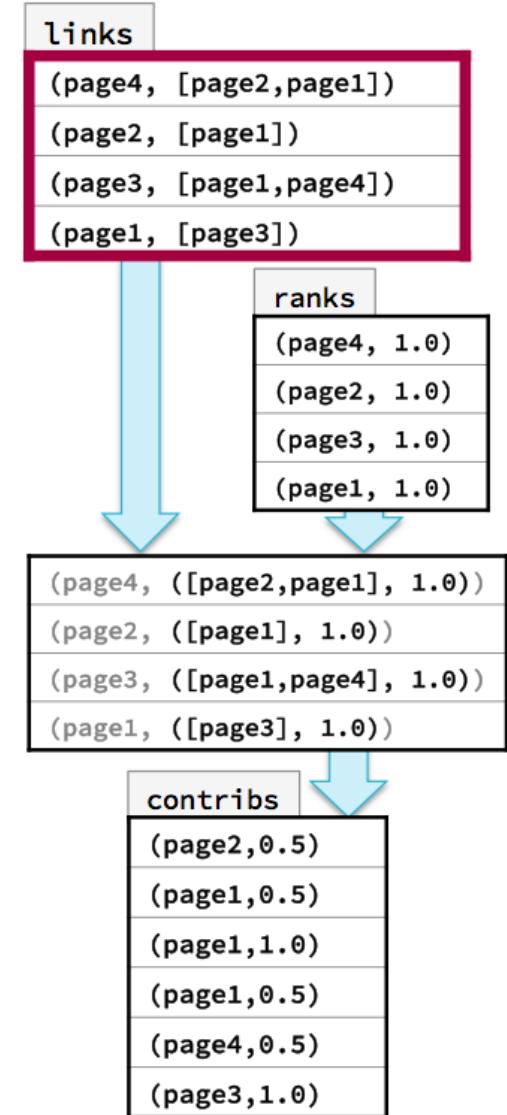
Language: Python



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))
```

Language: Python



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)

Language: Python

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))
```

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)

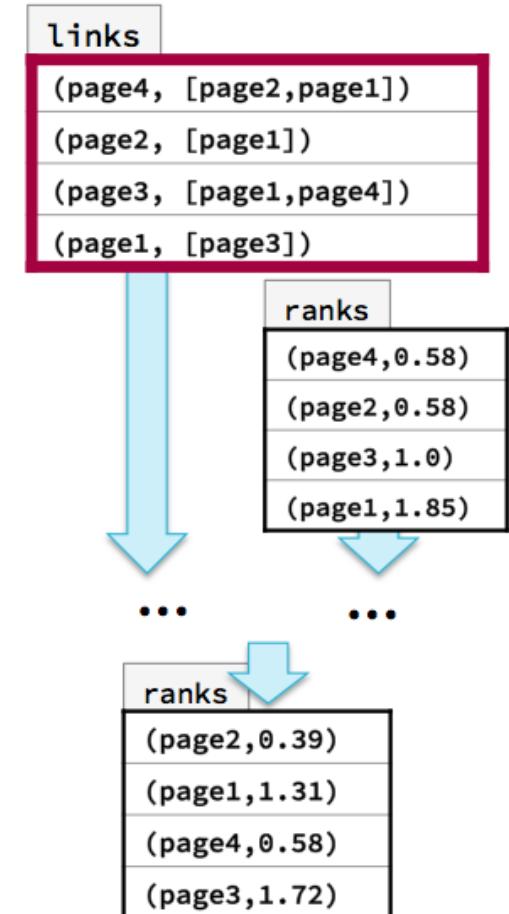
ranks
(page4,.58)
(page2,.58)
(page3,1.0)
(page1,1.85)

Language: Python

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)
```

Language: Python

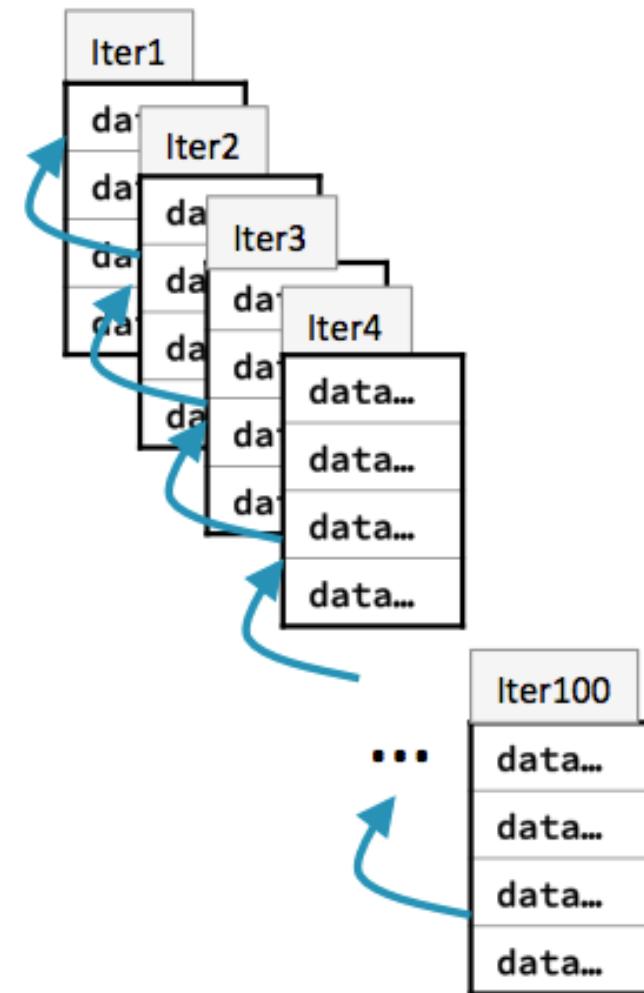


Checkpointing (1)

- In some cases, RDD lineages can get very long
 - For example: iterative algorithms, streaming
- Long lineages can cause problems
 - Recovery might be very expensive
 - Potential stack overflow

```
myrdd = ...initial-value...
for i in xrange(100):
    myrdd = myrdd.transform(...)
myrdd.saveAsTextFile(dir)
```

Language: Python

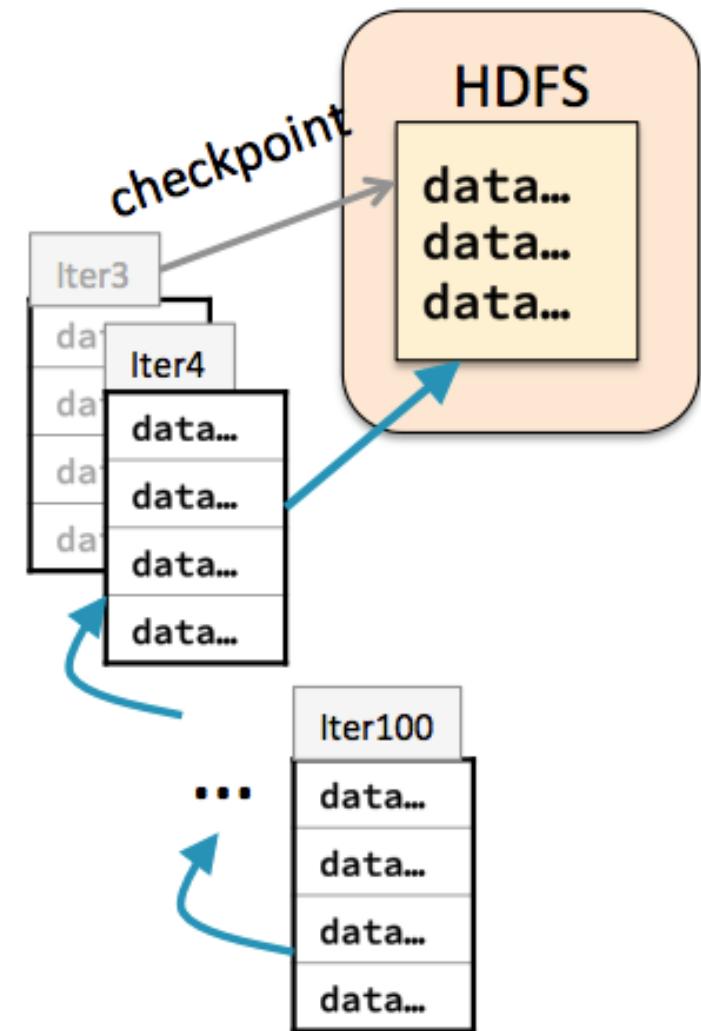


Checkpointing (2)

- 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(dir)
myrdd = ...initial-value...
for x in xrange(100):
    myrdd = myrdd.transform(...)
    if x % 3 == 0:
        myrdd.checkpoint()
        myrdd.count()
myrdd.saveAsTextFile(dir)
```

Language: Python



Chapter Topics

Common Patterns in Spark Data Processing

- Common Apache Spark Use Cases
- Iterative Algorithms in Apache Spark
- **Machine Learning**
- Example: k-means
- Essential Points
- Hands-On Exercise: Implementing an Iterative Algorithm

Fundamentals of Computer Programming

- First consider how a typical program works
 - Hardcoded conditional logic
 - Predefined reactions when those conditions are met

```
#!/usr/bin/env python

import sys
for line in sys.stdin:
    if "Make MONEY Fa$t At Home!!!" in line:
        print("This message is likely spam")
    if "Happy Birthday from Aunt Betty" in line:
        print("This message is probably OK")
```

- The programmer must consider all possibilities at design time
- An alternative technique is to have computers *learn* what to do

What is Machine Learning?

- **Machine learning is a field within artificial intelligence (AI)**
 - AI: “The science and engineering of making intelligent machines”
- **Machine learning focuses on automated knowledge acquisition**
 - Primarily through the design and implementation of algorithms
 - These algorithms require empirical data as input
- **Machine learning algorithms “learn” from data and often produce a predictive model as their output**
 - Model can then be used to make predictions as new data arrives
- **For example, consider a predictive model based on credit card customers**
 - Build model with data about customers who did/did not default on debt
 - Model can then be used to predict whether new customers will default

Types of Machine Learning

- **Three established categories of machine learning techniques:**
 - Collaborative filtering (recommendations)
 - Clustering
 - Classification

What is Collaborative Filtering?

- **Collaborative filtering is a technique for making recommendations**
- **Helps users find items of relevance**
 - Among a potentially vast number of choices
 - Based on comparison of preferences between users
 - Preferences can be either *explicit* (stated) or *implicit* (observed)

Applications Involving Collaborative Filtering

- **Collaborative filtering is domain agnostic**
- **Can use the same algorithm to recommend practically anything**
 - Movies (Netflix, Amazon Instant Video)
 - Television (TiVO Suggestions)
 - Music (several popular music download and streaming services)
- **Amazon uses CF to recommend a variety of products**
 - "Customers who bought items in your recent history also bought ..."

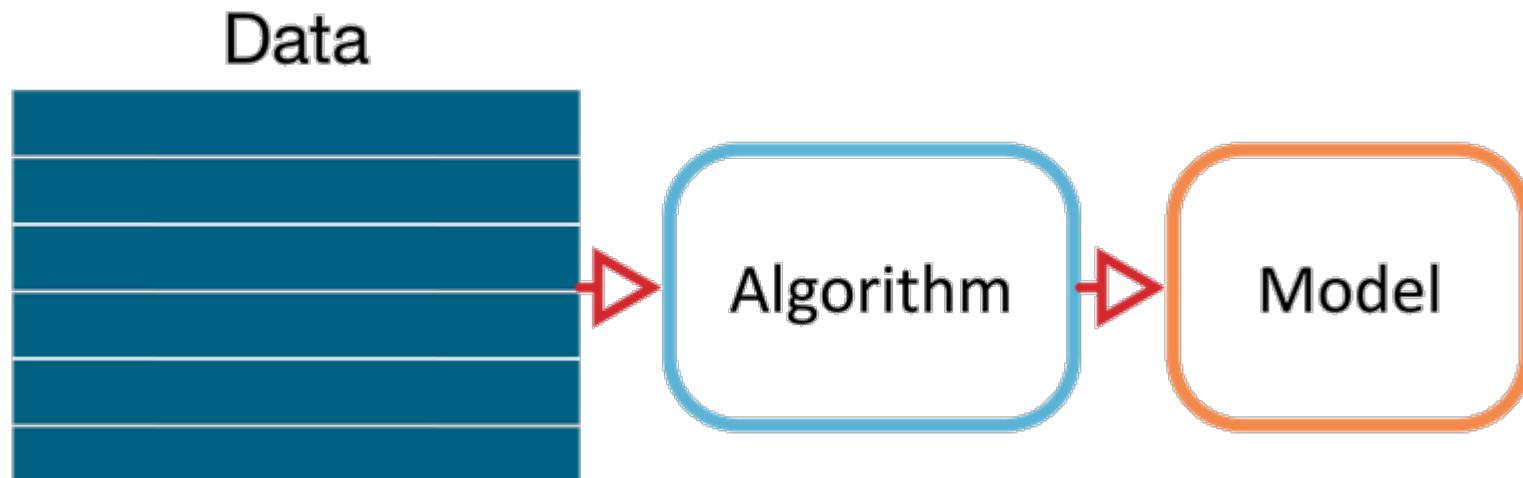
What is Clustering?

- **Clustering algorithms discover structure in collections of data**
 - Where no formal structure previously existed
- **They discover which clusters (“groupings”) naturally occur in data**
 - By examining various properties of the input data
- **Clustering is often used for exploratory analysis**
 - Divide huge amount of data into smaller groups
 - Can then tune analysis for each group



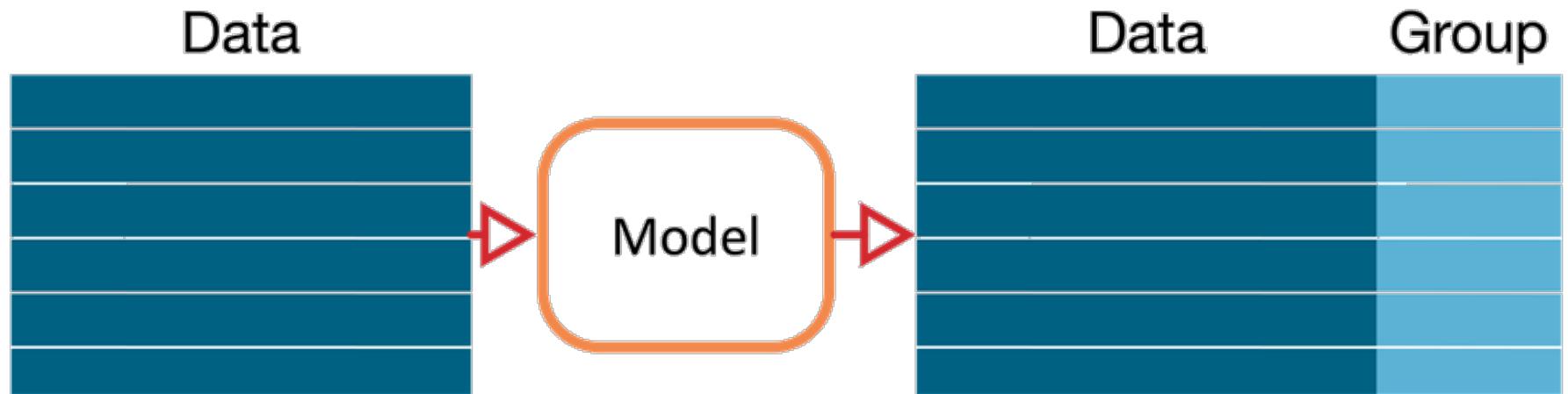
Unsupervised Learning (1)

- Clustering is an example of *unsupervised learning*
 - Begin with a data set that has no apparent label
 - Use an algorithm to discover structure in the data



Unsupervised Learning (2)

- Once the model has been created, you can use it to assign groups

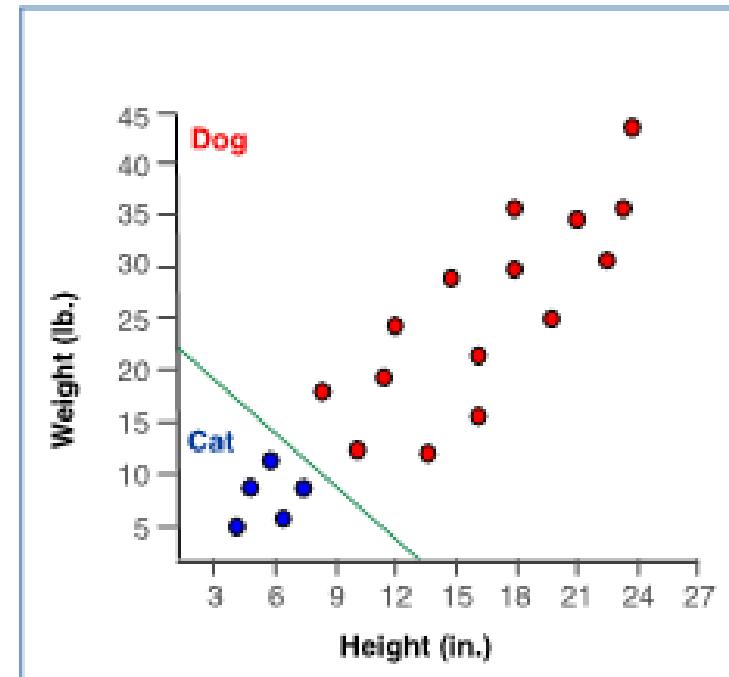


Applications Involving Clustering

- **Market segmentation**
 - Group similar customers in order to target them effectively
- **Finding related news articles**
 - Google News
- **Epidemiological studies**
 - Identifying a “cancer cluster” and finding a root cause
- **Computer vision (groups of pixels that cohere into objects)**
 - Related pixels clustered to recognize faces or license plates

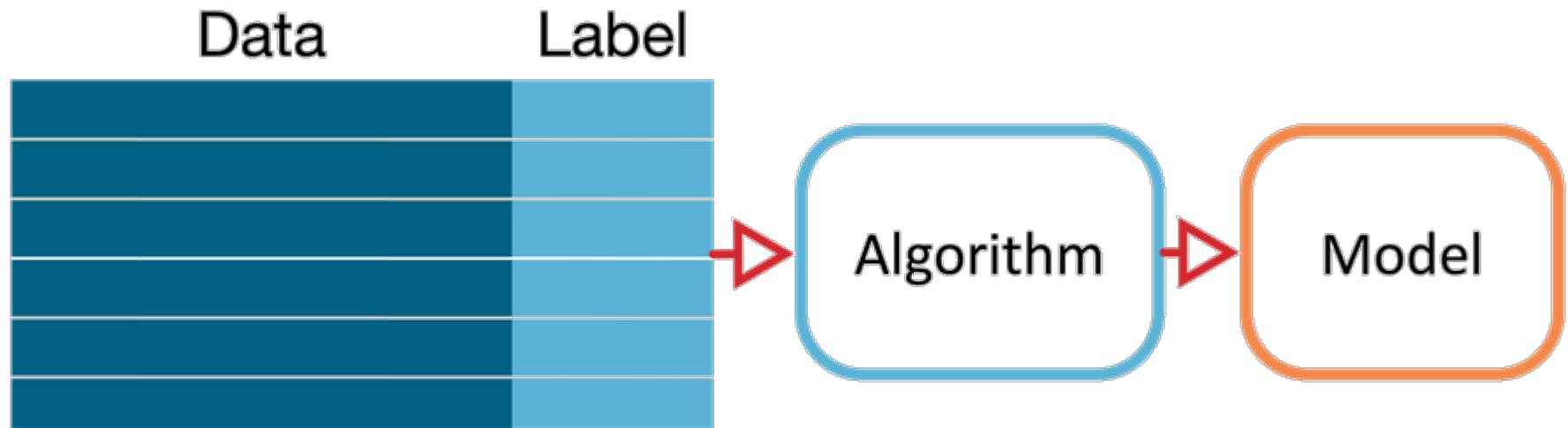
What is Classification?

- Classification is a form of *supervised learning*
 - This requires training with data that has known labels
 - A classifier can then label new data based on what it learned in training
- This example depicts how a classifier might identify animals
 - In this case, it learned to distinguish between these two classes of animals based on height and weight



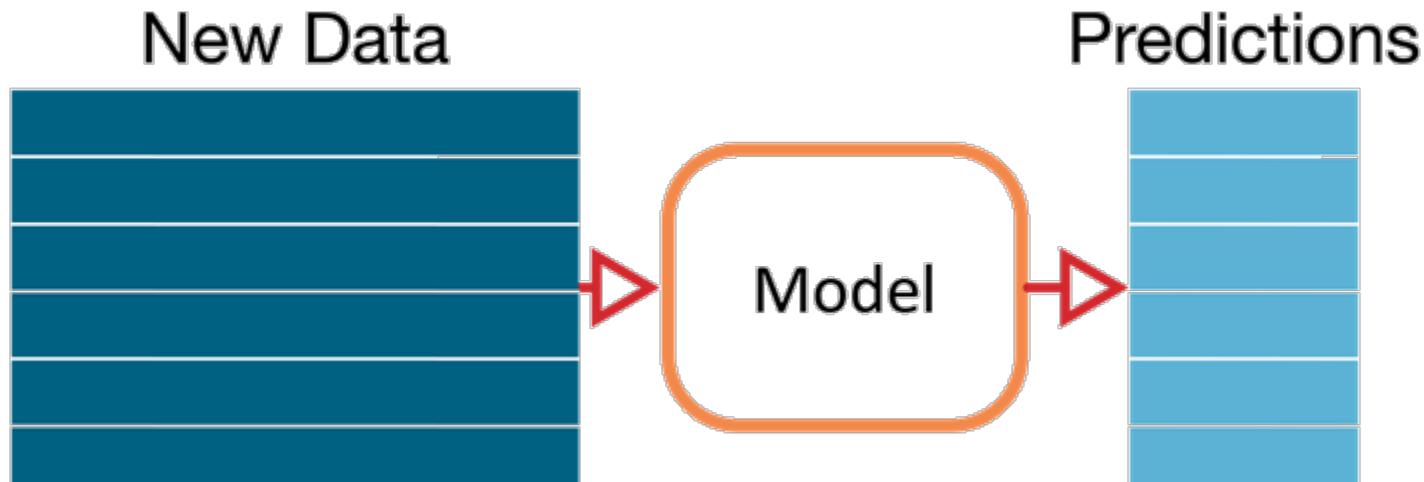
Supervised Learning (1)

- Classification is an example of supervised learning
 - Begin with a data set that includes the value to be predicted (the label)
 - Use an algorithm to train a predictive model using the data-label pairs



Supervised Learning (2)

- Once the model has been trained, you can make predictions
 - This will take new (previously unseen) data as input
 - The new data will not have labels



Applications Involving Classification

- **Spam filtering**

- Train using a set of spam and non-spam messages
 - System will eventually learn to detect unwanted email

- **Oncology**

- Train using images of benign and malignant tumors
 - System will eventually learn to identify cancer

- **Risk Analysis**

- Train using financial records of customers who do/don't default
 - System will eventually learn to identify risk customers

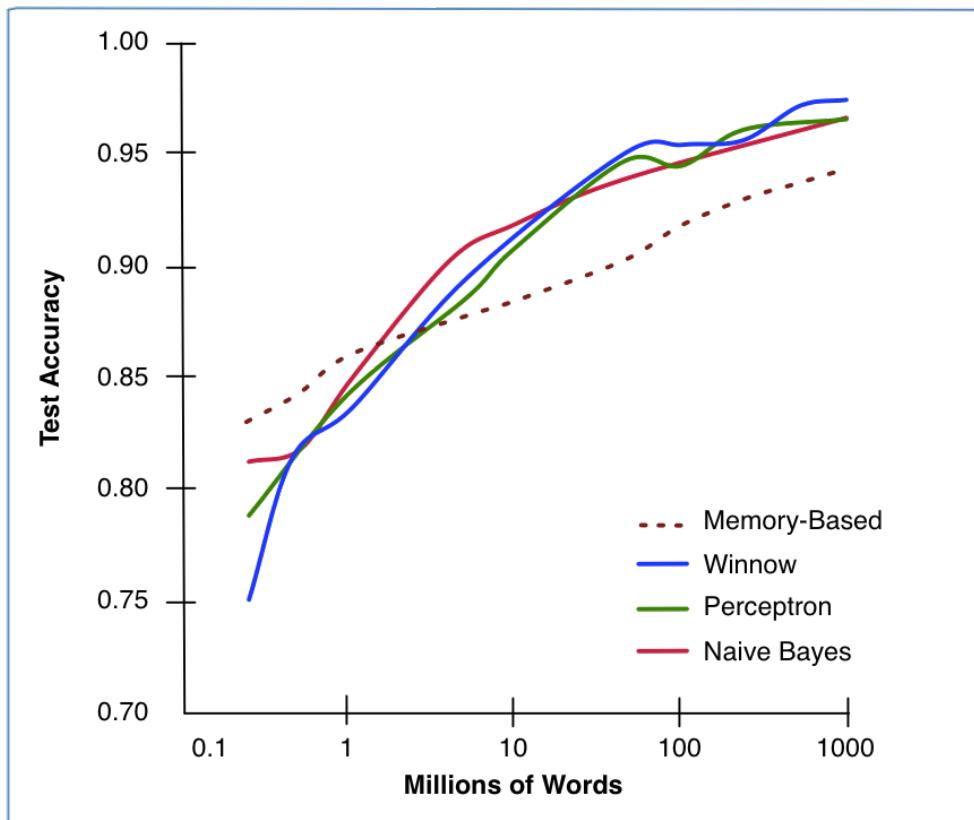
Relationship of Algorithms and Data Volume (1)

- **There are many algorithms for each type of machine learning**
 - There's no overall "best" algorithm
 - Each algorithm has advantages and limitations
- **Algorithm choice is often related to data volume**
 - Some scale better than others
- **Most algorithms offer better results as volume increases**
 - Best approach = simple algorithm + lots of data
- **Spark is an excellent platform for machine learning over large data sets**
 - Resilient, iterative, parallel computations over distributed data sets

Relationship of Algorithms and Data Volume (2)

It's not who has the best algorithms that wins. It's who has the most data.

—Banko and Brill, 2001



Machine Learning Challenges

- **Highly computation-intensive and iterative**
- **Many traditional numerical processing systems do not scale directly to very large datasets**
 - Some make use of Spark to bridge the gap, such as MATLAB®

Spark MLlib and Spark ML

- **Spark MLlib is a Spark machine learning library**
 - Makes practical machine learning scalable and easy
 - Includes many common machine learning algorithms
 - Includes base data types for efficient calculations at scale
 - Supports scalable statistics and data transformations
- **Spark ML is a new higher-level API for machine learning pipelines**
 - Built on top of Spark's DataFrames API
 - Simple and clean interface for running a series of complex tasks
 - Supports most functionality included in Spark MLlib
- **Spark MLlib and ML support a variety of machine learning algorithms**
 - Such as ALS (alternating least squares), k-means, linear regression, logistic regression, gradient descent

Chapter Topics

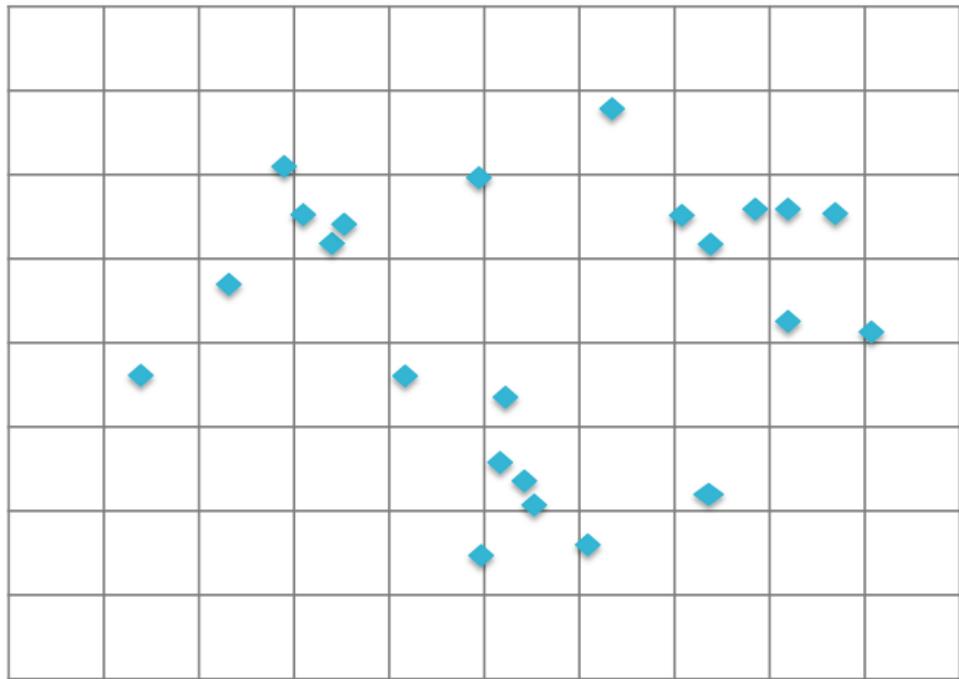
Common Patterns in Spark Data Processing

- Common Apache Spark Use Cases
- Iterative Algorithms in Apache Spark
- Machine Learning
- Example: k-means
- Essential Points
- Hands-On Exercise: Implementing an Iterative Algorithm

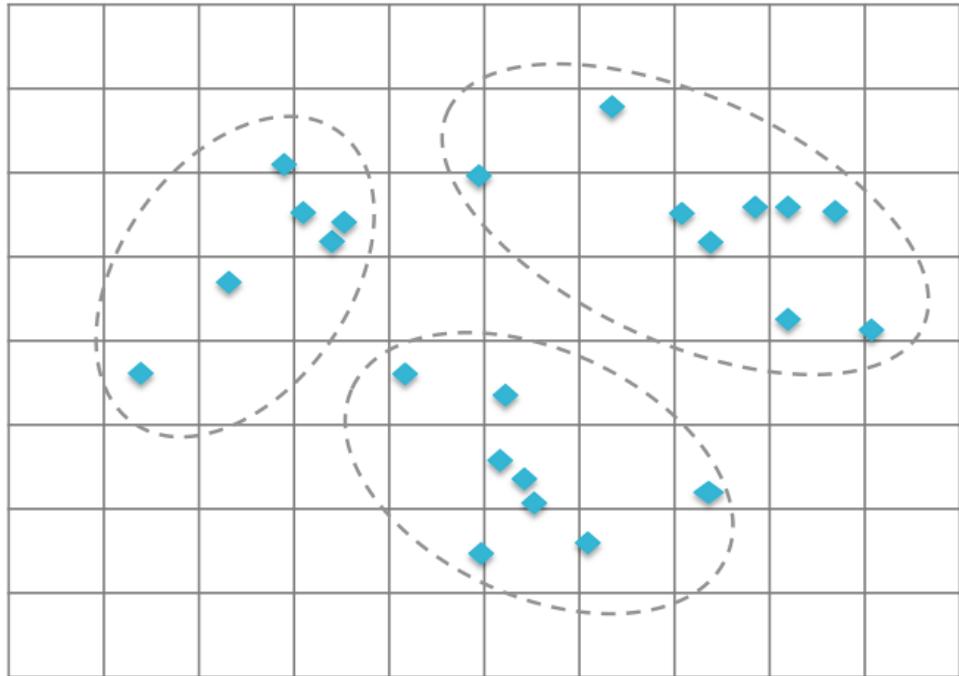
k-means Clustering

- **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)

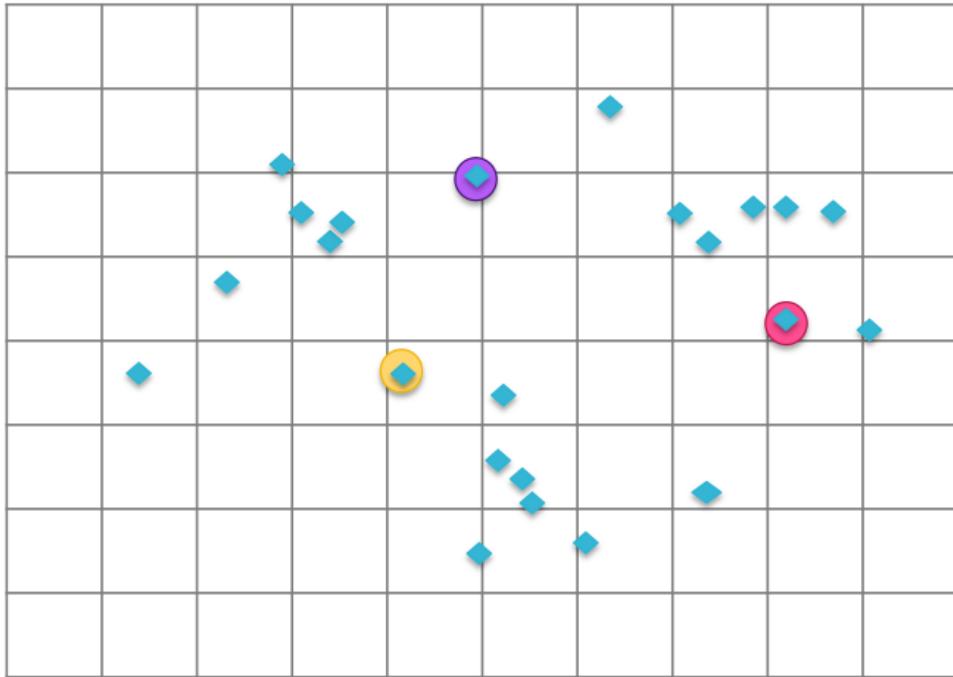


Clustering (2)



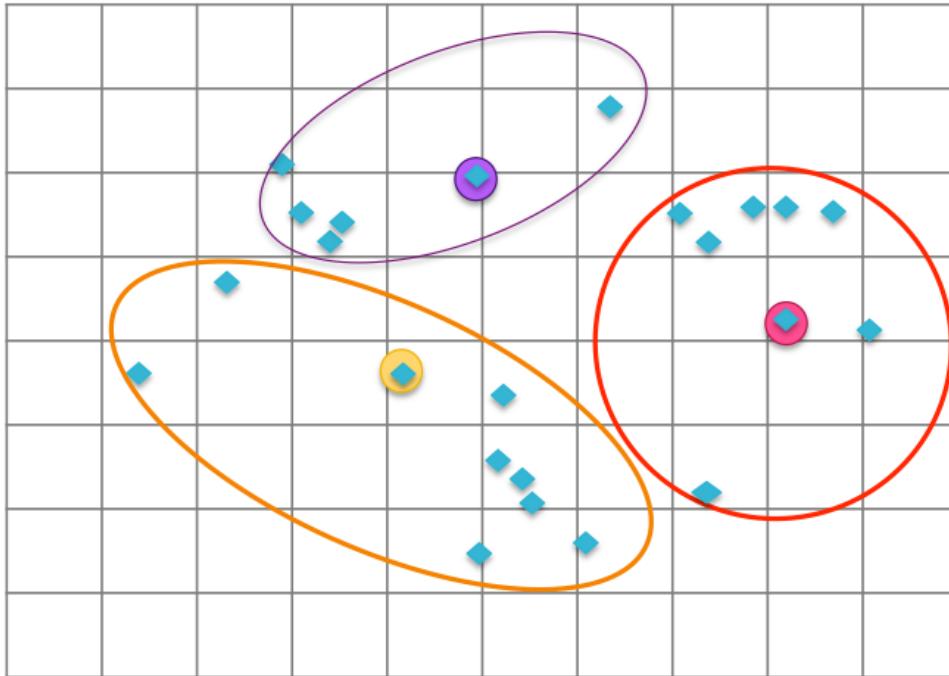
Goal: Find “clusters” of data points

Example: k-means Clustering (1)



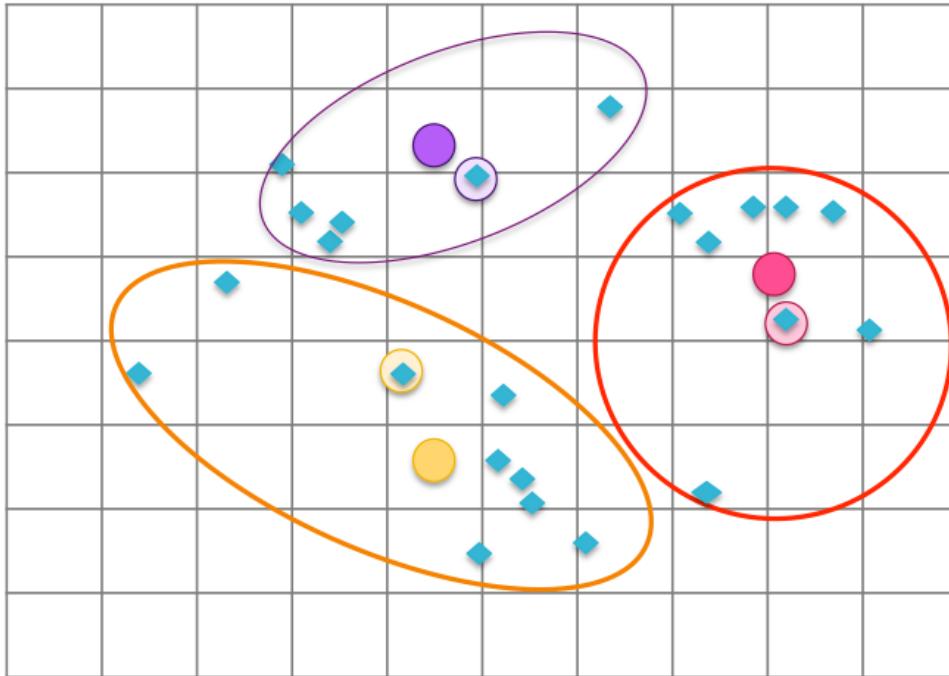
1. Choose k random points as starting centers

Example: k-means Clustering (2)



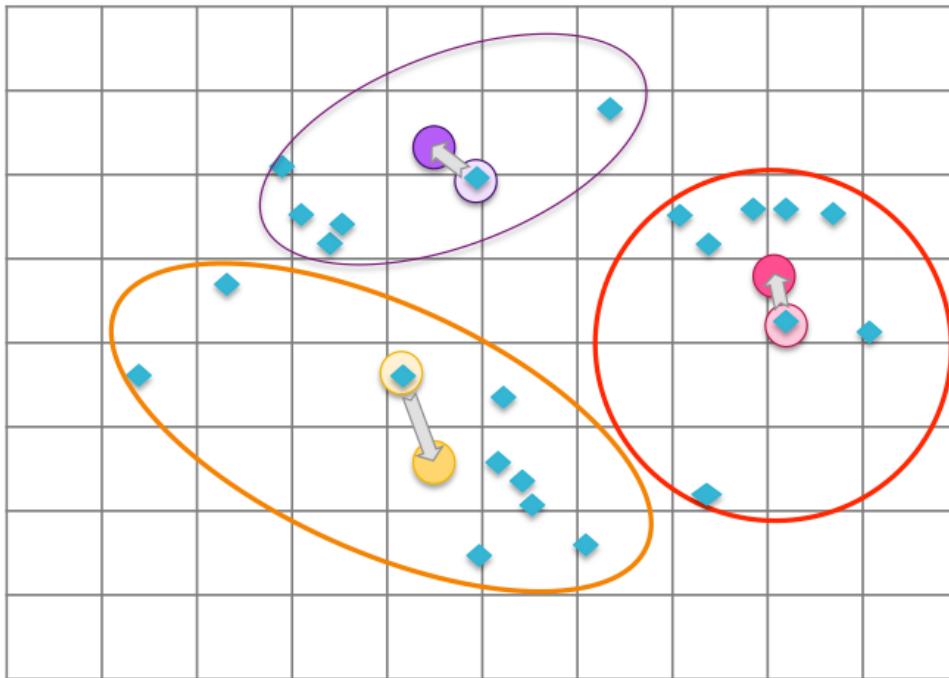
1. Choose k random points as starting centers
2. Find all points closest to each center

Example: k-means Clustering (3)



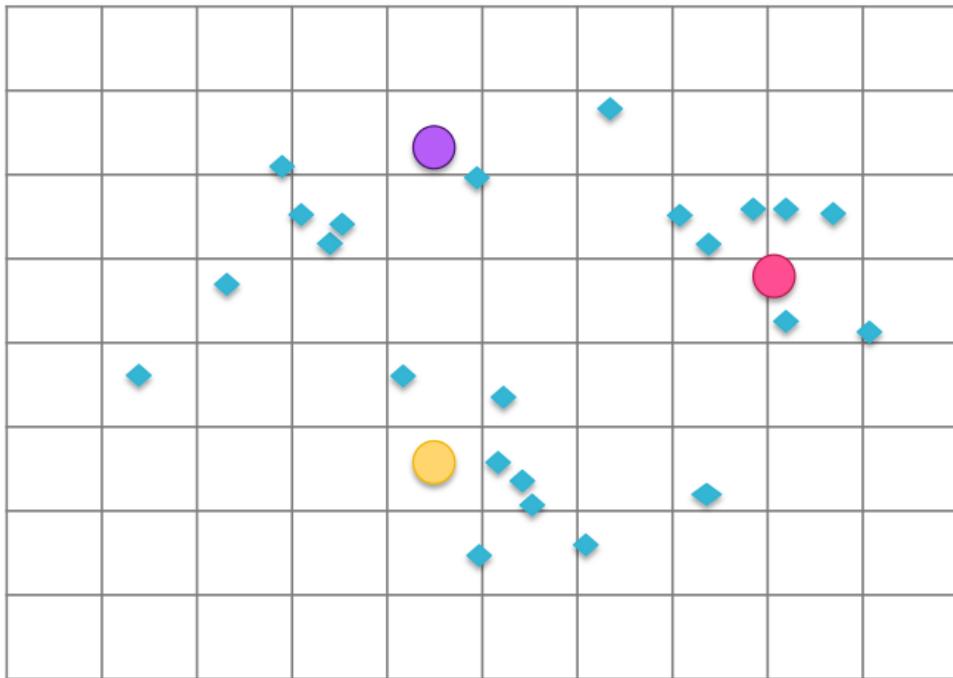
1. Choose k random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster

Example: k-means Clustering (4)



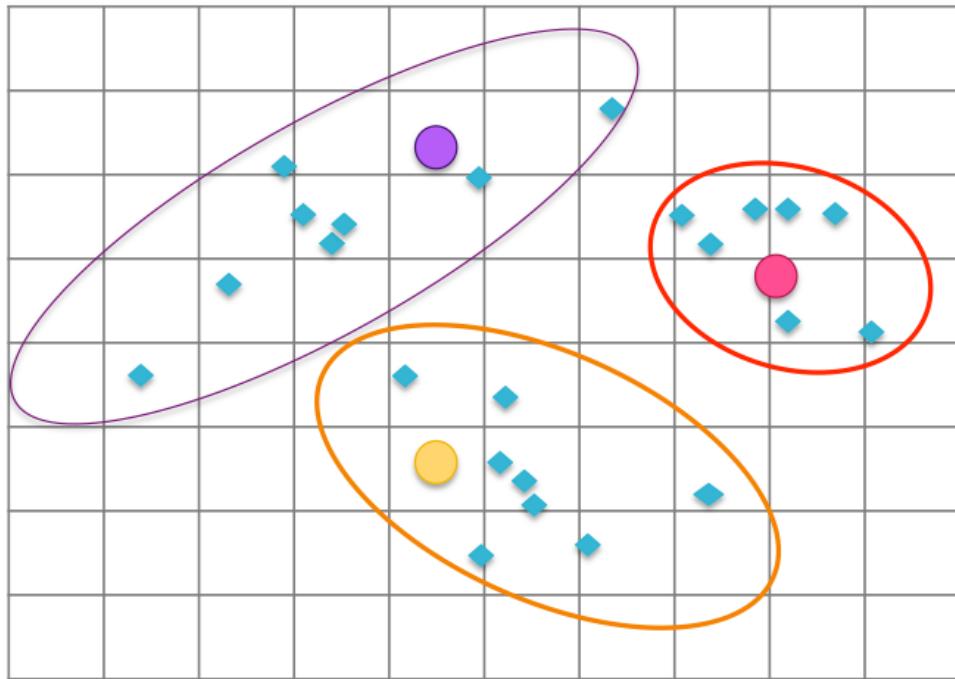
1. Choose k random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (5)



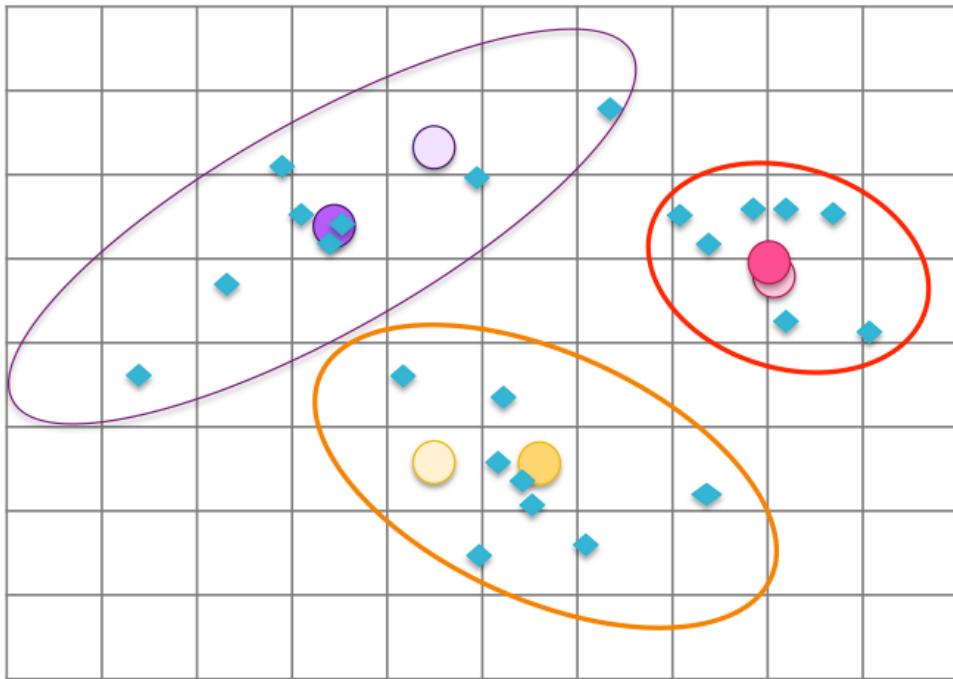
1. Choose k random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (6)



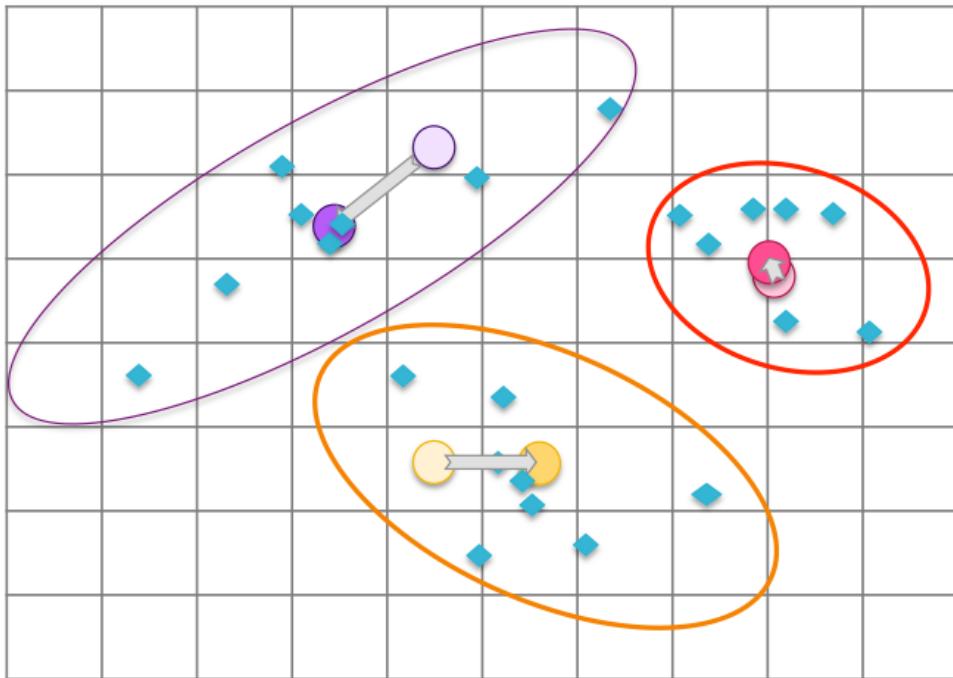
1. Choose k random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (7)



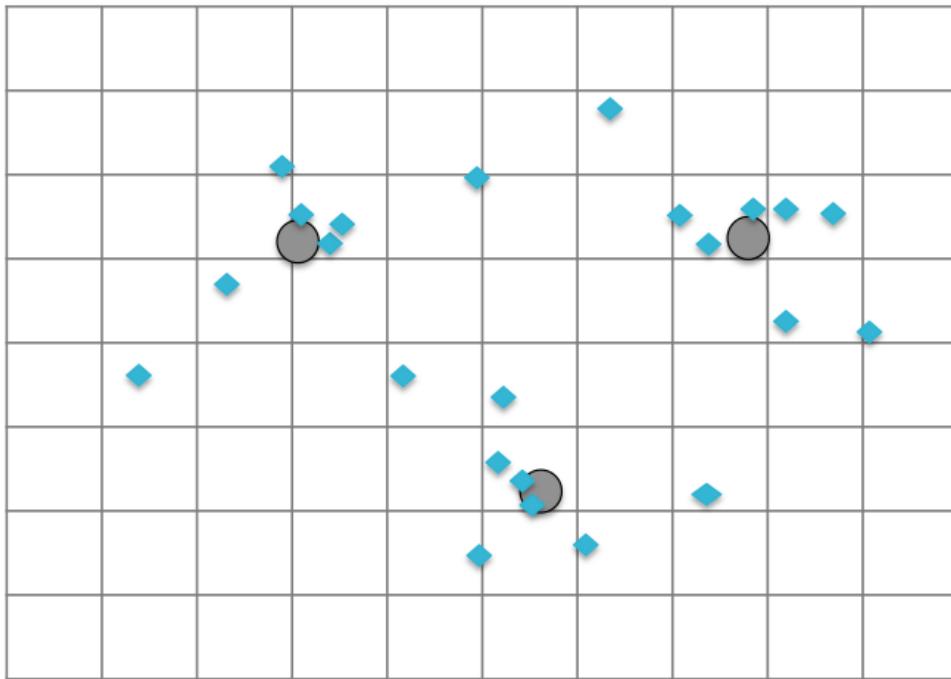
1. Choose k random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

Example: k-means Clustering (8)



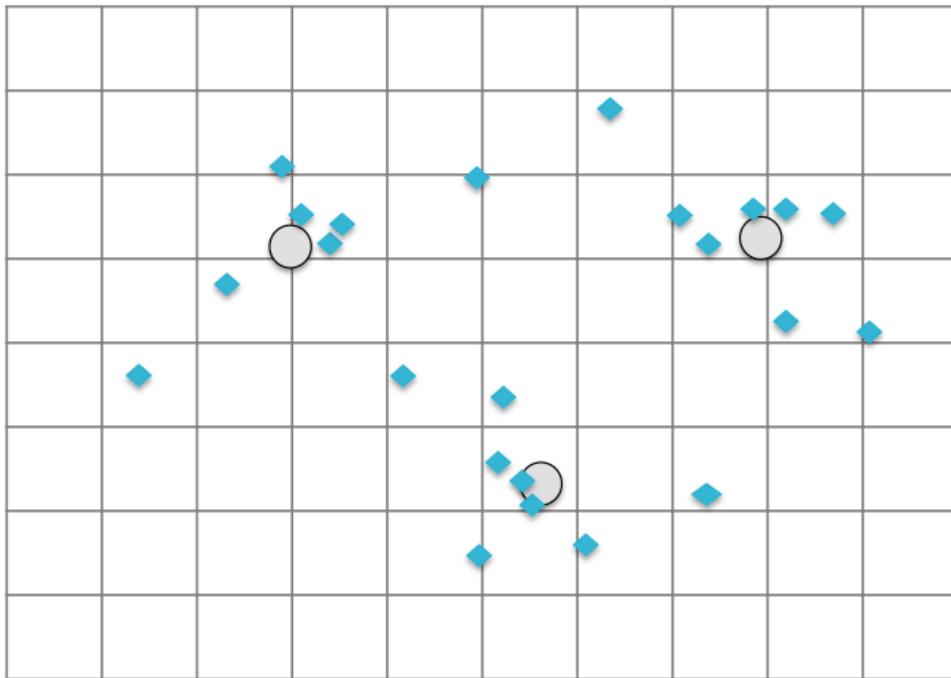
1. Choose k random points as starting centers
2. Find all points closest to each center
3. Find the center (mean) of each cluster
4. If the centers changed, iterate again

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

- Common Apache Spark Use Cases
- Iterative Algorithms in Apache Spark
- Machine Learning
- Example: k-means
- **Essential Points**
- Hands-On Exercise: Implementing an Iterative Algorithm

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**
 - For example, common algorithms such as PageRank and k-means
- **Spark is especially well-suited for implementing machine learning**
- **Spark includes MLlib and ML**
 - Specialized libraries to implement many common machine learning functions
 - Efficient, scalable functions for machine learning (for example: logistic regression, k-means)

Chapter Topics

Common Patterns in Spark Data Processing

- Common Apache Spark Use Cases
- Iterative Algorithms in Apache Spark
- Machine Learning
- Example: k-means
- Essential Points
- **Hands-On Exercise: Implementing an Iterative Algorithm**

Hands-On Exercise: Implementing an Iterative Algorithm

- In this exercise, you will implement the k-means algorithm in Spark
- Please refer to the Hands-On Exercise Manual for instructions



Introduction to Structured Streaming

Chapter 17

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- **Introduction to Structured Streaming**
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Introduction to Structured Streaming

After completing this chapter, you will be able to

- **Describe some use cases for streaming applications**
- **List the key features of Structured Streaming**
- **Create a streaming DataFrame to read data from a data source**
- **Use the DataFrame API to transform streaming data**
- **Output streaming data to a target sink**
- **Configure and execute streaming queries**

Chapter Topics

Introduction to Structured Streaming

- **Apache Spark Streaming Overview**
- Creating Streaming DataFrames
- Transforming DataFrames
- Executing Streaming Queries
- Essential Points
- Hands-On Exercise: Processing Streaming Data

Why Streaming?

- **Many big data applications need to process large data streams in real time, such as**
 - Continuous ETL
 - Website monitoring
 - Fraud detection
 - Advertisement monetization
 - Social media analysis
 - Financial market trends
 - Event-based data

Streaming Applications in Apache Spark

- **Spark includes two APIs for streaming applications**
 - DStreams API (also called “Spark Streaming”)
 - Initial streaming solution in Spark 1
 - API is in a separate Spark library
 - Structured Streaming API
 - Preview in Spark 2.0, supported in Spark 2.3
 - Integrated with DataFrames/Datasets API

Comparing Structured Streaming and DStreams API

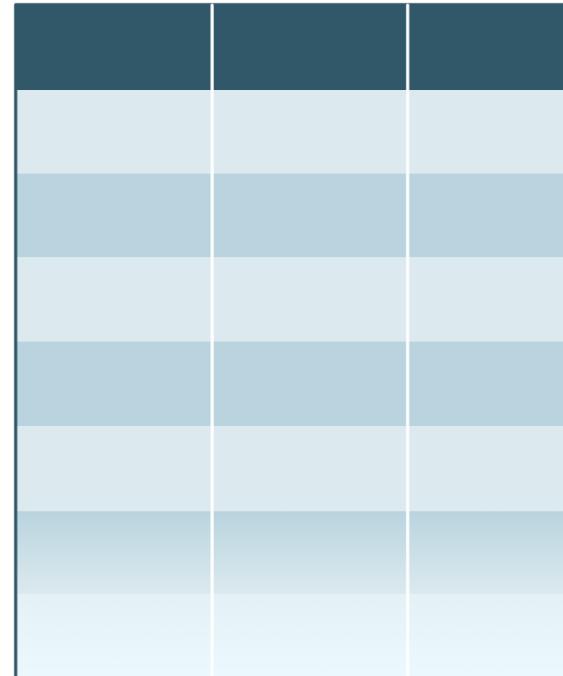
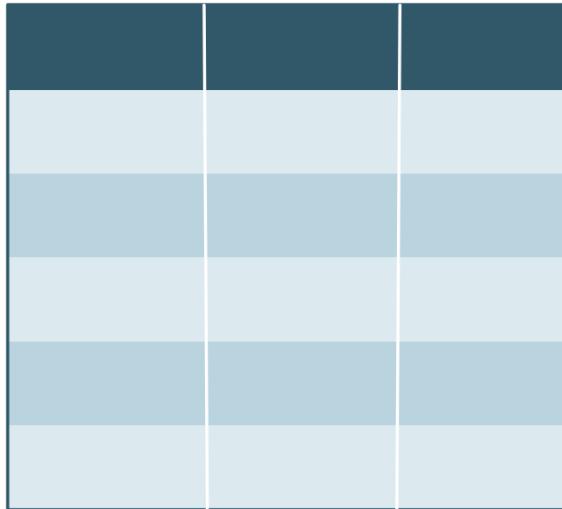
Structured Streaming	DStreams API
<ul style="list-style-type: none">■ DataFrame/Dataset-based■ Higher level API■ Best for structured or semi-structured data■ Provides SQL-like semantics■ Guarantees consistency between streaming and static queries■ Queries optimized by the Catalyst optimizer	<ul style="list-style-type: none">■ RDD-based■ Lower level API■ Best for unstructured data

Why Structured Streaming?

- **Designed for end-to-end, continuous, real-time data processing**
- **Ensures consistency**
 - Guarantees exactly-once handling
 - Query results are the same on static or streaming data
- **Built-in support for time-series data**
 - Handles out-of-order and late events

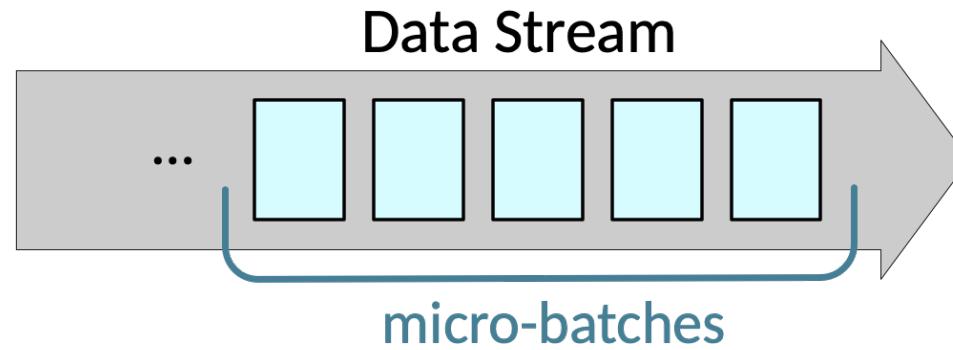
Streaming DataFrames

- Regular DataFrames model data as a *bounded* table
 - Data is immutable
- Streaming DataFrames model data as an *unbounded* table
 - Data grows over time



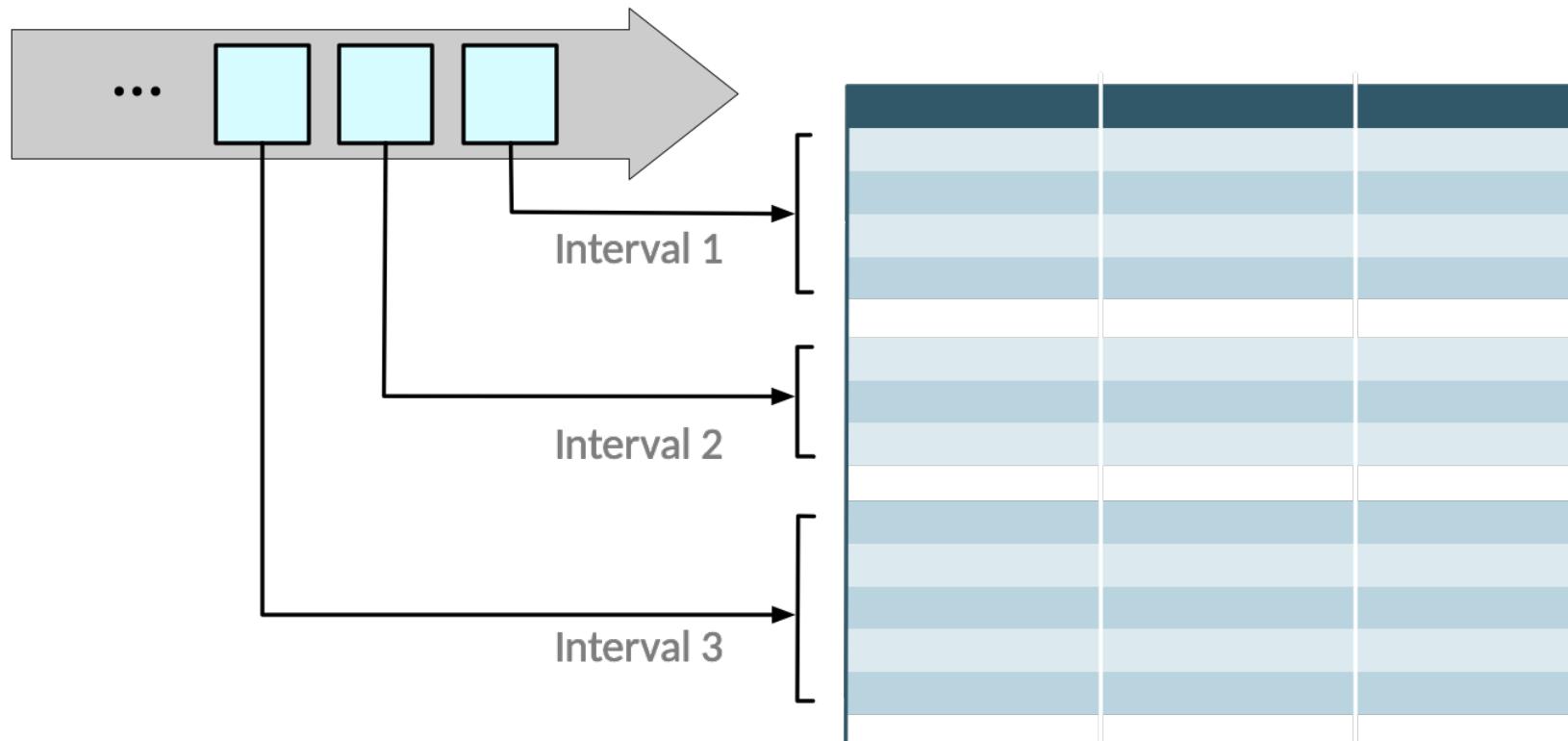
Micro-batches

- Spark collects all data received over an interval of time into *micro-batches*



Appending Streaming Data

- Each micro-batch is appended to the unbounded input table



Fault Tolerance

- **Fault tolerance is a key feature of Structured Streaming**
 - Ability to recover from failures such as hardware or network problems
 - Exactly-once guarantees for supported data sources and sinks
- **Spark uses write-ahead logs and checkpointing for fault tolerance**
 - Write-ahead logs ensure data is consistent after a failure
 - Checkpointing saves the current state of a query to files

Creating Structured Streaming Applications

- Typical steps in writing a structured streaming application
 1. Create a streaming DataFrame from a streaming data source
 2. Define a query including one or more transformations on the DataFrame
 3. Save or display query results

Example Application

- The next few sections will walk through an example application
 - Receives mobile device status data through a socket
 - CSV format

```
model,device-id,temperature,signal-strength,wifi,...
```

- Displays selected status information about a specified model

Chapter Topics

Introduction to Structured Streaming

- Apache Spark Streaming Overview
- **Creating Streaming DataFrames**
- Transforming DataFrames
- Executing Streaming Queries
- Essential Points
- Hands-On Exercise: Processing Streaming Data

Creating Streaming DataFrames

- Create streaming DataFrames using DataStreamReader API
 - Very similar to DataFrameReader
- Streaming and regular DataFrame objects are the same type: DataFrame (Python), Dataset [Row] (Scala/Java)
 - DataFrame `isStreaming` property is set to true

```
linesDF = spark.readStream...
```

Language: Python

Data Sources

- **Supported streaming data sources**
 - File—reads files from a specified directory
 - Socket—not fault-tolerant, for testing only
 - Rate—generates data for load testing
 - Apache Kafka

```
linesDF = spark.readStream.format("socket")...
```

Language: Python

Data Source Options

- Different data sources use different options
- Examples
 - File: read latest or oldest files first, number of files per batch
 - Socket: host name and port

```
linesDF = spark.readStream.format("socket"). \
    .option("host",hostname).option("port", port-number)...
```

Language: Python

Loading a Streaming DataFrame

- **Socket and Kafka streams: load**
- **File streams: use specific function for file format**
 - Such as `csv`, `orc`, `parquet`, `json`, `text`
 - Pass path to directory containing files
- **Note that this does not start reading data from the stream**
 - Spark does not start receiving data until a streaming query is started

```
linesDF = spark.readStream.format("socket"). \
    option("host",hostname).option("port",port-number).load()
```

Language: Python

Streaming DataFrame Schemas

- Schema of input DataFrame depends on type of source
 - Socket—single string column value
 - Elements in streaming text are line-delimited
 - Transform to parse lines into individual values
 - Files
 - Plain text files—same as socket streams
 - Structured formats such as Parquet, JSON, CSV—specify schema manually

```
...schema(mySchema).csv(directory-path)
```

- Kafka—fixed schema

Chapter Topics

Introduction to Structured Streaming

- Apache Spark Streaming Overview
- Creating Streaming DataFrames
- **Transforming DataFrames**
- Executing Streaming Queries
- Essential Points
- Hands-On Exercise: Processing Streaming Data

Streaming DataFrame Transformations (1)

- Transforming a streaming DataFrame returns a new streaming DataFrame
- Structured Streaming supports most DataFrame transformations, such as
 - select
 - where / filter
 - groupBy
 - orderBy / sort
 - join

Streaming DataFrame Transformations (2)

- Example: Parse CSV input into columns

```
from pyspark.sql.functions import *

statusDF = linesDF. \
    withColumn("model", split(linesDF.value, ",") [0]). \
    withColumn("dev_id", split(linesDF.value, ",") [1]). \
    withColumn("dev_temp", \
        split(linesDF.value, ",") [2].cast("integer")). \
    withColumn("signal", \
        split(linesDF.value, ",") [3].cast("integer")). \
    ...
```

Language: Python

Streaming DataFrame Transformations (3)

- Example: Select desired columns and rows

```
statusDF = linesDF. \
    withColumn("model", split(linesDF.value, ",") [0]). \
    withColumn("dev_id", split(linesDF.value, ",") [1]). \
    withColumn("dev_temp", \
        split(linesDF.value, ",") [2].cast("integer")). \
    withColumn("signal", \
        split(linesDF.value, ",") [3].cast("integer")). \
    ... \
statusFilterDF = statusDF. \
    select("dev_id","dev_temp","signal"). \
    where("model='Sorrento F41L'")
```

Language: Python

Chapter Topics

Introduction to Structured Streaming

- Apache Spark Streaming Overview
- Creating Streaming DataFrames
- Transforming DataFrames
- **Executing Streaming Queries**
- Essential Points
- Hands-On Exercise: Processing Streaming Data

Executing a Streaming Query

- DataFrame queries are triggered by actions
- Streaming DataFrames use DataStreamWriter
 - Similar to DataFrameWriter

```
statusQuery = statusDF.writeStream...
```

Language: Python

Output Formats

- **Output (*sink*) formats**

- Console—displays output to console (debugging only)
- Memory—save data to an in-memory table (debugging only)
- File—saves a set of files to a directory
 - Use specific function for target format
 - Such as `csv()`, `parquet()`, `json()`, `orc()`
- Kafka—sends result rows as Kafka messages
- Custom—use `foreach` or `foreachBatch` to write your own output code

Output Format Options

- Most options are specific to target formats
 - Examples: `sep` for CSV, `truncate` for console

```
statusQuery = statusDF.writeStream. \
    format("console").option("truncate","false")...
```

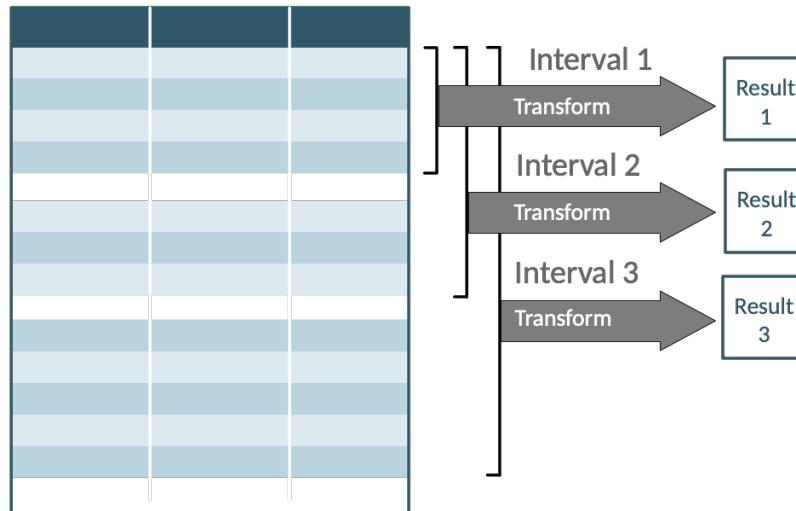
Language: Python

- Some types of output require checkpointing for fault tolerance
 - Such as writing to files
 - Set for individual queries using `option("checkpointLocation", "checkpoint-directory")`
 - Set for Spark session by configuring
`spark.sql.streaming.checkpointLocation`

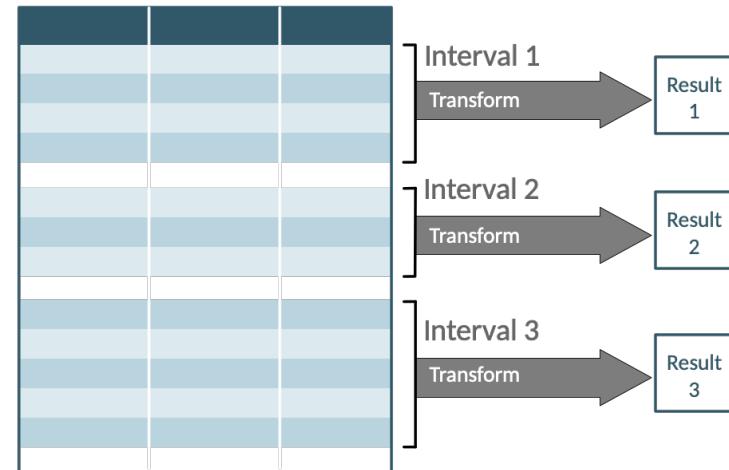
Output Mode

- **outputMode** specifies which results will be returned for each interval
 - **complete**—the entire result set
 - **append**—only new rows in the interval (default)
 - **update**—new and changed rows in the interval

Complete Output Mode



Append Output Mode



Output Formats and Supported Output Modes

Output Formats	Output Mode
File	append
Console	all
Foreach, ForeachBatch	all
Kafka	all
Memory	append, complete

```
statusQuery = statusDF.writeStream. \
    format("console").option("truncate","false"). \
    outputMode("append")...
```

Language: Python

Micro-batches

- Micro-batches contain all data received over a period of time
 - Queries are triggered once for each micro-batch
- Default—micro-batches generated as soon as previous micro-batch query is complete
- Fixed interval—queries are triggered at specified intervals
 - Set interval using trigger function
 - Will not trigger until previous query is complete

```
statusQuery = statusDF.writeStream. \
    format("console").option("truncate","false"). \
    outputMode("append").trigger(processingTime="2 seconds")...
```

Language: Python

```
import org.apache.spark.sql.streaming.Trigger.ProcessingTime
..trigger(ProcessingTime("2 seconds"))...
```

Language: Scala

Starting and Stopping Streams

- Use **start** function to start the stream
 - Spark will start receiving data
 - Queries will start executing for each micro-batch
- Returns a **StreamingQuery** object
 - **stop** function stops the stream
 - **status** function returns current stream status
 - **awaitTermination** function waits until query is stopped with **stop** or an exception
 - Several other functions explore and manage the query

```
statusQuery = statusDF.writeStream. \
    format("console").option("truncate","false"). \
    outputMode("append").trigger(processingTime="2 seconds"). \
    start()

statusQuery.awaitTermination()
```

Language: Python

Example Output (1)

Batch: 0

```
+-----+-----+-----+
| dev_id | dev_temp | signal |
+-----+-----+-----+
|         |         |       |
|         |         |       |
|         |         |       |
```

Batch: 1

```
+-----+-----+-----+
| dev_id           | dev_temp | signal |
+-----+-----+-----+
| 524a4c77-64f5-4fb0-a444-ceaef7079d9f | 29   | 48   |
| 64f745dd-f99e-4c01-83e3-c695365138f5 | 12   | 67   |
| 9d0ada1b-a354-4759-a470-1ed725264eb6 | 24   | 74   |
| adc591f2-10b8-49fd-a469-f6d9e55581bf | 23   | 49   |
+-----+-----+-----+
```

...

[Continued on next slide...](#)

Example Output (2)

Batch: 2

dev_id	dev_temp	signal
a98febad-683c-4b96-9e50-9615418ce1a2 27	38	
5b42e782-f0ab-4428-aea4-cbeaf77032ff 13	28	
7836c1a6-365b-4ec2-a402-65cdcb03a171 19	22	
598adc05-45ed-48e2-866e-119d4c8d5dda 20	43	
...		

Batch: 3

dev_id	dev_temp	signal
d79a37e7-96fb-4aa2-aa9a-fab01ee210d3 13	20	
56d87542-0c79-43c3-8184-01d0e595caed 12	53	
f8e2a12c-2a90-4c48-9420-d00805eb6912 13	34	
...		

Chapter Topics

Introduction to Structured Streaming

- Apache Spark Streaming Overview
- Creating Streaming DataFrames
- Transforming DataFrames
- Executing Streaming Queries
- **Essential Points**
- Hands-On Exercise: Processing Streaming Data

Essential Points

- The Structured Streaming API is integrated into the DataFrames/Datasets API
- Data streams are broken up into micro-batches
 - Using sequential or interval triggers
- Streaming DataFrames can read from files, sockets, Kafka, or a test source
 - Create a streaming DataFrame using DataStreamReader
- Basic streaming DataFrame transformations are the same as regular DataFrames
- Save or display streaming query results using DataStreamWriter
- Output mode determines which results are returned
 - Append, complete, and update

Chapter Topics

Introduction to Structured Streaming

- Apache Spark Streaming Overview
- Creating Streaming DataFrames
- Transforming DataFrames
- Executing Streaming Queries
- Essential Points
- **Hands-On Exercise: Processing Streaming Data**

Hands-On Exercise: Processing Streaming Data

- In this exercise, you will read and process streaming text data through a socket
- Please refer to the Hands-On Exercise Manual for instructions



Structured Streaming with Apache Kafka

Chapter 18

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- **Structured Streaming with Apache Kafka**
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Structured Streaming with Apache Kafka

After completing this chapter, you will be able to

- Summarize how Kafka receives and delivers messages using topics
- Describe the schema used for Kafka input and output streaming DataFrames
- Create a streaming DataFrame to receive or send Kafka messages
- Access message content from a Kafka message

Chapter Topics

Structured Streaming with Apache Kafka

- **Overview**
- Receiving Kafka Messages
- Sending Kafka Messages
- Essential Points
- Hands-On Exercise: Working with Apache Kafka Streaming Messages

Structured Streaming with Kafka

- Structured Streaming applications can read and write streams of Kafka messages
- Kafka sinks and sources provide fault tolerance and “at-least-once” guarantees

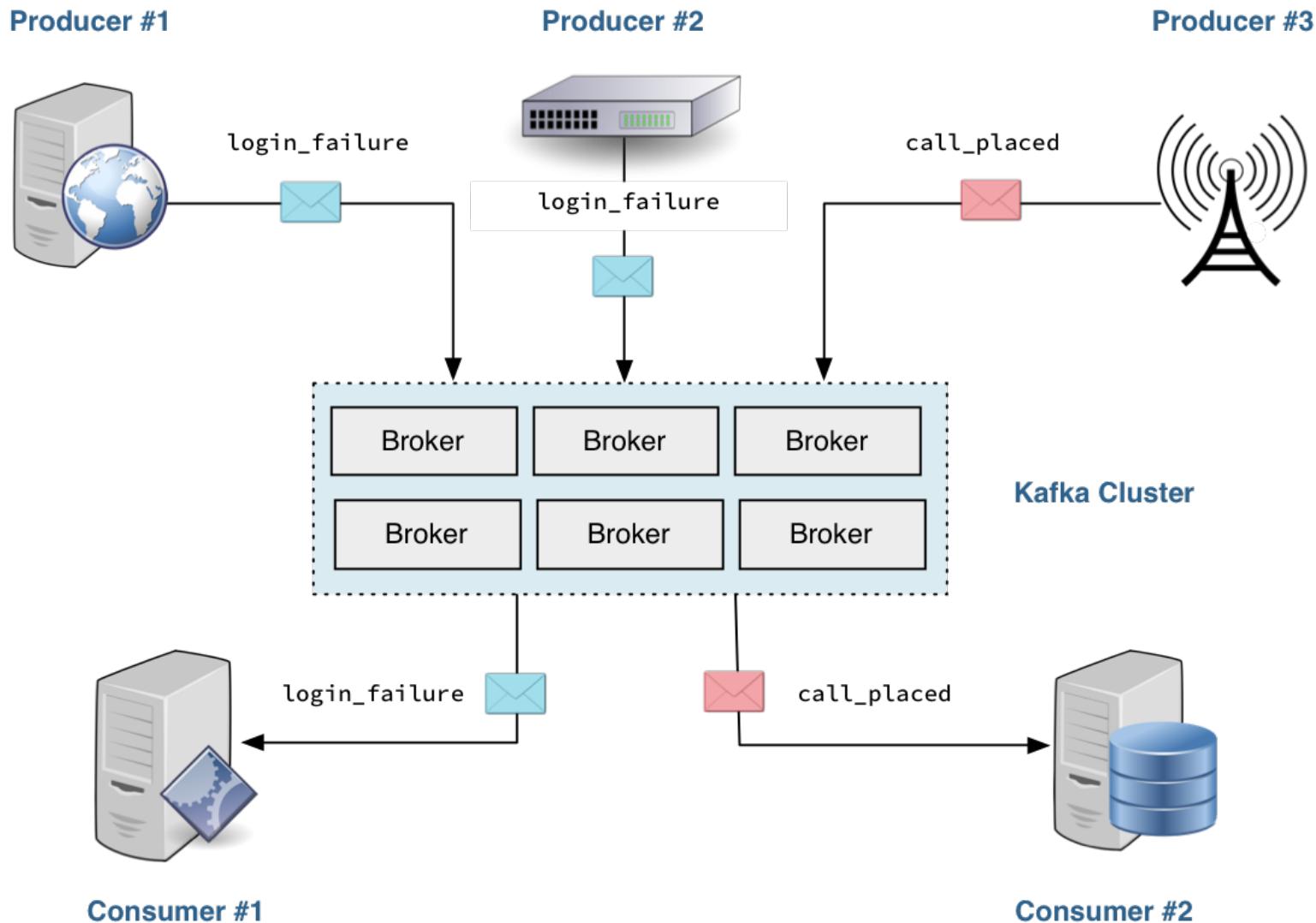
Kafka Basics (1)

- Apache Kafka is a fast, scalable, distributed publish-subscribe messaging system that provides
 - Durability by persisting data to disk
 - Fault tolerance through replication
- A *message* is a single data record passed by Kafka
- One or more *brokers* in a cluster receive, store, and distribute messages

Kafka Basics (2)

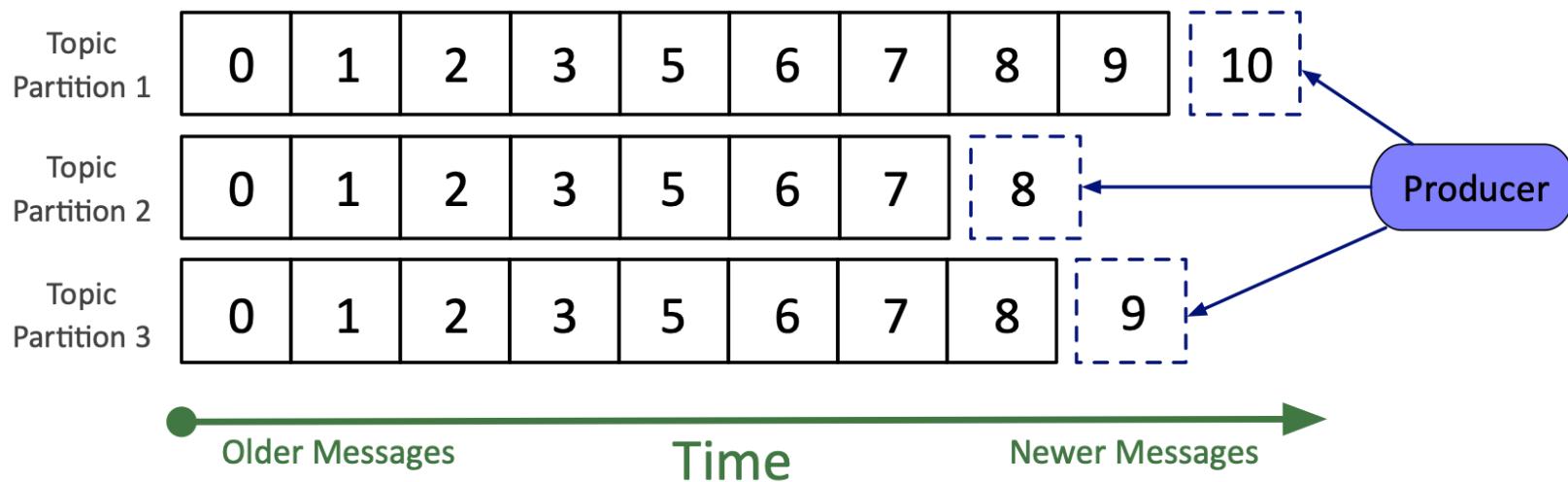
- A **topic** is a named feed or queue of messages
 - A Kafka cluster can include any number of topics
- **Producers** are programs that publish (send) messages to a topic
- **Consumers** are programs that subscribe to (receive messages from) a topic
- A Spark Streaming application can be a Kafka producer, consumer, or both
- **Kafka allows a topic to be partitioned**
 - Topic partitions are handled by different brokers for scalability
 - Note that topic partitions are not related to DataFrame or RDD partitions

Kafka Basics (3)



Kafka Basics (4)

- Kafka divides each topic into partitions
 - Topic partitioning improves scalability and throughput
- A topic partition is an ordered sequence of messages
 - New messages are appended to the partition as they are received
 - Each message is assigned a unique sequential ID known as an offset



Chapter Topics

Structured Streaming with Apache Kafka

- Overview
- **Receiving Kafka Messages**
- Sending Kafka Messages
- Essential Points
- Hands-On Exercise: Working with Apache Kafka Streaming Messages

Using Kafka Streaming Data Source

- **Use kafka format to create a streaming DataFrame**
 - Each Kafka message received becomes a single DataFrame element

```
kafkaDF = spark.readStream.\n    format("kafka")...
```

Language: Python

Key Kafka Source Options (1)

- **Kafka brokers (required)**
 - `kafka.bootstrap.servers`—comma-delimited list with hostname and port
 - Specify multiple servers for fault-tolerance
- **Topic subscriptions (required)—specify one of the following options**
 - `subscribe`—the topic or topics to subscribe to (comma-separated)
 - `subscribePattern`—regex pattern to match a set of topics
 - `assign`—specific partitions for topics (JSON string)

```
kafkaDF = spark.readStream. \
    format("kafka"). \
    option("kafka.bootstrap.servers", \
        "broker-host1:9092,broker-host2:9092"). \
    option("subscribe","status").load()
```

Language: Python

Key Kafka Source Options (2)

- **Offsets (optional)—where to start reading messages in a topic**
 - `startingOffsets`
 - `earliest`—start reading from the earliest available messages
 - `latest`—start reading messages sent after a new query starts
 - `maxOffsetsPerTrigger`—maximum number of messages in each micro-batch

Kafka Input DataFrame Schema (1)

- All DataFrames from Kafka input have the same schema
- Columns
 - key (binary)—used by some producers to allow consumer to sort or classify messages (might be empty)
 - value (binary)—the content of the Kafka message
 - topic (string)—the topic the message was received on
 - partition (int)—the partition the message was in

Kafka Input DataFrame Schema (2)

- **Columns (continued)**
 - `offset` (long)—offset within topic partition
 - `timestamp`—the timestamp of the message
 - `timestampType`
 - Whether the timestamp represents the time the producer created the message or the time the broker received it

Kafka Message Values

- **The value column contains the content of the message**
 - In binary form
- **The message producer determines the format of the content**
 - Common formats include CSV and JSON strings and Avro data format
- **You must transform the message into a usable form**
 - Such as parsing comma-separated values in a string to a set of column values

Example: Convert CSV String to Column Values (1)

- Parse comma-delimited string into array of strings

```
kafkaDF = spark.readStream...  
  
from pyspark.sql import functions  
valuesDF = kafkaDF. \  
    select(functions.split(kafkaDF.value, ","). \  
        alias("status_vals"))
```

Language: Python

Example: Convert CSV String to Column Values (2)

- Use parsed values in transformations

```
kafkaDF = spark.readStream...

from pyspark.sql import functions as f
valuesDF = kafkaDF. \
    select(functions.split(kafkaDF.value, ","). \
        alias("status_vals"))

modelTempsDF = valuesDF. \
    select(valuesDF.status_vals[0].alias("model"), \
        valuesDF.status_vals[2].cast("integer").alias("dev_temp"))
```

Language: Python

Chapter Topics

Structured Streaming with Apache Kafka

- Overview
- Receiving Kafka Messages
- **Sending Kafka Messages**
- Essential Points
- Hands-On Exercise: Working with Apache Kafka Streaming Messages

Sending Data to a Kafka Streaming Sink

1. Create a DataFrame containing output data

- Each DataFrame element is sent as a single Kafka message
- Elements must be in the correct format

2. Execute a query on the output data

- Send messages to the correct Kafka topic

Kafka Output DataFrame Schema

- Kafka output DataFrames have the following columns
 - key (string or binary)
 - Column is required but value may be empty string
 - value (string or binary)—content of the message
 - Required
 - Encode into desired format—such as JSON, CSV, or Avro data format
 - Consumer will extract required information
 - topic (string)—what Kafka topic(s) to send message to
 - Optional—topic(s) may be specified as an option when sending the message instead

Example: A Kafka Output DataFrame with CSV data

- Output DataFrame is based on existing streaming DataFrame with device status data
- DataFrame contains
 - key column with empty string literal values
 - value column with CSV strings containing device ID and model

```
statusDF = ...  
  
from pyspark.sql import functions  
statusValueDF = statusDF. \  
    select(functions.lit("").alias("key"),  
           functions.concat_ws(",","dev_id","model").alias("value"))
```

Language: Python

Writing Messages to Kafka

- Set the format to kafka
- Pass a list of Kafka brokers (hostname and port)
- Set a location to save checkpoint files
 - Required if `spark.sql.streaming.checkpointLocation` is not set
- Specify a topic
 - Required if DataFrame has no `topic` column
- Specify output mode
 - Kafka sink supports any output mode—default is append

```
statusValueQuery = statusValueDF. \
    writeStream.format("kafka"). \
    option("checkpointLocation", "/tmp/kafka-checkpoint"). \
    option("topic", "status-out"). \
    option("kafka.bootstrap.servers", "brokerhost:9092").start()
```

Language: Python

Chapter Topics

Structured Streaming with Apache Kafka

- Overview
- Receiving Kafka Messages
- Sending Kafka Messages
- **Essential Points**
- Hands-On Exercise: Working with Apache Kafka Streaming Messages

Essential Points

- **Kafka is a publish-subscribe messaging system**
 - Kafka brokers maintain ordered collections of related messages called *topics*
- **Structured Streaming applications can send (produce) and receive (consume) Kafka messages in a topic**
 - Using the same mechanism used to receive and send socket and file streaming data

Chapter Topics

Structured Streaming with Apache Kafka

- Overview
- Receiving Kafka Messages
- Sending Kafka Messages
- Essential Points
- **Hands-On Exercise: Working with Apache Kafka Streaming Messages**

Hands-On Exercise: Working with Apache Kafka Streaming Messages

- In this exercise, you will send and receive Apache Kafka messages**
- Please refer to the Hands-On Exercise Manual for instructions**



Aggregating and Joining Streaming DataFrames

Chapter 19

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames**
- Conclusion
- Appendix: Message Processing with Apache Kafka

Aggregating and Joining Streaming DataFrames

After completing this chapter, you will be able to

- **Describe how aggregation of streaming data works**
- **Explain how aggregating and non-aggregating streaming transformations differ**
- **Perform aggregation operations on streaming DataFrames**
- **Aggregate time-series data in a sliding window**
- **Use watermarking to limit how much streaming data must be buffered**
- **Join a streaming DataFrame with a static or streaming DataFrame**
- **Summarize the limitations on streaming joins**

Chapter Topics

Aggregating and Joining Streaming DataFrames

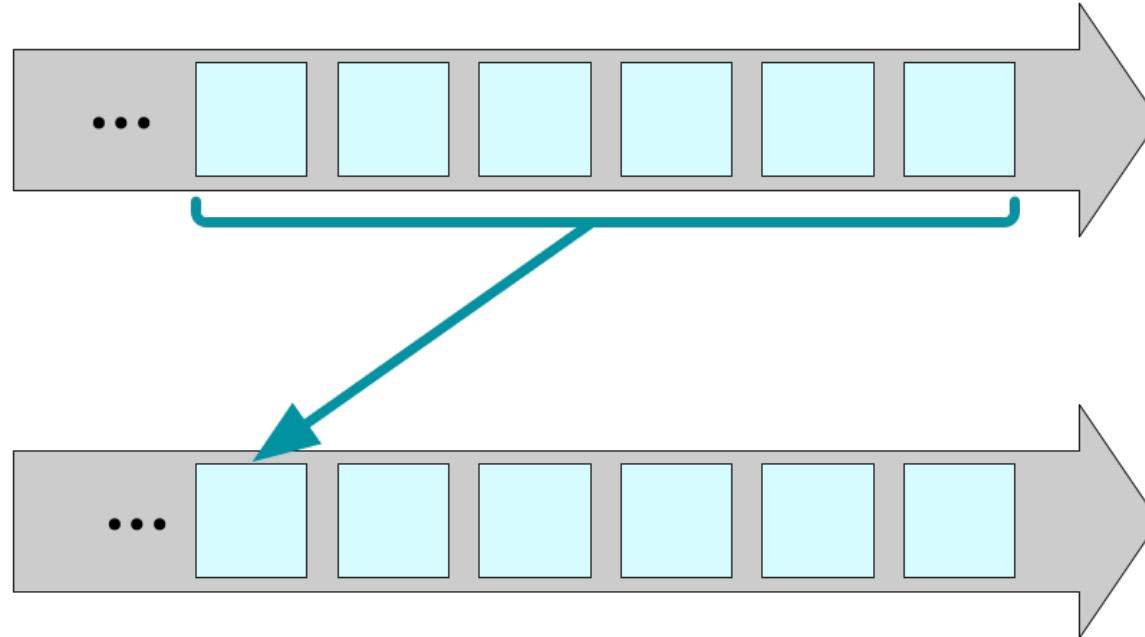
- **Streaming Aggregation**
- **Joining Streaming DataFrames**
- **Essential Points**
- **Hands-On Exercise: Aggregating and Joining Streaming DataFrames**

Streaming Aggregation

- Transformations create child streaming DataFrames by transforming data in the parent DataFrame
- Aggregation transformations calculate results from multiple input rows
- Streaming aggregation can be based on all or a subset of rows
- All static DataFrame aggregation transformations work with streaming DataFrames
 - But you cannot chain aggregation transformations

Full Aggregation

- By default, aggregation queries return results based all the data received up to the current trigger
 - Each transformation result will be based on more input data than the previous one



Example: Count Total Status Messages by Model

- Count status messages for selected models
 - Count totals reflect all status messages ever received by application

```
kafkaDF = spark.readStream.format("kafka")...

from pyspark.sql.functions import *
modelsDF = kafkaDF. \
    select(split(kafkaDF.value, ",")[0].alias("model"))

roninCountsDF = modelsDF. \
    where(modelsDF.model.startswith("Ronin S")). \
    groupBy("model").count()

roninCountsQuery = roninCountsDF.writeStream. \
    outputMode("complete").format("console").start()
```

Language: Python

Total Count Example Output

Batch: 1

model	count
Ronin S3	2
Ronin S1	3

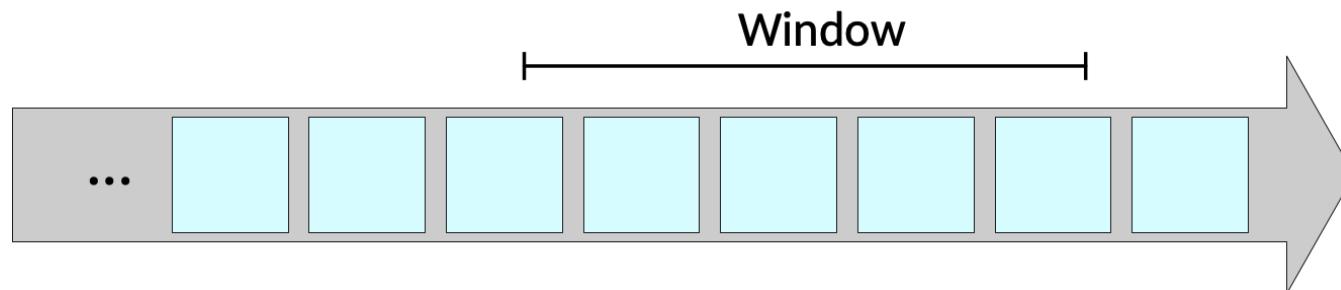
Batch: 2

model	count
Ronin S3	2
Ronin S2	2
Ronin S1	4

...

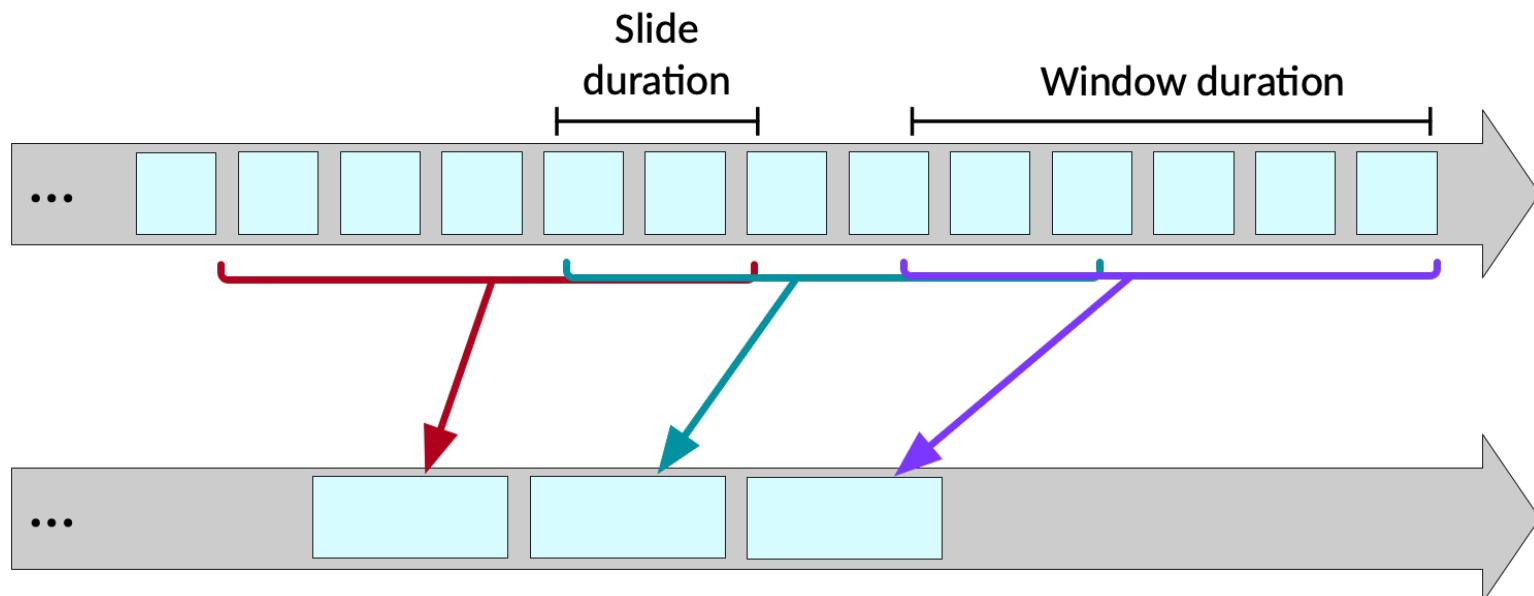
Sliding Window Aggregation (1)

- You can also aggregate on data from events that occurred within a specific time frame
 - Called a *window*
 - May contain data from any number of micro-batches
 - Input data must include a column with an event timestamp



Sliding Window Aggregation (2)

- As new data arrives, the window *slides*
 - Aggregate values are calculated for data within each window
- Sliding windows have
 - A *window duration*—the length of time the window covers
 - A *slide duration*—how often a new window is calculated



Sliding Window Aggregation (3)

- Use the `functions.window` function to do window aggregation
- Parameters
 - `timeColumn`—reference to column containing time event occurred
 - `windowDuration` and `slideDuration`—window time length and how often windows are created
 - Specified as strings such as "1 minute" or "30 seconds"
 - `slideDuration` is optional
 - Default uses the `windowDuration` value

Example: Count Status Messages by Model per Window

- Count status messages for selected models for every 10 minute period
 - Update count every five minutes
- Use the timestamp column in status Kafka messages to group by window

```
kafkaDF = spark.readStream.format("kafka")...

from pyspark.sql.functions import *
modelsTimeDF = kafkaDF. \
    select("timestamp", \
        split(kafkaDF.value, ",") [0].alias("model"))

windowRoninCountsDF = modelsTimeDF. \
    where(modelsTimeDF.model.startswith("Ronin S")). \
    groupBy(window("timestamp", "10 minutes", \
        "5 minutes"), "model").count()

windowRoninCountsQuery = windowRoninCountsDF.writeStream. \
    outputMode("complete").format("console").start()
```

Language: Python

Windowed Model Count Example Output (1)

Batch: 1

window	model	count
[2019-05-28 01:35:00, 2019-05-28 01:45:00]	Ronin S3	1
[2019-05-28 01:35:40, 2019-05-28 01:45:00]	Ronin S1	3
[2019-05-28 01:40:00, 2019-05-28 01:50:00]	Ronin S1	3
[2019-05-28 01:40:00, 2019-05-28 01:50:00]	Ronin S3	2
[2019-05-28 01:45:00, 2019-05-28 01:55:00]	Ronin S3	1
[2019-05-28 02:00:00, 2019-05-28 02:10:00]	Ronin S1	1
[2019-05-28 02:00:00, 2019-05-28 02:10:00]	Ronin S3	1
[2019-05-28 02:05:00, 2019-05-28 02:15:00]	Ronin S1	1
[2019-05-28 02:05:00, 2019-05-28 02:15:00]	Ronin S3	1

Continued on next slide...

Windowed Model Count Example Output (2)

Batch: 2

window	model	count
[2019-05-28 01:35:00, 2019-05-28 01:45:00]	Ronin S3	1
[2019-05-28 01:35:40, 2019-05-28 01:45:00]	Ronin S1	3
[2019-05-28 01:40:00, 2019-05-28 01:50:00]	Ronin S3	2
[2019-05-28 01:45:00, 2019-05-28 01:55:00]	Ronin S3	1
[2019-05-28 01:50:00, 2019-05-28 02:00:00]	Ronin S1	3
[2019-05-28 01:55:00, 2019-05-28 02:05:00]	Ronin S3	1
[2019-05-28 02:00:00, 2019-05-28 02:10:00]	Ronin S1	3
[2019-05-28 02:00:00, 2019-05-28 02:10:00]	Ronin S3	2
[2019-05-28 02:00:00, 2019-05-28 02:10:00]	Ronin S2	1
[2019-05-28 02:05:00, 2019-05-28 02:15:00]	Ronin S1	1
[2019-05-28 02:05:00, 2019-05-28 02:15:00]	Ronin S3	2
[2019-05-28 02:15:00, 2019-05-28 02:25:00]	Ronin S1	3
[2019-05-28 02:15:00, 2019-05-28 02:25:00]	Ronin S3	1
[2019-05-28 02:20:00, 2019-05-28 02:50:00]	Ronin S1	2

Late Data

- **Sometimes Spark receives events “late”**
 - Late = occurring in a window earlier than the current one
- **Example:**
 - Current window: 02:15 - 02:25
 - New event time: 02:16 (on time)
 - New event time: 02:09 (late)
- **groupBy groups new events with appropriate window automatically**
 - But this requires Spark to maintain prior window states indefinitely
 - Can use a lot of memory

Watermarking

- **Watermarking sets a threshold of how late an event can be**
 - Beyond the threshold, events are ignored in aggregation
 - Spark does not need to maintain intermediate results past the watermark threshold

Supported Output Modes for Aggregation Queries

- Not all output modes can be used for all types of queries

Query Type	Supported Output Modes
Non-aggregation queries	append, update
Aggregation without watermarks	update, complete
Aggregation with watermarks	all

Example: Windowed Count with Watermarking

```
modelsTimeDF = ...  
  
watermarkRoninCountsDF = modelsTimeDF. \  
    where(modelsTimeDF.model.startswith("Ronin S")). \  
        withWatermark("timestamp", "1 minute"). \  
        groupBy(window("timestamp","10 seconds",  
            "5 seconds"),"model").count()  
  
watermarkRoninCountsQuery =  
    watermarkRoninCountsDF.writeStream. \  
        outputMode("complete").format("console"). \  
        option("truncate","false").start()
```

Language: Python

Chapter Topics

Aggregating and Joining Streaming DataFrames

- Streaming Aggregation
- **Joining Streaming DataFrames**
- Essential Points
- Hands-On Exercise: Aggregating and Joining Streaming DataFrames

Joining Streaming DataFrames

- Structured Streaming provides the ability to join a streaming DataFrame with
 - A static DataFrame
 - Another streaming DataFrame (added in Spark 2.3)

Streaming Join Limitations

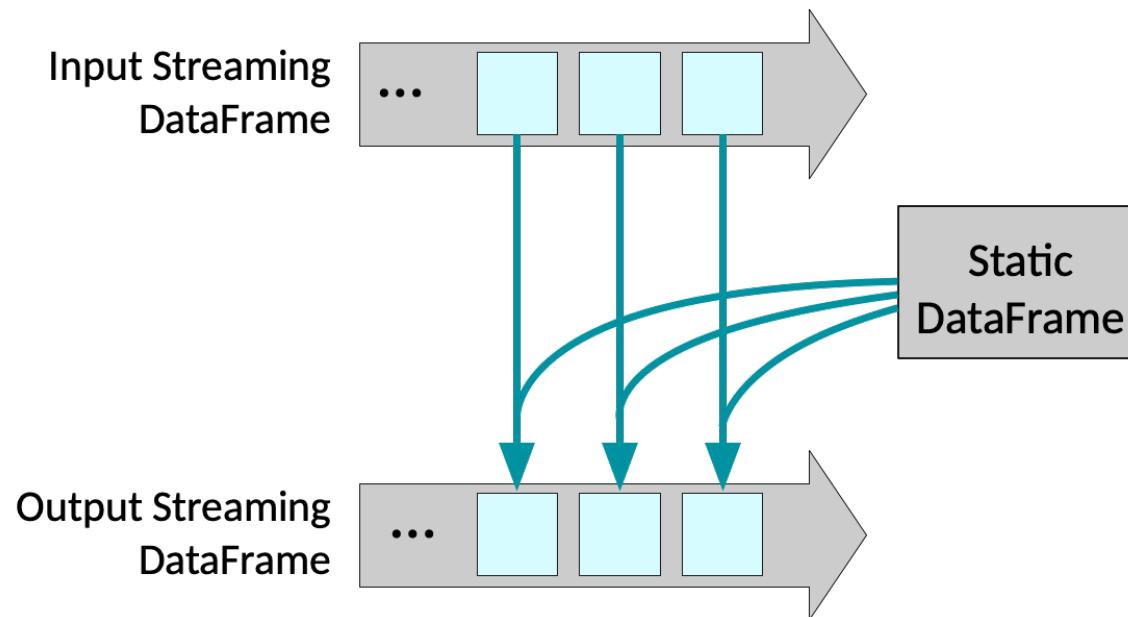
- Only append output mode is supported
- You cannot join an aggregated streaming DataFrame
 - Do not perform operations such as `groupBy` before a join operation
- Not all types of joins are supported for streaming joins

Supported Types of Joins

Left DataFrame	Right DataFrame	Supported Joins
streaming	static	inner left outer
static	streaming	inner right outer
streaming	streaming	inner left outer right outer

Static/Streaming Joins

- Each streaming micro-batch is joined with static DataFrame
- **Joins are not stateful**
 - Each calculation is based on a single micro-batch
 - No intermediate data is stored (buffered)



Example: Join Status Data with Account Data

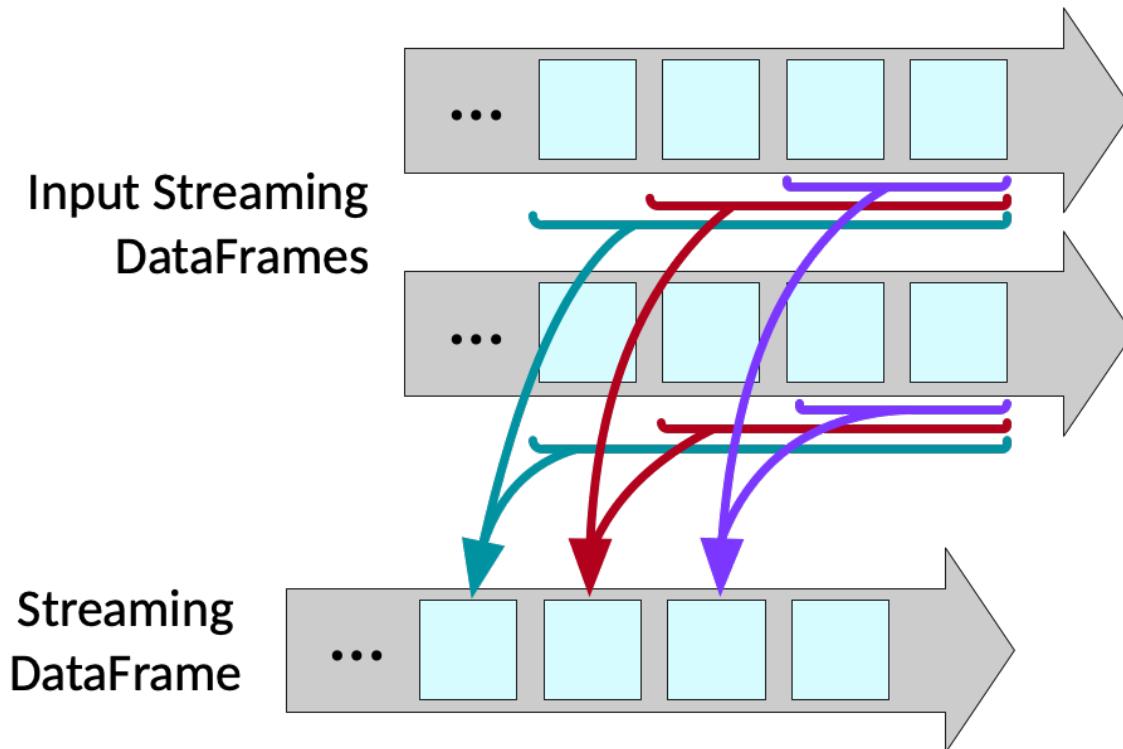
- `statusStreamDF`—streaming status messages including device IDs
- `accountDevStaticDF`—static data containing account numbers with IDs of the account's devices
- Output: status messages with account number of device

```
statusStreamDF = spark.readStream....  
accountDevStaticDF = spark.read....  
  
statusWithAccountDF = statusStreamDF. \  
    join(accountDevStaticDF,  
        accountDevStaticDF.account_device_id ==  
        statusStreamDF.device_id)  
  
statusWithAccountQuery = statusWithAccountDF. \  
    writeStream.outputMode("append"). \  
    format("console").start()
```

Language: Python

Streaming/Streaming Joins

- Join calculated using all available data from both DataFrames
 - Intermediate data is kept indefinitely by default



Example: Join Activation and Status Message Streams

- Join activation messages with status messages with the same device ID
 - Inner join (default)
 - Status messages with no activation record will be excluded

```
statusStreamDF = spark.readStream...
activationsStreamDF = spark.readStream...

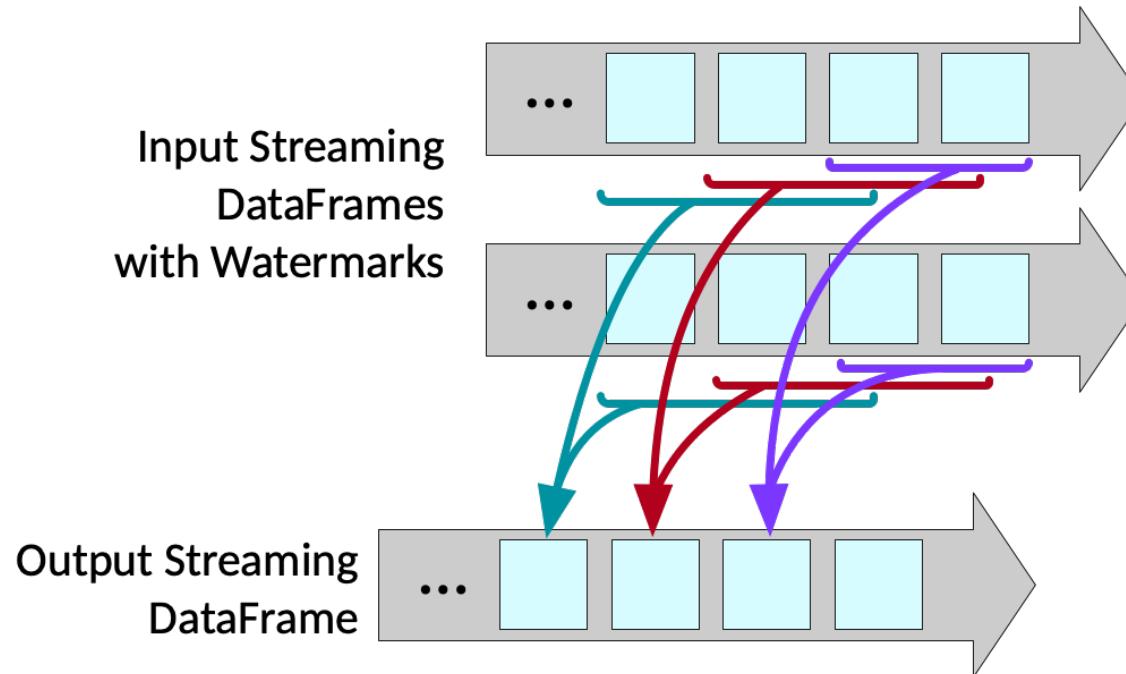
joinedDF = statusStreamDF.join(activationsStreamDF,"dev_id")

joinQuery = joinedDF.writeStream. \
    outputMode("append"). \
    format("console").start()
```

Language: Python

Streaming/Streaming Joins using Watermarking (1)

- Limit intermediate storage using watermarking
 - One or both input DataFrames may be watermarked



Streaming/Streaming Joins using Watermarking (2)

- **Watermarking is optional for inner joins**
- **Watermarking is required for left and right outer joins**
 - The “nullable” DataFrame must define a watermark
 - Left DataFrame for right joins, right DataFrame for left joins
 - The join condition must specify a time range
 - Using the watermarking timestamp column

Example: Use Left Outer Join with Two Streaming DataFrames

- **Left outer join on device IDs**
 - Left DataFrame: status messages, no watermark
 - Right DataFrame: activation messages, with watermark

```
statusStreamDF = spark.readStream...
activationsStreamDF = spark.readStream...

activationsStreamWDF = activationsStreamDF. \
    withWatermark("timestamp_act","10 minutes")

joinedDF = statusStreamDF.join(activationsStreamWDF,
    (activationsStreamWDF.dev_id ==
        statusStreamDF.dev_id)
    & (statusStreamDF.timestamp_status >
        activationsStreamWDF.timestamp_act),
    "left_outer")

joinQuery = ...
```

Language: Python

Chapter Topics

Aggregating and Joining Streaming DataFrames

- Streaming Aggregation
- Joining Streaming DataFrames
- **Essential Points**
- Hands-On Exercise: Aggregating and Joining Streaming DataFrames

Essential Points

- **Streaming aggregation operations calculate results across multiple micro-batches**
- **Windowed aggregations use data from events occurring within a specified time period**
- **Watermarks define the maximum time threshold to include “late” events in results**
 - Limits the amount of intermediate data Spark must buffer
- **Streaming join operations join data in a streaming DataFrame with data in either a static or streaming DataFrame**

Chapter Topics

Aggregating and Joining Streaming DataFrames

- Streaming Aggregation
- Joining Streaming DataFrames
- Essential Points
- **Hands-On Exercise: Aggregating and Joining Streaming DataFrames**

Hands-On Exercise: Aggregating and Joining Streaming DataFrames

- In this exercise, you will practice aggregating and joining streaming DataFrames
- Please refer to the Hands-On Exercise Manual for instructions



Conclusion

Chapter 20

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Message Processing with Apache Kafka

Course Objectives

During this course, you have learned

- How the Apache Hadoop ecosystem fits in with the data processing lifecycle
- How data is distributed, stored, and processed in a Hadoop cluster
- How to write, configure, and deploy Apache Spark applications on a Hadoop cluster
- How to use the Spark shell and Spark applications to explore, process, and analyze distributed data
- How to query data using Spark SQL, DataFrames, and Datasets
- How to use Spark Streaming to process a live data stream

Which Course to Take Next

- **For developers**
 - *Cloudera Search Training*
 - *Cloudera Training for Apache HBase*
- **For system administrators**
 - *Cloudera Administrator Training for Apache Hadoop*
 - *Cloudera Security Training*
- **For data analysts and data scientists**
 - *Cloudera Data Analyst Training*
 - *Cloudera Data Scientist Training*



Message Processing with Apache Kafka

Appendix A

Course Chapters

- Introduction
- Introduction to Apache Hadoop and the Hadoop Ecosystem
- Apache Hadoop File Storage
- Distributed Processing on an Apache Hadoop Cluster
- Apache Spark Basics
- Working with DataFrames and Schemas
- Analyzing Data with DataFrame Queries
- RDD Overview
- Transforming Data with RDDs
- Aggregating Data with Pair RDDs
- Querying Tables and Views with SQL
- Working with Datasets in Scala
- Writing, Configuring, and Running Spark Applications
- Spark Distributed Processing
- Distributed Data Persistence
- Common Patterns in Spark Data Processing
- Introduction to Structured Streaming
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- **Appendix: Message Processing with Apache Kafka**

Message Processing with Apache Kafka

In this appendix, you will learn

- **What Apache Kafka is and what advantages it offers**
- **About the high-level architecture of Kafka**
- **How to create topics, publish messages, and read messages from the command line**

Chapter Topics

Message Processing with Apache Kafka

- **What Is Apache Kafka?**
- Apache Kafka Overview
- Scaling Apache Kafka
- Apache Kafka Cluster Architecture
- Apache Kafka Command Line Tools
- Essential Points
- Hands-On Exercise: Producing and Consuming Apache Kafka Messages

What Is Apache Kafka?

- **Apache Kafka is a distributed commit log service**
 - Widely used for data ingest
 - Conceptually similar to a publish-subscribe messaging system
 - Offers scalability, performance, reliability, and flexibility
- **Originally created at LinkedIn, now an open source Apache project**
 - Donated to the Apache Software Foundation in 2011
 - Graduated from the Apache Incubator in 2012
 - Supported by Cloudera for production use with CDH in 2015



Characteristics of Kafka

- **Scalable**
 - Kafka is a distributed system that supports multiple nodes
- **Fault-tolerant**
 - Data is persisted to disk and can be replicated throughout the cluster
- **High throughput**
 - Each broker can process hundreds of thousands of messages per second*
- **Low latency**
 - Data is delivered in a fraction of a second
- **Flexible**
 - Decouples the production of data from its consumption

*Using modest hardware, with messages of a typical size

Kafka Use Cases

- **Kafka is used for a variety of use cases, such as**
 - Log aggregation
 - Messaging
 - Web site activity tracking
 - Stream processing
 - Event sourcing

Chapter Topics

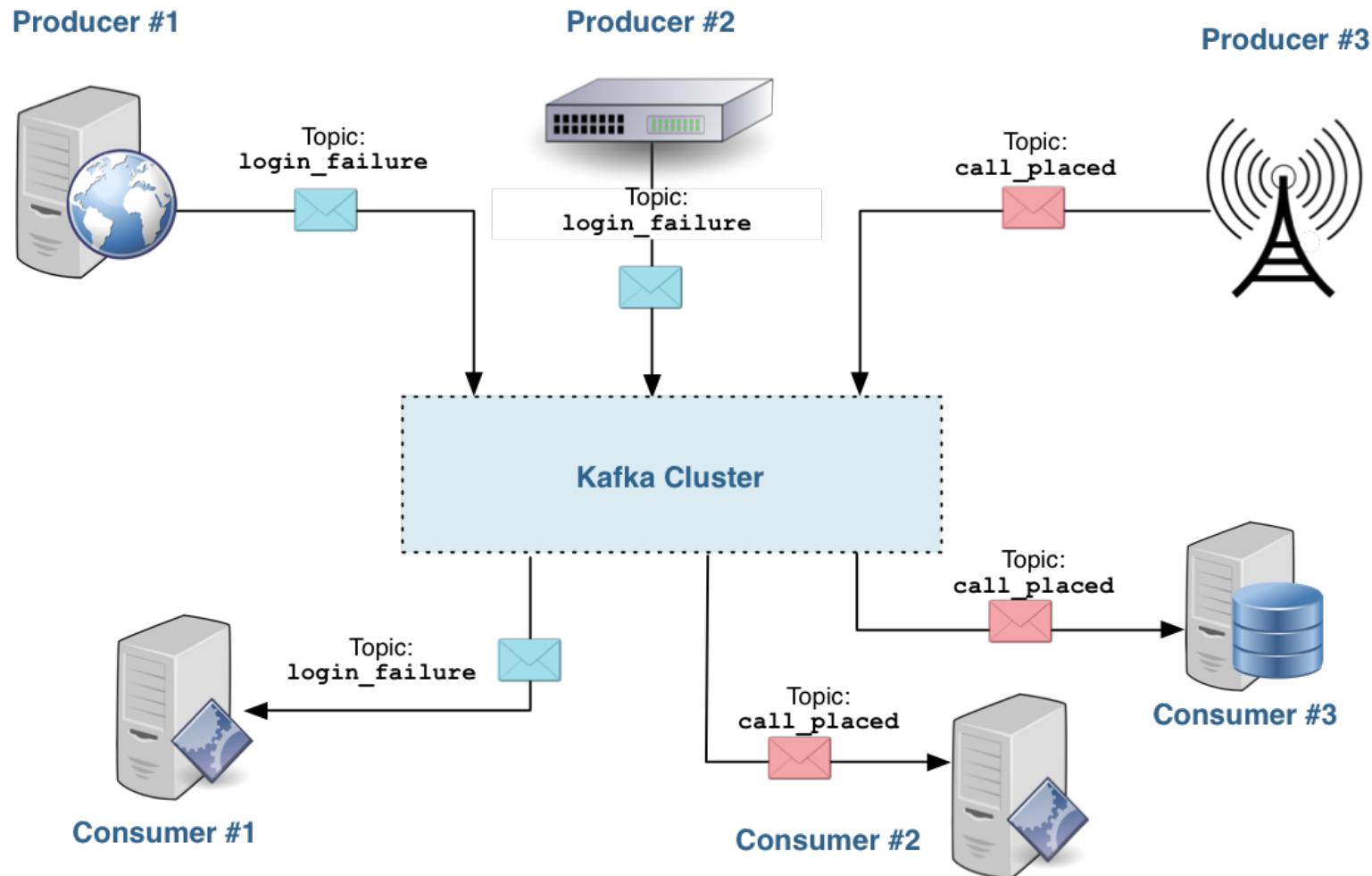
Message Processing with Apache Kafka

- What Is Apache Kafka?
- **Apache Kafka Overview**
- Scaling Apache Kafka
- Apache Kafka Cluster Architecture
- Apache Kafka Command Line Tools
- Essential Points
- Hands-On Exercise: Producing and Consuming Apache Kafka Messages

Key Terminology

- **Message**
 - A single data record passed by Kafka
- **Topic**
 - A named log or feed of messages within Kafka
- **Producer**
 - A program that writes messages to Kafka
- **Consumer**
 - A program that reads messages from Kafka

Example: High-Level Architecture



Messages (1)

- **Messages in Kafka are variable-size byte arrays**
 - Represent arbitrary user-defined content
 - Use any format your application requires
 - Common formats include free-form text, JSON, and Avro
- **There is no explicit limit on message size**
 - Optimal performance at a few KB per message
 - Practical limit of 1MB per message

Messages (2)

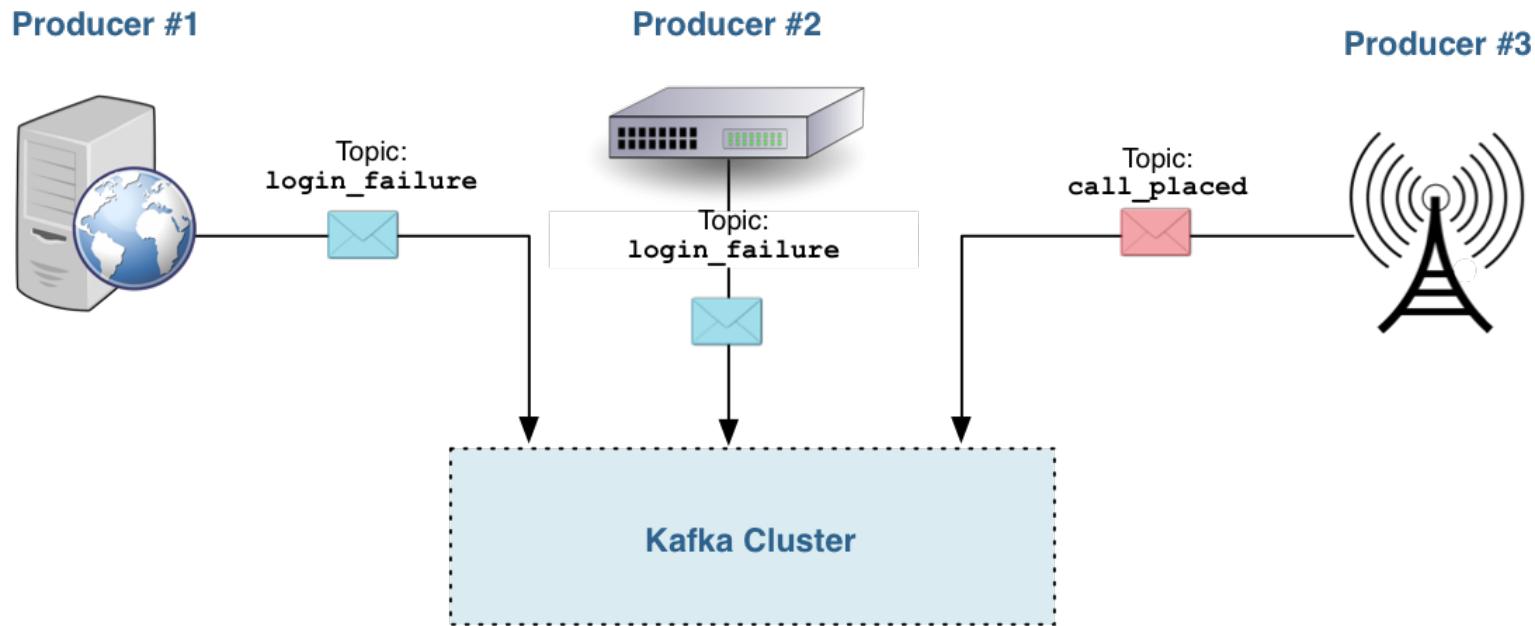
- **Kafka retains all messages for a defined time period and/or total size**
 - Administrators can specify retention on global or per-topic basis
 - Kafka will retain messages regardless of whether they were read
 - Kafka discards messages automatically after the retention period or total size is exceeded (whichever limit is reached first)
 - Default retention is one week
 - Retention can reasonably be one year or longer

Topics

- **There is no explicit limit on the number of topics**
 - However, Kafka works better with a few large topics than many small ones
- **A topic can be created explicitly or simply by publishing to the topic**
 - This behavior is configurable
 - Cloudera recommends that administrators disable auto-creation of topics to avoid accidental creation of large numbers of topics

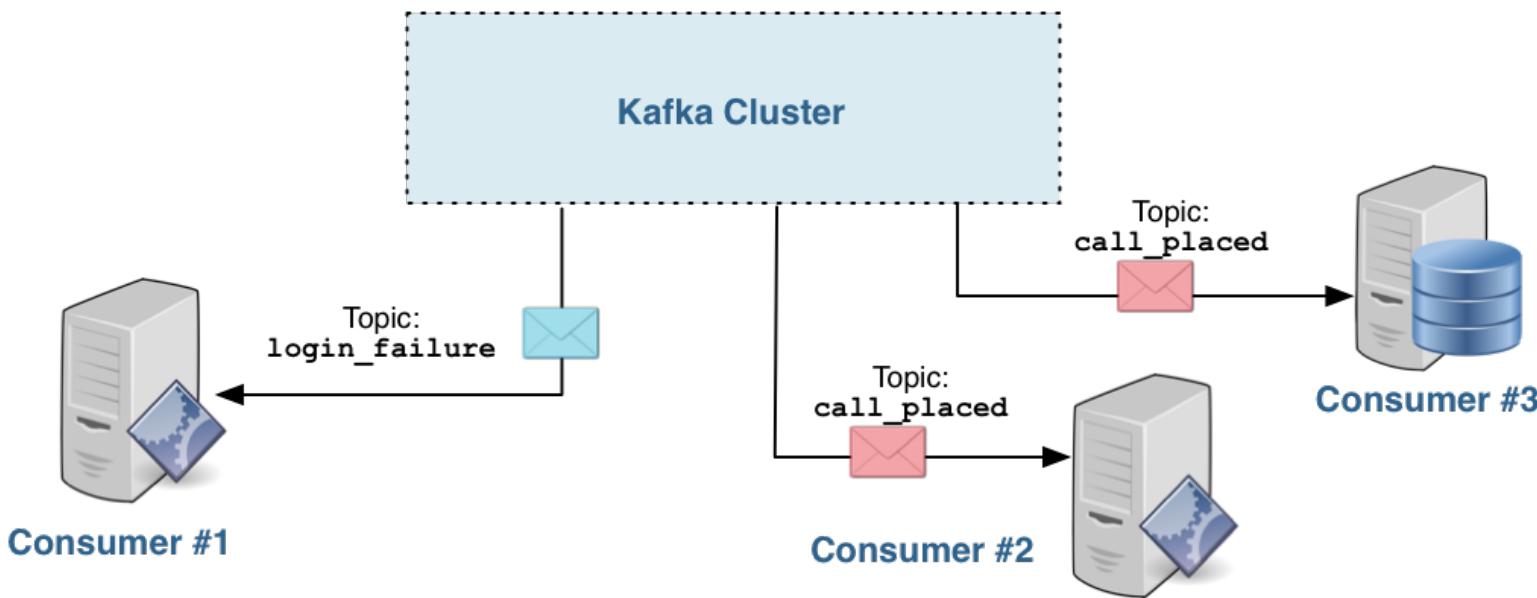
Producers

- Producers publish messages to Kafka topics
 - They communicate with Kafka, not a consumer
 - Kafka persists messages to disk on receipt



Consumers

- A consumer reads messages that were published to Kafka topics
 - They communicate with Kafka, not any producer
- Consumer actions do not affect other consumers
 - For example, having one consumer display the messages in a topic as they are published does not change what is consumed by other consumers
- They can come and go without impact on the cluster or other consumers



Producers and Consumers

- **Tools available as part of Kafka**
 - Command-line producer and consumer tools
 - Client (producer and consumer) Java APIs
- **A growing number of other APIs are available from third parties**
 - Client libraries in many languages including Python, PHP, C/C++, Go, .NET, and Ruby
- **Integrations with other tools and projects include**
 - Apache Flume
 - Apache Spark
 - Amazon AWS
 - syslog
- **Kafka also has a large and growing ecosystem**

Chapter Topics

Message Processing with Apache Kafka

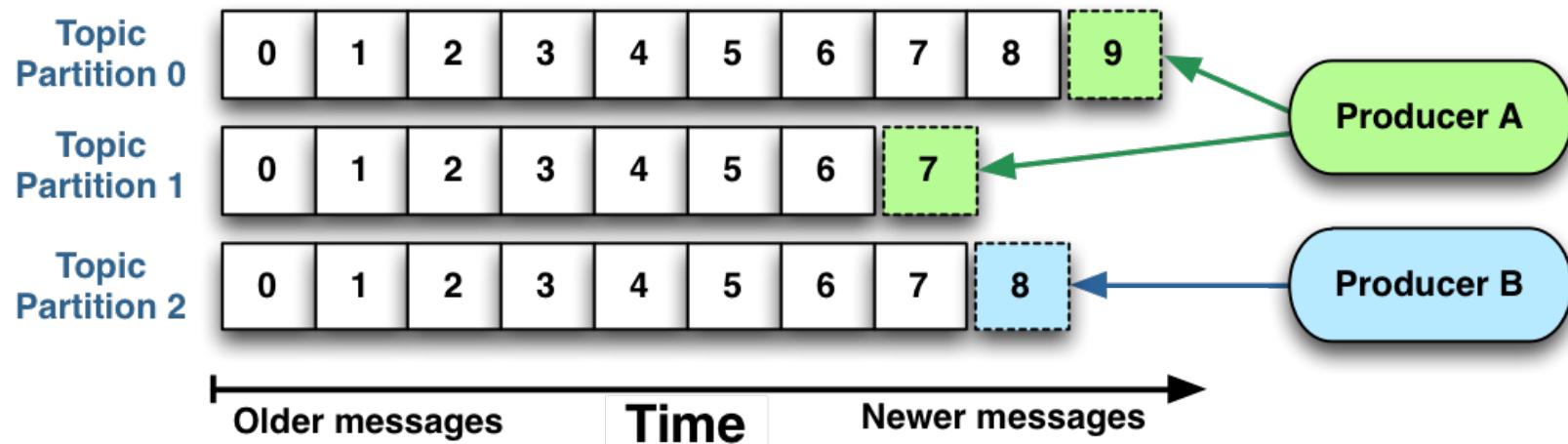
- What Is Apache Kafka?
- Apache Kafka Overview
- **Scaling Apache Kafka**
- Apache Kafka Cluster Architecture
- Apache Kafka Command Line Tools
- Essential Points
- Hands-On Exercise: Producing and Consuming Apache Kafka Messages

Scaling Kafka

- **Scalability is one of the key benefits of Kafka**
- **Two features let you scale Kafka for performance**
 - Topic partitions
 - Consumer groups

Topic Partitioning

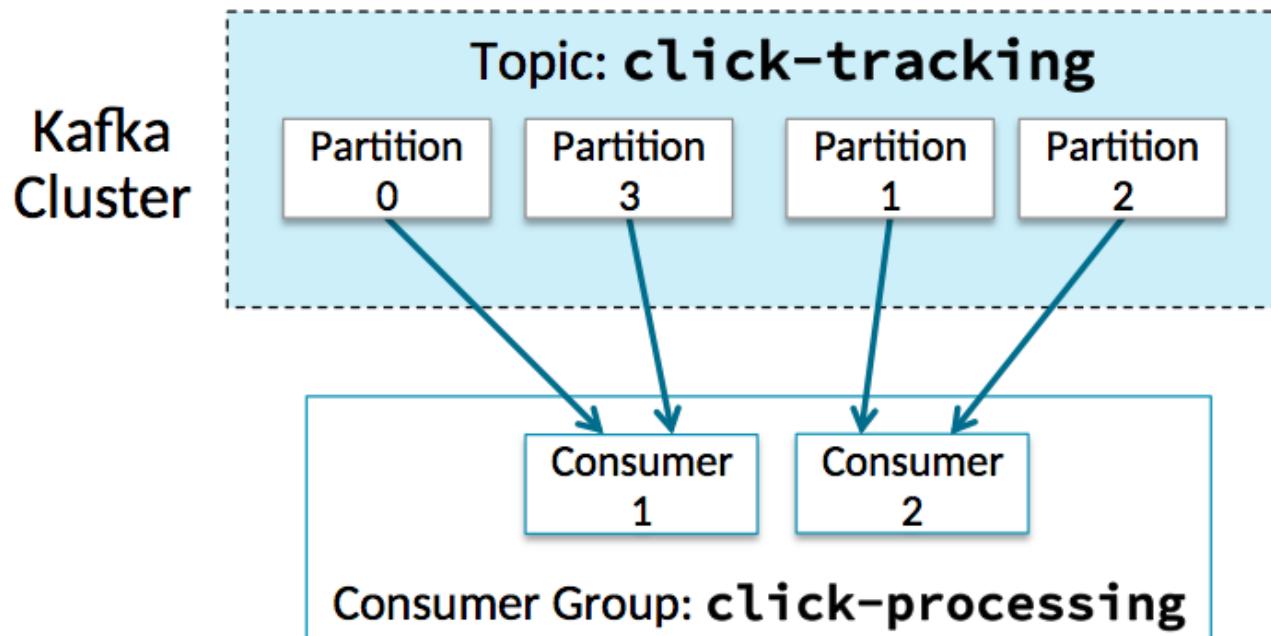
- Kafka divides each topic into some number of partitions*
 - Topic partitioning improves scalability and throughput
- A topic partition is an ordered and immutable sequence of messages
 - New messages are appended to the partition as they are received
 - Each message is assigned a unique sequential ID known as an offset



*Note that this is unrelated to partitioning in HDFS or Spark

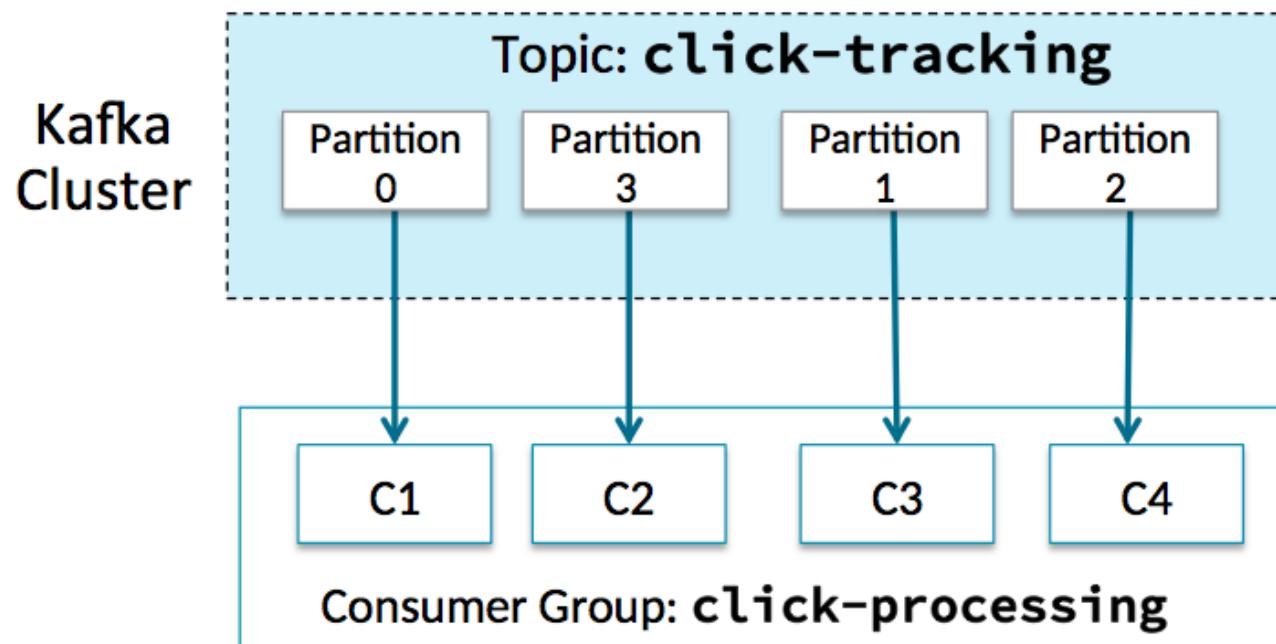
Consumer Groups

- One or more consumers can form their own consumer group that work together to consume the messages in a topic
- Each partition is consumed by only one member of a consumer group
- Message ordering is preserved per partition, but not across the topic



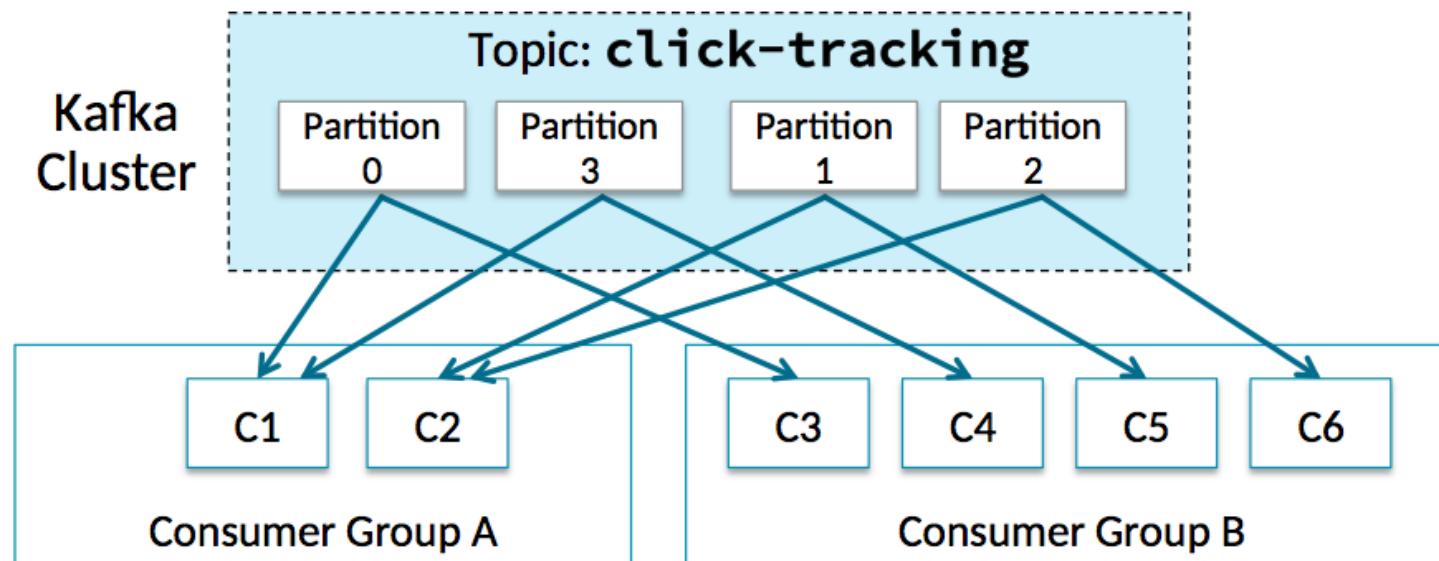
Increasing Consumer Throughput

- Additional consumers can be added to scale consumer group processing
- Consumer instances that belong to the same consumer group can be in separate processes or on separate machines



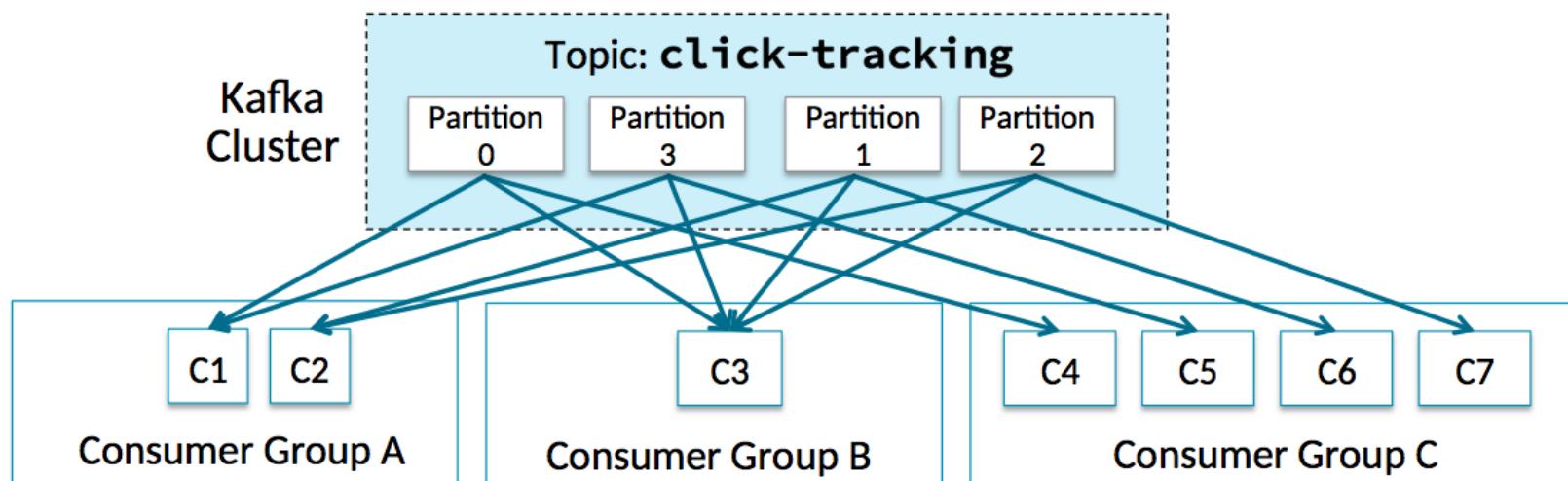
Multiple Consumer Groups

- Each message published to a topic is delivered to one consumer instance within each subscribing consumer group
- Kafka scales to large numbers of consumer groups and consumers



Publish and Subscribe to Topic

- Kafka functions like a traditional queue when all consumer instances belong to the same consumer group
 - In this case, a given message is received by one consumer
- Kafka functions like traditional publish-subscribe when each consumer instance belongs to a different consumer group
 - In this case, all messages are broadcast to all consumer groups



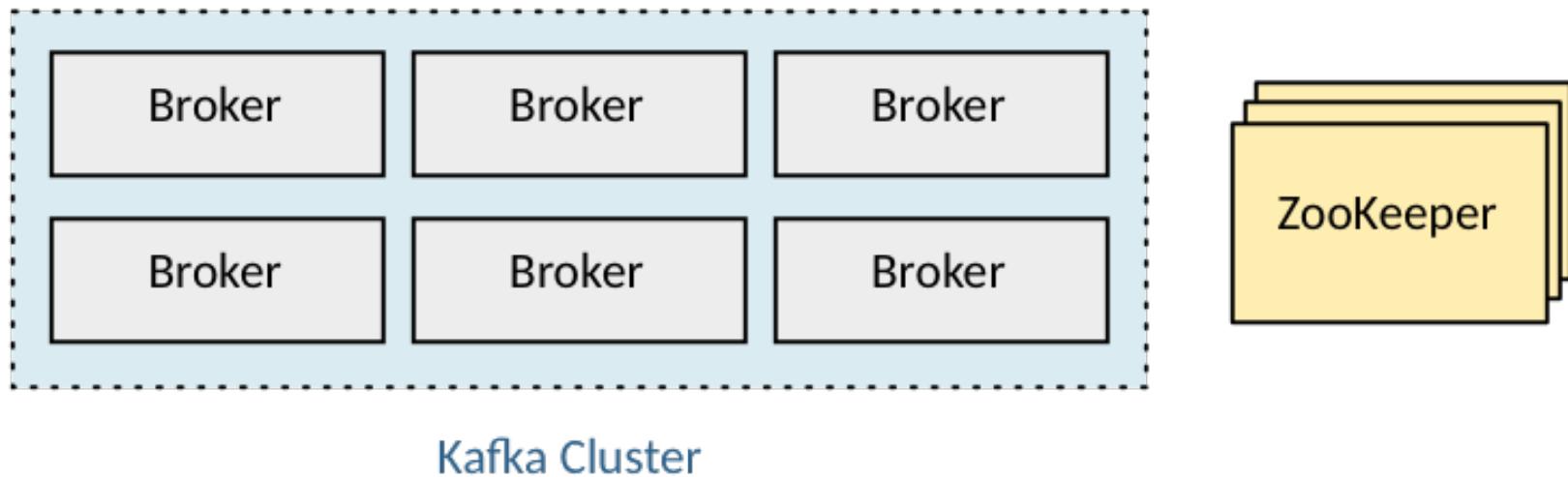
Chapter Topics

Message Processing with Apache Kafka

- What Is Apache Kafka?
- Apache Kafka Overview
- Scaling Apache Kafka
- **Apache Kafka Cluster Architecture**
- Apache Kafka Command Line Tools
- Essential Points
- Hands-On Exercise: Producing and Consuming Apache Kafka Messages

Kafka Clusters

- A Kafka cluster consists of one or more *brokers*—servers running the Kafka broker daemon
- Kafka depends on the Apache ZooKeeper service for coordination



Apache ZooKeeper

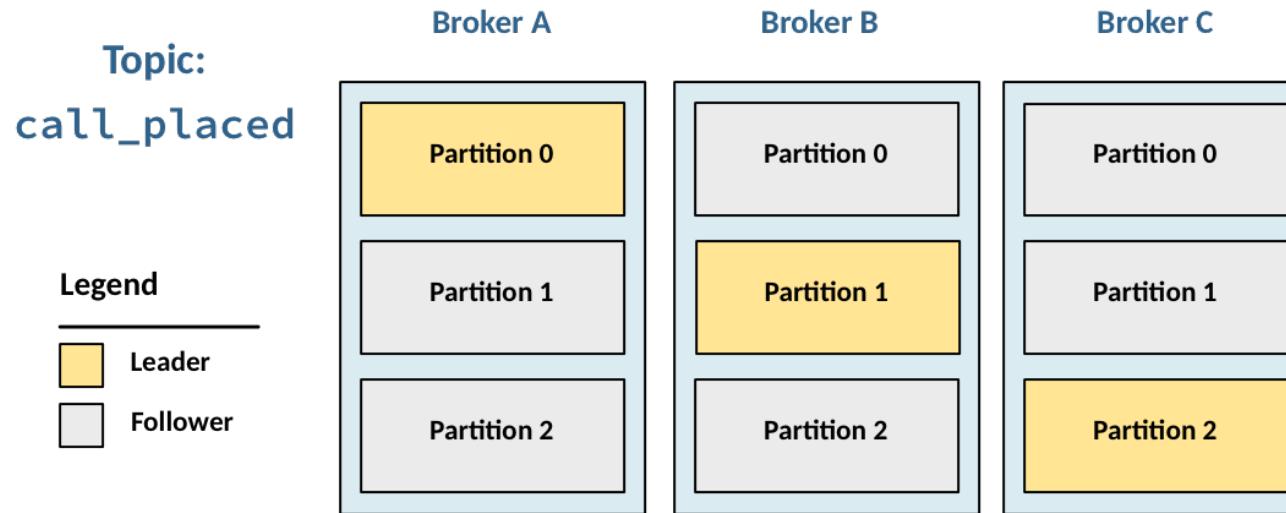
- Apache ZooKeeper is a coordination service for distributed applications
- Kafka depends on the ZooKeeper service for coordination
 - Typically running three or five ZooKeeper instances
- Kafka uses ZooKeeper to keep track of brokers running in the cluster
- Kafka uses ZooKeeper to detect the addition or removal of consumers

Kafka Brokers

- **Brokers are the fundamental daemons that make up a Kafka cluster**
- **A broker fully stores a topic partition on disk, with caching in memory**
- **A single broker can reasonably host 1000 topic partitions**
- **One broker is elected controller of the cluster (for assignment of topic partitions to brokers, and so on)**
- **Each broker daemon runs in its own JVM**
 - A single machine can run multiple broker daemons

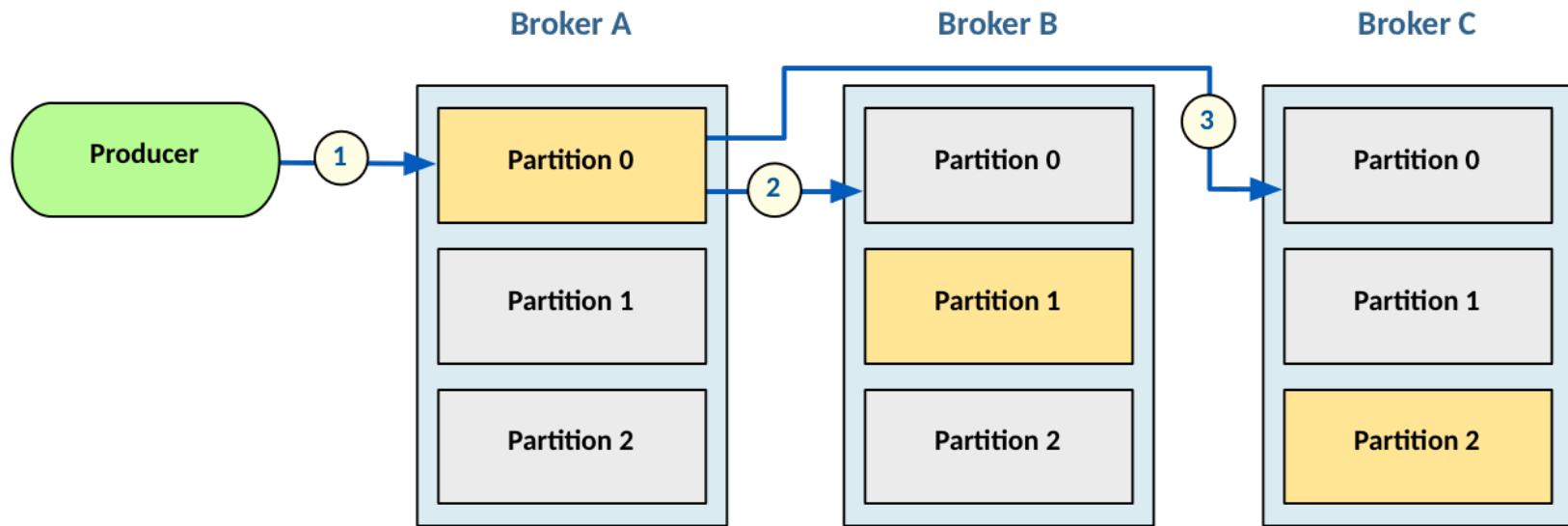
Topic Replication

- At topic creation, a topic can be set with a replication count
 - Doing so is recommended, as it provides fault tolerance
- Each broker can act as a leader for some topic partitions and a follower for others
 - Followers passively replicate the leader
 - If the leader fails, a follower will automatically become the new leader



Messages Are Replicated

- Configure the producer with a list of one or more brokers
 - The producer asks the first available broker for the leader of the desired topic partition
- The producer then sends the message to the leader
 - The leader writes the message to its local log
 - Each follower then writes the message to its own log
 - After acknowledgements from followers, the message is committed



Chapter Topics

Message Processing with Apache Kafka

- What Is Apache Kafka?
- Apache Kafka Overview
- Scaling Apache Kafka
- Apache Kafka Cluster Architecture
- **Apache Kafka Command Line Tools**
- Essential Points
- Hands-On Exercise: Producing and Consuming Apache Kafka Messages

Creating Topics from the Command Line

- Kafka includes a convenient set of command line tools
 - These are helpful for exploring and experimentation
- The `kafka-topics` command offers a simple way to create Kafka topics
 - Provide the topic name of your choice, such as `device_status`
 - You must also specify the ZooKeeper connection string for your cluster

```
$ kafka-topics --create \  
--zookeeper zkhost1:2181,zkhost2:2181,zkhost3:2181 \  
--replication-factor 3 \  
--partitions 5 \  
--topic device_status
```

Displaying Topics from the Command Line

- Use the `--list` option to list all topics

```
$ kafka-topics --list \
--zookeeper zkhost1:2181,zkhost2:2181,zkhost3:2181
```

- Use the `--help` option to list all `kafka-topics` options

```
$ kafka-topics --help
```

Running a Producer from the Command Line (1)

- You can run a producer using the `kafka-console-producer` tool
- Specify one or more brokers in the `--broker-list` option
 - Each broker consists of a hostname, a colon, and a port number
 - If specifying multiple brokers, separate them with commas
- You must also provide the name of the topic

```
$ kafka-console-producer \
  --broker-list brokerhost1:9092,brokerhost2:9092 \
  --topic device_status
```

Running a Producer from the Command Line (2)

- You may see a few log messages in the terminal after the producer starts
- The producer will then accept input in the terminal window
 - Each line you type will be a message sent to the topic
- Until you have configured a consumer for this topic, you will see no other output from Kafka

Writing File Contents to Topics Using the Command Line

- **Using UNIX pipes or redirection, you can read input from files**
 - The data can then be sent to a topic using the command line producer
- **This example shows how to read input from a file named alerts.txt**
 - Each line in this file becomes a separate message in the topic

```
$ cat alerts.txt | kafka-console-producer \
  --broker-list brokerhost1:9092,brokerhost2:9092 \
  --topic device_status
```

- **This technique can be an easy way to integrate with existing programs**

Running a Consumer from the Command Line

- You can run a consumer with the `kafka-console-consumer` tool
- This requires the ZooKeeper connection string for your cluster
 - Unlike starting a producer, which instead requires a list of brokers
- The command also requires a topic name
- Use `--from-beginning` to read *all* available messages
 - Otherwise, it reads only new messages

```
$ kafka-console-consumer \
  --bootstrap-server brokerhost:9092 \
  --topic device_status \
  --from-beginning
```

Chapter Topics

Message Processing with Apache Kafka

- What Is Apache Kafka?
- Apache Kafka Overview
- Scaling Apache Kafka
- Apache Kafka Cluster Architecture
- Apache Kafka Command Line Tools
- **Essential Points**
- Hands-On Exercise: Producing and Consuming Apache Kafka Messages

Essential Points

- Producers publish messages to categories called topics
- Messages in a topic are read by consumers
- Topics are divided into partitions for performance and scalability
 - These partitions are replicated for fault tolerance
- Consumer groups work together to consume the messages in a topic
- Nodes running the Kafka service are called brokers
- Kafka includes command-line tools for managing topics, and for starting producers and consumers

Bibliography

The following offer more information on topics discussed in this chapter

- The Apache Kafka web site
 - <http://kafka.apache.org/>
- Real-Time Fraud Detection Architecture
 - <http://tiny.cloudera.com/kmc01a>
- Kafka Reference Architecture
 - <http://tiny.cloudera.com/kmc01b>
- The Log: What Every Software Engineer Should Know...
 - <http://tiny.cloudera.com/kmc01c>

Chapter Topics

Message Processing with Apache Kafka

- What Is Apache Kafka?
- Apache Kafka Overview
- Scaling Apache Kafka
- Apache Kafka Cluster Architecture
- Apache Kafka Command Line Tools
- Essential Points
- **Hands-On Exercise: Producing and Consuming Apache Kafka Messages**

Appendix Hands-On Exercise: Producing and Consuming Apache Kafka Messages

- In this exercise, you will use the Kafka command-line tools to pass messages between a producer and consumer
- Please refer to the Hands-On Exercise Manual for instructions