

# Cloudera Data Engineering: Developing Applications with Apache Spark



## Introduction

---

### Chapter 1

# Course Chapters

- **Introduction**
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

## 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 Hadoop

Apache NiFi

Apache Spark

Apache Avro

Apache HBase

Apache Oozie

Apache Sqoop

Apache Airflow

Apache Hive

Apache ORC

Apache Tez

Apache Atlas

Apache Impala

Apache Parquet

Apache Zeppelin

Apache Bigtop

Apache Kafka

Apache Phoenix

Apache ZooKeeper

Apache Druid

Apache Knox

Apache Ranger

Apache Flink

Apache Kudu

Apache Solr

- 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

# About This Course

---

- **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

# Agenda

---

Day 1	Day 2	Day 3	Day 4
Class Introduction	Spark DataFrames	Integration with Hive	Introduction to Streaming DataFrames
Zeppelin Introduction	Reading DataFrames	Visualization with Zeppelin	Kafka Introduction
HDFS Introduction	Working with Columns	Distributed Processing	Integration with Kafka
YARN Introduction	Transforming DataFrames	Distributed Persistence	Aggregating and Joining Streaming DataFrames
Distributed Processing History	Working with UDFs	Building Spark Applications	(*) Appendix: Scala Datasets
Spark RDDs	Working with Windows		

(\*) if time allows

# 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?
  - What is your experience with Spark and 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

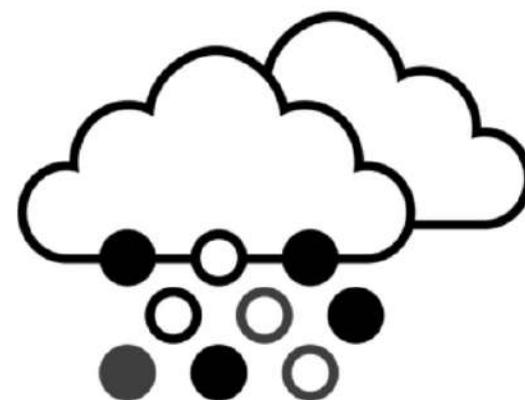
- About This Course
- Introductions
- **About Cloudera**
- About Cloudera Educational Services
- Course Logistics



- **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
- **In 2021 Cloudera was acquired to become a private company**

# CLOUDERA

THE ENTERPRISE DATA CLOUD COMPANY



Any Cloud



Data Lifecycle

CLOUDERA  
SDX



Secure & Governed

Open

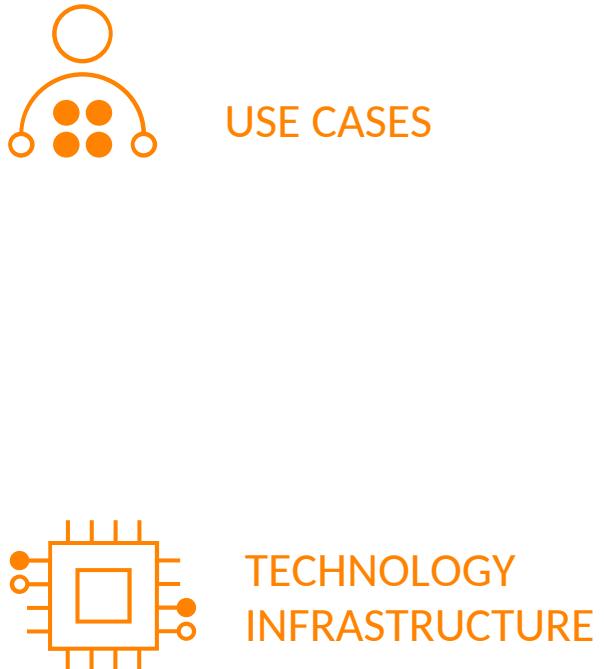
# Considerations For Data Are Evolving (1)

## The new realities of managing data and workloads across clouds

### Decade 1

#### Hadoop on-prem and on the cloud

- Need to efficiently store & process data
- Batch process “big data”
- Co-locate compute and storage to use commodity hardware and avoid costly network transfers



### Decade 2

#### Hadoop powered data clouds

- Need to integrate the entire lifecycle
- Industrialize data-driven decision making
- High performance analytics with remote disaggregated storage with memory and SSD caching

# Considerations For Data Are Evolving (2)

## Decade 1

Hadoop on-prem and on the cloud

- Deploy software in months and quarters
- Network perimeter & physical access controls are the norm
- Simplicity over robust mechanisms



USER EXPERIENCE



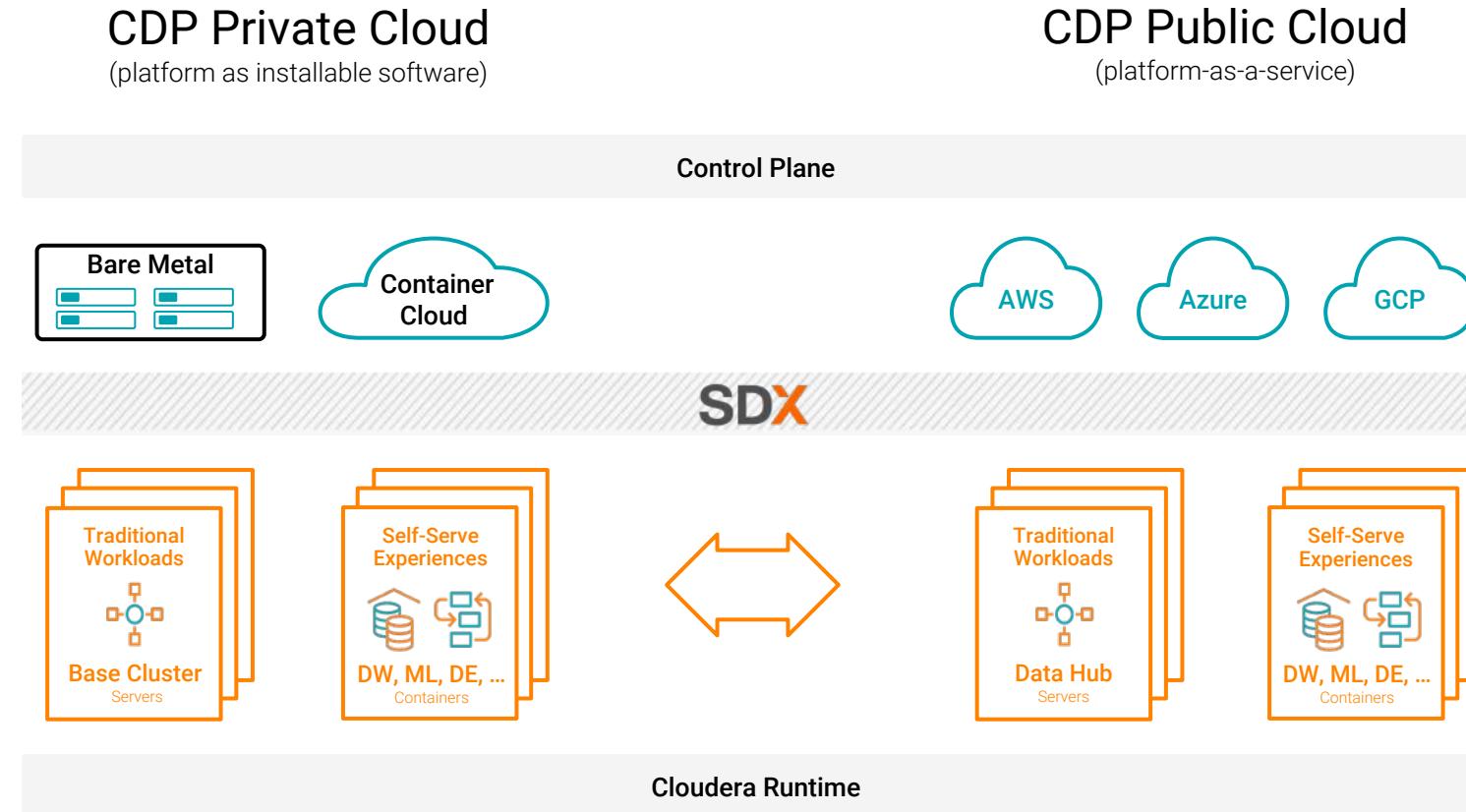
PRIVACY,  
SECURITY &  
GOVERNANCE

## Decade 2

Hadoop-powered data clouds

- Spin up services in minutes
- Security at the workload, data, and metadata layer
- Solutions for new regulations (GDPR)

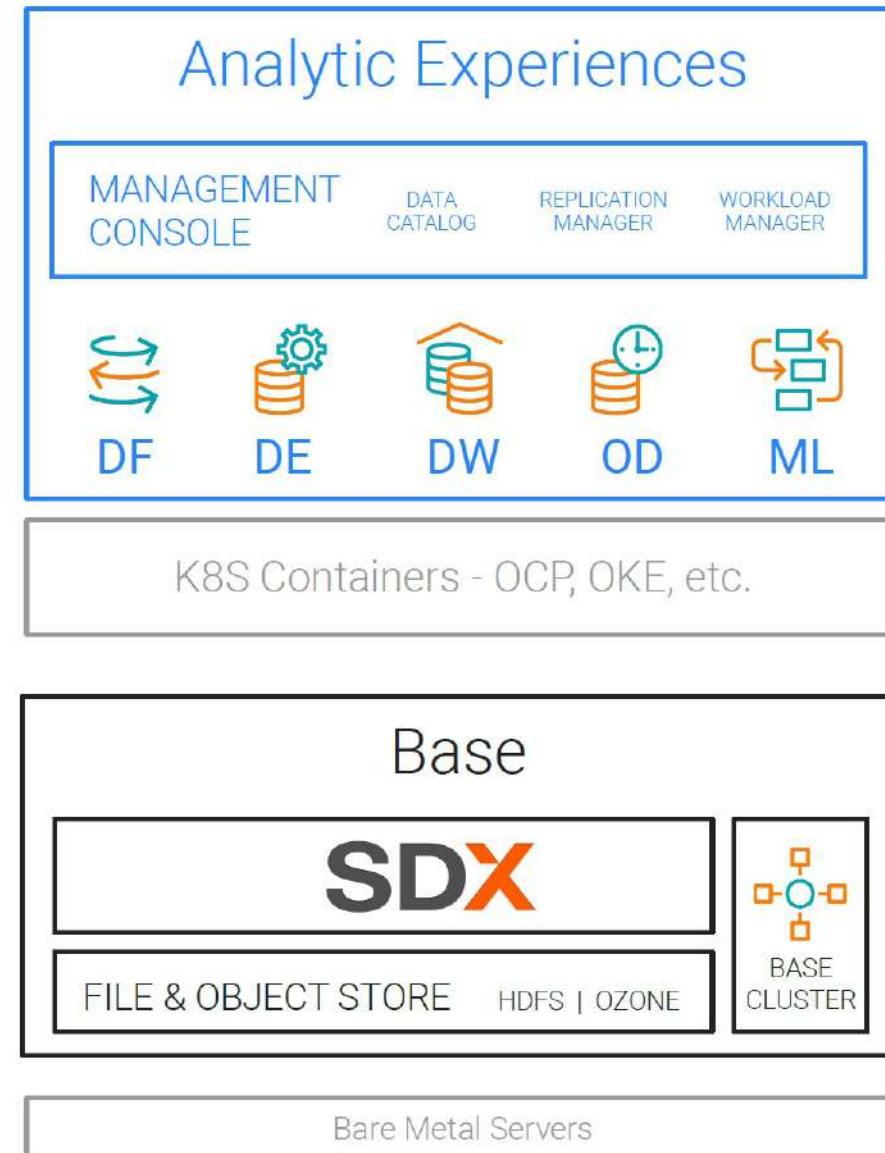
# Cloudera Data Platform



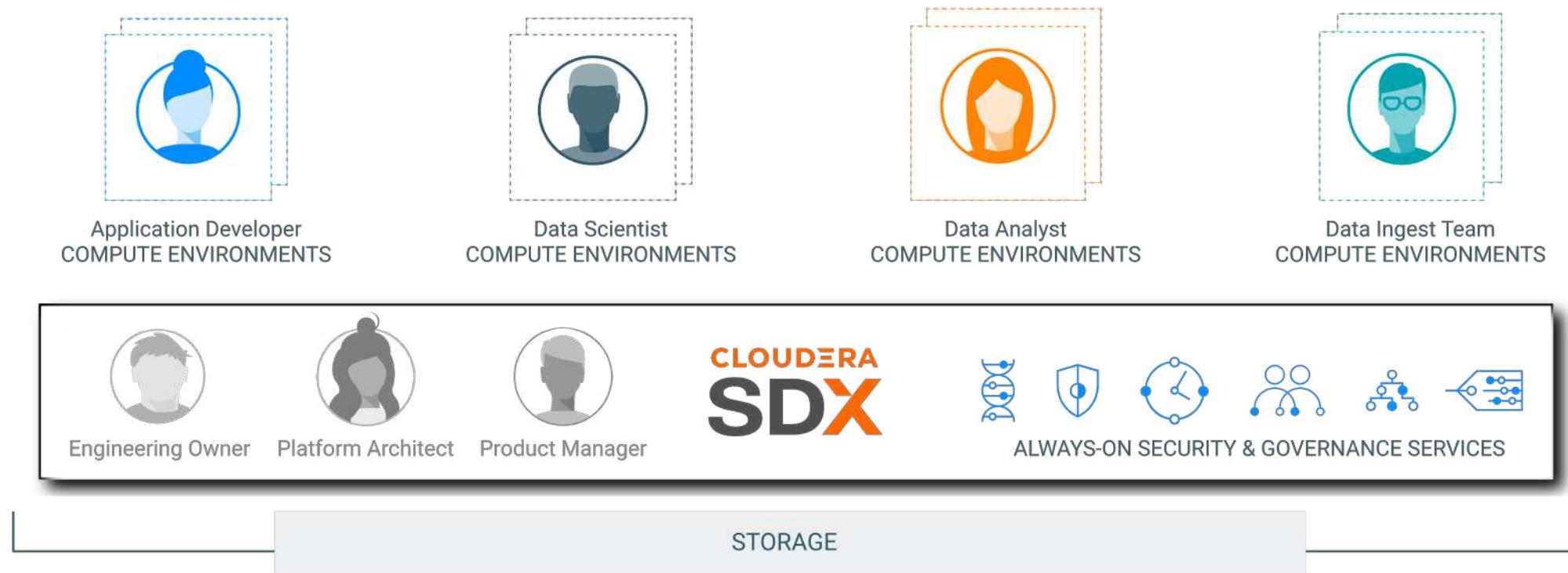
- A suite of products to collect, curate, report, serve, and predict
  - Cloud native or bare metal deployment
  - Powered by open source
  - Analytics from the Edge to AI
  - Unified data control plane
  - Shared Data Experience (SDX)

# CDP Private Cloud

- Cloud-native architecture with containerized Experiences and Base cluster foundation
- CDP Private Base is identical to and supersedes CDP Data Center
- Admin and user experience is consistent across CDP Private & CDP Public for true hybrid cloud



# 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**

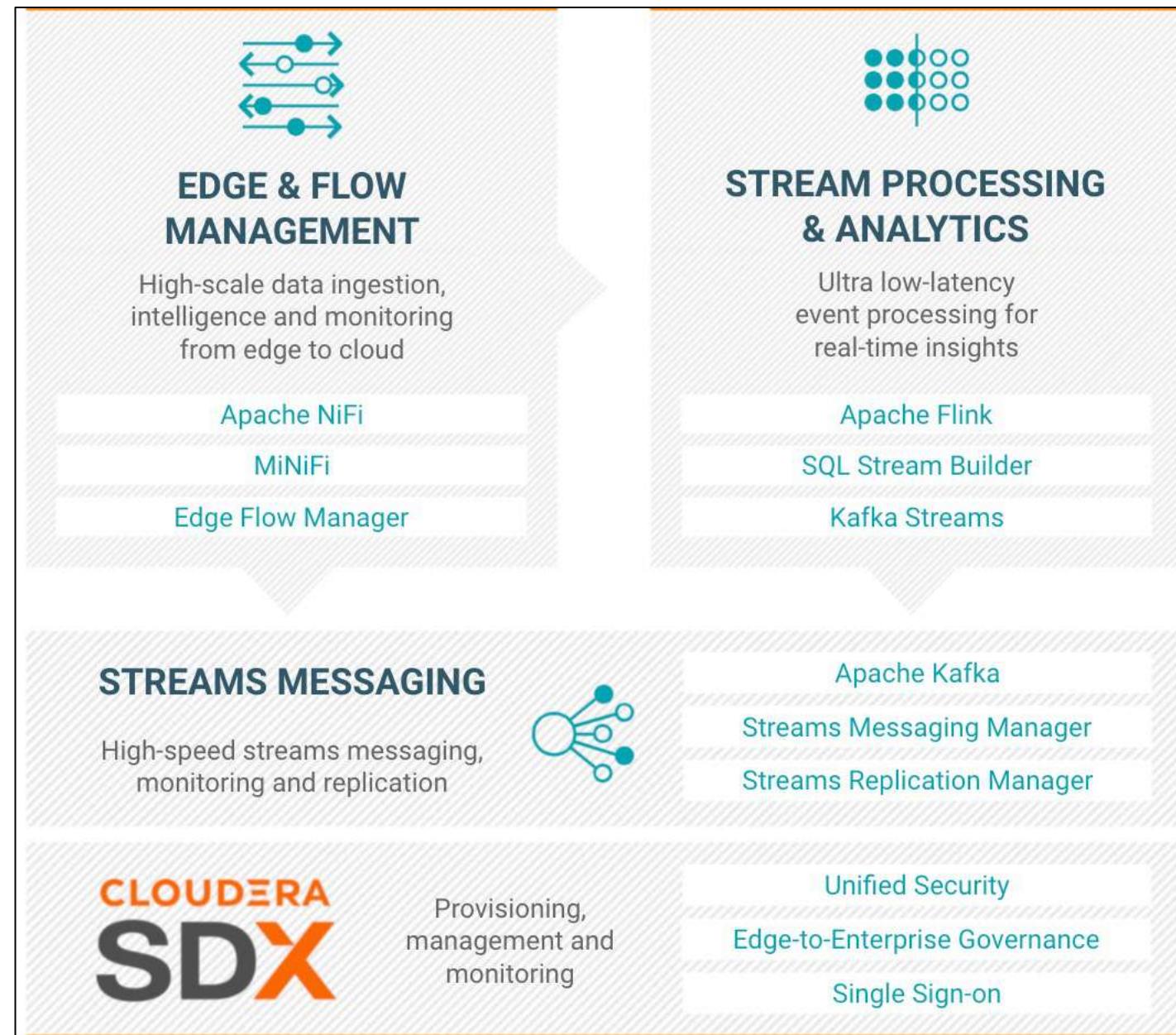
# Self-Serve Experiences for Cloud Form Factors

- 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

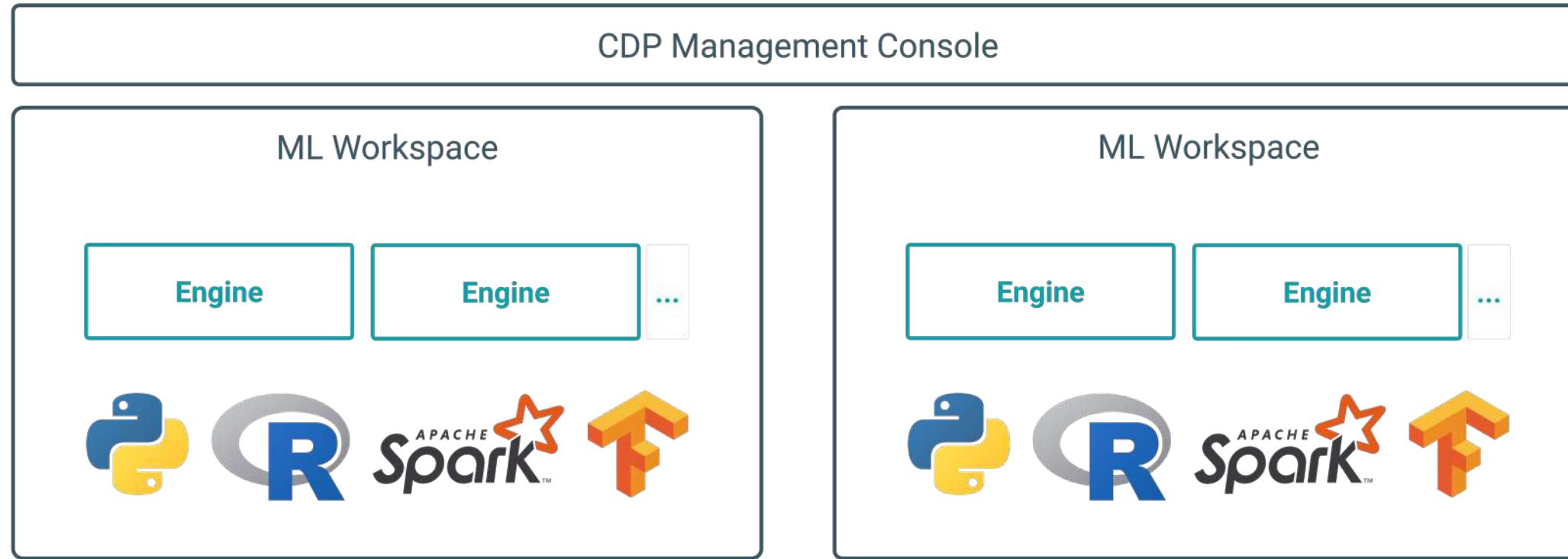


# Cloudera DataFlow

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

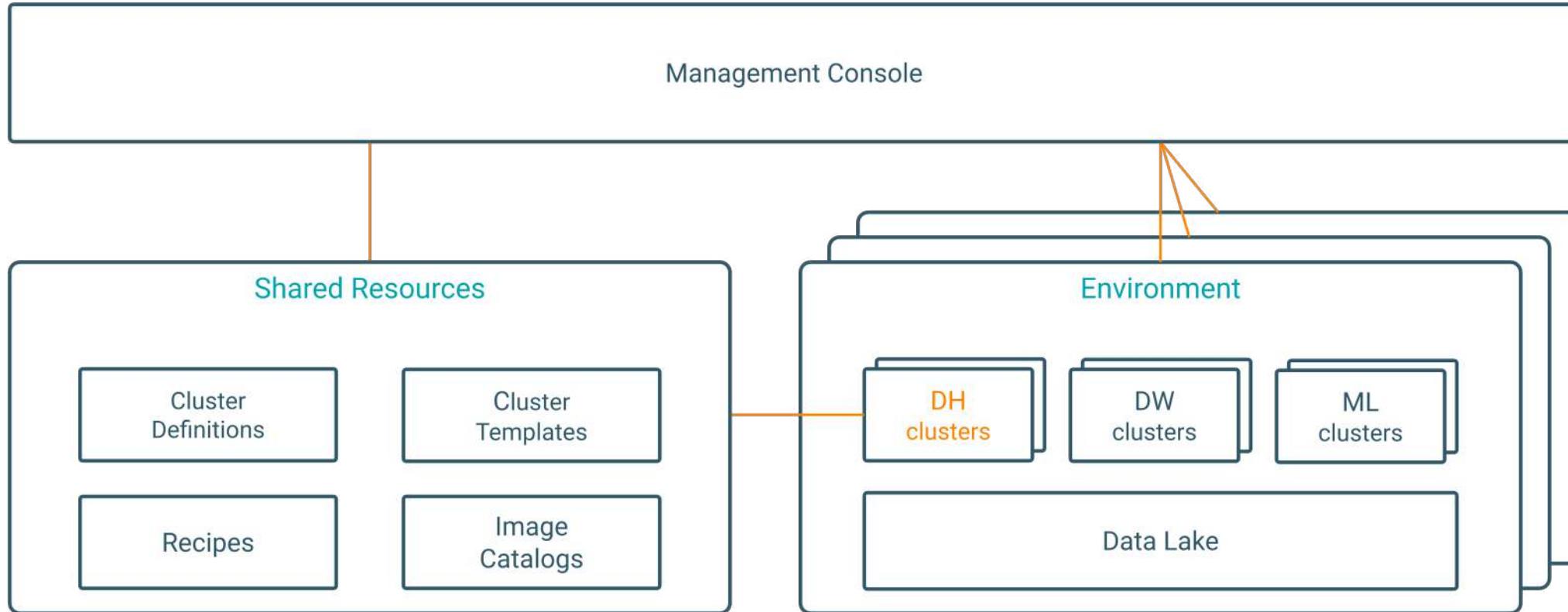


# Cloudera Machine Learning



- **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	Administrator CDH ●●○ HDP ●●	Administrator Private Cloud Base CDP 7.1 ●●○	Administrator CDP Public Cloud ○ AWS   Azure	Kafka Operations CDP 7.1 ●					
Data Steward	Data Governance CDP ●●○								
Data Analyst	Data Analyst: Hive/Impala CDH ○ CDP ●●	Hive LLAP HDP ●	CDP Data Visualization ○						
Developer Data Engineer	Spark Development CDH ○ CDP ●●	Spark Performance Tuning CDP ●●	NiFi Flow Management CDF ●●○	CDF Stream Processing CDF ○	Architecture CDP ●●	HBase CDH ●●○	Flink CDP ●●		
Data Scientist	CDSW CDSW ○	Data Science CDH ●●	CML CDP ○						
General	OnDemand Library CDH   HDP   CDP   CDF ○	“CDF” ○ Essentials, Pvt Cloud Fundamentals, HDP/CDH to CDP, AWS	“Just Enough” ○ Python, Git, Scala	Tech Overviews ○ CDW, Kudu, Kafka, Cloudera Manager, CDH on Azure	HDP ○ Self Paced Library				

● Live  
○ OnDemand

- **Our OnDemand catalog includes**
  - Courses for developers, data analysts, administrators, and data scientists
  - Exclusive OnDemand-only courses, such as those covering security and Cloudera Data Science Workbench
  - Free courses 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
- **Purchase access to a library of courses or individual courses**
  - [OnDemand information page](#)
  - [OnDemand course catalog](#)

# Accessing Cloudera OnDemand

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

The screenshot shows a web browser displaying a Cloudera OnDemand course page. The header includes the Cloudera logo, user information (CDP: admin-PvCbase), and a navigation bar with 'Available Training'. Below the header, a breadcrumb trail shows the course structure: Course > CDP Private Cloud Base Installation > Installation Overview > Installation Overview. The main content area has tabs for 'Course', 'Important Info', 'About This Course', 'Discussion', and 'Progress', with 'Course' being the active tab. A navigation bar at the top of the content area includes 'Previous' and 'Next' buttons, and a central button with a play icon and a checkmark. The main title is 'Installation Overview', with a 'Bookmark this page' button below it. The content section is titled 'Installation Overview' and contains a list of 'CDP Private Cloud Base Requirements' with five items: Hardware Requirements, Operating System Requirements, Database Requirements, Java Requirements, and Networking and Security Requirements. To the right of this list is a video player with a play button. The video summary text reads: 'First, an overview of the installation. We'll discuss the hardware requirements, operating system requirements, database and Java requirements and the network and security requirements for installation.' Below the video, there are several questions: 'To access the hardware and resource allocations for your cluster, there's some things to keep in mind.', 'What type of workloads are you going to be supporting?', 'What's the runtime components that are going to be used?', and 'The size of the data to be stored and processed,'.

# CDP Certification Program

---

- **New role-based certifications**
  - CDP Certified Generalist
  - CDP Certified Administrator
    - Private Cloud
    - Public Cloud
  - CDP Certified Developer
  - CDP Certified Analyst
- **Convenient online, proctored exams**
- **Digital badges**
- [www.cloudera.com/about/training/cdp-certification.html](http://www.cloudera.com/about/training/cdp-certification.html)



# 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

# Downloading the Course Materials

## 1. Log in using <https://university.cloudera.com/user>

- If necessary, use the **Register Now** link to create an account
- If you have forgotten your password, use the **Forgot your password** link

## 2. Scroll down to find this course

- If necessary, click **My Learning**
- You may also want to use the **Current** filter

## 3. Select the course title

## 4. Click the Resources tab

## 5. Click a file to download it

- Course materials are available for 30 days after your course

The screenshot shows the 'Please Log In' page from the Cloudera University website. The page features the Cloudera logo at the top left. It has fields for 'Email Address\*' and 'Password\*', both with their respective icons. A 'Log In' button is located below the password field. At the bottom right of the page, there is a red-bordered box containing links for 'Forgot your password?' and 'Don't have an account? Register now'.

## Introduction to Zeppelin

---

Chapter 2



# Course Chapters

---

- Introduction
- **Introduction to Zeppelin**
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Objectives

---

- **By the end of this chapter, you will be able to:**
  - Understand the motivation for notebooks
  - Understand the structure of a Zeppelin notebook
  - Run and create Zeppelin notebooks using best practices

# Chapter Topics

---

## Introduction to Zeppelin

- Why Notebooks?
- Zeppelin Notes
- Demo: Apache Spark In 5 Minutes

# Required For a Scientific Approach to Data Analysis

---

- To be able to perform analysis on data in a scientific manner, we need a tool that allows us to perform **reproducible data analysis** in which executable code is interspersed with paragraphs of text and visualizations that document our train of thoughts (literate programming)

- Several tools meet those requirements, here are the most popular ones:

- RStudio



- Jupyter



- Zeppelin



- All of them can be used as IDEs with our **Cloudera Data Science Workbench** tool

# Chapter Topics

---

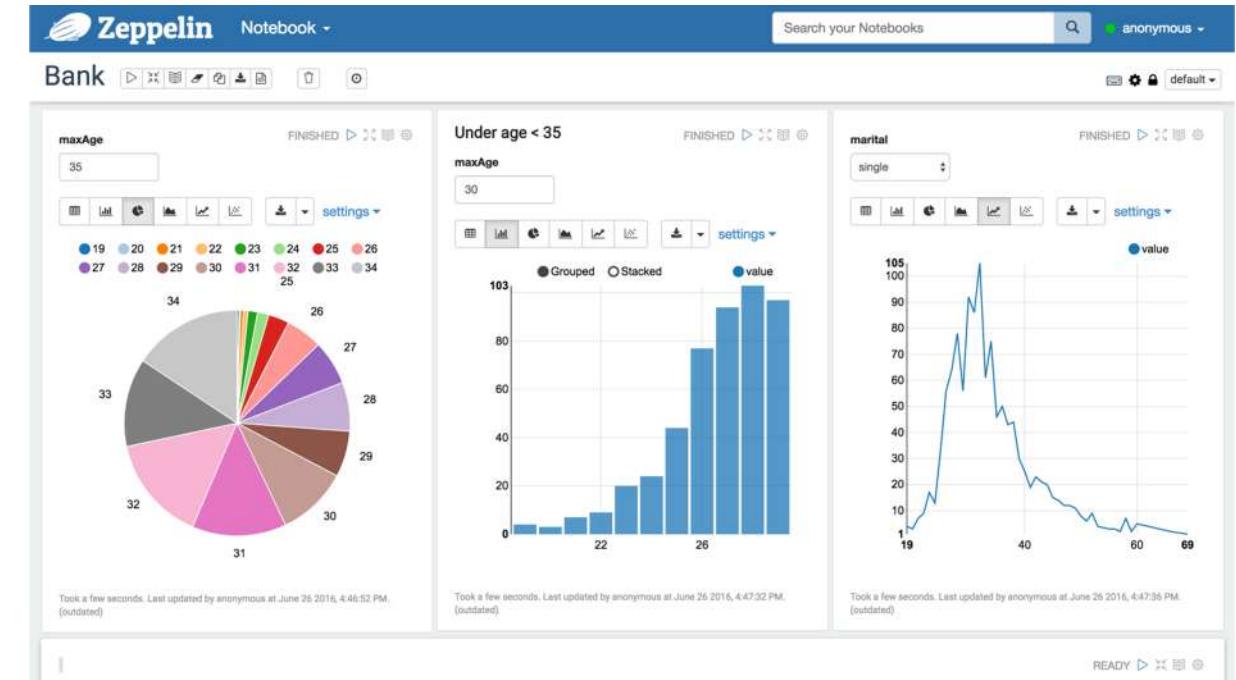
## Introduction to Zeppelin

- Why Notebooks?
- **Zeppelin Notes**
- Demo: Apache Spark In 5 Minutes

# What is Apache Zeppelin?



- An open source project from the Apache Software Foundation
  - 2013, NFLabs started the Zeppelin project
  - 2014-12-23, the Zeppelin project became incubation project in Apache Software Foundation
  - 2016-06-18, the Zeppelin project graduated incubation and became a Top Level Project in Apache Software Foundation
- A multi-purpose notebook system for
  - Data ingestion
  - Data discovery
  - Data analytics
  - Data visualization and collaboration



# Anatomy of a Note

---

- A Zeppelin note is a sequence of paragraphs
- Each paragraph is bound to an interpreter
- Each note has a list of available interpreters which is a subset of all the interpreters installed on the Zeppelin server
- The first element of the list is the default interpreter for the note
- Each paragraph should start by specifying the interpreter to which it is bound unless it is the default

ApacheSparkIn5Minutes

Settings

Interpreter binding

Bind interpreter for this note. Click to Bind/Unbind interpreter. Drag and drop to reorder interpreters. The first interpreter on the list becomes default. To create/remove interpreters, go to [Interpreter](#) menu.

- spark (%spark (default), %sql, %pyspark, %ipyspark, %r, %ir, %shiny, %kotlin)
- md (%md)
- sh (%sh, %sh.terminal)
- angular (%angular, %angular.ng)

Save Cancel

# Available Interpreters

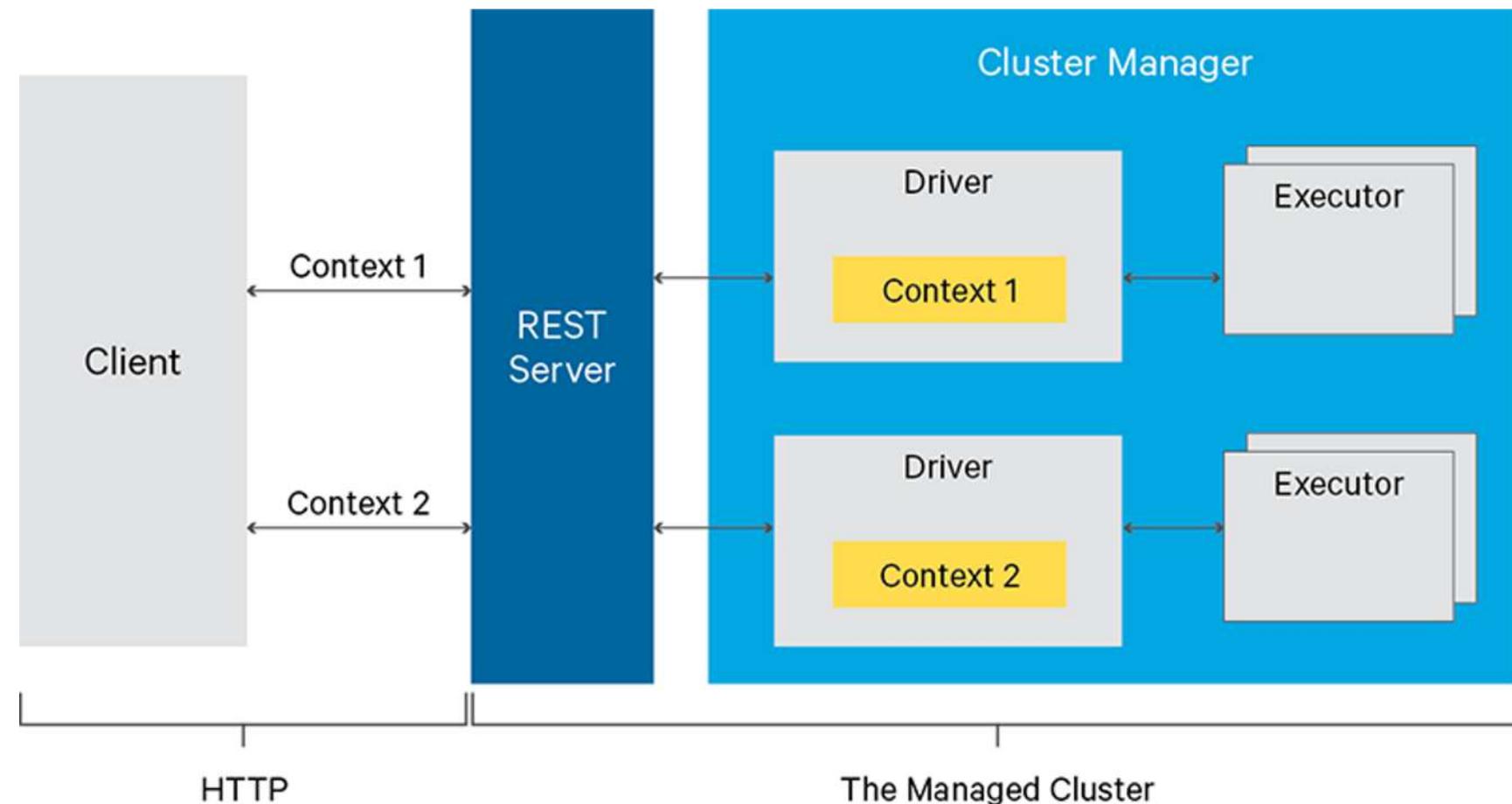
---

- The latest version of Zeppelin contains over 30 interpreters
- In CDP DC the following interpreters are available by default:
  - angular
  - livy
  - md
- The default one is livy
- This is a direct consequence of the secured design of CDP DC
- If you need an additional interpreter check with your favourite admin whether it can be installed securely



# What is Apache Livy?

- Livy enables multitenant and secure communication with a Spark cluster over a REST interface



# Note Formatting

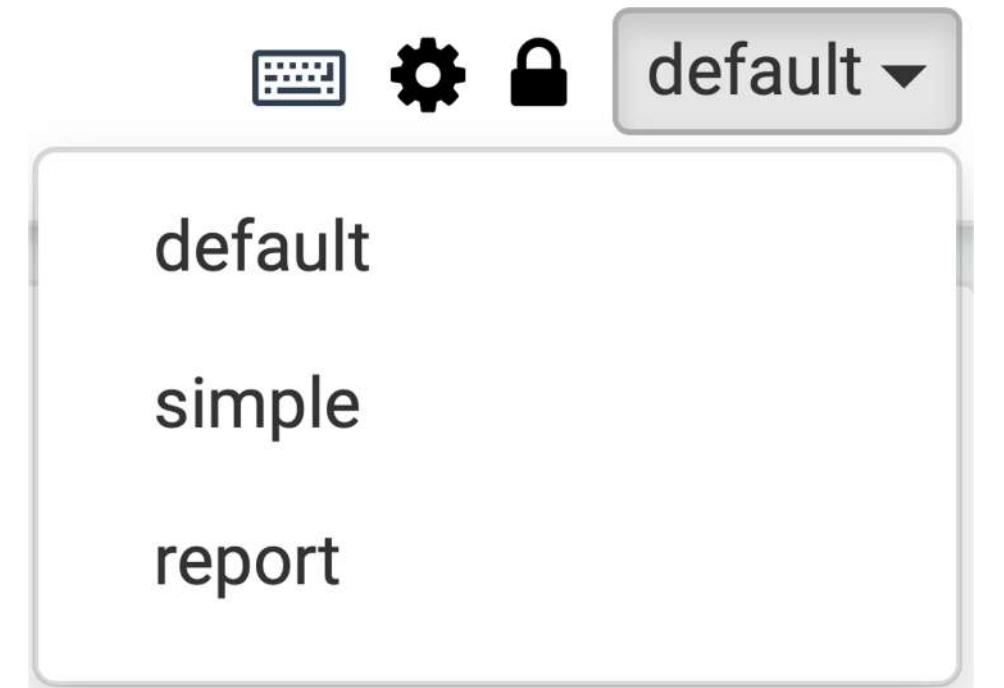
- Note owners can control all paragraphs at the note level, including:

- Hide/Show all code
- Hide/Show all output
- Clear all output



- There are also two additional note views

- Simple: Removes note-level controls
- Report: Removes note-level controls and all code



# Paragraph Formatting

- Paragraphs also contain formatting settings, including:

- Hide/Show paragraph code
- Hide/Show paragraph output
- Clear paragraph output is available  
in the settings menu (gear icon)



The screenshot shows a context menu for a paragraph. At the top right are buttons for 'FINISHED', navigation (right arrow, double arrows), and settings. Below is a timestamp: 20180313-232401\_119794092. The menu items are:

Action	Shortcut
Width	12 ▲▼
Font size	9 ▲▼
Move up	Ctrl+Option+K
Move down	Ctrl+Option+J
Insert new	Ctrl+Option+B
Run all above	Ctrl+Shift+Enter
Run all below	Ctrl+Shift+Enter
Clone paragraph	Ctrl+Shift+C
Hide title	Ctrl+Option+T
Show line numbers	Ctrl+Option+M
Disable run	Ctrl+ Option+R
Link this paragraph	Ctrl+Option+W
<b>Clear output</b>	<b>Ctrl+Option+L</b>
Remove	Ctrl+Option+D

The 'Clear output' option is highlighted with a red oval.

# Paragraph Enhancement - Width

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

The screenshot shows a Jupyter Notebook interface with three main sections:

- Code Cell 1:** Contains Python code using PySpark to load a dataset from a table named "bankdataperm". The output shows a table with columns: age, balance, and marital.
- Code Cell 2:** Contains SQL code to filter rows where age is between 30 and 55, and marital status is married. The output shows a histogram of balances for married individuals, with a total count of 24,598.
- Sidebar Menu:** A panel on the right side of the interface, titled "FINISHED", contains various options for managing the paragraph:
  - Width: Set to 5
  - Move Up
  - Move Down
  - Insert New
  - Show title
  - Show line numbers
  - Disable run
  - Link this paragraph
  - Clear output
  - Remove

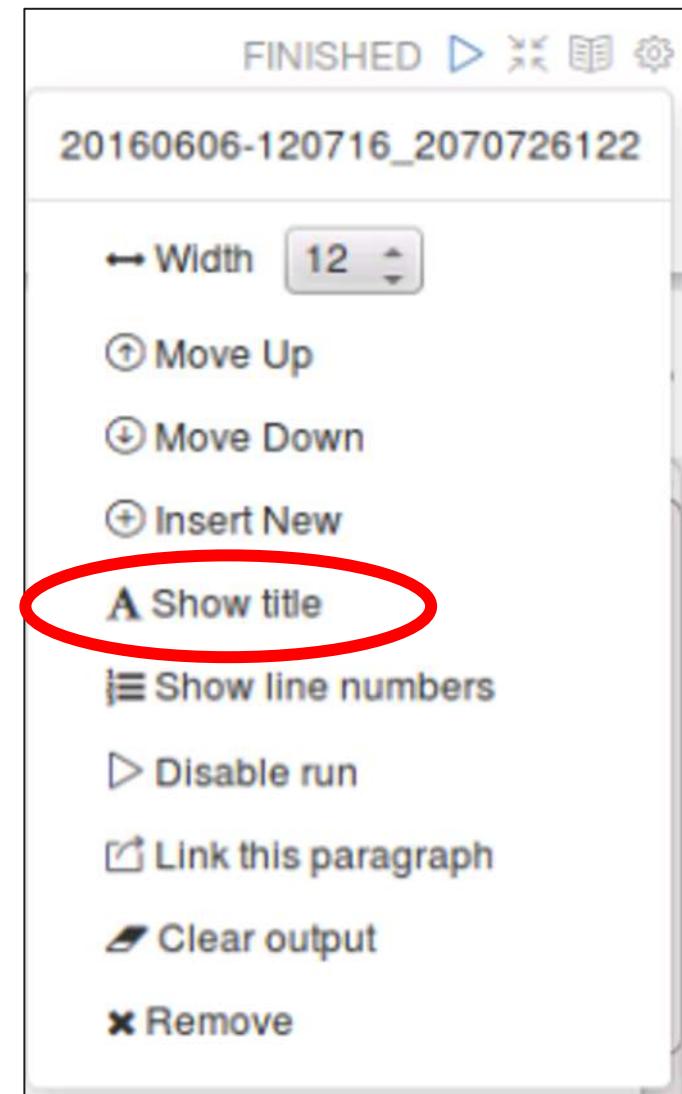
## Paragraph Enhancement - Show Title

- Paragraph titles can be added for clarity

Untitled FINISH

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

marital Maximum Age

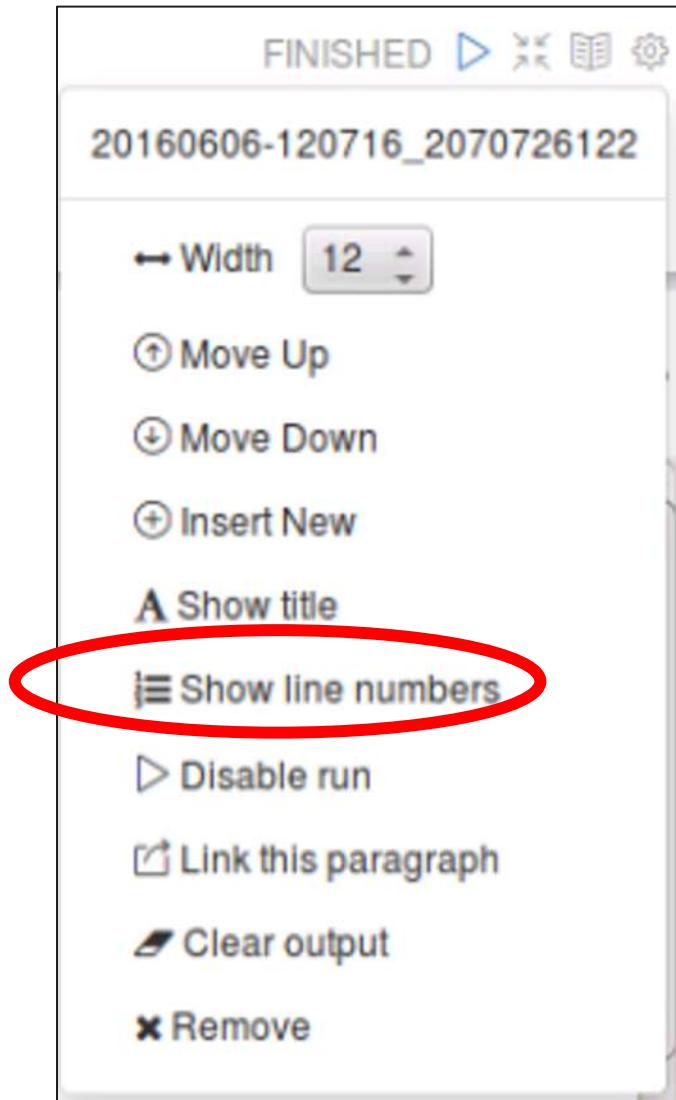


# Paragraph Enhancement - Line Numbers

- Line numbers can be added to paragraph code

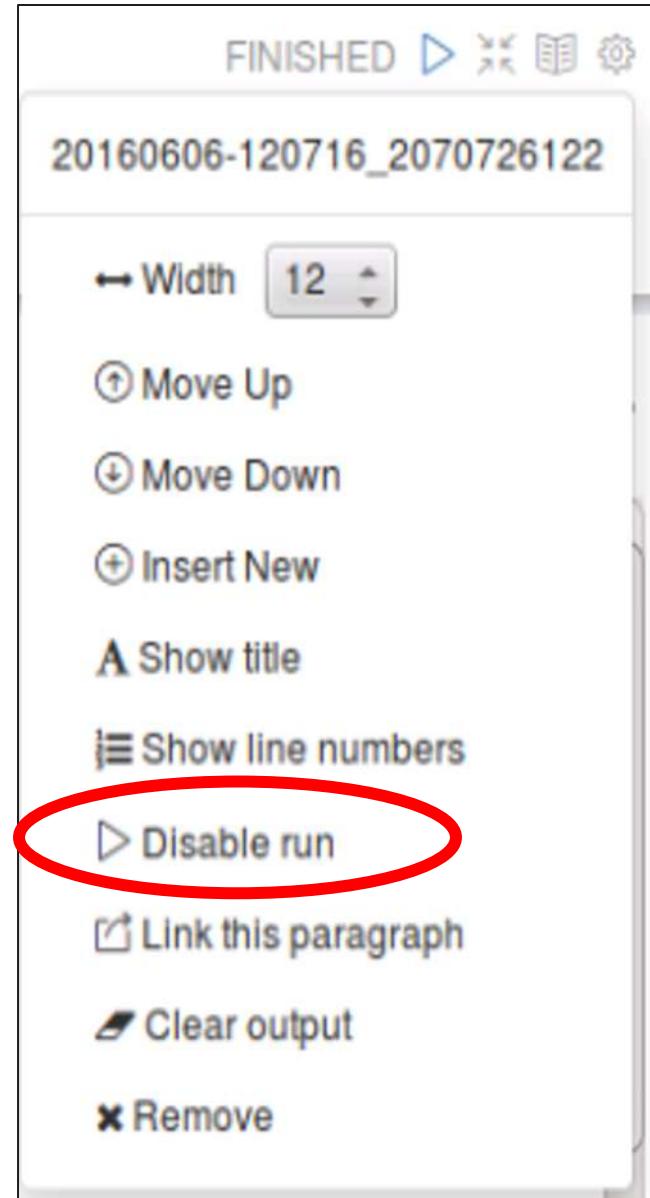
```
1 %sql  
2 select * from bankdataperm where age >= ${Min Age}  
and age <= ${Maximum Age=100} and marital =  
  
marital
```

Maximum Age



# Disable Paragraph Output Changes

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



# Best Practices

---

- **Save your notes outside of the Zeppelin home folder**
- **Clear the outputs of your notes before exporting them**
  - Saves disk space
  - Reduces the risk of not being able to import them back in
- **Minimize the dependencies of your notes with your local environment to increase reproducibility**
  - Download your data from cloud storage if possible
- **Make your note error free when run twice**
  - Make the extra effort to avoid triggering an error if the same paragraph runs twice
- **Disable your markdown paragraphs and hide their codes**
  - Saves CPU cycles and looks cleaner
- **Give your paragraphs titles**
- **Make good use of markdown paragraphs and the visualization features to tell your story**

# Chapter Topics

---

## Introduction to Zeppelin

- Why Notebooks?
- Zeppelin Notes
- **Demo: Apache Spark In 5 Minutes**



## HDFS Introduction

---

### Chapter 3

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- **HDFS Introduction**
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Lessons Objectives

---

**By the end of this chapter, you will be able to:**

- **Present an overview of the Hadoop Distributed File System (HDFS)**
- **Detail the major architectural components and their interactions**
  - NameNode
  - DataNode
  - Clients

# Chapter Topics

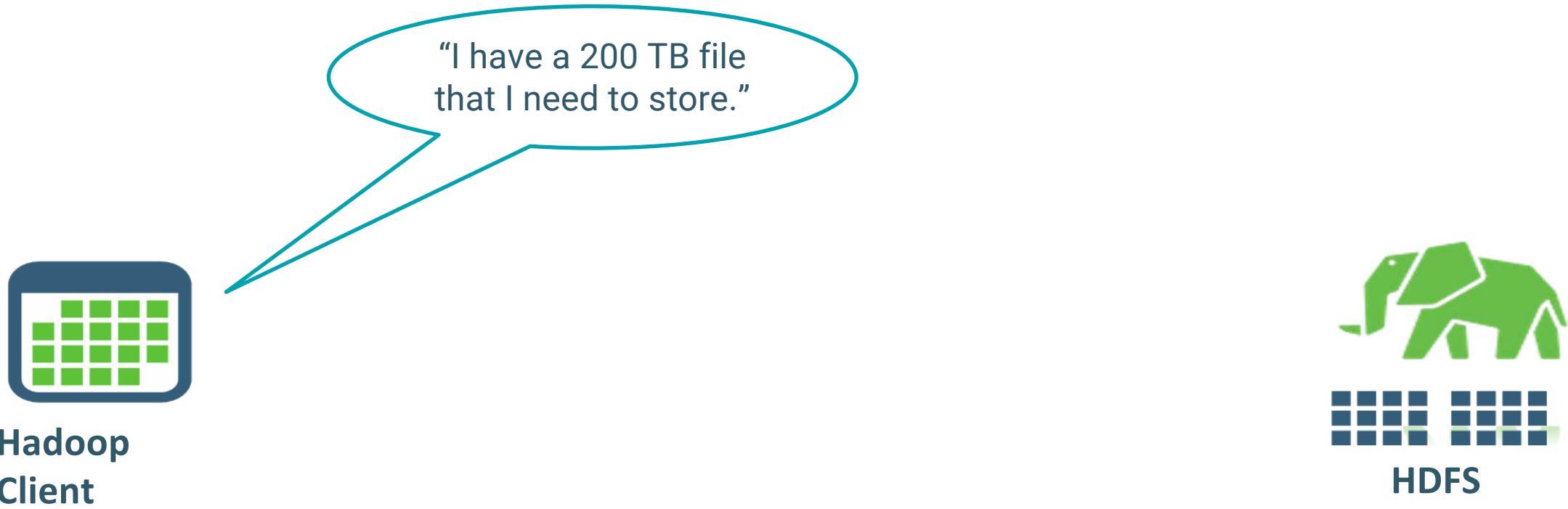
---

## HDFS Introduction

- **HDFS Overview**
- HDFS Components and Interactions
- Additional HDFS Interactions
- Ozone Overview
- Exercise: Working with HDFS

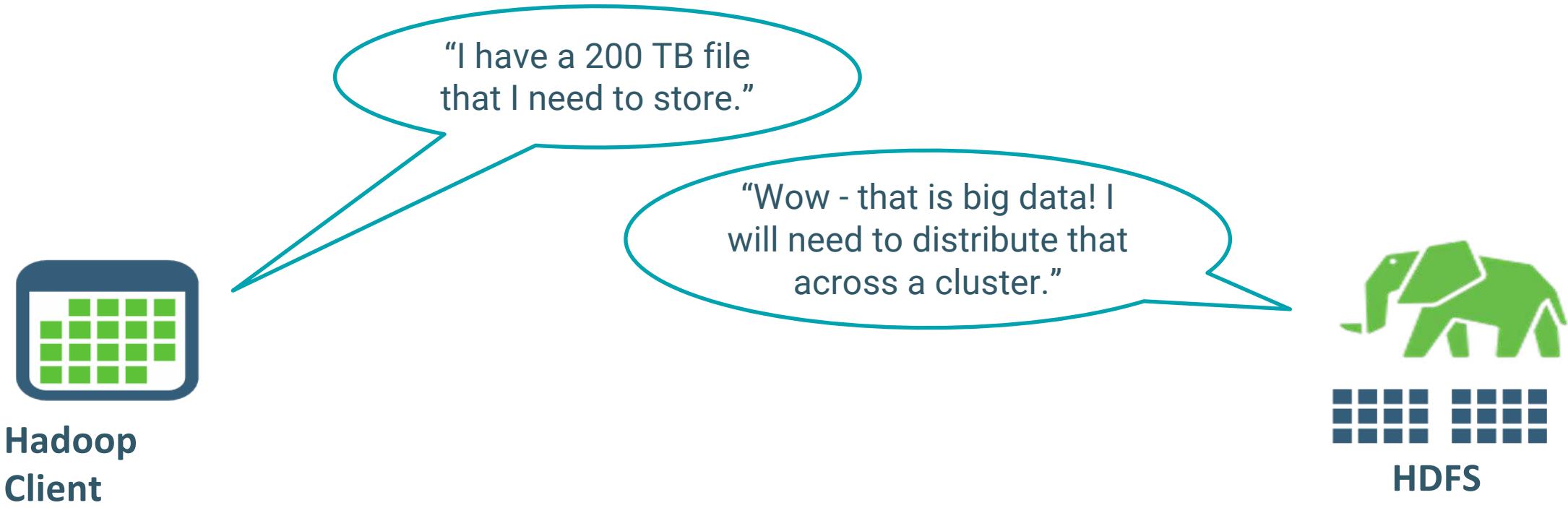
# What is HDFS?

---

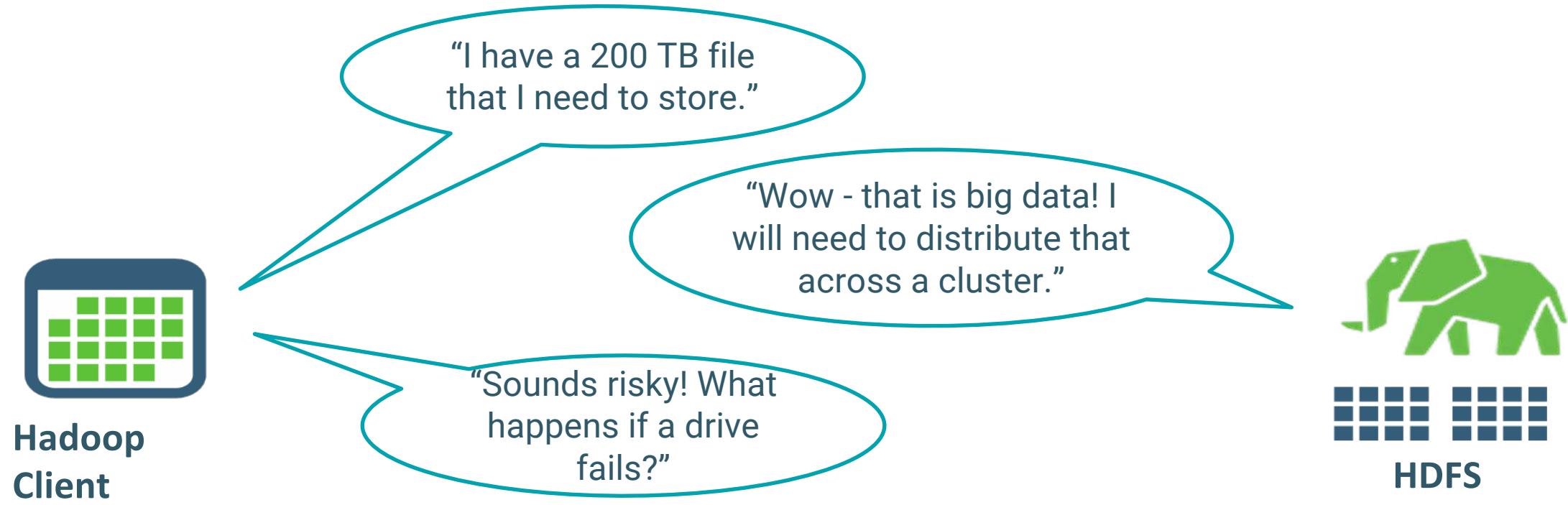


# What is HDFS?

---



# What is HDFS?

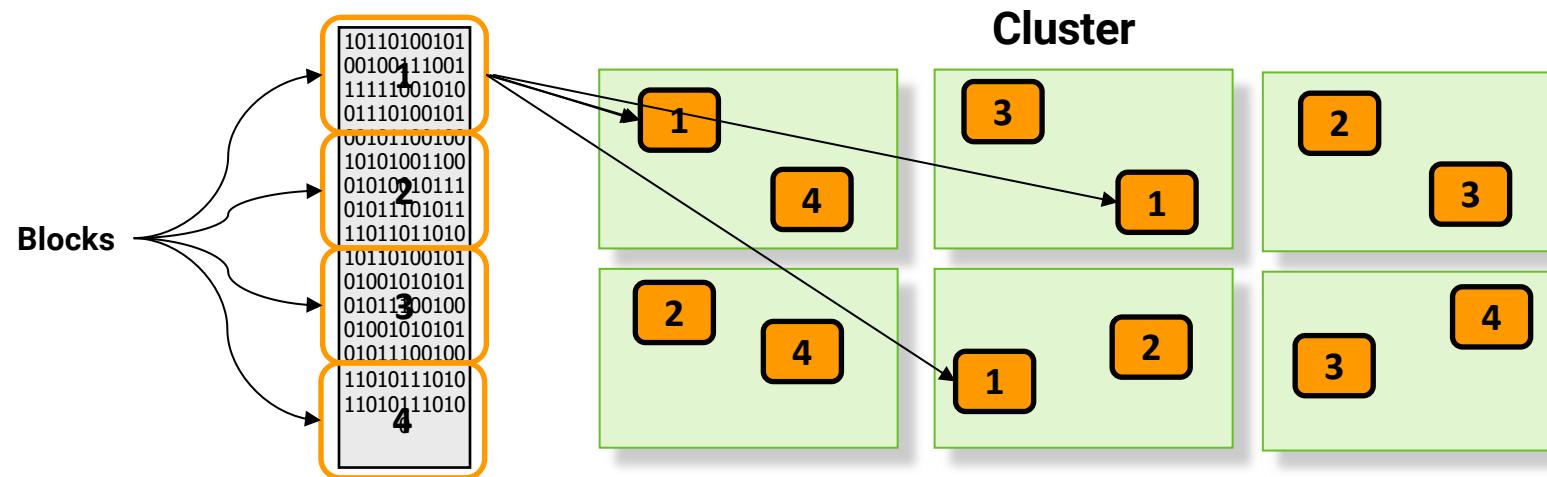


# What is HDFS?



## ■ Key Ideas

- Write Once, Read Many times (WORM)
- Divide files into big blocks and distribute across the cluster
- Store multiple replicas of each block for reliability
- Programs can ask "where do the pieces of my file live?"



# It Looks Like a File System

The screenshot shows a file browser interface with the following details:

- Path:** / user / it1 / geolocation
- Buttons:** + New directory, Browse..., Select files to upload, X
- Search:** Search File Names, magnifying glass icon
- Table Headers:** Name, Size, Last Modified, Owner, Group, Permission
- Table Data:**

Name	Size	Last Modified	Owner	Group	Permission
..					
geolocation.csv	514.3 kB	2016-03-13 19:42	maria_dev	hdfs	-rw-r--r--
trucks.csv	59.9 kB	2016-03-13 19:41	maria_dev	hdfs	-rw-r--r--
- SSH Terminal:**

```
[[it1@sandbox ~]$ hdfs dfs -ls /user/it1/geolocation
Found 2 items
-rw-r--r--  3 maria_dev hdfs      526677 2016-03-13 23:42 /user/it1/geolocation/geolocation.csv
-rw-r--r--  3 maria_dev hdfs      61378 2016-03-13 23:41 /user/it1/geolocation/trucks.csv
[it1@sandbox ~]$ ]]
```

# It Acts Like a File System

---

- A few of the almost 30 HDFS commands:

**hdfs dfs –command [args]**

- cat: display file content (uncompressed)
- text: just like cat but works on compressed files
- chgrp,-chmod,-chown: changes file permissions
- put,-get,-copyFromLocal,-copyToLocal: copies files from the local file system to the HDFS and vice versa.
- ls, -ls -R: list files/directories
- mv,-moveFromLocal,-moveToLocal: moves files
- stat: statistical info for any given file (block size, number of blocks, file type, etc.)

# Chapter Topics

---

## HDFS Introduction

- HDFS Overview
- **HDFS Components and Interactions**
- Additional HDFS Interactions
- Ozone Overview
- Exercise: Working with HDFS

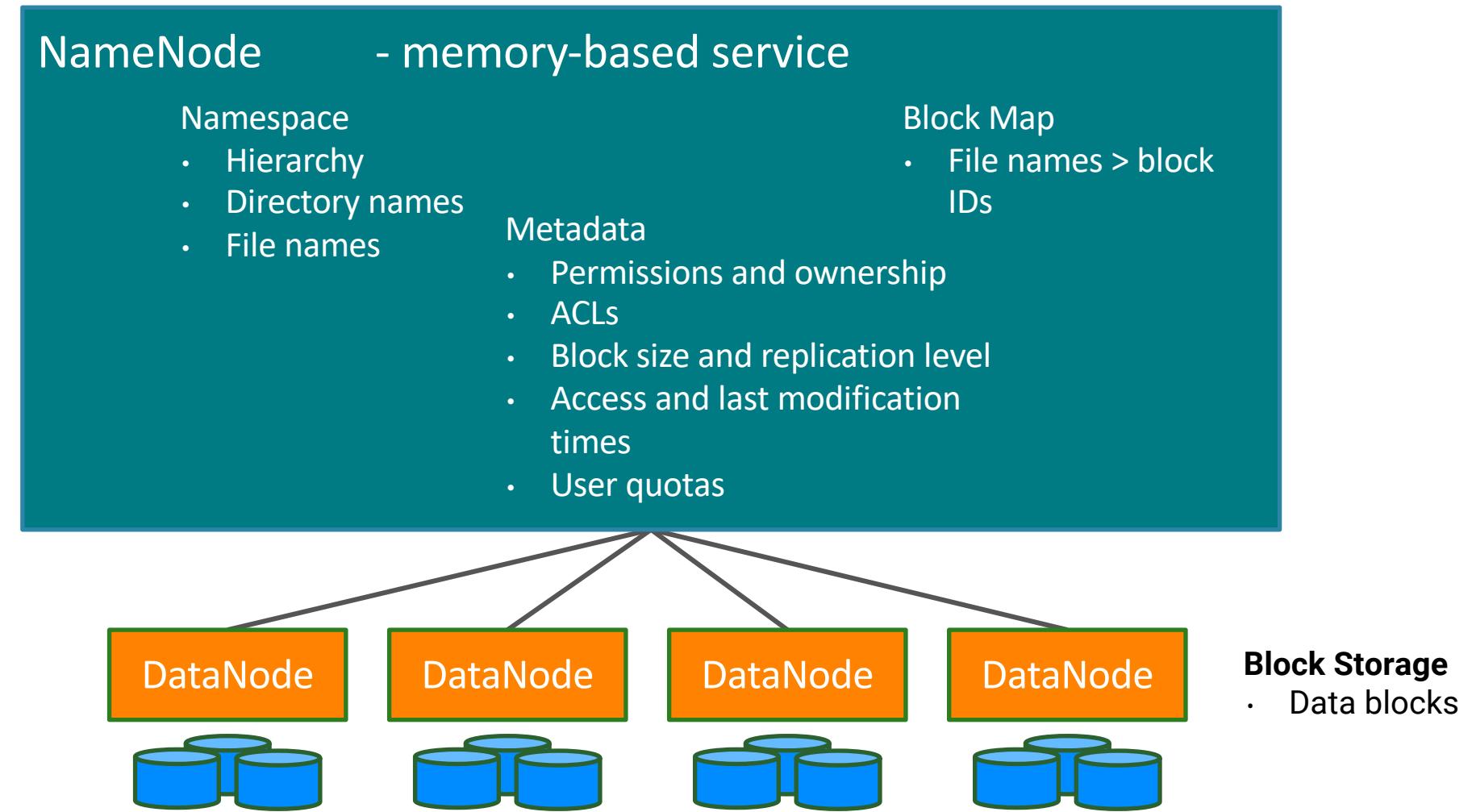
# HDFS Components

---

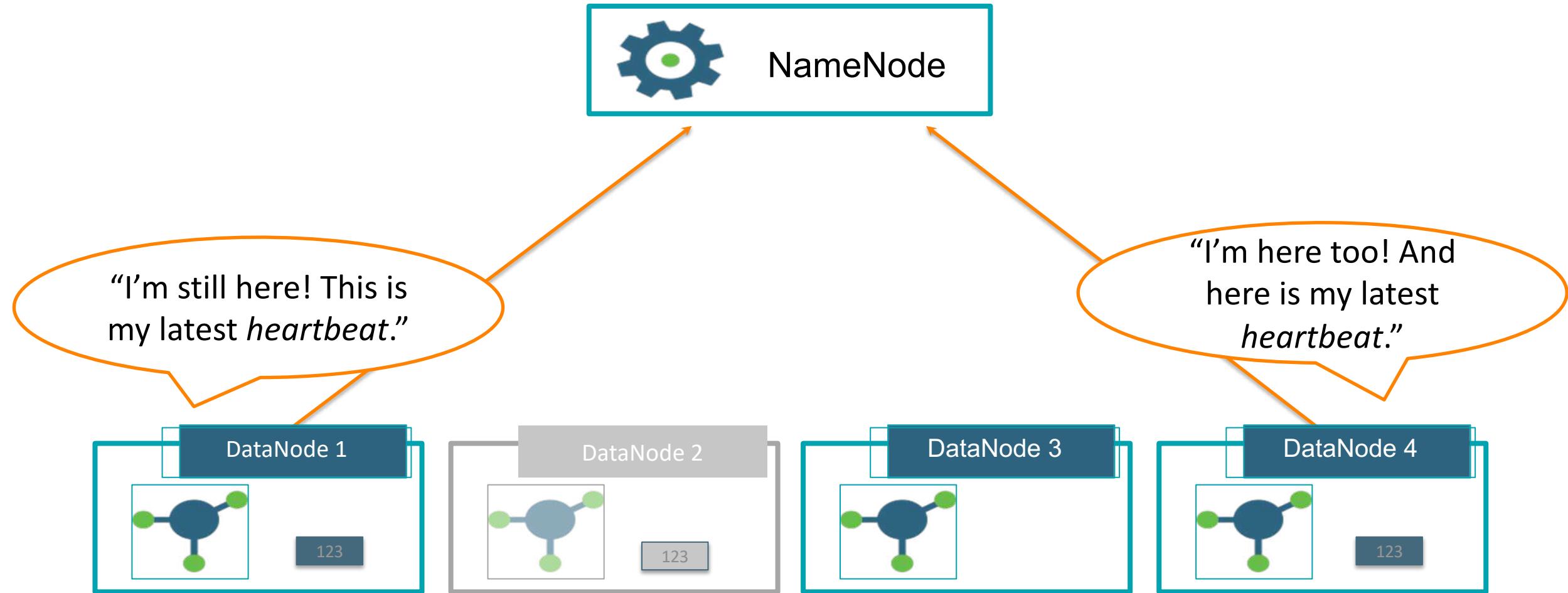
- **NameNode**
  - Is the master service of HDFS
  - Determines and maintains how the chunks of data are distributed across the DataNodes
- **DataNode**
  - Stores the chunks of data, and is responsible for replicating the chunks across other DataNodes

# HDFS Architecture

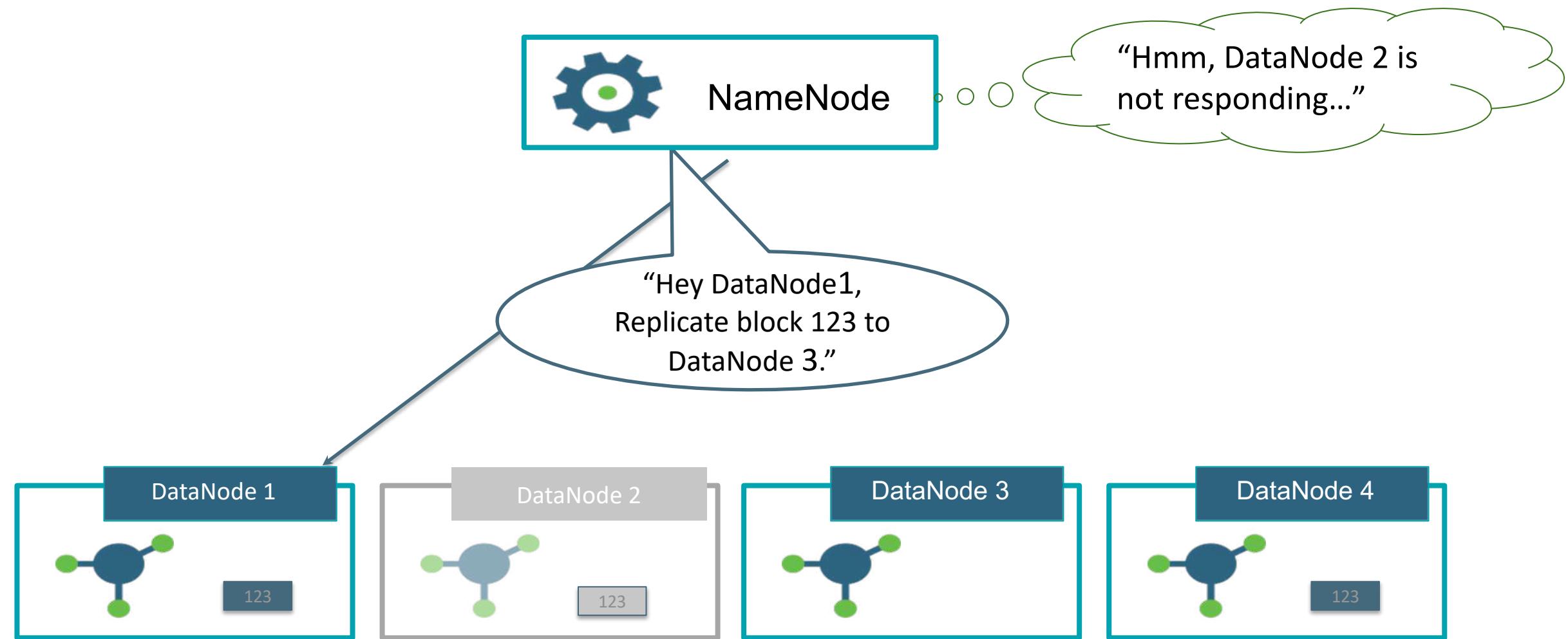
- The NameNode (master node) and DataNodes (worker nodes) are daemons running in a Java virtual machine.



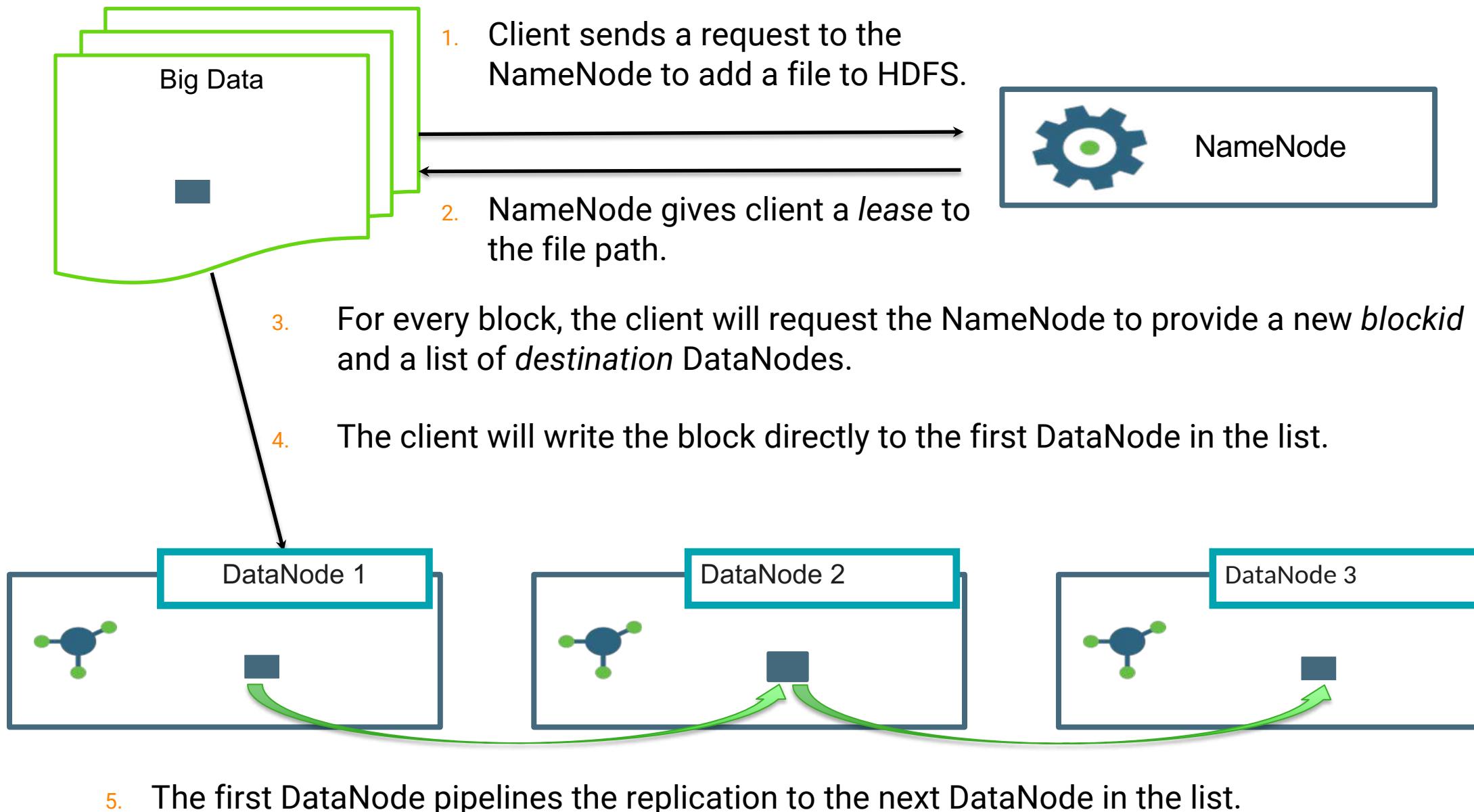
# The DataNodes



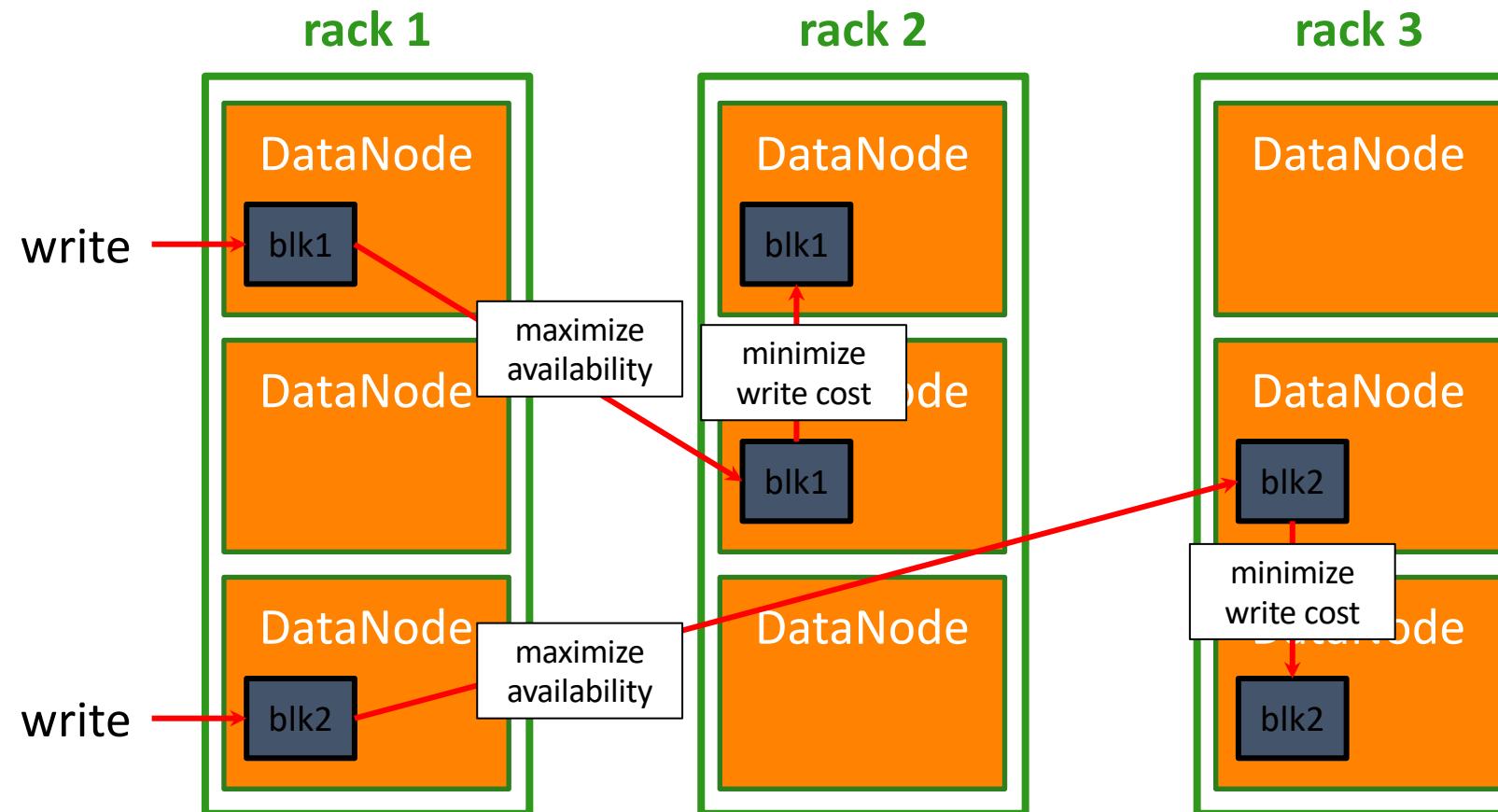
# The DataNodes



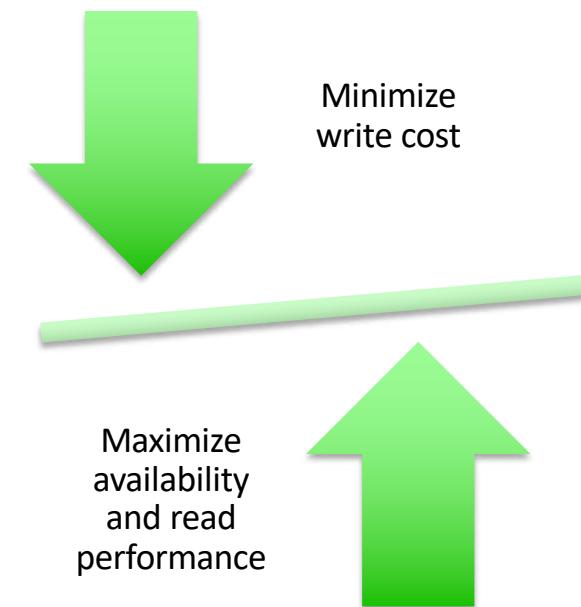
# Writing to HDFS



# Replication and Block Placement



HDFS is designed to assume, and handle, disk and system failures.



# Chapter Topics

---

## HDFS Introduction

- HDFS Overview
- HDFS Components and Interactions
- **Additional HDFS Interactions**
- Ozone Overview
- Exercise: Working with HDFS

# NameNode High Availability

---

- **The HDFS NameNode is a single point of failure.**
  - The entire cluster is unavailable if the NameNode:
    - Fails or becomes unreachable
    - Is stopped to perform maintenance
- **NameNode HA:**
  - Uses a redundant NameNode
  - Is configured in an Active/Standby configuration
  - Enables fast failover in response to NameNode failure
  - Permits administrator-initiated failover for maintenance
  - Is configured by Cloudera Manager

# HDFS Multi-Tenant Controls

---

- **Security**
  - Classic POSIX permissioning (ex: -rwxr-xr--)
  - Extended Access Control Lists (ACL) for richer scenarios
  - Centralized authorization policies and audit available via Ranger plug-in
  
- **Quotas**
  - Easy to understand data size quotas
  - Additional option for controlling the number of files

# Chapter Topics

---

## HDFS Introduction

- HDFS Overview
- HDFS Components and Interactions
- Additional HDFS Interactions
- **Ozone Overview**
- Exercise: Working with HDFS

# Ozone Overview

---

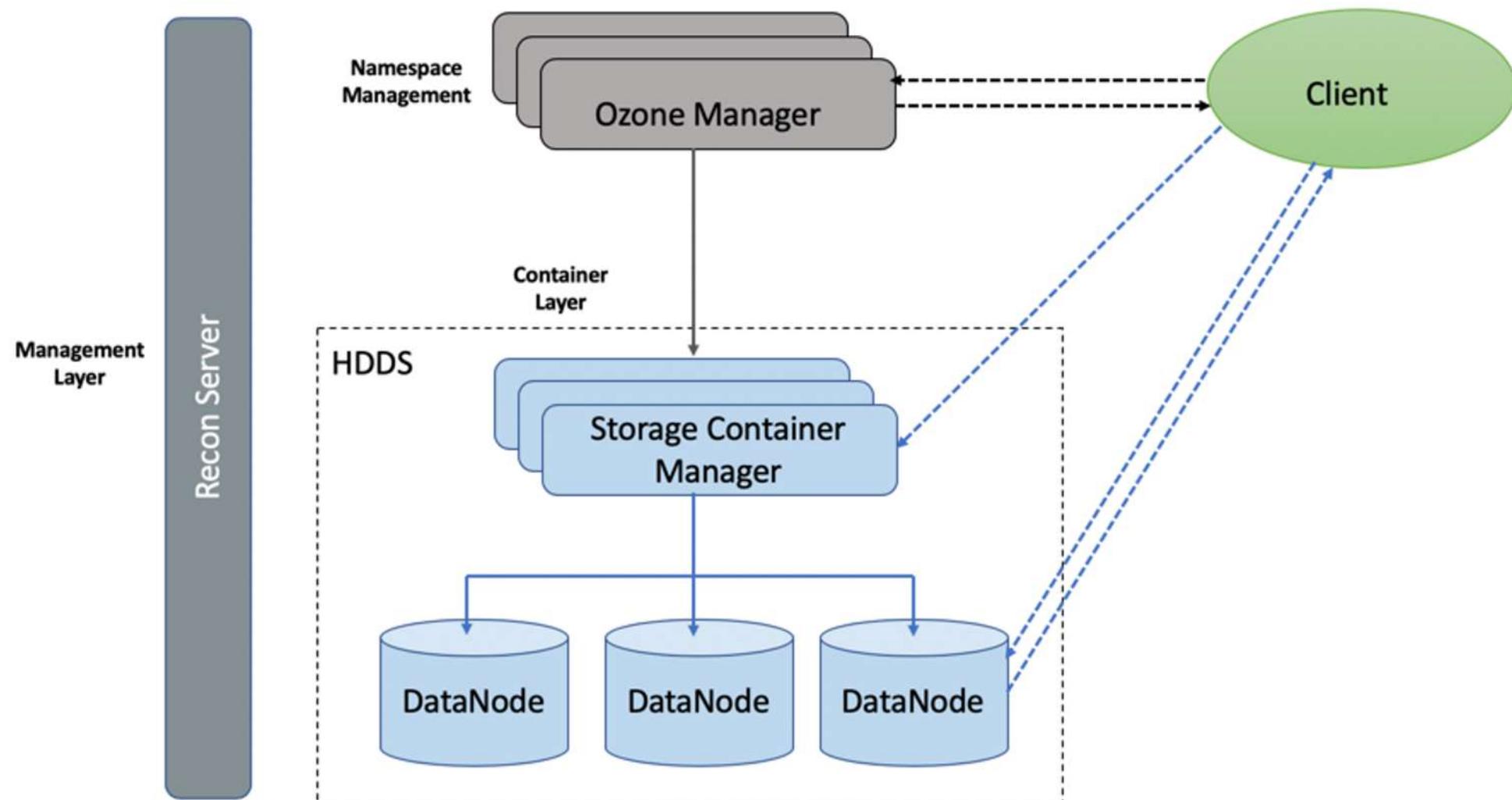
- **Distributed key-value store**
- **Efficiently manages both small and large files**
- **Designed to work well with the existing Apache Hadoop ecosystem**
- **Open-source**
- **Scales to thousands of nodes and billions of objects in a single cluster**
- **Based on the good HDFS ideas, but fixing HDFS bottlenecks**
  - Small files
  - Many files
  - Block Reports

# Ozone Architecture

---

- **Layered architecture**
  - Key/Value Objects
  - Blocks
  - Containers
  - Buckets
  - Volumes
- **Separating the HDFS Namenode monolithic architecture**
  - Managing Namespace (Ozone Manager (OM))
  - Block Storage (Storage Container Management (SCM))
- **Simplified architecture for High Availability**
  - No more Zookeeper, Failover Controllers and Journal Nodes needed

# Ozone Architecture

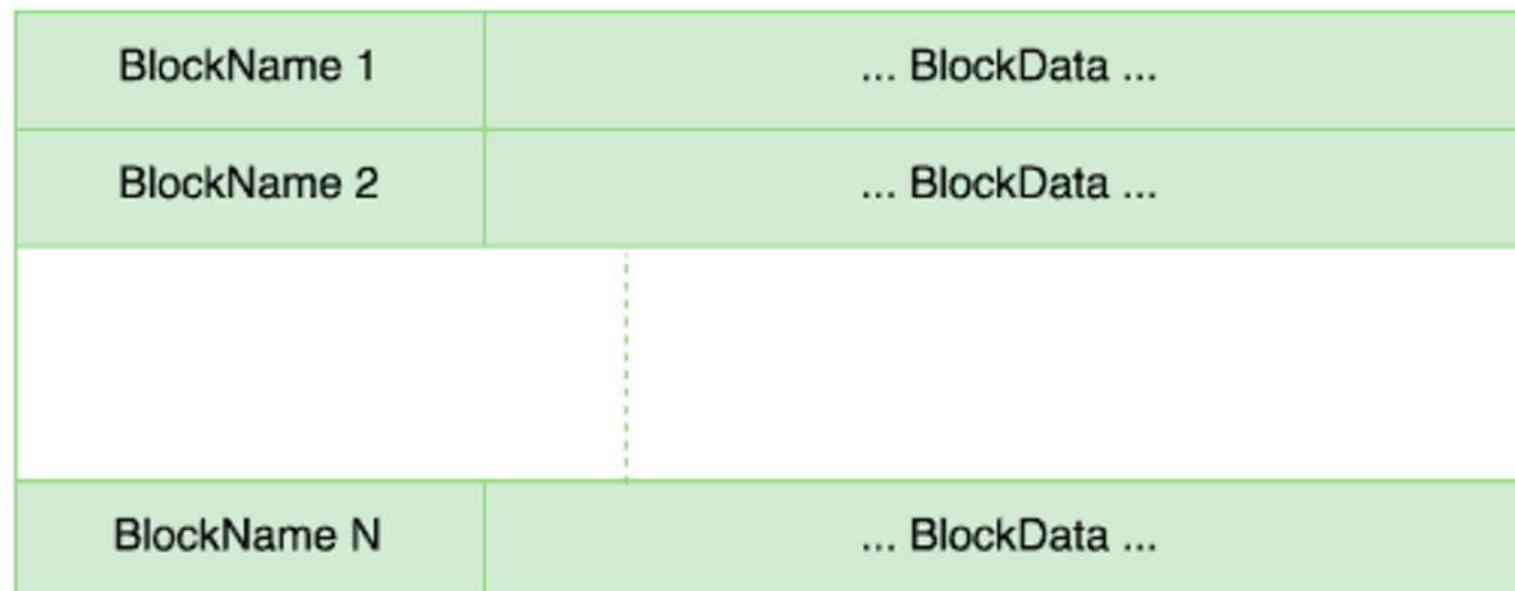


# Key/Values, Blocks and Containers

---

## ■ Lowest Layer of Ozone

- Key/Value objects can be stored, read and deleted
- Objects are split into one or more blocks
- Multiple Blocks get stored into a container



**Figure 1: Container Layout**

## Knowledge Check

---

- 1. HDFS breaks files into \_\_\_\_\_ and persists multiple \_\_\_\_\_ across the cluster to aid in the file system's \_\_\_\_\_ and the to help programs obtain \_\_\_\_\_.**
- 2. What is the primary master node service?**
- 3. What is the worker node service?**
- 4. True/False? Clients avoid writing data through the NameNode.**
- 5. True/False? Clients write replica copies directly to each DataNode.**

# Essential Points

---

- HDFS breaks files into blocks and replicates them for reliability and processing data locality
- The primary components are the master NameNode service and the worker DataNode service
- The NameNode is a memory-based service
- The NameNode automatically takes care of recovery missing and corrupted blocks
- Clients interact with the NameNode to get a list, for each block, of DataNodes to write data to
- Ozone is the future of distributed storage

# Chapter Topics

---

## HDFS Introduction

- HDFS Overview
- HDFS Components and Interactions
- Additional HDFS Interactions
- Ozone Overview
- **Exercise: Working with HDFS**



## YARN Introduction

---

### Chapter 4

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction**
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Lesson Objectives

---

**By the end of this chapter, you will be able to:**

- **Describe the purpose and components of YARN**
- **Describe the major architectural components and their interactions**
  - ResourceManager
  - NodeManager
  - ApplicationManager
- **Describe additional YARN features**
  - High Availability
  - Resource request model
  - Schedulers

# Chapter Topics

---

## YARN Introduction

- **YARN Overview**
- **YARN Components and Interaction**
- **Working with YARN**
- **Exercise: Working with YARN**

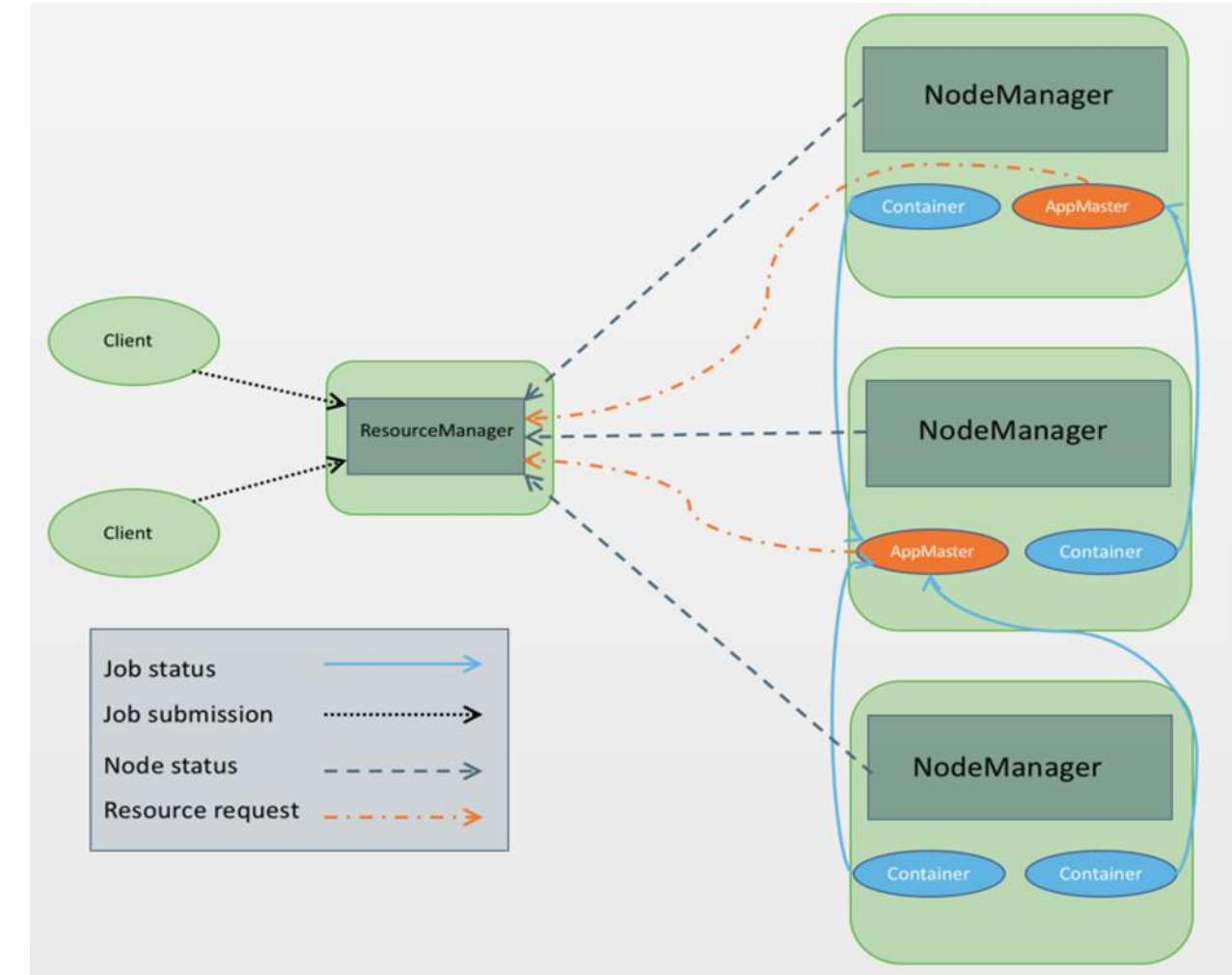
# YARN Resource Management

---

- Yet Another Resource Negotiator
- Architectural center of Enterprise Hadoop
- Provides centralized resource management and job scheduling across multiple types of processing workloads
- Enables multi tenancy

# YARN Architectural Components

- **Resource Manager**
  - Global resource scheduler
  - Hierarchical queues
- **Node Manager**
  - Per-machine agent
  - Manages the life-cycle of container
  - Container resource monitoring
- **Application Master**
  - Per-application
  - Manages application scheduling and task execution
  - E.g. MapReduce Application Master



# Chapter Topics

---

## YARN Introduction

- YARN Overview
- **YARN Components and Interaction**
- Working with YARN
- Exercise: Working with YARN

# YARN Architecture – Big Picture View

---

- Master node component
- Centrally manages cluster resources for all YARN applications

Resource  
Manager

NodeManager

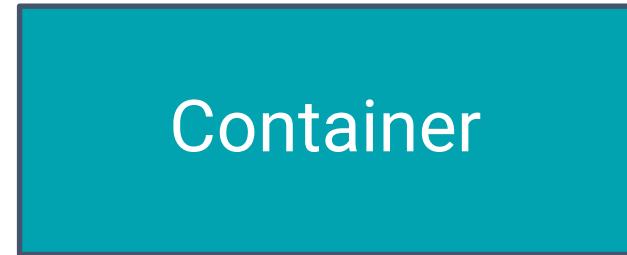
- Worker node component
- Manages local resources at the direction of the ResourceManager
- Launches containers

# 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



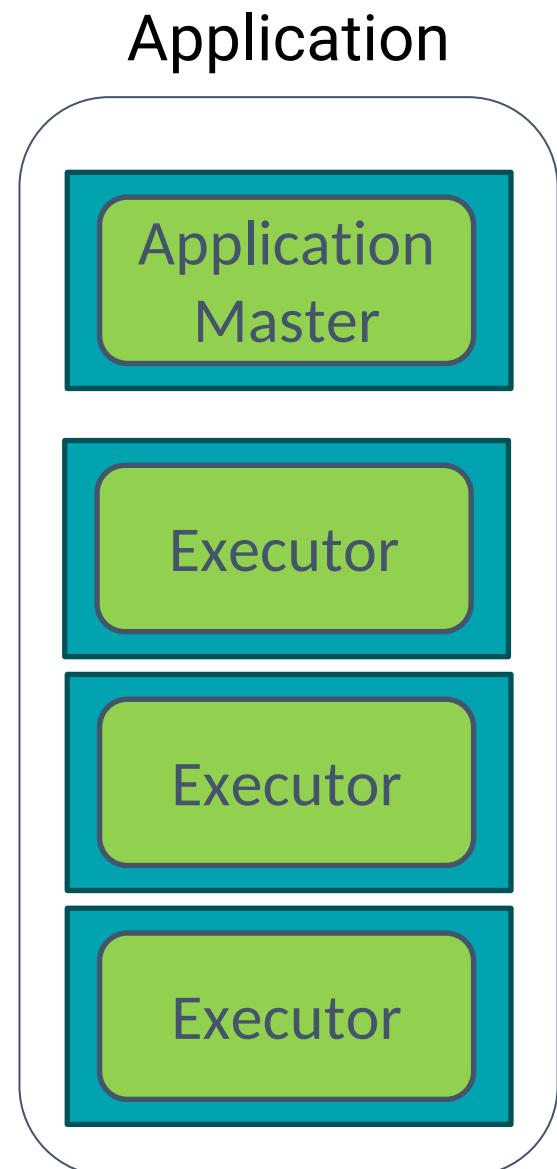
- **ApplicationMaster (AM)**

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

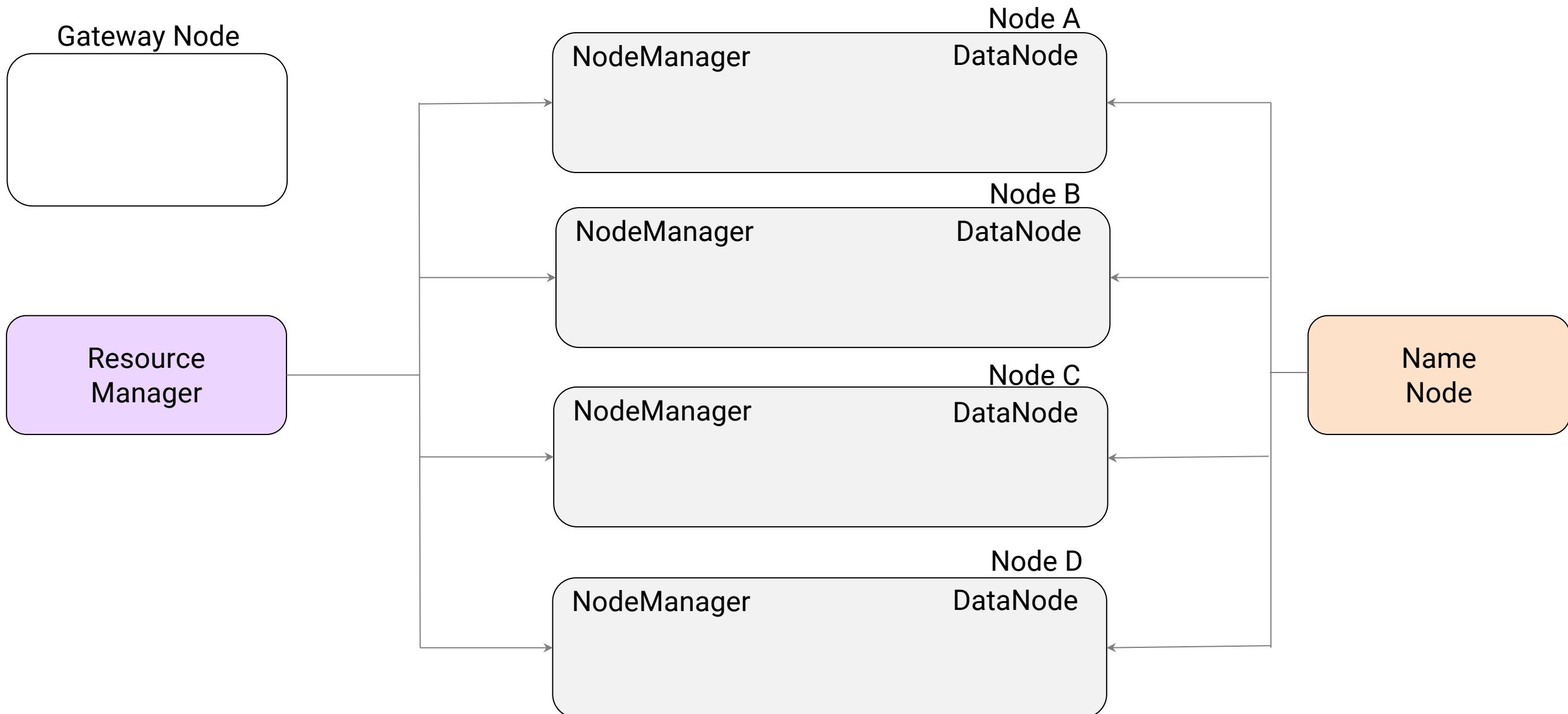


## Applications on YARN (2)

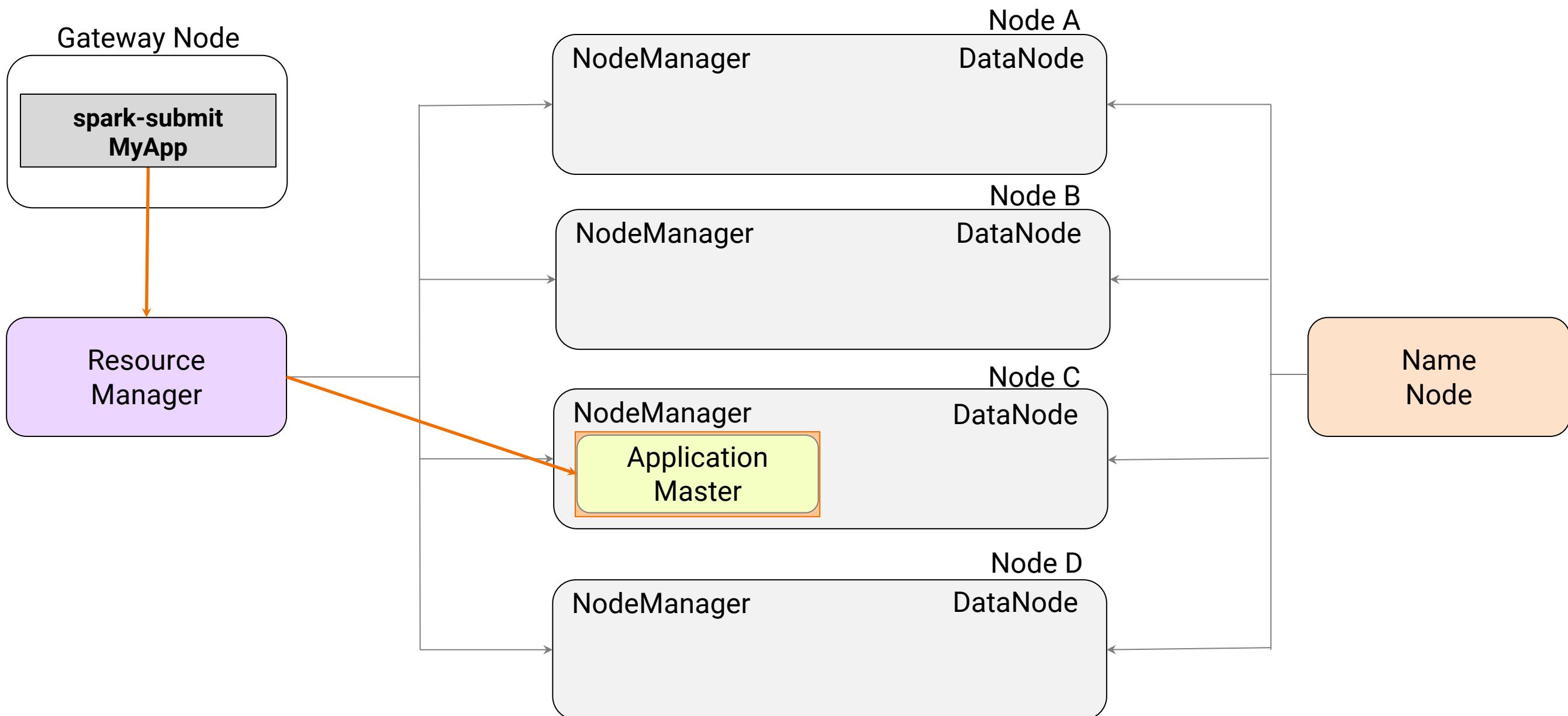
- **Each application consists of one or more containers**
  - The ApplicationMaster runs in one container
  - The application's distributed processes (JVMs) run in other containers
    - The processes run in parallel, and are managed by the AM
    - The processes are called executors in Apache Spark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



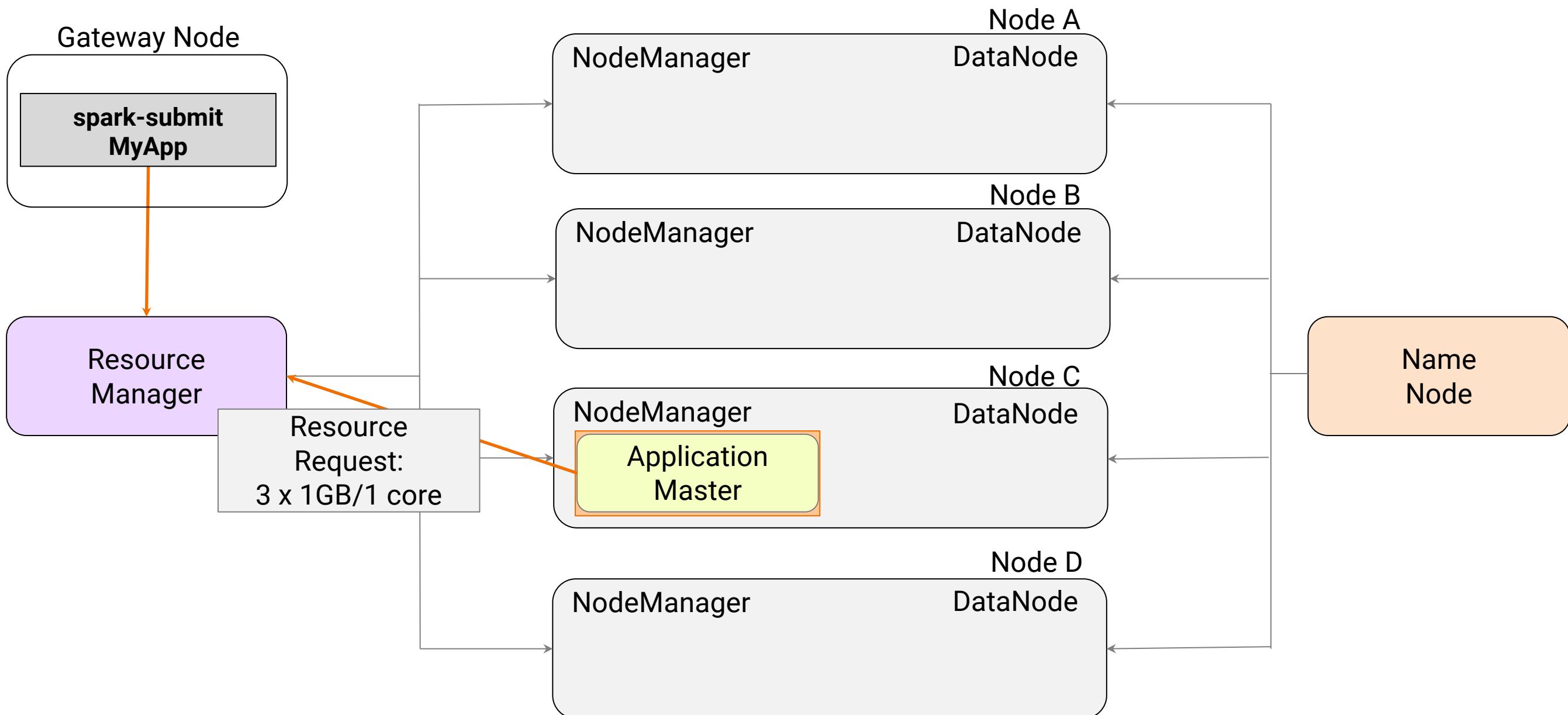
# Running an Application on YARN (1)



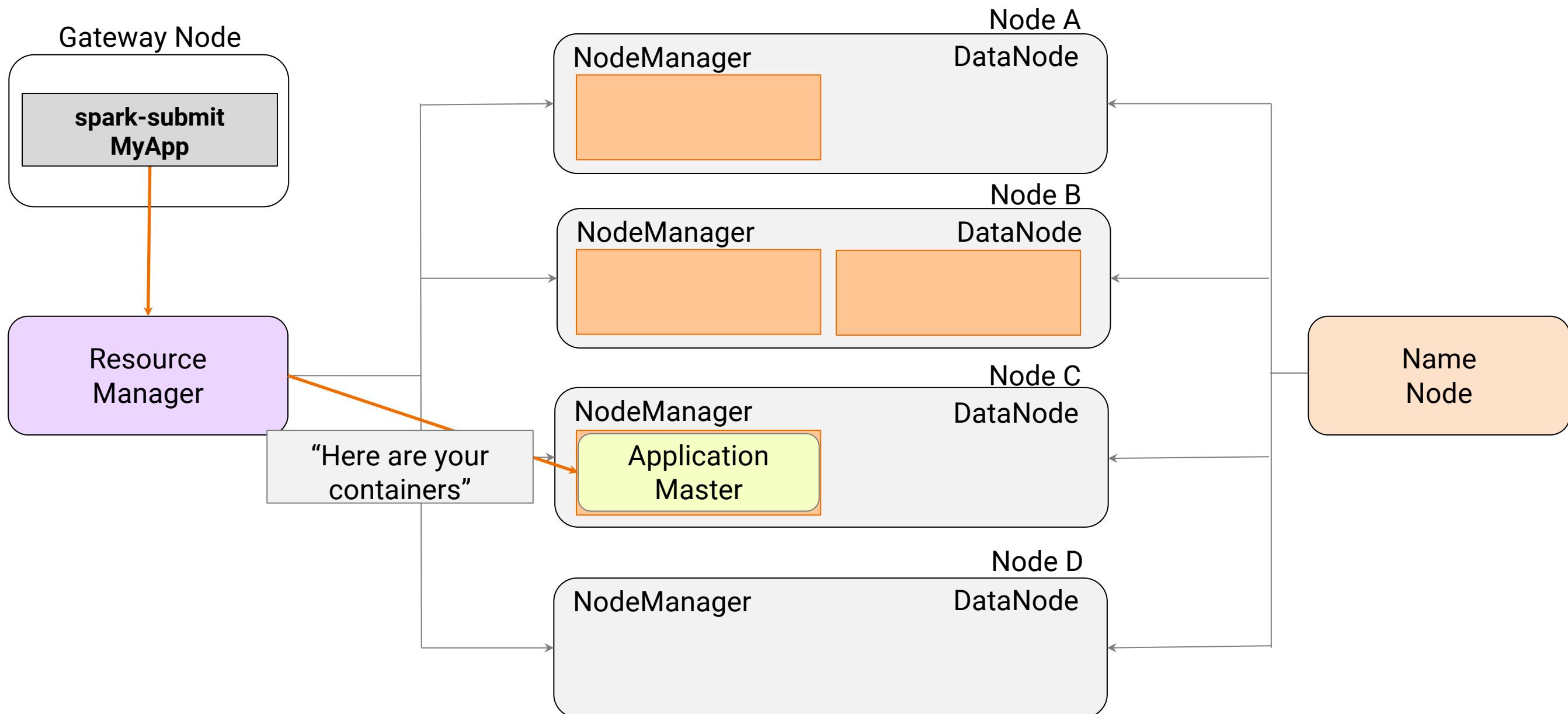
## Running an Application on YARN (2)



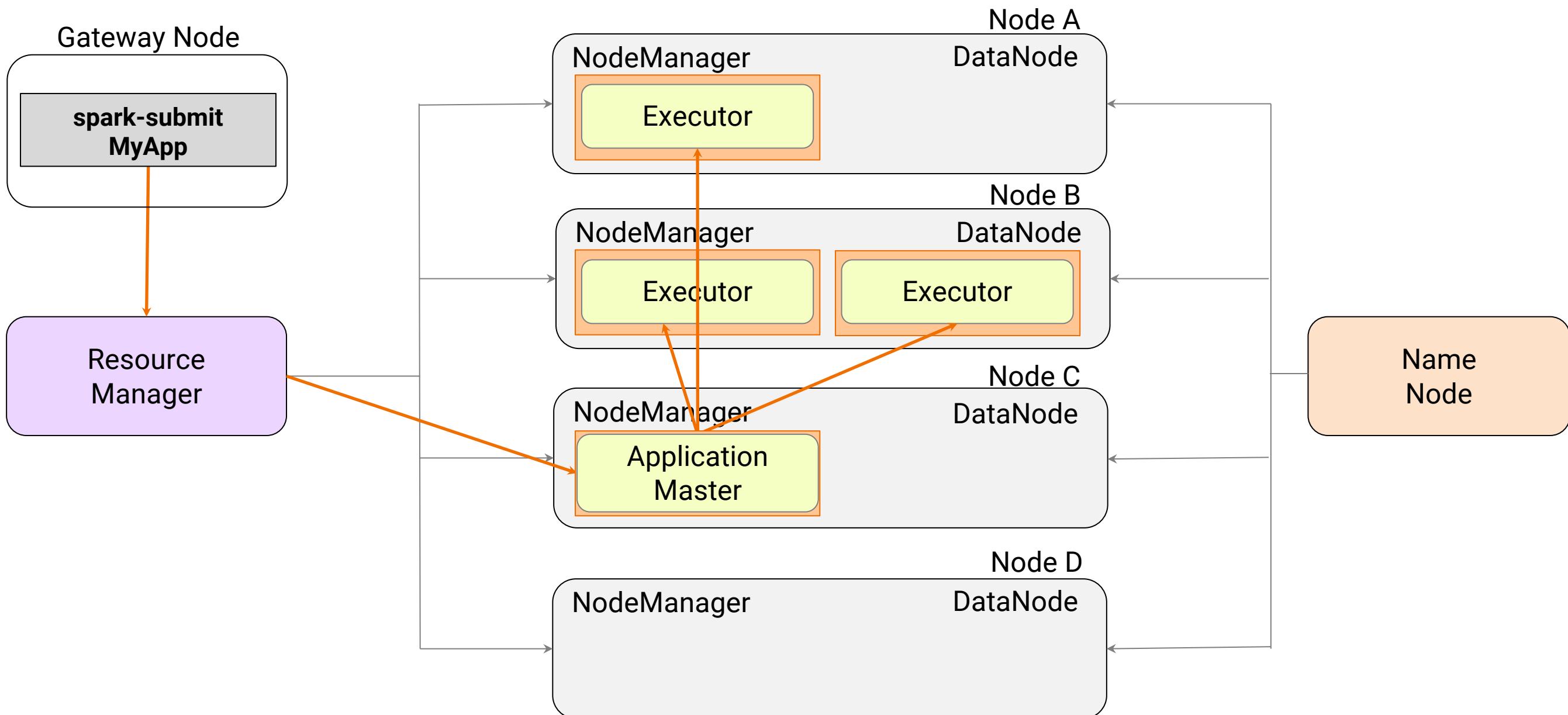
## Running an Application on YARN (3)



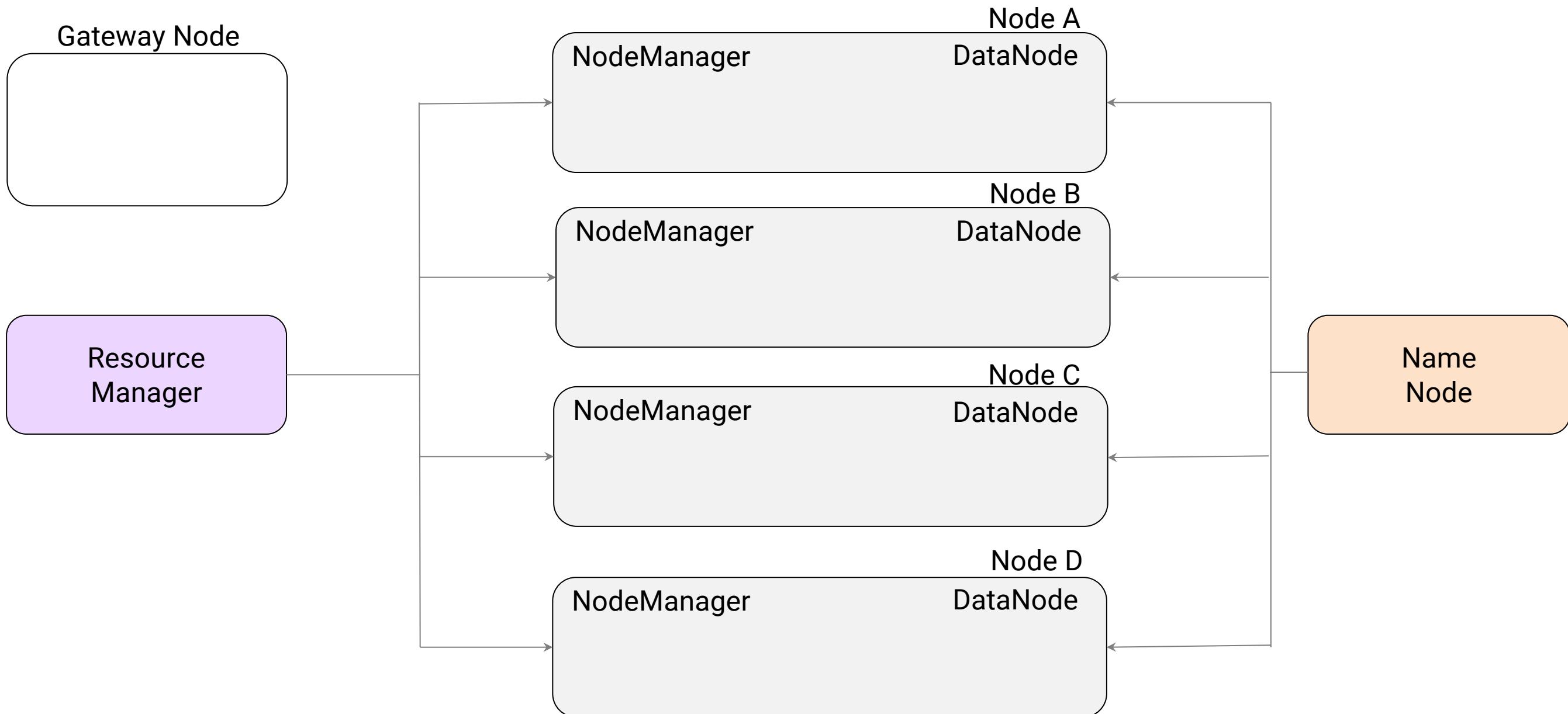
## Running an Application on YARN (4)



# Running an Application on YARN (5)



# Static Resource Allocation (6): Application Terminates



# Chapter Topics

---

## YARN Introduction

- YARN Overview
- YARN Components and Interaction
- **Working with YARN**
- Exercise: Working with YARN

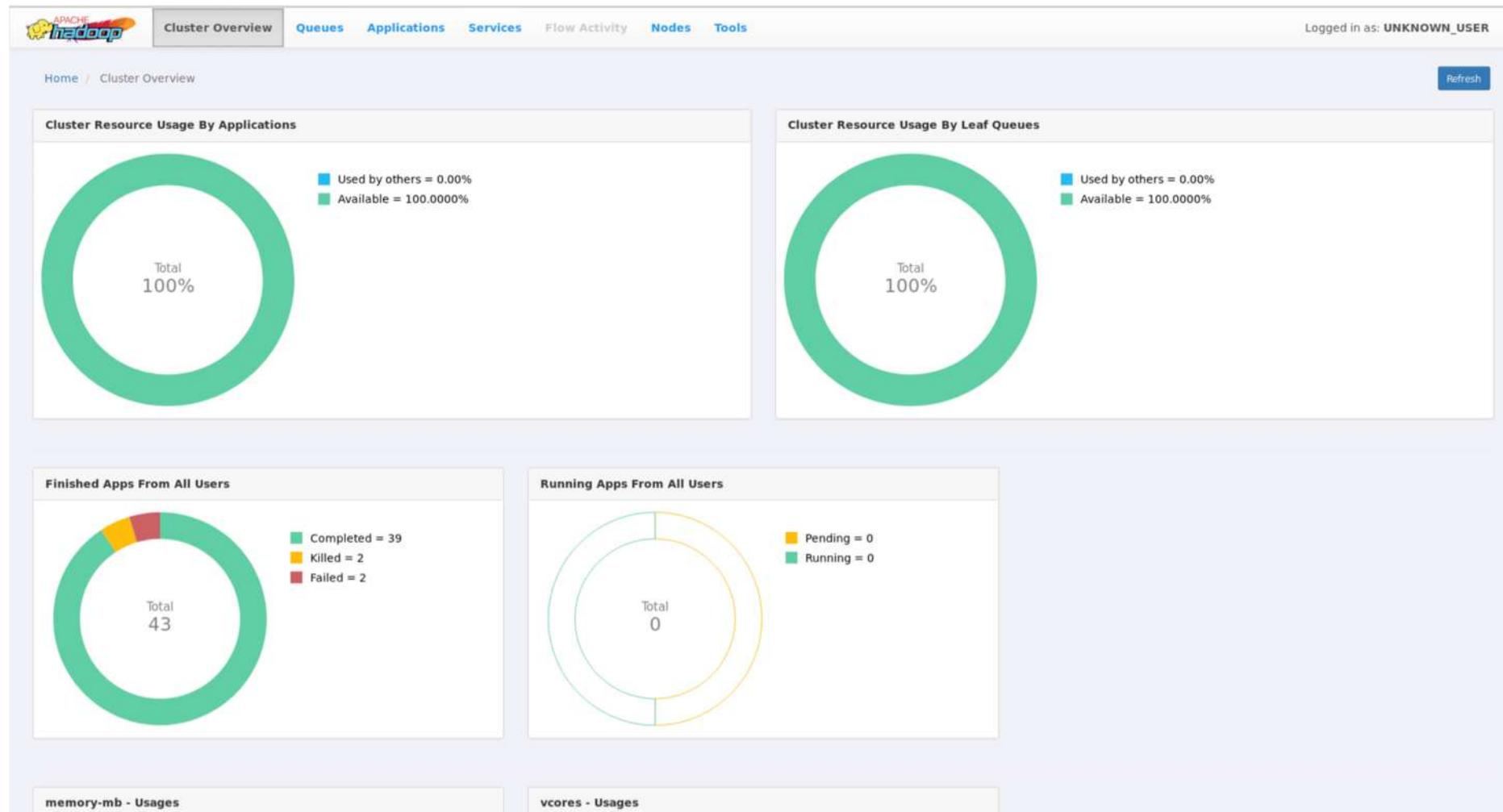
# Working with YARN

---

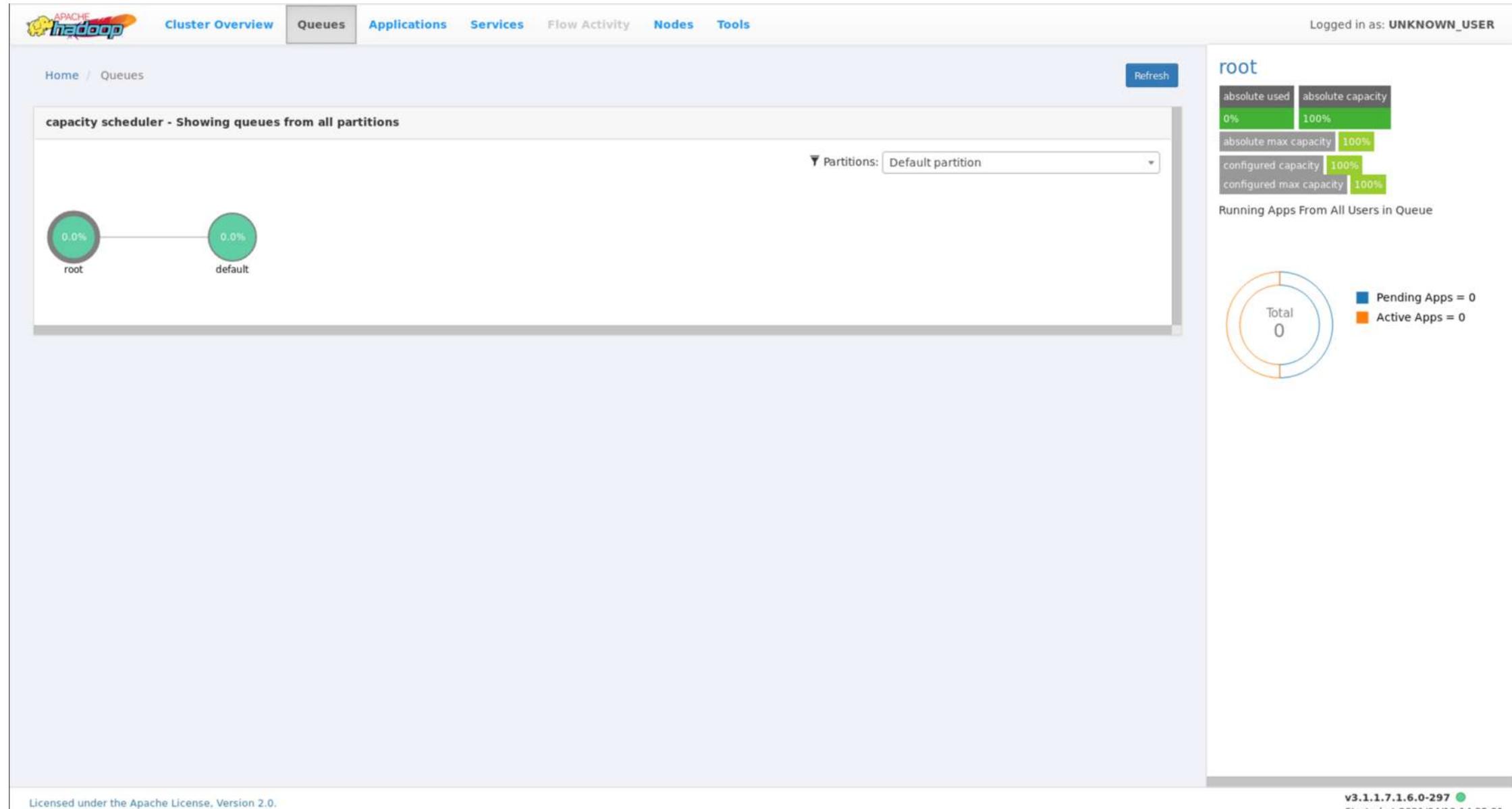
- **Developers need to be able to**
  - Submit jobs (applications) to run on the YARN cluster
  - Monitor and manage jobs
- **YARN tools for developers**
  - The YARN web UI
  - The YARN command line interface

# The YARN Web UI - Cluster Overview

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



# The YARN Web UI - Queues



# The YARN Web UI - Applications

The screenshot shows the Apache Hadoop YARN Web UI Applications page. The top navigation bar includes links for Cluster Overview, Queues, Applications (selected), Services, Flow Activity, Nodes, and Tools. It also displays the user 'Logged in as: UNKNOWN\_USER'. Below the navigation is a breadcrumb trail: Home / Applications, and a Refresh button.

On the left, there are two filter panels:

- User (5):** Includes checkboxes for 'livy' (332), 'hive' (45), 'training' (44), 'admin' (21), and 'hdfs' (5). A 'More' link is present.
- State (3):** Includes checkboxes for 'FINISHED' (373), 'KILLED' (39), and 'FAILED' (35). A 'More' link is present.

Below these filters are 'Apply' and 'Clear' buttons.

The main content area is a table listing 24 completed applications (labeled 1-24) with the following details:

Application ID	Application Type	Application Token	Application Name	User	State	Queue	Progress	Start Time	Elapsed Time
application_1618320331610_0043	SPARK	livy-session-11...	livy-session-11	livy	Finished	default	100%	2021/04/15 12...	3h 6m 50s 652...
application_1618320331610_0042	SPARK	livy-session-10...	livy-session-10	livy	Finished	default	100%	2021/04/14 06...	1h 2m 3s 464ms
application_1618320331610_0041	SPARK	livy-session-9...	livy-session-9	livy	Finished	default	100%	2021/04/14 04...	1h 42m 16s 76...
application_1618320331610_0040	SPARK	livy-session-8...	livy-session-8	livy	Finished	default	100%	2021/04/14 04...	3m 35s 419ms
application_1618320331610_0039	SPARK	livy-session-7...	livy-session-7	livy	Finished	default	100%	2021/04/14 04...	1m 30s 876ms
application_1618320331610_0038	SPARK	livy-session-6...	livy-session-6	livy	Finished	default	100%	2021/04/14 04...	3m 55s 124ms
application_1618320331610_0037	SPARK	livy-session-5...	livy-session-5	livy	Finished	default	100%	2021/04/14 04...	3m 36s 264ms
application_1618320331610_0036	SPARK	livy-session-4...	livy-session-4	livy	Finished	default	100%	2021/04/14 04...	1m 29s 781ms
application_1618320331610_0035	SPARK	livy-session-3...	livy-session-3	livy	Finished	default	100%	2021/04/14 00...	2h 26m 45s 50...
application_1618320331610_0034	SPARK	livy-session-2...	livy-session-2	livy	Finished	default	100%	2021/04/14 00...	26m 17s 947ms
application_1618320331610_0033	SPARK	livy-session-1...	livy-session-1	livy	Finished	default	100%	2021/04/13 21...	2h 14m 34s 72...
application_1618320331610_0032	SPARK	livy-session-27...	livy-session-279	livy	Finished	default	100%	2021/04/13 21...	5m 58s 1ms
application_1618320331610_0031	SPARK	livy-session-27...	livy-session-278	livy	Finished	default	100%	2021/04/13 17...	4m 9s 346ms
application_1618320331610_0030	SPARK	livy-session-27...	livy-session-277	livy	Finished	default	100%	2021/04/13 17...	4m 13s 489ms
application_1618320331610_0029	SPARK	livy-session-27...	livy-session-276	livy	Failed	default	100%	2021/04/13 17...	1m 36s 989ms
application_1618320331610_0028	SPARK	livy-session-27...	livy-session-275	livy	Finished	default	100%	2021/04/13 17...	1m 22s 587ms
application_1618320331610_0027	SPARK	livy-session-27...	livy-session-274	livy	Finished	default	100%	2021/04/13 17...	3m 53s 743ms
application_1618320331610_0026	SPARK	livy-session-27...	livy-session-273	livy	Finished	default	100%	2021/04/13 17...	56s 483ms
application_1618320331610_0025	SPARK	livy-session-27...	livy-session-272	livy	Finished	default	100%	2021/04/13 16...	18m 9s 832ms
application_1618320331610_0024	SPARK	livy-session-27...	livy-session-271	livy	Finished	default	100%	2021/04/13 16...	3m 5s 38ms

# The YARN Web UI - Nodes

APACHE  
hadoop

Cluster Overview   Queues   Applications   Services   Flow Activity   Nodes   Tools

Logged in as: UNKNOWN\_USER

Home / Nodes   Refresh

Information   Node Status   Nodes Heatmap Chart

Search...   Search   1   25 Rows

Node Label (1)   Node State (1)

default    RUNNING

More   More

Apply   Clear

Node Label	Rack	Node State	Node Address	Node HTTP Address	Containers	Mem Used	Mem Available	Vcores Use
default	/default	RUNNING	localhost.localdomain:8041	localhost.localdomain:8042	0	0 B	4 GB	0

Licensed under the Apache License, Version 2.0.

v3.1.1.7.1.6.0-297   Started at 2021/04/13 14:25:31

# YARN Command Line

---

- **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`
  - View the logs of the specified application
    - `$ yarn logs -applicationId app-id`
  - View the full list of command options
    - `$ yarn -help`

# Kubernetes - K8s

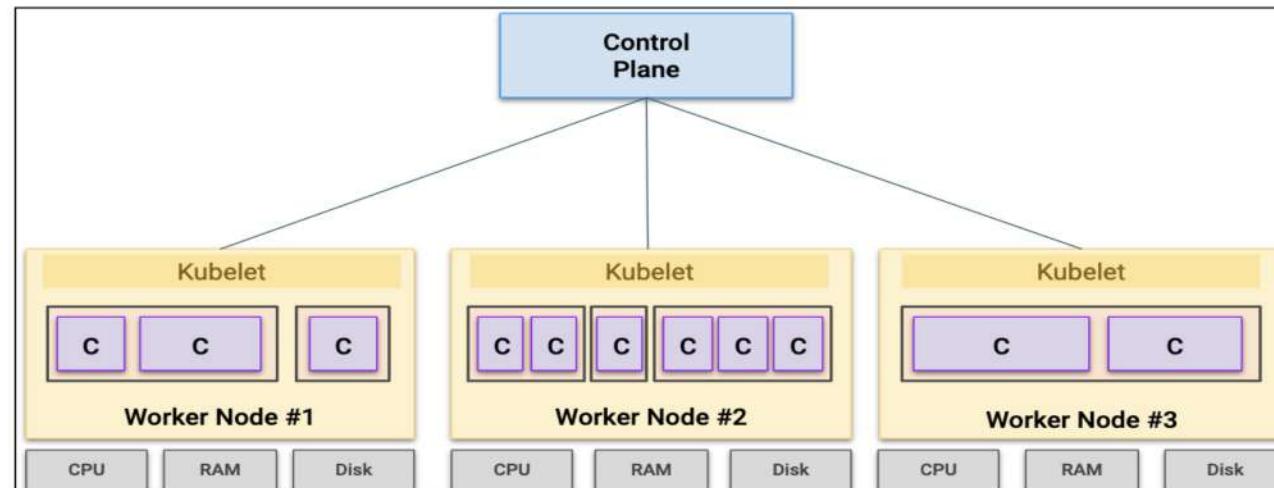
---

- **Kubernetes = Pilot in Greek**
- **Open source Container cluster manager developed at Google**
- **Deployment, maintenance and scaling of containerized applications**
- **Uses Docker**
- **Software system to deploy, scale, and manage containerized applications**
  - Highly efficient elastic Cloud deployment. Scale up and down in seconds
- **Supported by all major cloud providers and private cloud or burst to cloud**
- **Collection of machines running Kubernetes software is called a "cluster"**
  - Groups applications in Pods. Scalable Docker applications
- **CDP based experiences are using Kubernetes for resource management**



# Kubernetes Hierarchy of Concepts

- **Kubelet - container agent**
  - Uses a set of container manifest (yaml) that each describes a pod
  - Reacts to file, http endpoint, watch on a etcd server, http server requests
- **Pod -The smallest and simplest Kubernetes object. A Pod represents a set of running containers on your cluster**
- **Service - Logical set of pods**



## Knowledge Check

---

- 1. The master node service is called the \_\_\_\_\_ and the \_\_\_\_\_ runs on the worker nodes.**
  
- 2. Which Container resource type is the driver for most resource requests?**
  
- 3. True/False? ApplicationMasters execute on master nodes.**
  
- 4. What component is responsible for dealing with a Container failure?**
  
- 5. True/False? Capacity Scheduler queues are aligned with specific worker nodes.**

## Essential Points

---

- YARN enables multiple workloads to execute simultaneously in the cluster
- The ResourceManager is the master process responsible for fulfilling resource requests and the NodeManager resides on the worker nodes along with the actual Containers that fulfill job functions
- The ApplicationMaster resides within a Container and is the process responsible for running a job (batch or long-lived service) and making appropriate resource requests
- The Capacity Scheduler allows for resource sharing that enables SLA-compliant multi-tenancy
- Use the YARN ResourceManager web UI or the `yarn` command to monitor applications
- Kubernetes is the future of cluster resource management

# Chapter Topics

---

## YARN Introduction

- YARN Overview
- YARN Components and Interaction
- Working with YARN
- **Exercise: Working with YARN**



# Distributed Processing History

---

Chapter 5

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- **Distributed Processing History**
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Lesson Objectives

---

**By the end of this chapter, you will be able to:**

- **Describe how MapReduce works**
  - Explain the reliance on the Key Value Pair (KVP) paradigm
  - Illustrate the MapReduce framework with simple examples
- **Understand the shortcomings of this design and how second generation distributed frameworks addressed them**

# Chapter Topics

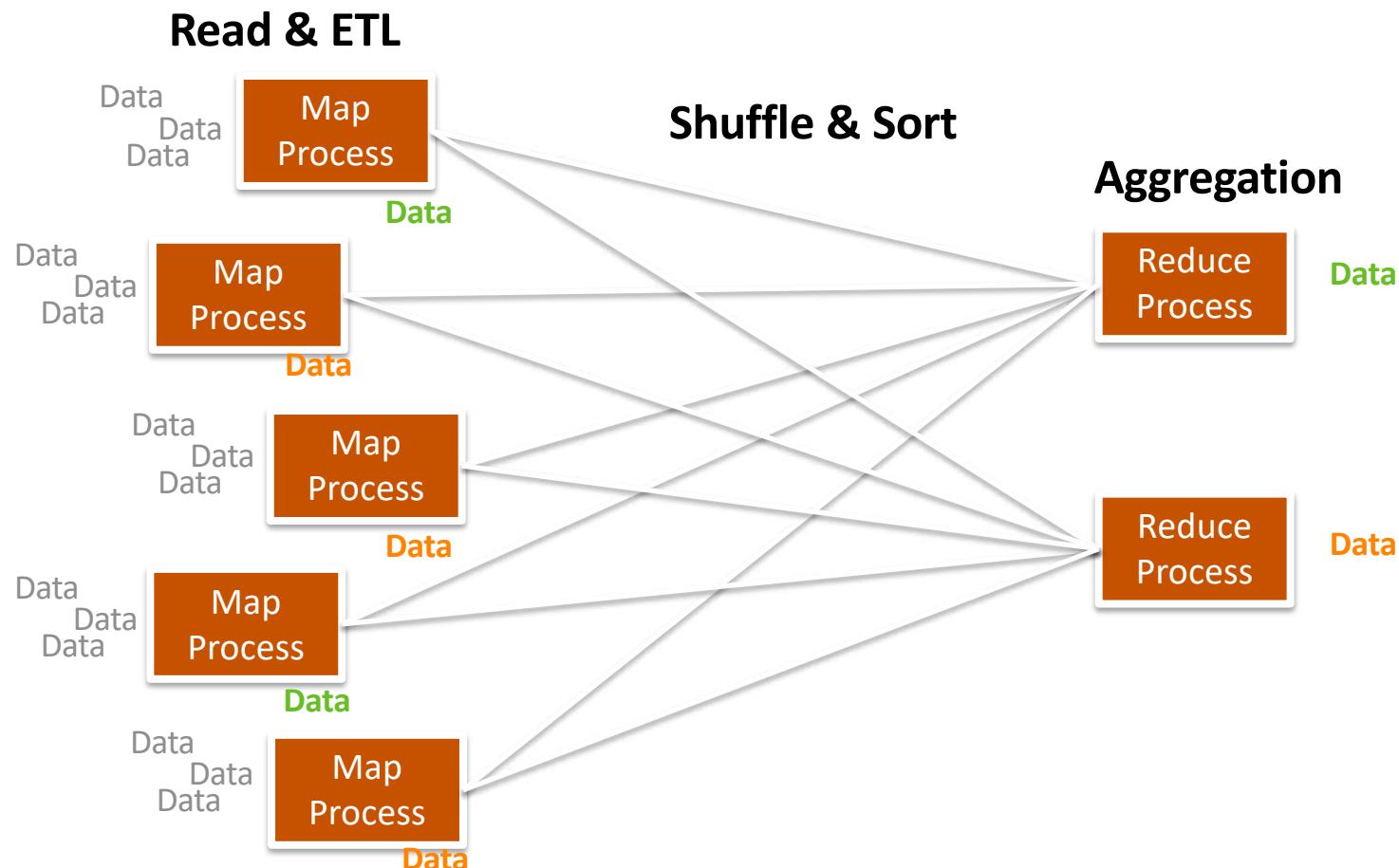
---

## Distributed Processing History

- **The Disk Years: 2000 ->2010**
- **The Memory Years: 2010 ->2020**
- **The GPU Years: 2020 ->**

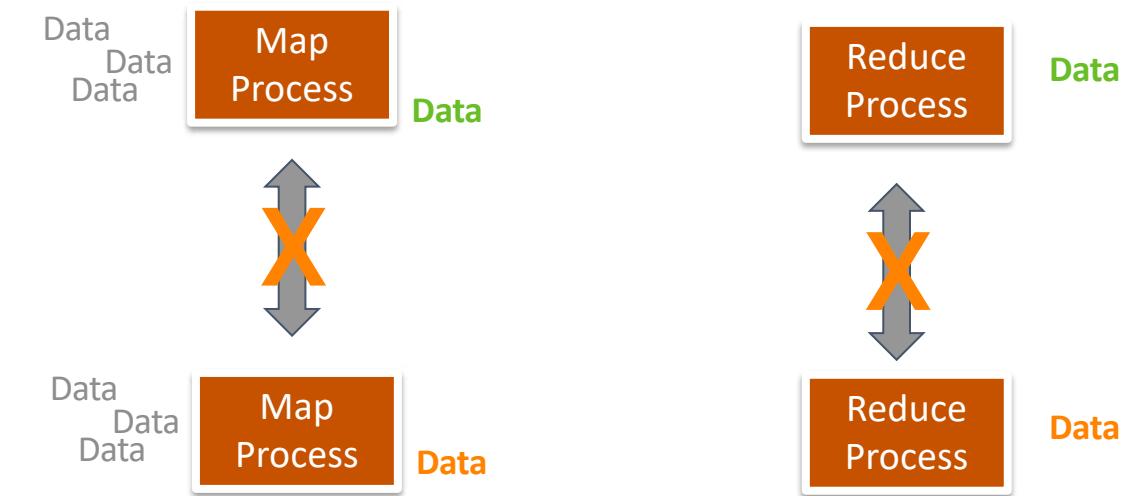
# What is MapReduce?

Breaking a large problem into sub-solutions



# Shared Nothing Model

- MapReduce uses a simple distributed processing model called a Shared Nothing Model
- In this model
  - Mappers do not communicate with each other
  - Reducers do not communicate with each other
- This model does not work for some problems
  - Like solving a puzzle
- But it does work for what programs have been doing for years
  - Counting things!



# Simple Algorithm

---

1. Review stack of quarters
2. Count each year that ends in an even number

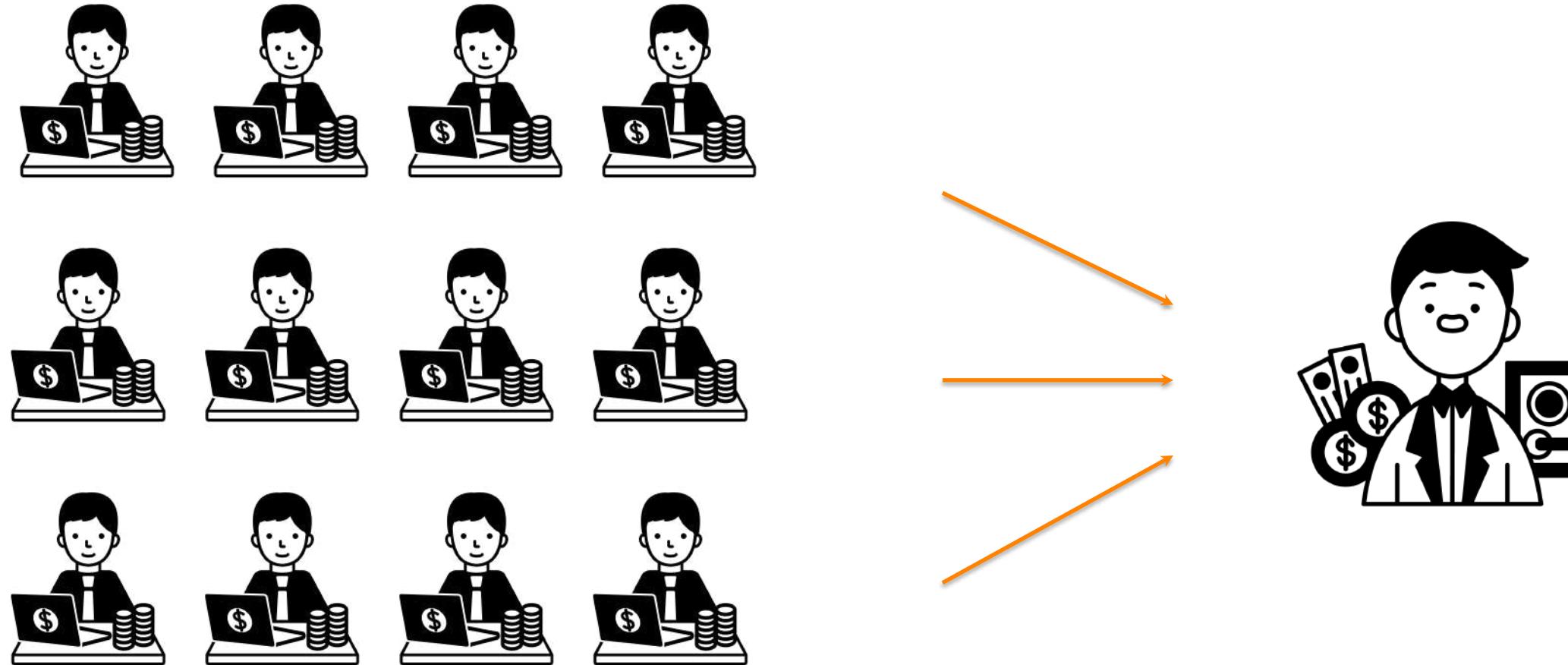


# Processing at Scale

---



# Distributed Algorithm – MapReduce



# The Mapper

---

- **The Mapper reads data in the form of key/value pairs (KVPs)**
- **It outputs zero or more KVPs**
- **The Mapper may use or completely ignore the input key**
  - For example, a standard pattern is to read a line of a file at a time
    - The key is the byte offset into the file at which the line starts
    - The value is the contents of the line itself
    - Typically the key is considered irrelevant with this pattern
- **If the Mapper writes anything out, it must in the form of KVPs**
  - This “intermediate data” is NOT stored in HDFS (local storage only without replication)

# MapReduce Example – Map Phase

---

## Input to Mapper

(8675, 'I will not eat green eggs and ham')  
(8709, 'I will not eat them Sam I am')  
...

## Output from Mapper

('I', 1), ('will', 1), ('not', 1), ('eat', 1), ('green', 1),  
(eggs', 1), ('and', 1),  
(ham', 1), ('I', 1), ('will', 1),  
(not', 1), ('eat', 1),  
(them', 1), ('Sam', 1),  
(I', 1), ('am', 1)

- Ignoring the key
  - It is just an offset
- In this example
  - The size of the output > size of the input
  - No attempt is made to optimize within a record in this example
    - This is a great use case for a “Combiner”

# The Shuffle

---

- After the Map phase is over, all the outputs from the mappers are sent to reducers
- KVPs with the same key will be sent to the same reducer
  - By default  $(k,v)$  will be sent to the reducer number  $\text{hash}(k) \% \text{numReducers}$
- This can potentially generate a lot of network traffic on your cluster
  - In our word count example the size of the output data is of the same order of magnitude as our input data
- Some very common operations like join, or group by require a lot of shuffle by design
- Optimizing these operations is an important part of mastering distributed processing programming
- CPU and RAM can scale by adding worker nodes, network can't

# MapReduce Example – Reduce Phase

---

## Input to Reducer

(‘I’, [1, 1, 1])  
('Sam', [1])  
('am', [1])  
('and', [1])  
('eat', [1, 1])  
('eggs', [1])  
('green', [1])  
('ham', [1])  
('not', [1, 1])  
('them', [1])  
('will', [1, 1])

## Output from Reducer

(‘I’, 3)  
('Sam', 1)  
('am', 1)  
('and', 1)  
('eat', 2)  
('eggs', 1)  
('green', 1)  
('ham', 1)  
('not', 2)  
('them', 1)  
('will', 2)

- Notice keys are sorted and associated values for same key are in a single list
  - Shuffle & Sort did this for us

- All done!

# The Reducer

---

- After the Shuffle phase is over, all the intermediate values for a given intermediate key are sorted and combined together into a list
- This list is given to a Reducer
  - There may be a single Reducer, or multiple Reducers
  - All values associated with a particular intermediate key are guaranteed to go to the same Reducer
  - The intermediate keys, and their value lists, are passed in sorted order
- The Reducer outputs zero or more KVPs
  - These are written to HDFS
  - In practice, the Reducer often emits a single KVP for each input key

# Recap of Word Count

---

GreenEggsAndHam.txt



HDFS

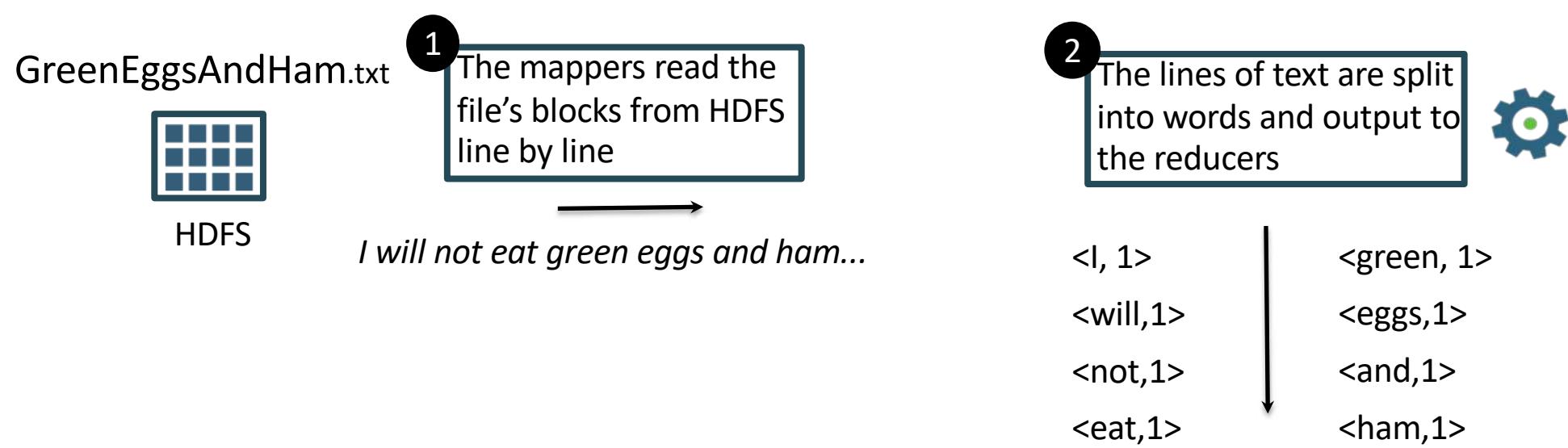
1

The mappers read the  
file's blocks from HDFS  
line by line

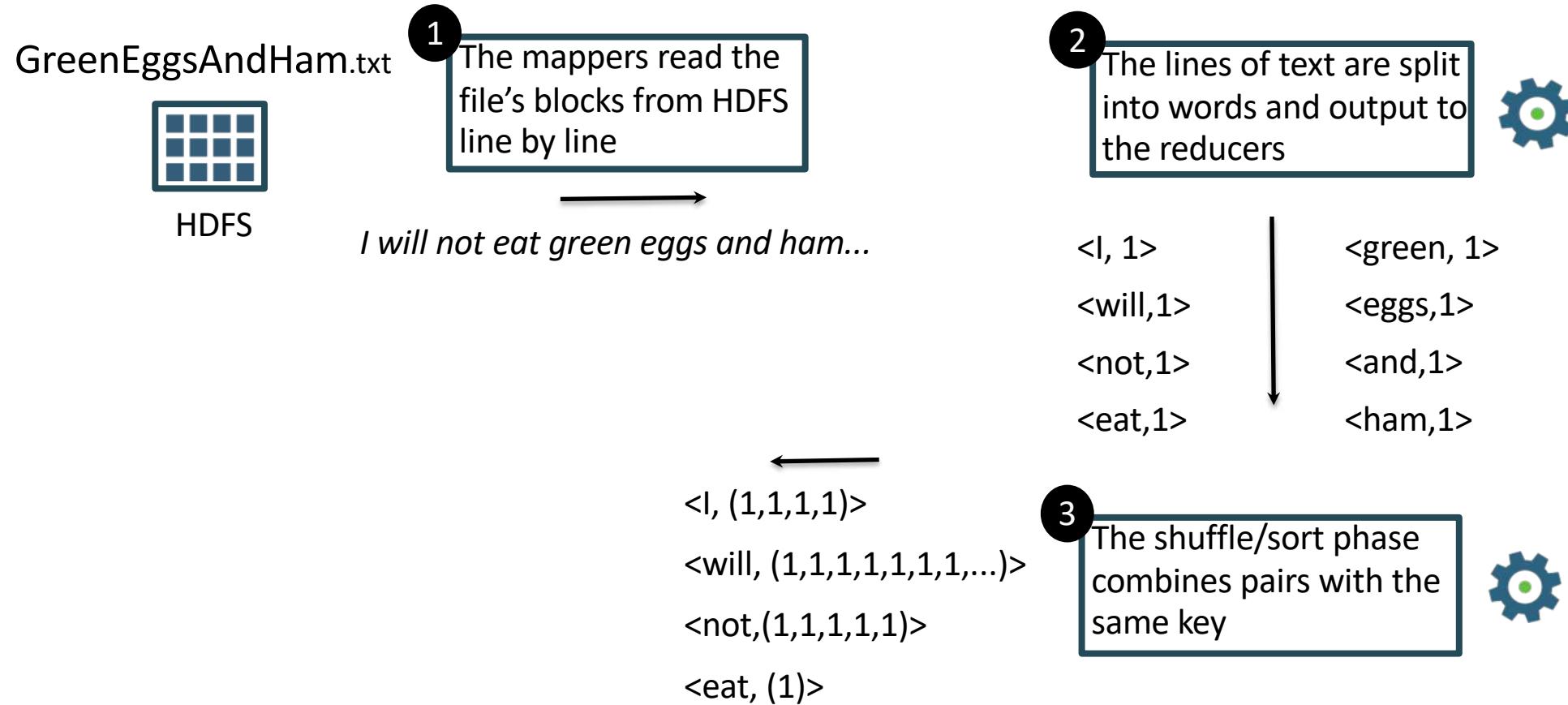


*I will not eat green eggs and ham...*

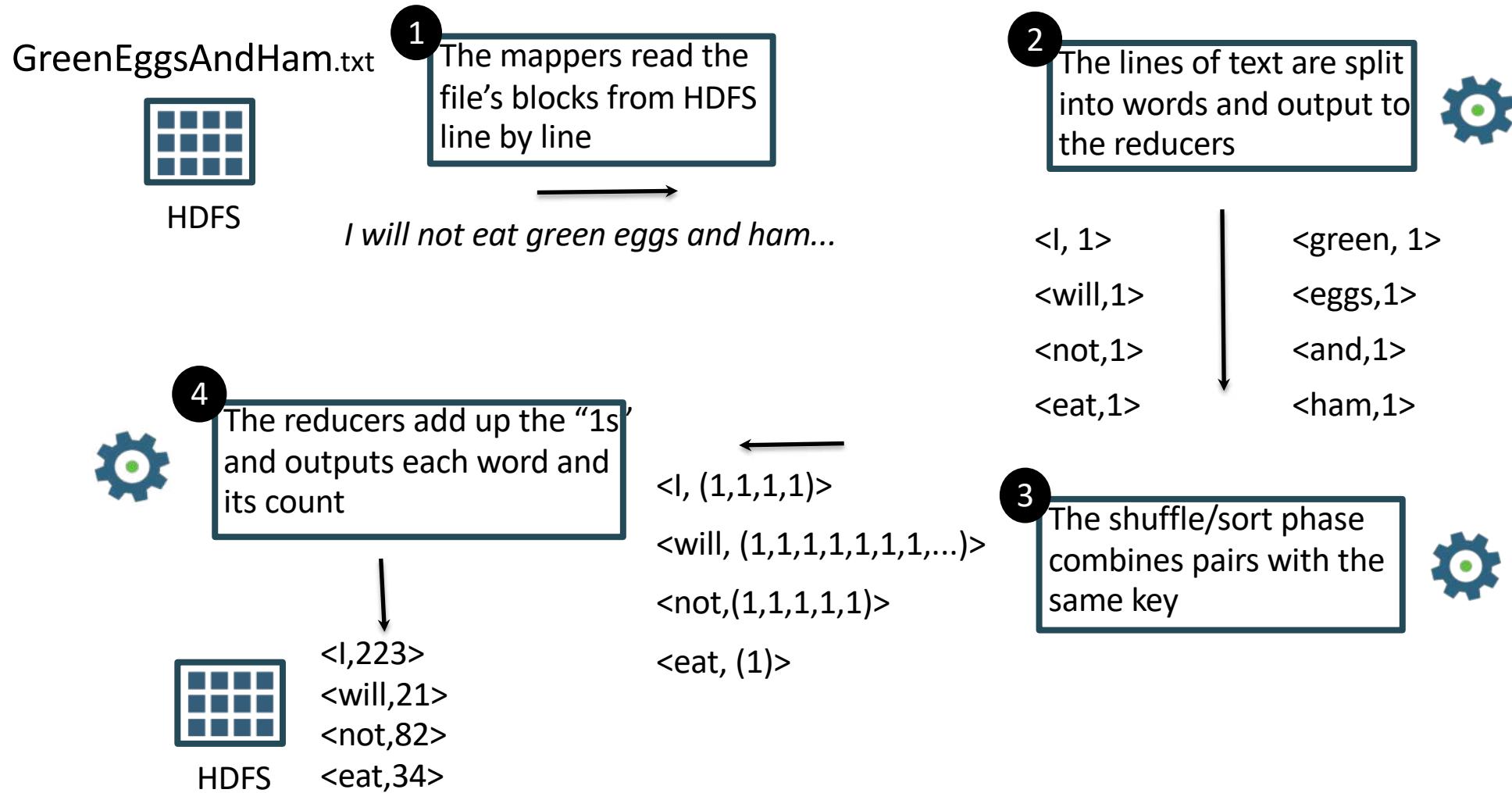
# Recap of Word Count



# Recap of Word Count



# Recap of Word Count



# MapReduce Example – Word Count

## The Mapper

```
public class WordCount {  
  
    public static class TokenizerMapper  
        extends Mapper<Object, Text, Text, IntWritable>{  
  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
  
        public void map(Object key, Text value, Context context  
                        ) throws IOException, InterruptedException {  
            StringTokenizer itr = new StringTokenizer(value.toString());  
            while (itr.hasMoreTokens()) {  
                word.set(itr.nextToken());  
                context.write(word, one);  
            }  
        }  
    }  
}
```

# MapReduce Example – Word Count

## The Reducer

```
public static class IntSumReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                      Context context
                      ) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

# MapReduce Example – Word Count

## The Main

```
public static void main(String[] args) throws Exception {  
    Configuration conf = new Configuration();  
    Job job = Job.getInstance(conf, "word count");  
    job.setJarByClass(WordCount.class);  
    job.setMapperClass(TokenizerMapper.class);  
    job.setCombinerClass(IntSumReducer.class);  
    job.setReducerClass(IntSumReducer.class);  
    job.setOutputKeyClass(Text.class);  
    job.setOutputValueClass(IntWritable.class);  
    FileInputFormat.addInputPath(job, new Path(args[0]));  
    FileOutputFormat.setOutputPath(job, new Path(args[1]));  
    System.exit(job.waitForCompletion(true) ? 0 : 1);  
}
```

# Chapter Topics

---

## Distributed Processing History

- The Disk Years: 2000 ->2010
- **The Memory Years: 2010 ->2020**
- The GPU Years: 2020 ->

# The Second Generation of Distributed Processing

---

- The MapReduce framework unlocked the power of distributed processing but its design was built around the shortcomings of the hardware of 2000 :
  - Not reliable enough + not a lot of memory => save everything to disk all the time
  - Limited to a Map Reduce graph with a single shuffle
- In 2010 the hardware landscape has changed, Moore's law no longer applies but you have more
  - memory
  - disk
  - network bandwidth
- In 2009 a graduate student at Berkeley started a project to create a general purpose data processing engine based on in-memory distributed computing to overcome the shortcomings of the MapReduce framework

# Why Spark?

---

- **Elegant Developer APIs: DataFrames/SQL, Machine Learning, Graph algorithms and streaming**
  - Scala, Python, Java and R
  - Single environment for importing, transforming, and exporting data
- **In-memory computation model**
  - Effective for iterative computations
- **High level API**
  - Allows users to focus on the business logic and not internals
- **Supports wide variety of workloads**
  - MLlib for Data Scientists
  - Spark SQL for Data Analysts
  - Spark Streaming for micro batch use cases
  - Spark Core, SQL, Streaming, Mllib, and GraphX for Data Processing Applications
- **Integrated fully with Hadoop and an open source tool**
- **Faster than MapReduce**

# Spark vs MapReduce

- Higher level API
- In-memory data storage
  - Up to 100 x performance improvement



## Pyspark

```
text_file = spark.textFile("hdfs://...")  
counts = text_file.flatMap(lambda line: line.split(" ")) \  
    .map(lambda word: (word, 1)) \  
    .reduceByKey(lambda a, b: a + b)  
counts.saveAsTextFile("hdfs://...")
```

## Java MapReduce

```
package org.myorg;  
  
import java.io.IOException;  
import java.util.*;  
  
import org.apache.hadoop.fs.Path;  
import org.apache.hadoop.conf.*;  
import org.apache.hadoop.io.*;  
import org.apache.hadoop.mapreduce.*;  
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;  
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;  
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;  
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;  
  
public class WordCount {  
  
    public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {  
        private final static IntWritable one = new IntWritable(1);  
        private Text word = new Text();  
  
        public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {  
            String line = value.toString();  
            StringTokenizer tokenizer = new StringTokenizer(line);  
            while (tokenizer.hasMoreTokens()) {  
                word.set(tokenizer.nextToken());  
                context.write(word, one);  
            }  
        }  
    }  
  
    public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {  
  
        public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {  
            int sum = 0;  
            for (IntWritable val : values) {  
                sum += val.get();  
            }  
            context.write(key, new IntWritable(sum));  
        }  
    }  
  
    public static void main(String[] args) throws Exception {  
        Configuration conf = new Configuration();  
  
        Job job = new Job(conf, "wordcount");  
  
        job.setOutputKeyClass(Text.class);  
        job.setOutputValueClass(IntWritable.class);  
  
        job.setMapperClass(Map.class);  
        job.setReducerClass(Reduce.class);  
  
        job.setInputFormatClass(TextInputFormat.class);  
        job.setOutputFormatClass(TextOutputFormat.class);  
  
        FileInputFormat.addInputPath(job, new Path(args[0]));  
        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
  
        job.waitForCompletion(true);  
    }  
}
```

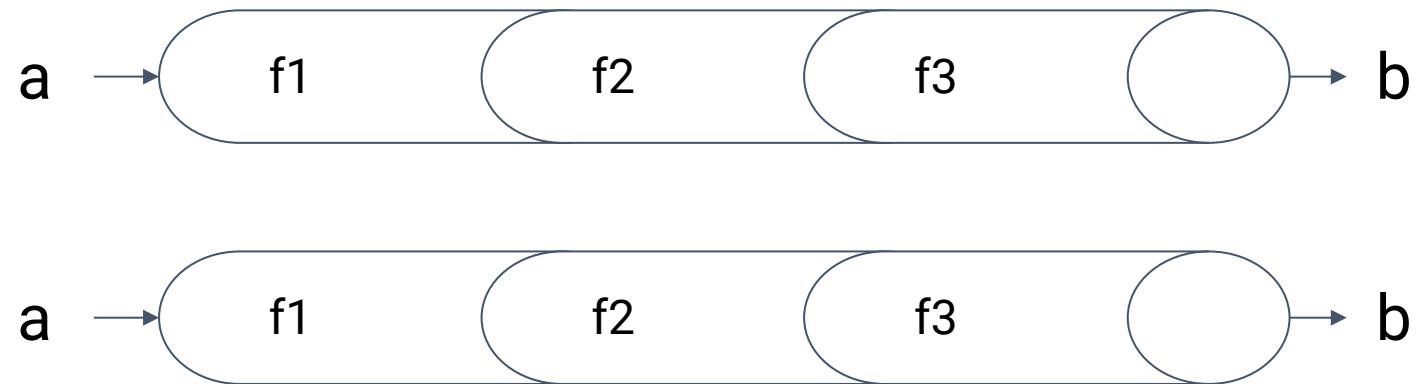
# Why Functional Programming?

---

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

# Immutability + Functional Programming = Distributed Processing!

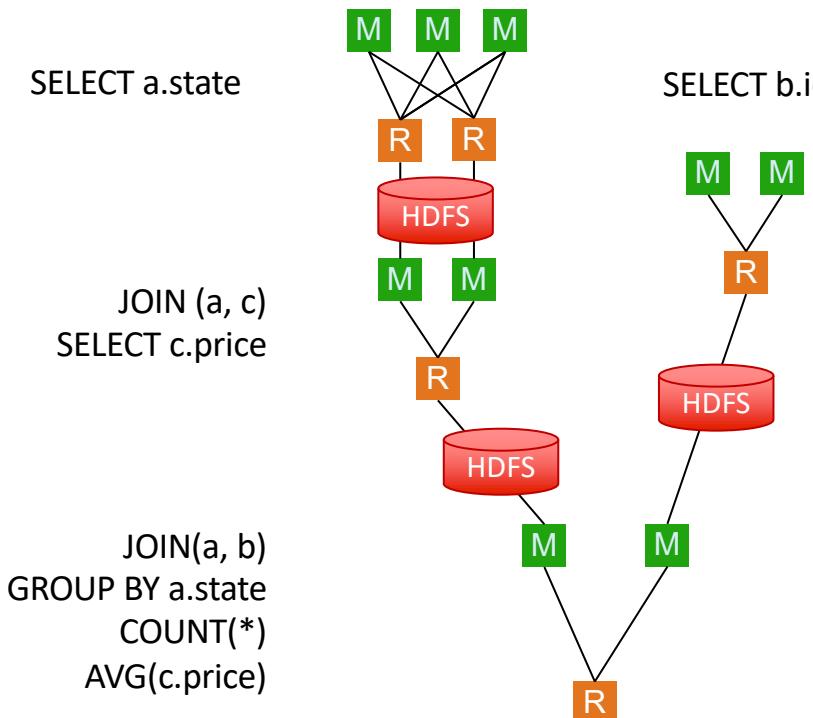
- Pipelines of functions create the same outputs from immutable inputs
  - This in turn allows for safe aggregation



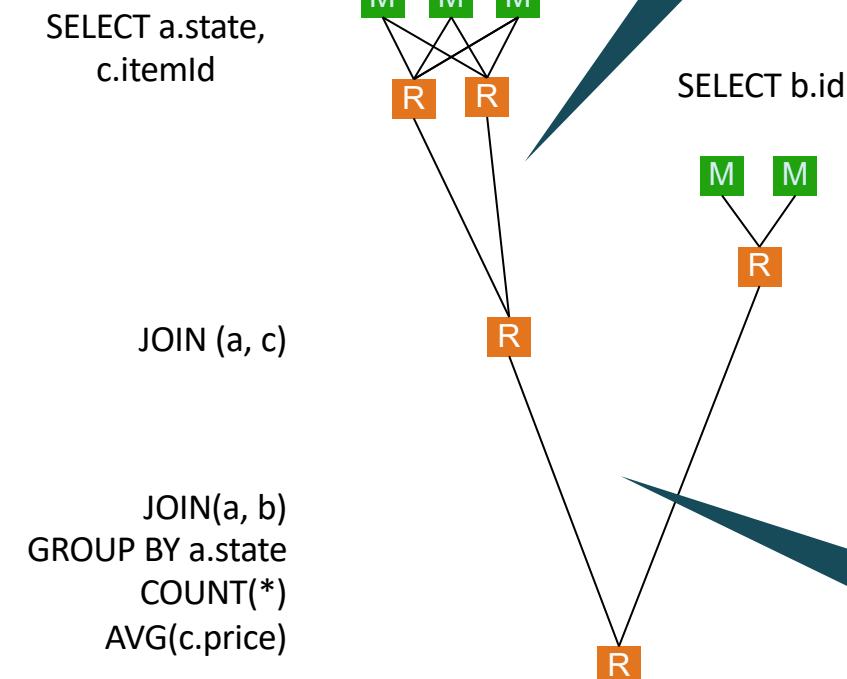
```
SELECT a.state, COUNT(*), AVG(c.price) FROM a
JOIN b ON (a.id = b.id)
JOIN c ON (a.itemId = c.itemId)
GROUP BY a.state
```

Tez avoids unneeded writes to HDFS

### Hive - MapReduce



### Hive - Tez



Tez allows reduce-only jobs

# Chapter Topics

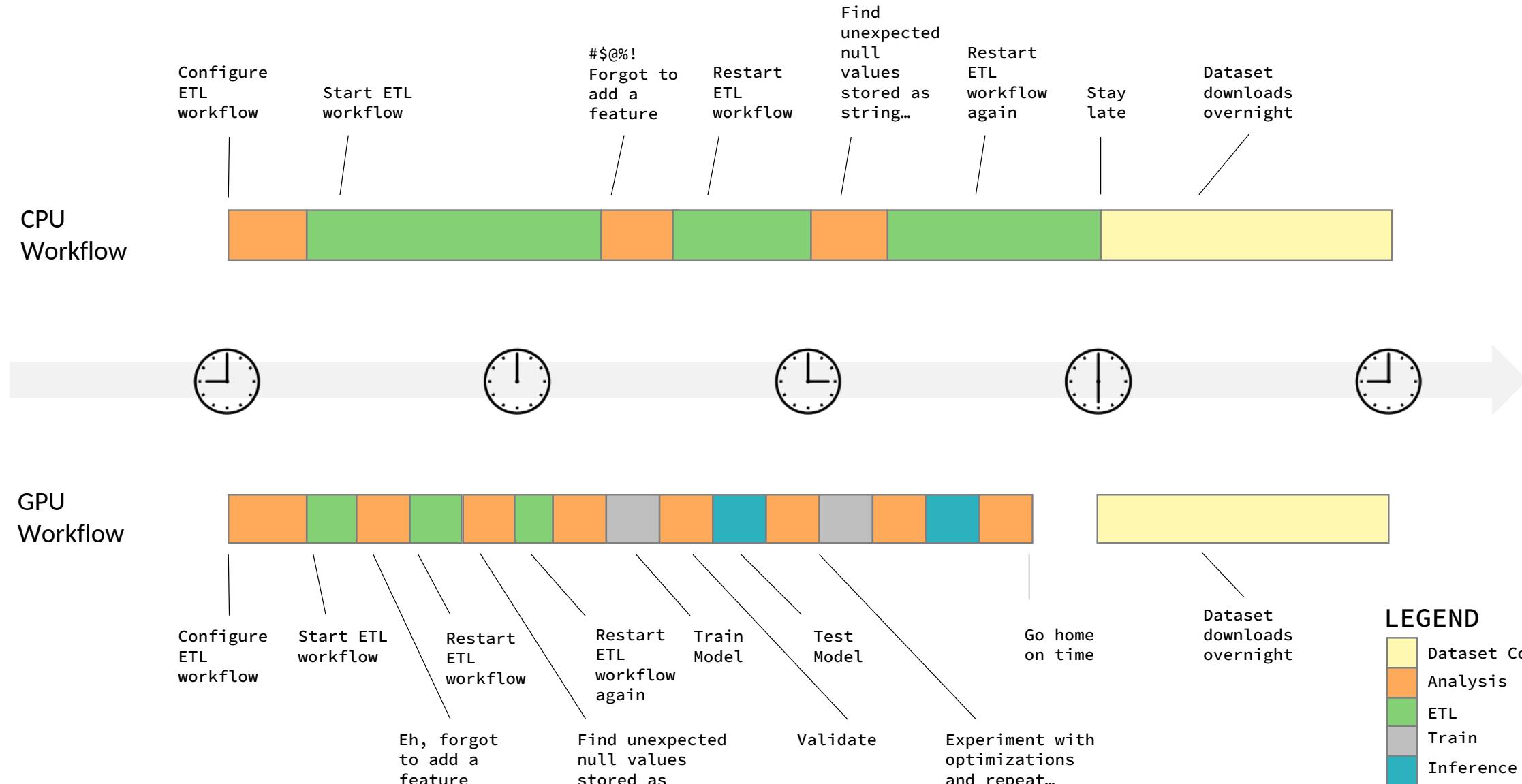
---

## Distributed Processing History

- The Disk Years: 2000 ->2010
- The Memory Years: 2010 ->2020
- **The GPU Years: 2020 ->**

# The Challenges of Data Workflows Today

- The Average Data Scientist Spends 80% of their Time in ETL as Opposed to Training Models



# Accelerating Apache Spark 3.0 with GPUs and RAPIDS

---

NVIDIA has worked with the Apache Spark community to implement GPU acceleration through the release of Spark 3.0 and the opensource RAPIDS Accelerator for Spark. In this post, we dive into how the [RAPIDS Accelerator for Apache Spark](#) uses GPUs to:

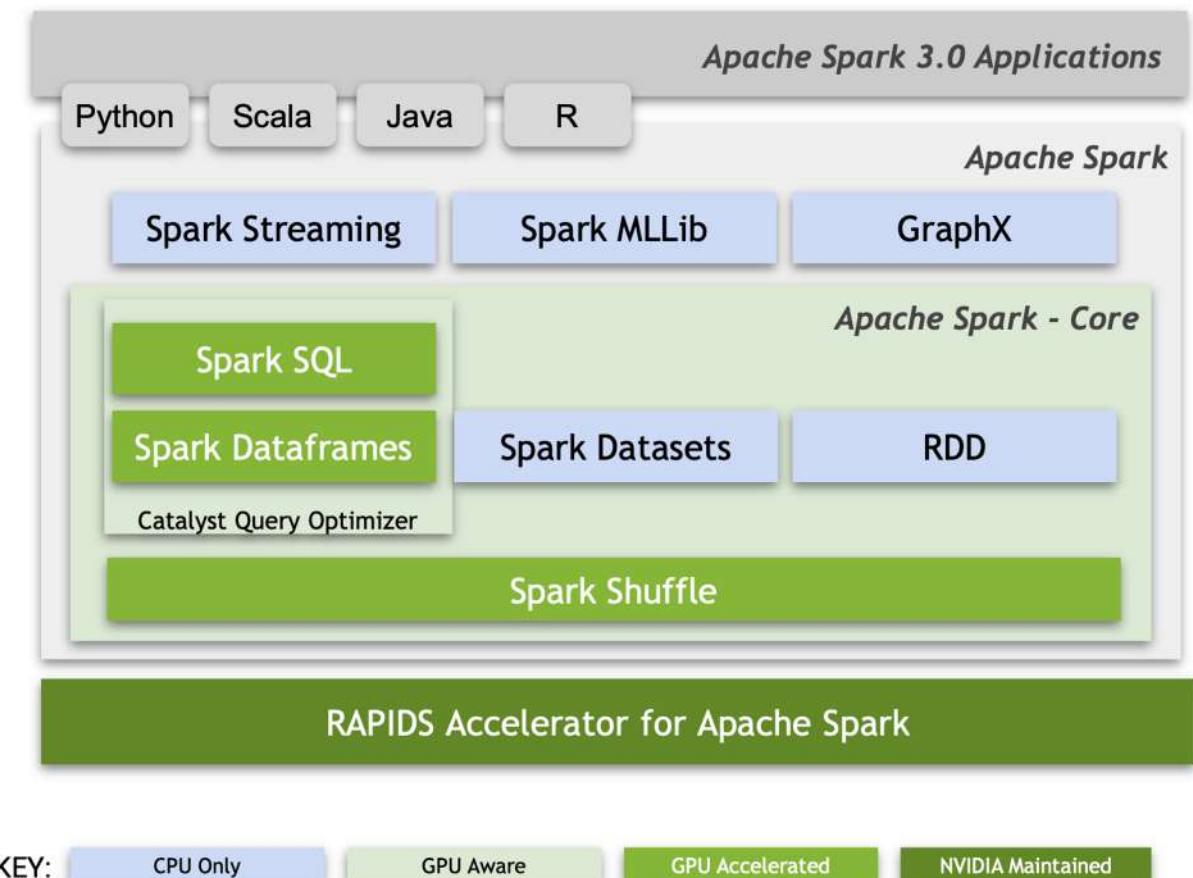
- Accelerate end-to-end data preparation and model training on the same Spark cluster.
- Accelerate Spark SQL and DataFrame operations without requiring any code changes.
- Accelerate data transfer performance across nodes (Spark shuffles).

Source: <https://developer.nvidia.com/blog/accelerating-apache-spark-3-0-with-gpus-and-rapids/>

# Data Workflow Software at NVIDIA

## GPU-Accelerated Apache Spark Implementation

- Transparent acceleration of any Apache Spark application - without code changes!
- Requires:
  - Apache Spark - Version 3.0 or later
  - RAPIDS Accelerator for Apache Spark
  - Hardware with NVIDIA GPUs
- Accelerated Operations:
  - Any code using the Spark SQL or Spark Dataframe interfaces, and any Spark Shuffle operations
  - Maximum benefit for longer running, highly compute bound Spark applications
- Continuous/Ongoing Improvement:
  - Applications will inherit performance gains as NVIDIA expands coverage of Spark interfaces, operations, data types (etc) and other optimizations



# End to End GPU Acceleration for Spark Workloads

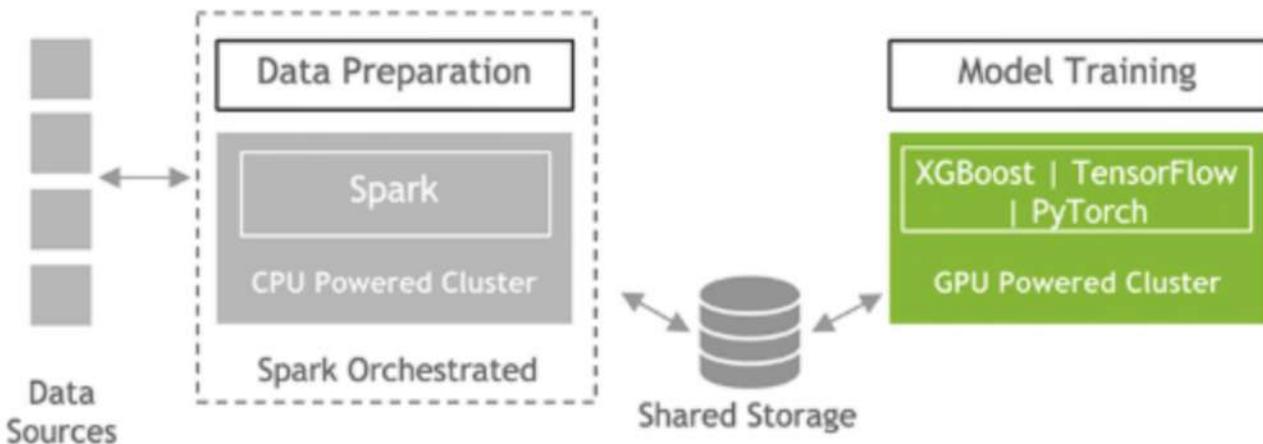
CLOUDERA

NVIDIA.

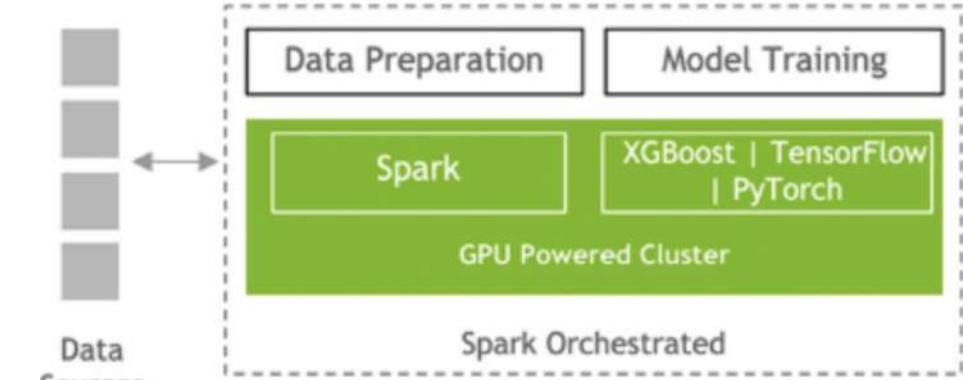
APACHE  
Spark™

RAPIDS

## Spark 2.x



## Spark 3.0



Source: <https://developer.nvidia.com/blog/accelerating-apache-spark-3-0-with-gpus-and-rapids/>

## Knowledge Check

---

- What are the two primary phases of the MapReduce framework? No, this is not a trick question.
- What dictates the number of Mappers that are run? Same question for the Reducers.
- How many input & output KVPs are passed into, and emitted out of, the Mappers? Same question for the Reducers.
- True/False? It is possible to have a Reducer-only job.
- Why were frameworks like Pig and Hive built on top of MapReduce? Again, not a trick question...
- Spark introduced completely new concepts in distributed processing which account for the performance increase
- The RAPIDS library requires you to change your Spark code to leverage GPUs

## Essential Points

---

- MapReduce is the foundational framework for processing data at scale because of its ability to break a large problem into any smaller ones
- Mappers read data in the form of KVPs and each call to a Mapper is for a single KVP; it can return 0..m KVPs
- The framework shuffles & sorts the Mappers' outputted KVPs with the guarantee that only one Reducer will be asked to process a given Key's data
- Reducers are given a list of Values for a specific Key; they can return 0..m KVPs
- Due to the fine-grained nature of the framework, many use cases are better suited for higher-order tools



# Working with RDDs

---

## Chapter 6

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs**
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Lesson Objectives

---

**By the end of this chapter, you will be able to:**

- Explain what RDDs are
- Load and save RDDs with a variety of data source types
- Transform RDD data and return query results

# Chapter Topics

---

## Working with RDDs

- **Resilient Distributed Datasets (RDDs)**
- Exercise: Working with RDDs

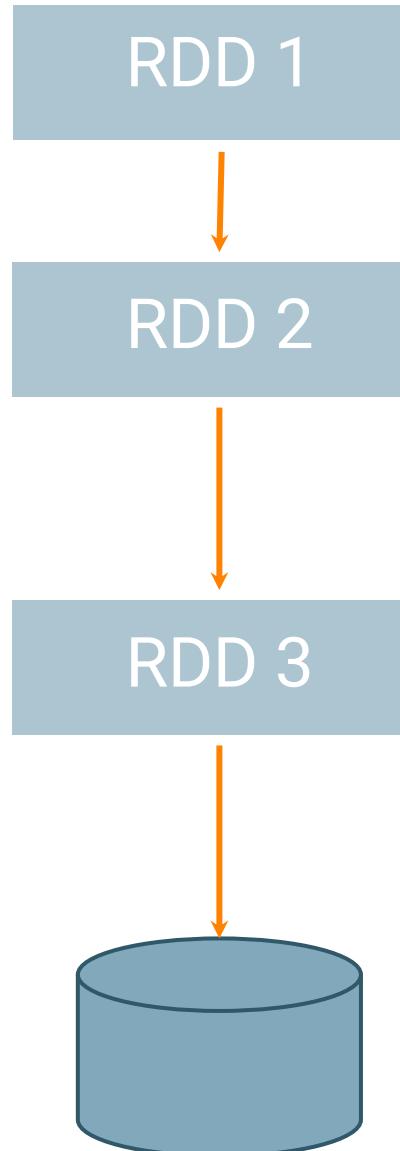
# Resilient Distributed Datasets (RDDs)

---

- **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
- **NOTE: DataFrames/Datasets have replaced RDD as the preferred API for Spark programming**
  - However, they are still a core part of the Spark engine, and useful to know in understanding Spark
    - DataFrame/Dataset code is translated to RDD code
    - There is also still plenty of RDD code in use
    - You may run into it

# RDD Lifecycle

- **RDD is created by either:**
  - Loading an external dataset
  - Distributing a local collection
- **RDD is transformed:**
  - e.g. filter out elements
  - Result: A new RDD
  - Often have a sequence of transformations
- **Data is eventually extracted**
  - By an action on an RDD
  - e.g. save the data
- **On the right, we read/transform a log file, then save the result**



Create: Read a log file  
(e.g. from HDFS)  
sc.textFile("server.log")

Filter: Keep lines starting with  
"ERROR"

Filter: Keep lines containing  
"mysql"

Action: Write result to storage

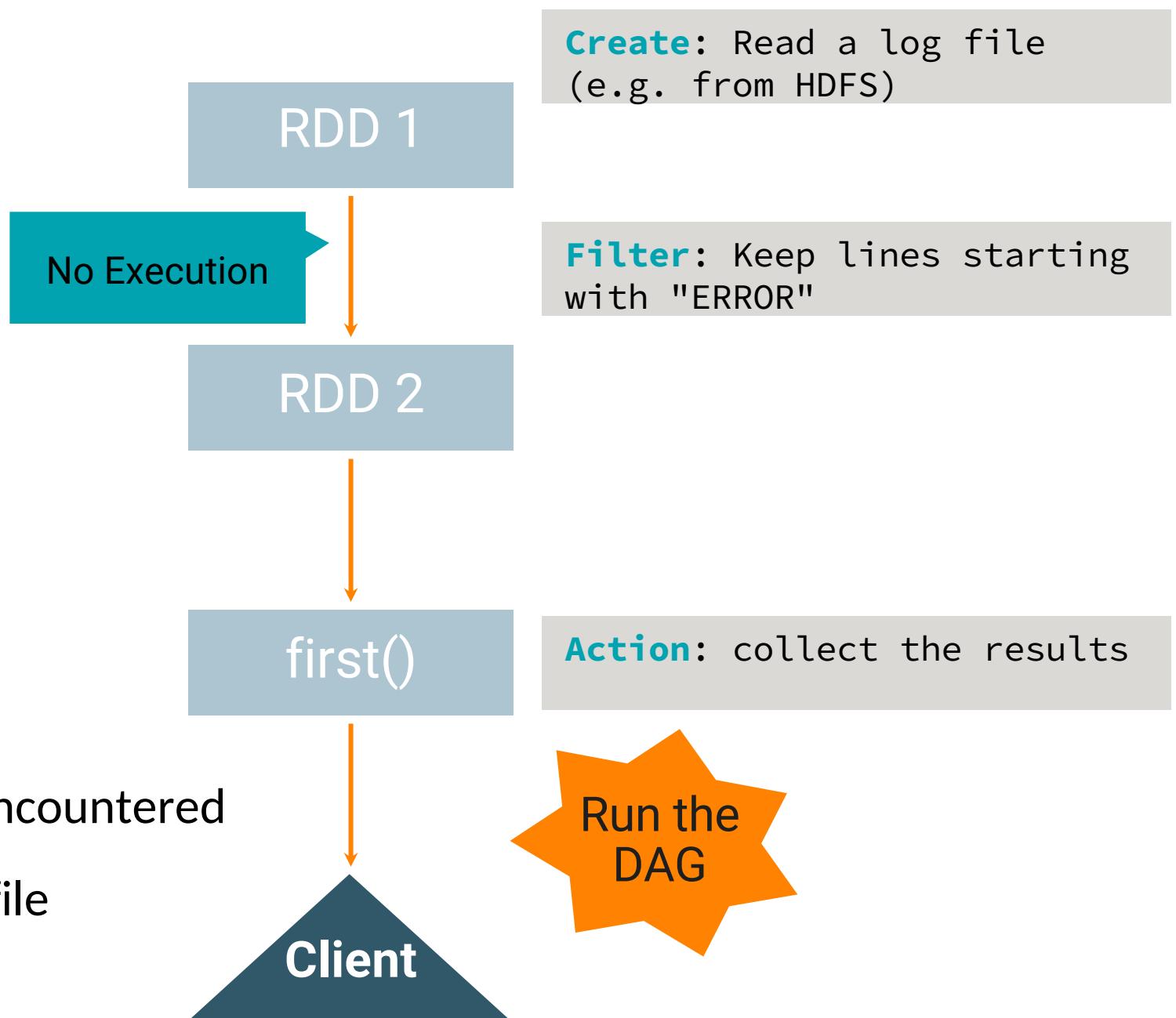
# All Transformations are Lazy

---

- **Spark doesn't immediately compute results**
  - Transformations stored as a graph (DAG) from a base RDD
  - Consider an RDD to be a set of operations
    - The ops required to produce data from a base
  - It's not really a container for specific data
- **The DAG is executed when an action occurs**
  - When it needs to provide data
- **Allows Spark to:**
  - Optimize required calculations (we'll view this soon)
  - Efficiently recover RDDs on node failure (more on this later)

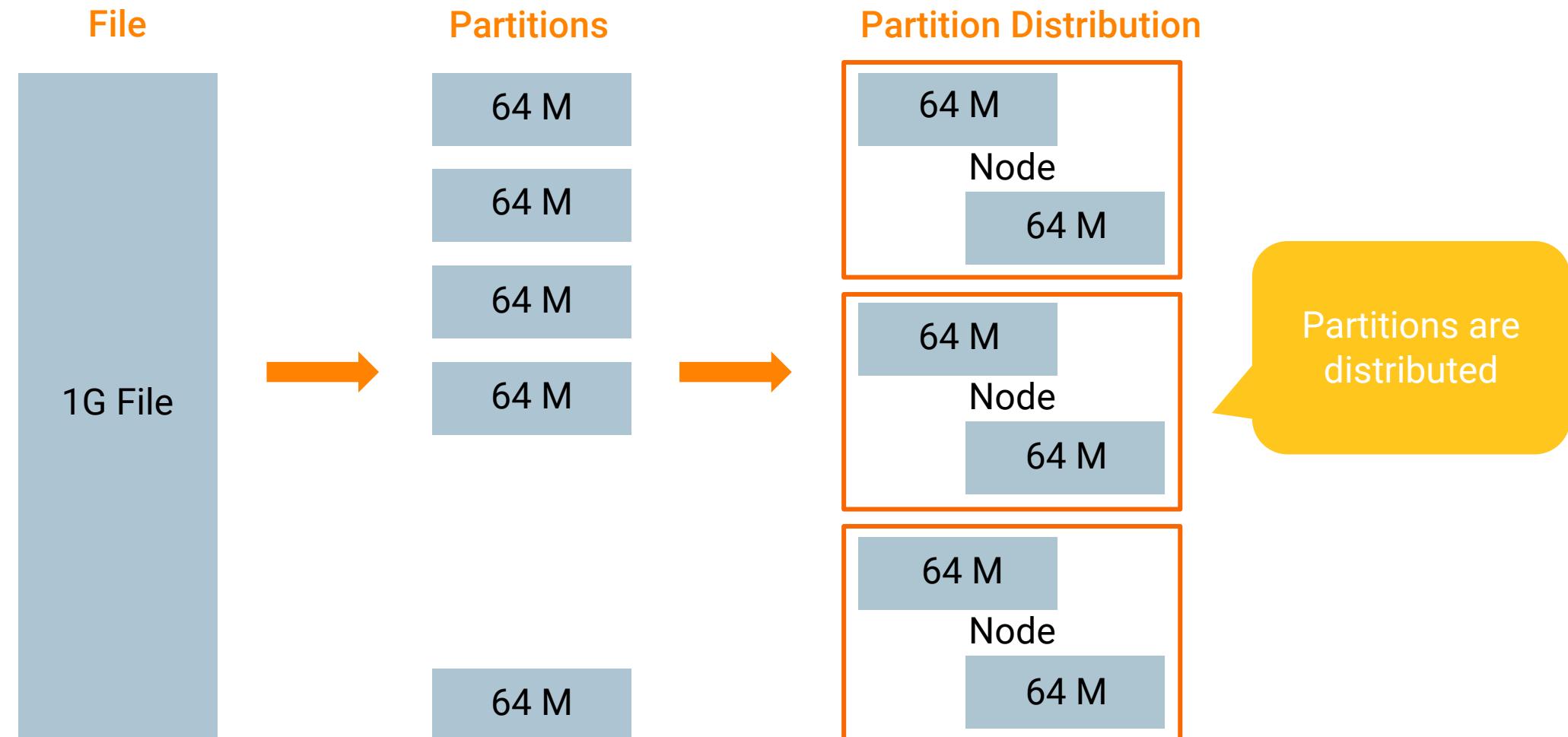
# Lazy Evaluation

- This example reads a log file
  - It filters out all but error lines
- At this point, no work done
- Client requests the first line
  - Triggers evaluation of the DAG
  - Here, the work is done
  - Result is sent to client
- Many possible optimizations
  - Stop filtering after the first ERROR line encountered
  - Doesn't even need to read all of the log file



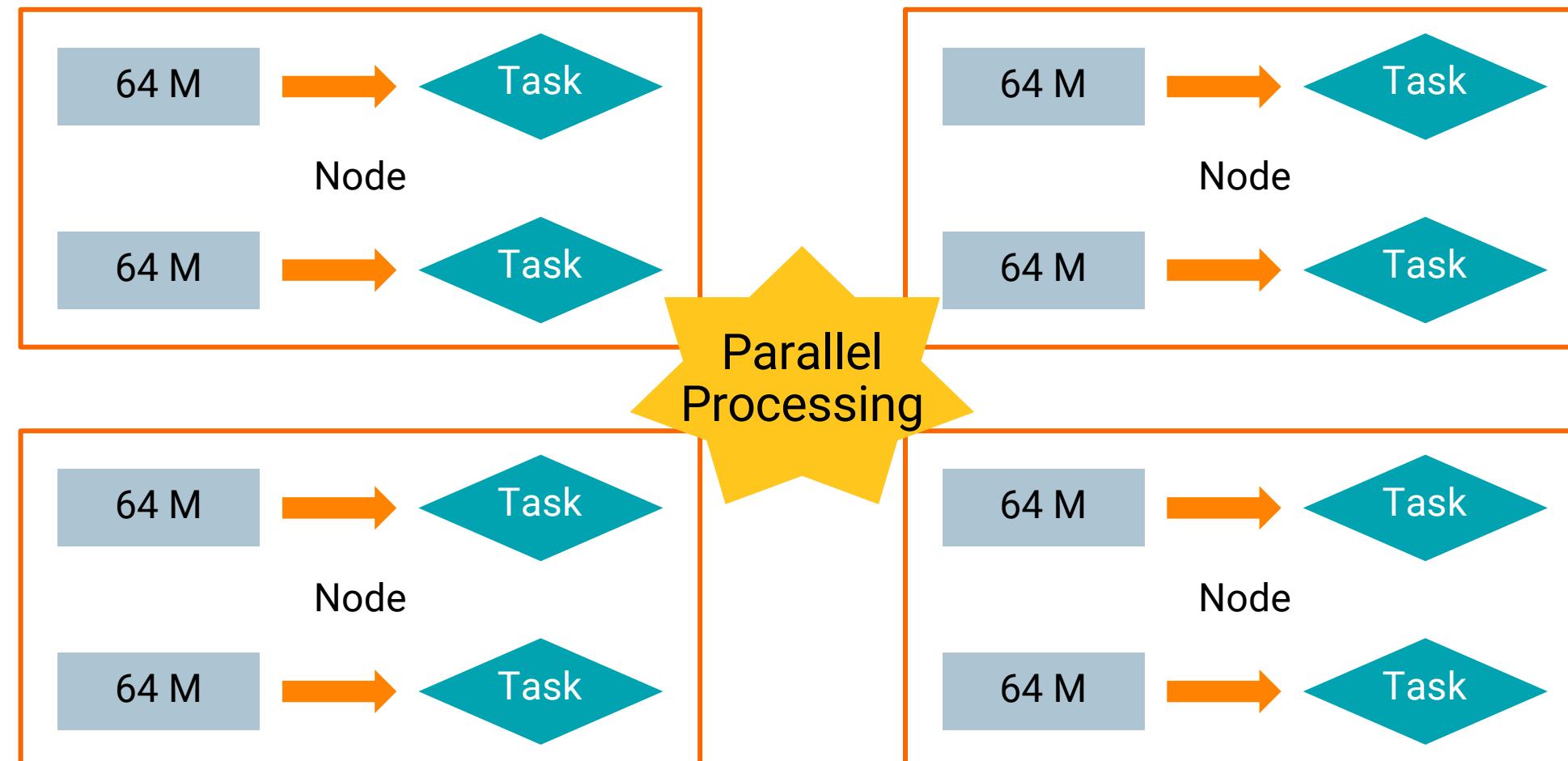
# RDD Partitioning

- Data in an RDD is partitioned around the cluster
  - e.g. With HDFS, Spark creates RDD partitions from HDFS blocks



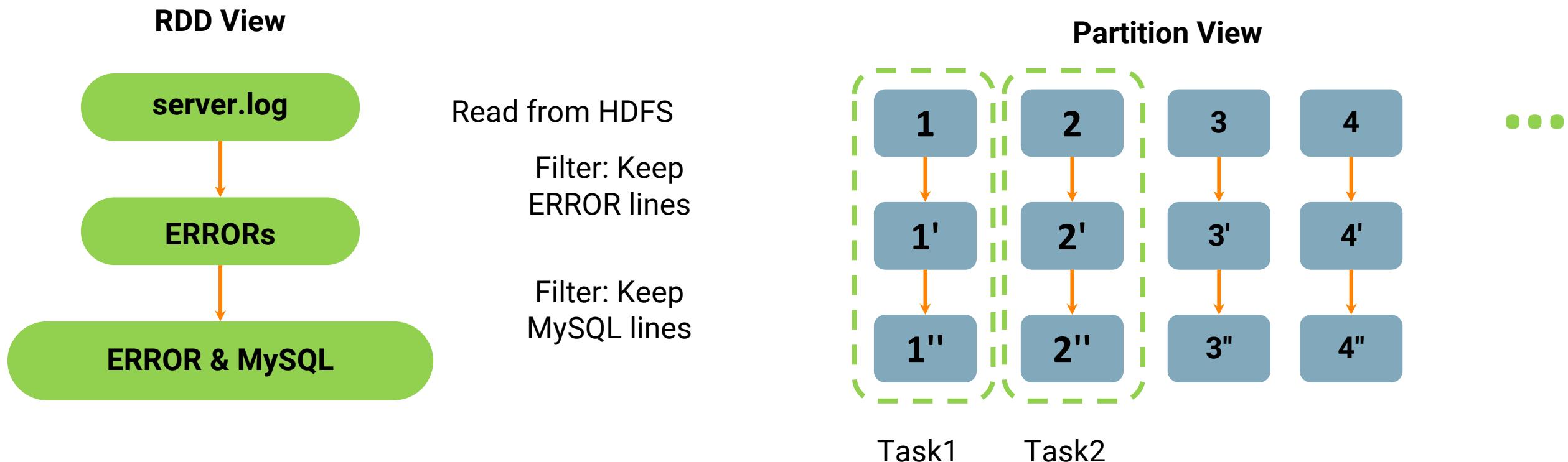
# RDD Partitions and Parallel Processing

- Nodes execute tasks in parallel on the partitions
  - Spark will co-locate tasks with their data (HDFS block if HDFS)



# Transformations Generate New Partitions

- A transformation on a partition creates a partition of the new RDD
- Succeeding transformations may be pipelined
- Often, it can all be done with in-memory data (fast)
- Some transformations require data shuffling (covered later)



## Example: RDD Partitions

---

- Below, we illustrate partitioning of our server log RDD
  - Including sample log messages
  - We label the messages for tracking them in the next examples
    - e.g. msg1, msg2, etc.

Partition 1	Partition 2	Partition 3
INFO, (msg1 ...MySQL...)	INFO (msg6 ...)	ERROR, (msg11 ...MySQL ...)
ERROR, (msg2 ...)	ERROR, (msg7 ...)	WARN, (msg12, ...)
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)	INFO, (msg13 ...)
INFO, (msg4 ...)	WARN, (msg9 ...)	WARN, (msg14 ...)
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)	INFO, (msg15 ...)

# Example: RDD Transformation

- We transform (filter) the RDD — each partition works on its own data

Partition 1	Partition 2	Partition 3	RDD
INFO, (msg1 ...MySQL...)	INFO (msg6 ...)	ERROR, (msg11 ...MySQL ...)	
ERROR, (msg2 ...)	ERROR, (msg7 ...)	WARN, (msg12, ...)	
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)	INFO, (msg13 ...)	
INFO, (msg4 ...)	WARN, (msg9 ...)	WARN, (msg14 ...)	
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)	INFO, (msg15 ...)	



Filter: Keep ERROR lines

Partition 1'	Partition 2'	Partition 3'	RDD '
		ERROR, (msg11 ...MySQL ...)	
ERROR, (msg2 ...)	ERROR, (msg7 ...)		
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)		
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)		

# Example: RDD Transformation

- We transform (filter) the RDD again

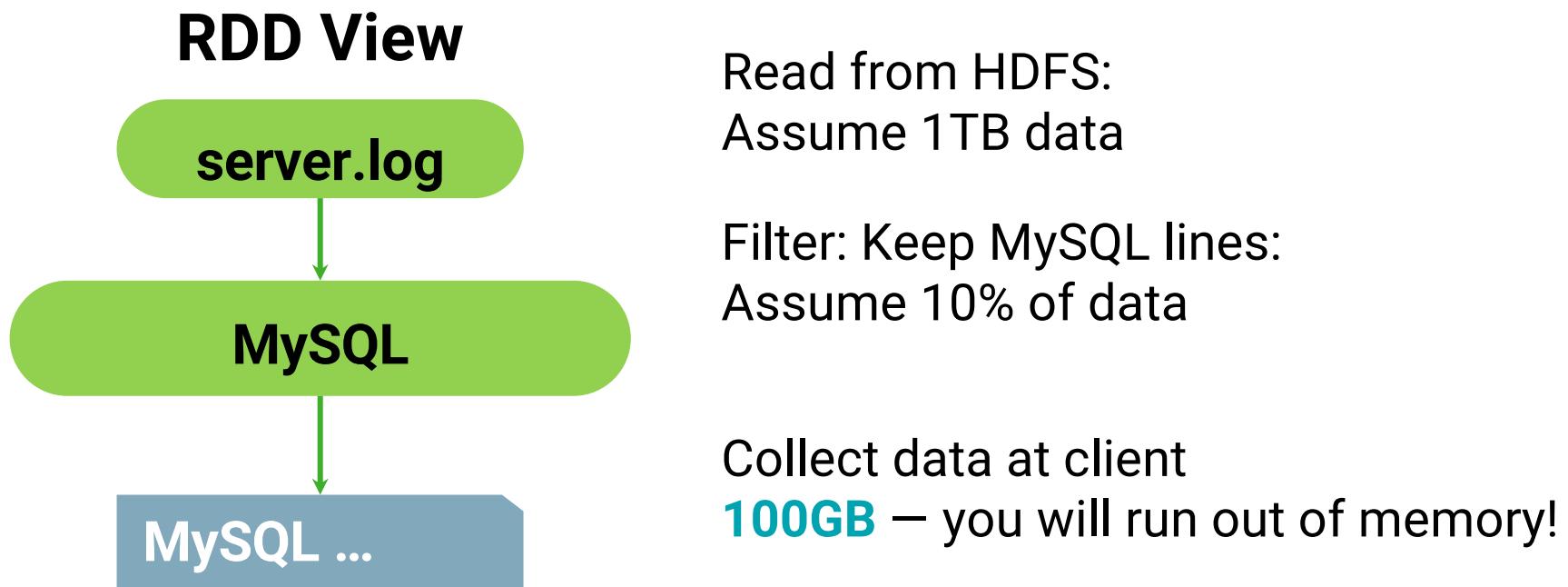
Partition 1'	Partition 2'	Partition 3'	RDD '
		ERROR, (msg11 ...MySQL ...)	
ERROR, (msg2 ...)	ERROR, (msg7 ...)		
ERROR, (msg3 ...MySQL ...)	ERROR, (msg8 ...)		
ERROR, (msg5 ...MySQL...)	ERROR, (msg10 ...)		

↓

Partition 1''	Partition 2''	Partition 3''	RDD ''
		ERROR, (msg11 ...MySQL ...)	
ERROR, (msg3 ...MySQL ...)			
ERROR, (msg5 ...MySQL...)			

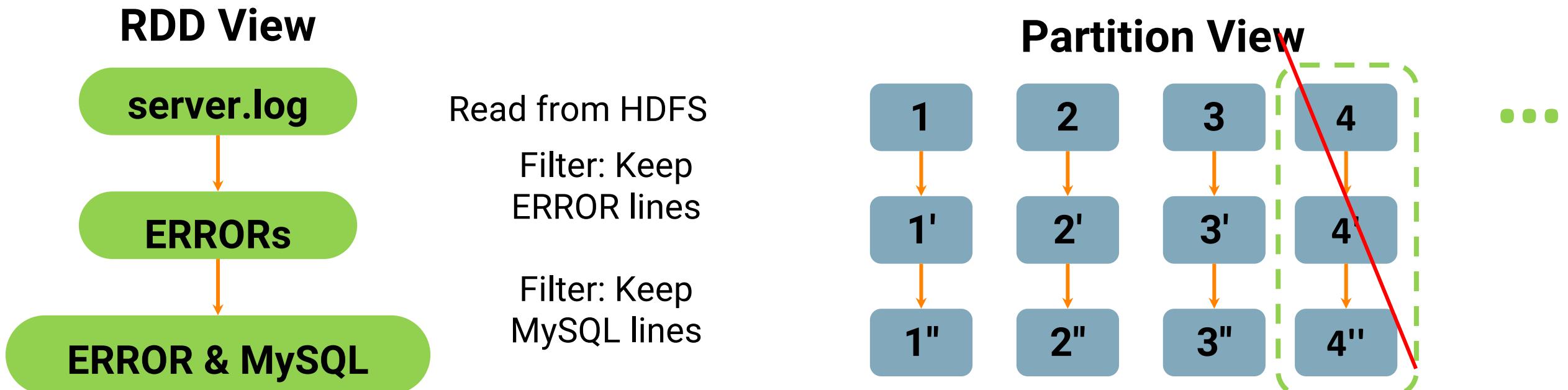
# Careful with Data to Client

- Don't transfer large data sets to your client!
  - You can easily run out of memory
  - OK to save results to a distributed file system like HDFS



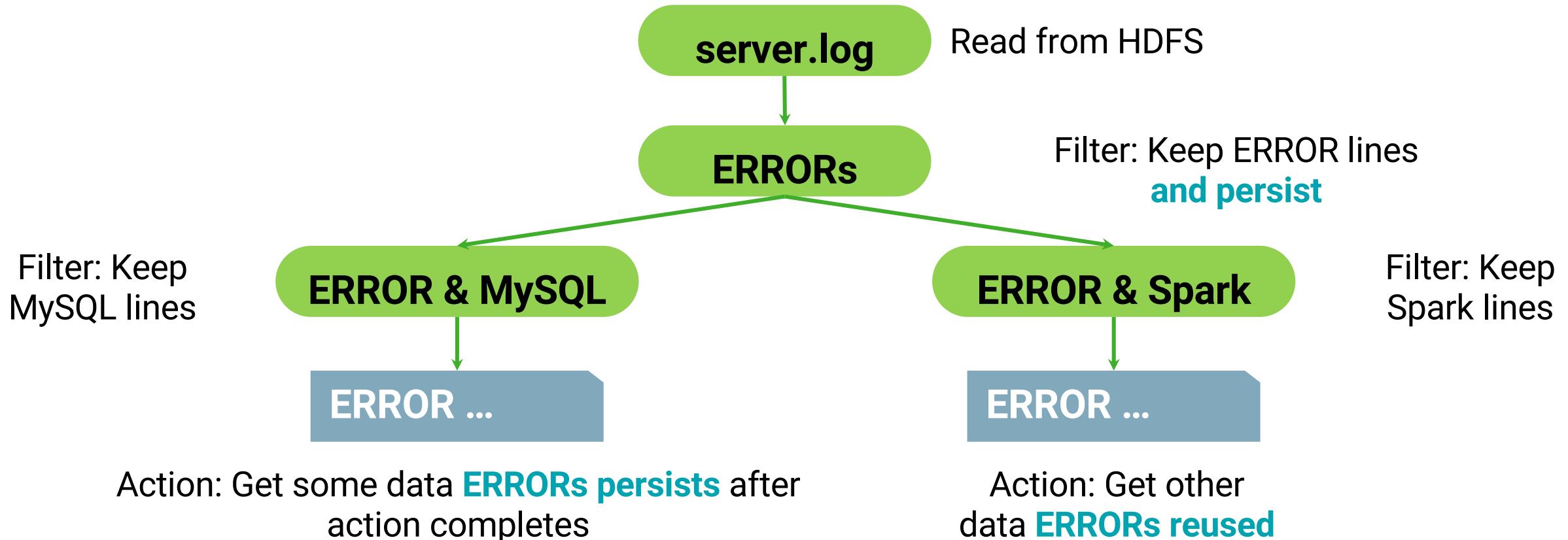
# Fault Tolerance

- Spark tracks transformations that create an RDD
  - Lineage: The series of transformations producing an RDD
- A lost partition can be rebuilt from its lineage
  - e.g. If partition 4 is lost, Spark can read the HDFS block again, apply the transformations, and recover the partition
  - Efficient, and adds little overhead to normal operation



# RDDs Are Transient By Default

- Once an action completes, its RDDs disappear by design
  - If you need one again, it's recomputed
- You can tell Spark to persist an RDD to keep it in memory
  - Useful for reusing an RDD that's expensive to create



# 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

# Creating RDDs from Text Files

- **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")`

```
// Create from local file
> val oneFile = spark.sparkContext.textFile("README.md")
```

**Language:** Scala

```
// Create from multiple files
> val multiFile = spark.sparkContext.textFile("data/mllib/*.txt")
```

```
// Create from a file in HDFS
> val hdfsFile = spark.sparkContext.textFile("hdfs://namehost:9000/logs/log201703.txt")
```

## Example: Using `textFile`

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

```
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 = spark.sparkContext.textFile("purplecow.txt")
```

Language: Python

myRDD

```
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.
```

## Example: Using wholeTextFiles

- wholeTextFiles maps 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"  
}
```

```
> 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", ...)
```

# Creating RDDs from Collections

---

- You can create RDDs from collections instead of files
  - `spark.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
  - The specified save directory cannot already exist

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

Language: Python

# 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

# RDD Actions Summary

**RDD r = {1,2,3,3}**

Action	Description	Example	Result
count()	Counts all records in an rdd	r.count()	4
first()	Extract the first record	r.first()	1
take(n)	Take first N lines	r.take(3)	[1,2,3]
collect()	Gathers all records for RDD. <b>All data has to fit in memory of ONE machine =&gt; don't use for big datasets</b>	r.collect()	[1,2,3,3]
saveAsTextFile()	Save to storage		
	... Many more – see docs ...		

# RDD Transformation Operations

---

- **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 Transformations Summary (1)

**RDD r = {1,2,3,3}**

Transformation	Description	Example (Scala)	Result
map(func)	apply func to each element in RDD	r.map(x => x*2)	{2,4,6,6}
filter(func)	Filters through each element when func is true (aka grep)	r.filter( x=> x % 2 == 1)	{1,3,3}
distinct	Removes dupes	r.distinct()	{1,2,3}
flatMap	Like map, but can output more than one result per element		
mapPartitions	Like map, but runs on the whole partition not on each element		

## RDD Transformations Summary (2)

**RDD r1 = {1,2,3,3}    RDD r2 = {2,4}**

Transformation	Description	Example	Result
union(RDD)	Merges two RDDs (duplicates are kept)	r1.union(r2)	{1,2,3,3,2,4}
intersection (RDD)	Returns common elements in two RDDs	r1.intersection(r2)	{2}
subtract(RDD)	Takes away elements from one	r1.subtract(r2)	{1,3,3}
sample	Take a small sample from RDD		

# Knowledge Check

---

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

# 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

---

## Working with RDDs

- Resilient Distributed Datasets (RDDs)
- **Exercise: Working with RDDs**

---

# Exercise: Working with RDDs





# Working with DataFrames

---

## Chapter 7

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- **Working with DataFrames**
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Lesson Objectives

---

**By the end of this chapter, you will be able to:**

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

# Chapter Topics

---

## Working with DataFrames

- **Introduction to DataFrames**
- **Exercises**
  - Introducing DataFrames
  - Reading and Writing DataFrames
  - Working with Columns
  - Working with Complex Types
  - Combining and Splitting DataFrames
  - Summarizing and Grouping DataFrames
  - Working with UDFs
  - Working with Windows

# RDDs Limitations

---

- **Low level API**

- You specify details (the how) not intent (the what)
  - API has little intelligence for dealing with common data formats
    - e.g. JSON

- **Opaque to Spark engine (uses arbitrary lambdas)**

- Queries can't easily be optimized by Spark
  - Not hard to write inefficient transformations
  - Often not obvious
  - And Spark can't make them better

- **No support for SQL-like querying**

- Limits applicability
  - SQL is well known

# 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.

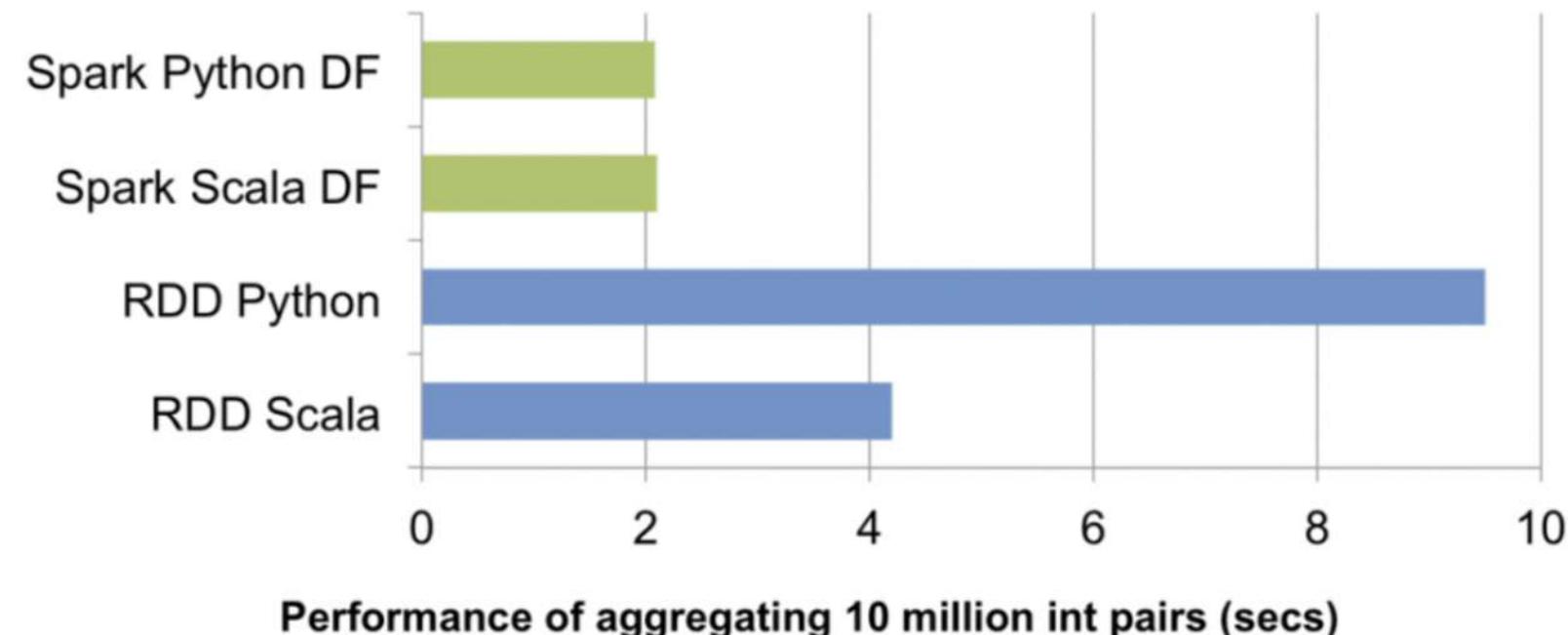
# DataFrames and Datasets: Some History

---

- **Introduced as SchemaRDD (1.0), renamed to DataFrame (1.3)**
- **Dataset type introduced in Spark 1.6**
  - Separate type from DataFrame for backwards compatibility
- **Dataset and DataFrame unified in 2.0**
  - DataFrame is now a typedef for Dataset[Row] (Scala)
  - The API is defined in Dataset, and divided into sections
    - Untyped operations derive historically from DataFrame
    - Typed operations derive historically from Dataset
- **A bit confusing — especially if you worked with early releases**
- **We'll use DataFrame to refer to the untyped API**

## Why DataFrames (and Datasets)? - More efficient code

- DataFrames based code is translated to RDD code through a process that involves a sequence of optimisers (Catalyst and Tungsten) that produce the best RDD code possible
- Catalyst understands the structure of data & semantics of operations and performs optimizations accordingly
- The Tungsten storage format is 4 times more efficient than its Java counterpart



# Why DataFrames (and Datasets)? - Cleaner Code

- The syntax is easier on the eyes: these three snippets of code perform the same processing

```
> val folksRDD=sc.textFile("people.txt") // Get the data
> folksRDD.map(_.split(" ")) // Split it into fields
  .map(x => (x(1), Array(x(2).toDouble, 1))) // Pair up the data
  .reduceByKey( (x,y) => Array(x(0)+y(0), x(1)+y(1)) ) // Count
  .map(x => Array(x._1, x._2(0)/x._2(1))) // Figure average
  .collect // Get results
```

Language: Scala

```
> val folksDF=spark.read.json("people.json") // Get the data
> folksDF.groupBy($"gender") // Group by gender
  .agg(avg($"age")) // Get average age of groups
  .show // Display data
```

Language: Scala

```
> val folksDF=spark.read.json("people.json") // Get the data
> folksDF.createOrReplaceTempView("people") // Setup for using SQL
> spark.sql("SELECT gender, avg(age) FROM people GROUP BY gender").show
```

Language: Scala

# 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

# Spark Session

---

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

## 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

# 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)

---

- Some common DataFrame 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
  - collect: returns all the rows in the DataFrame as an array
  - write: save the data to a file or other data source

## 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

```
> nameAgeDF = 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

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

Language: Scala

## Knowledge Check

---

- 1. What is Spark SQL, and why do we use it?**
  
- 2. How do you read data using the SparkSession? What kinds of data can be read?**
  
- 3. What is a DataFrame? A Dataset?**
  
- 4. True or False? - DataFrames are convenient but will never outperform native RDD based code.**

## Essential Points

---

- **DataFrames create another abstraction between the developers and data**
- **DataFrames have built in optimizers and outperforms core spark in speed**
- **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

---

## Working with DataFrames

- Introduction to DataFrames
- **Exercises**
  - Introducing DataFrames
  - Reading and Writing DataFrames
  - Working with Columns
  - Working with Complex Types
  - Combining and Splitting DataFrames
  - Summarizing and Grouping DataFrames
  - Working with UDFs
  - Working with Windows



## Introduction to Apache Hive

---

Cloudera 8



# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- **Introduction to Apache Hive**
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Lesson Objectives

---

**After completing this lesson, students should be able to:**

- **Understand where Hive comes from and how it has evolved to its current state**
- **Know the various Hive components**
- **Understand the differences between external and managed tables**
- **Understand how Hive manages ACID transactions**
- **Use the Beeline shell**

# Chapter Topics

---

## Introduction to Apache Hive

- About Hive

# What is Hive?

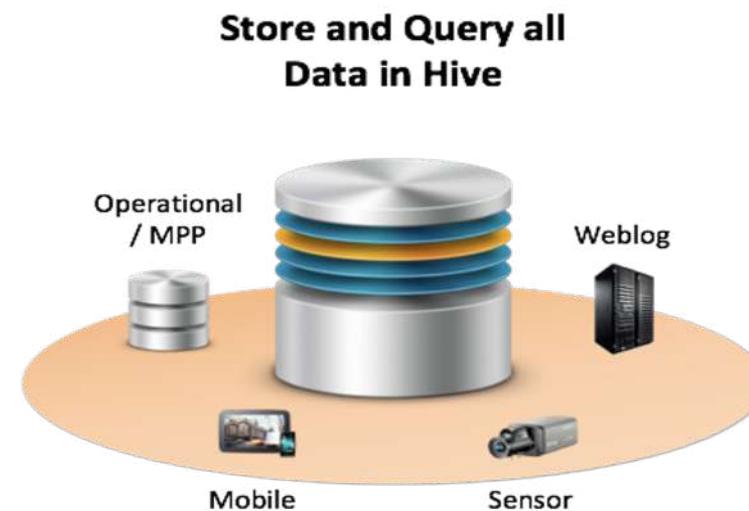
---

- **SQL Semantic Layer on Hadoop**
- **“De facto SQL Interface” for Hadoop**
- **Originally developed by Facebook**
- **Original Appeal**
  - Schema on Read
  - SQL to Map Reduce (Reduce complexity of Map Reduce)
  - Familiar Programming Context with SQL

# About Hive

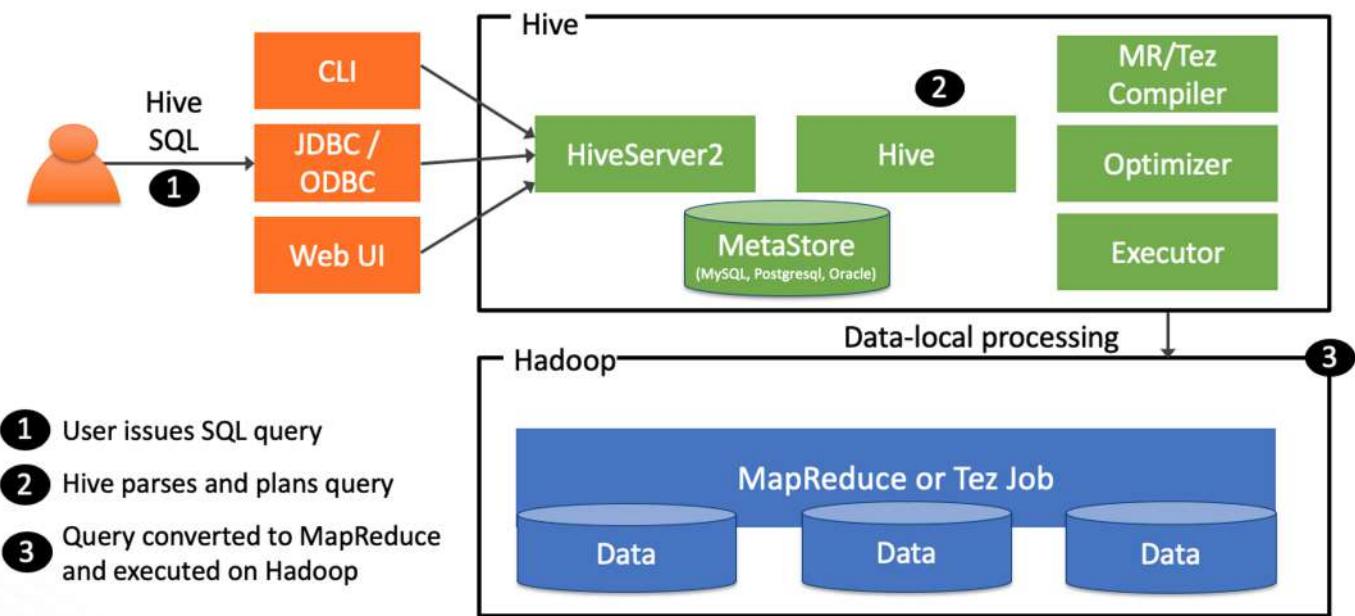
---

- It is a data warehouse system for Hadoop
- It maintains metadata information about your big data stored on HDFS
- Big data can be queried as tables
- It performs SQL-like operations on the data using a scripting language called HiveQL



# Hive Architecture

- **Hive CLI - Client Access, Direct HDFS integration**
  - Being retired in Hive 3.0
- **Hive Server 2**
  - ODBC/JDBC gateway to SQL on Hadoop
  - File System Abstraction
- **Metastore**
  - Hive's Catalog (System Tables)
- **Metastore DB**
  - RDBMS store for “system tables”
- **LLAP (Low Latency Analytical Processing)**
  - Low Latency, Always On, Shared Cache Service



## HiveServer 2

---

- **HiveServer2 (HS2) is a service that enables clients to execute queries against Hive**
- **HiveServer2 supports multi-client concurrency and authentication. It is designed to provide better support for open API clients like JDBC and ODBC**
- **HiveServer2 is a single process running as a composite service, which includes the Thrift-based Hive service (TCP or HTTP) and a Jetty web server for web UI**
- **Thrift is an RPC framework for building cross-platform services. Its stack consists of 4 layers: Server, Transport, Protocol, and Processor**

# Hive Metastore

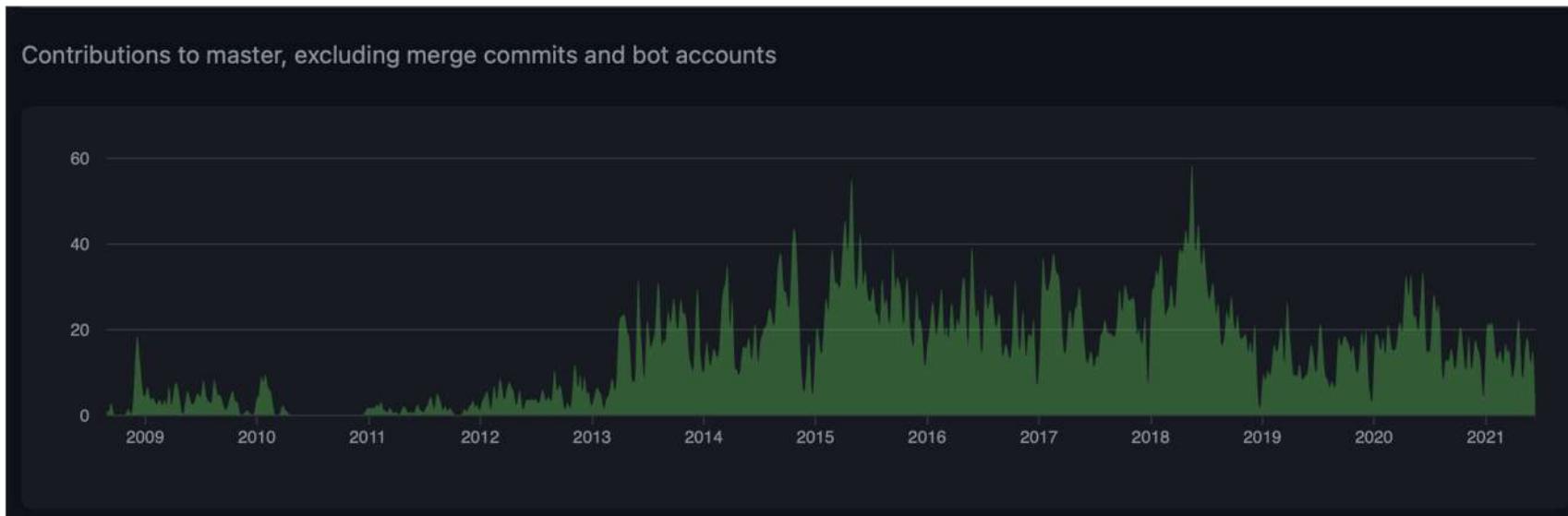
---

- **Metastore is the central repository of Apache Hive metadata. It stores metadata for Hive tables (like their schema and location) and partitions in a relational database**
- **The Metastore provides client access to this information by using metastore service API**
- **The Hive Metastore has two fundamental purposes:**
  - A service that provides metastore access to other Apache Hive services.
  - Disk storage for the Hive metadata which is separate from HDFS storage.
- **Three modes for Hive Metastore deployment**
  - Embedded Metastore
  - Local Metastore
  - Remote Metastore

# Hive History of Improvements

---

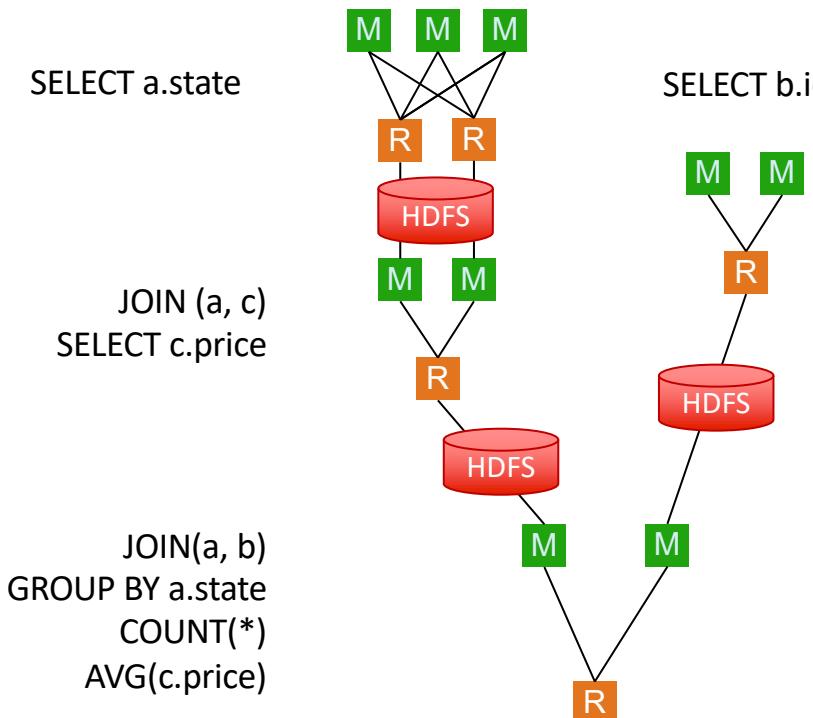
- Throughout its long history, Hive has undergone many improvements
  - The TEZ execution engine
  - Vectorization
  - The ORC format
  - LLAP
  - Transactional tables
  - Materialized Views



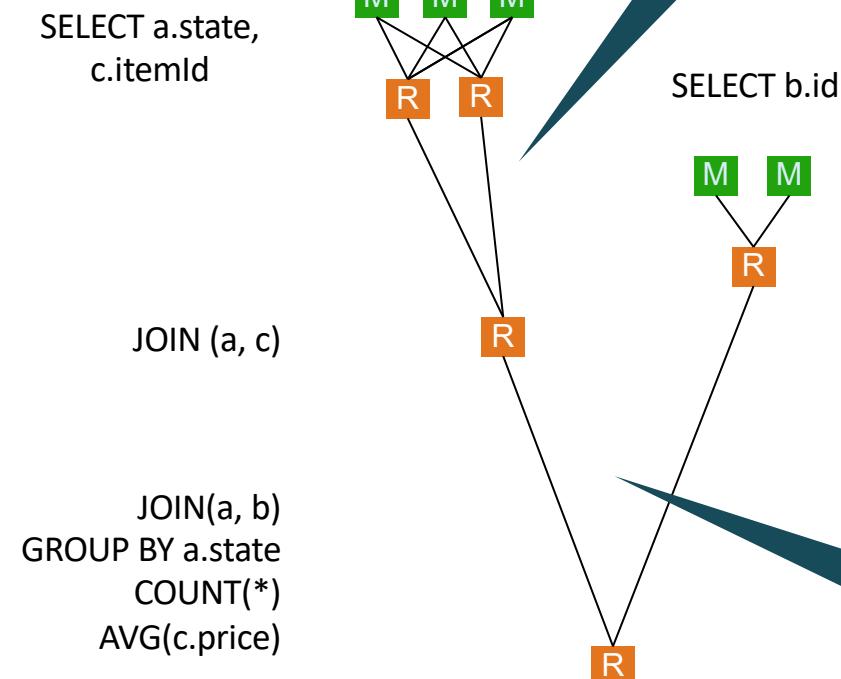
```
SELECT a.state, COUNT(*), AVG(c.price) FROM a
JOIN b ON (a.id = b.id)
JOIN c ON (a.itemId = c.itemId)
GROUP BY a.state
```

Tez avoids unneeded writes to HDFS

### Hive - MapReduce



### Hive - Tez



## Hive Optimizations - Vectorization

---

- **Vectorization improves query performance for operations like scans, aggregations, filters and joins by performing operations in batches of 1024 rows at a time.**
- **Below are the hive parameters that will enable Vectorization.**
  - set `hive.vectorized.execution.enabled=true;`
  - set `hive.vectorized.execution.reduce.enabled=true;`

# ORC-Backed Hive Tables

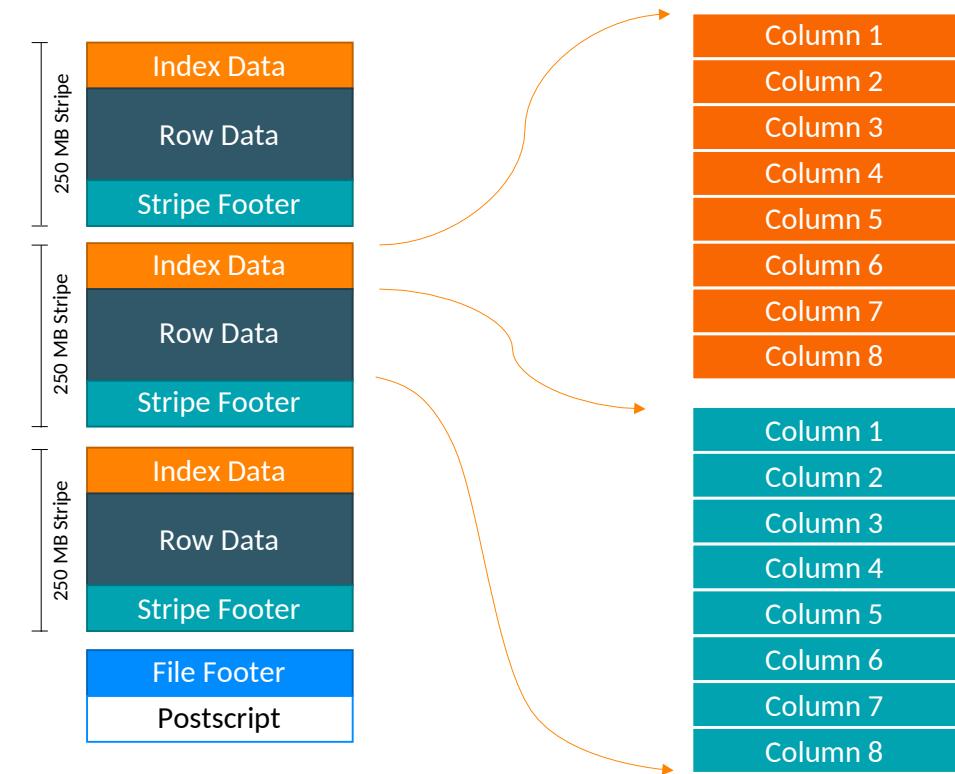
---

- The Optimized Row Columnar (ORC) file format provides highly efficient way to store Hive data.
- Hive type support including datetime,decimal, and complex types(struct, list, map and union).
- Light-weight indexes stored within the ORC file format.
- Block-mode compression based on data type.



# ORC File- Columnar Format

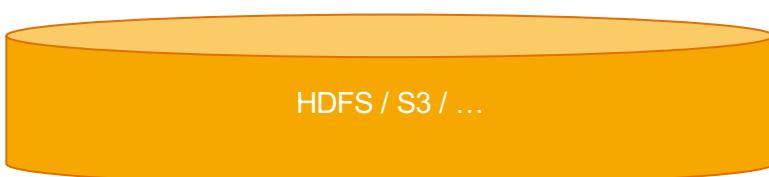
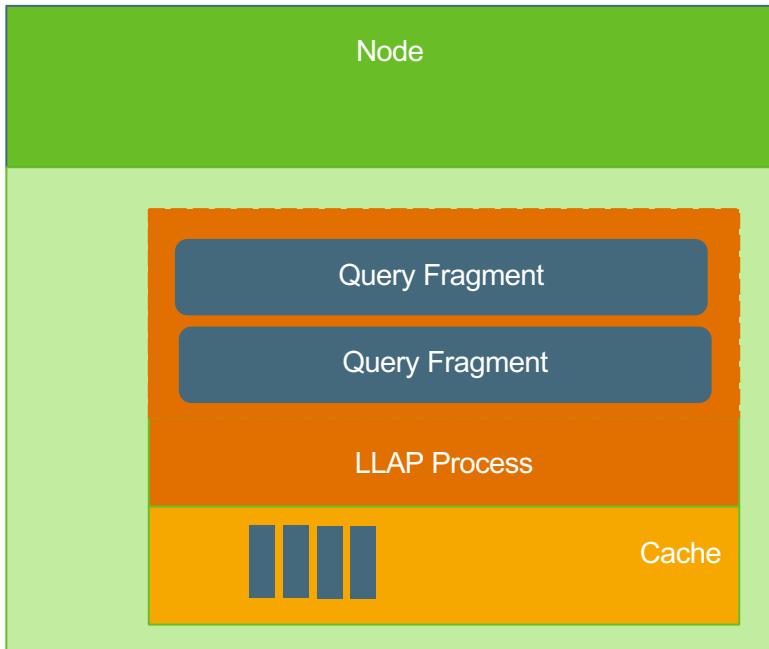
- **High Performance: Columnar storage File.**
- **Efficient Reads: Break into large “stripes” of data efficient read.**
- **Fast Filtering: Built in index, min/max, metadata for fast filtering blocks.**
- **Efficient Compression: Decompose complex row types into primitives: massive compression and efficient comparisons for filtering.**
- **Precomputation: Built in aggregates per block ( min, max, count, sum, etc.)**
- **Proven at 300 PB scale: Facebook uses ORC for their 300 PB Hive Warehouse.**



# Low Latency Analytical Processing (LLAP)



- LLAP is a set of persistent daemons that execute fragments of Hive queries. (Not an execution engine like Tez)
- LLAP enables as fast as sub-second SQL analytics on Hadoop by intelligently caching data in memory with persistent servers that instantly process SQL queries
- Query execution on LLAP is very similar to Hive without LLAP, except that worker tasks run inside LLAP daemons, and not in containers.
- Intelligent memory caching for quick startup and data sharing.
- Persistent server used to execute queries. ( LLAP daemons)



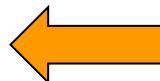
# Hive Beeline

---

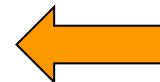
- **Hive CLI - Retired in Hive 3.0**  
e.g. --- \$HIVE\_HOME/bin/hive
- **Use Beeline to connect to client**  
e.g.--- \$HIVE\_HOME/bin/beeline -u jdbc:hive2://
- **Beeline connects to [HiveServer2](#) instance using JDBC**
- **HiveQL can be executed the same way as Hive CLI**
- **Beeline supports embedded mode**

# Beeline Example

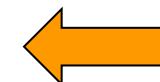
```
% bin/beeline
Hive version 0.11.0-SNAPSHOT by Apache
beeline> !connect jdbc:hive2://localhost:10000 scott tiger
!connect jdbc:hive2://localhost:10000 scott tiger
Connecting to jdbc:hive2://localhost:10000
Connected to: Hive (version 0.10.0)
Driver: Hive (version 0.10.0-SNAPSHOT)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://localhost:10000> show tables;
show tables;
+-----+
| tab_name |
+-----+
| primitives |
| src |
| src1 |
| src_json |
| src_sequencefile |
| src_thrift |
| srcbucket |
| srcbucket2 |
| srcpart |
+-----+
9 rows selected (1.079 seconds)
```



JDBC connection to Hive



HiveQL command



Results

## Hive Internal Managed Tables

---

- Hive internally managed tables are the default tables in Hive
- The default table will be created in the /warehouse/tablespace/managed/hive directory of HDFS.
- Deleting Managed tables removes both the table data and the metadata for the table from HDFS.

```
CREATE TABLE customer (
    customerID INT,
    firstName STRING,
    lastName STRING,
    birthday TIMESTAMP
) ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ',';
```

## Hive External Tables

---

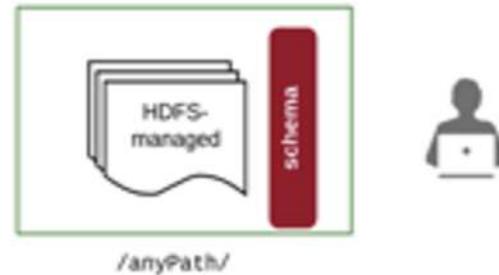
- The data in Hive external tables can reside in any HDFS directory.
- When you drop an External Hive table only the metadata is deleted not the data.

```
CREATE EXTERNAL TABLE SALARIES (
    gender string,
    age int,
    salary double,
    zip int
) ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LOCATION '/user/train/sALARIES/';
```

# Hive External vs Internal Tables

## External Table

- Data life cycle not managed by Hive.

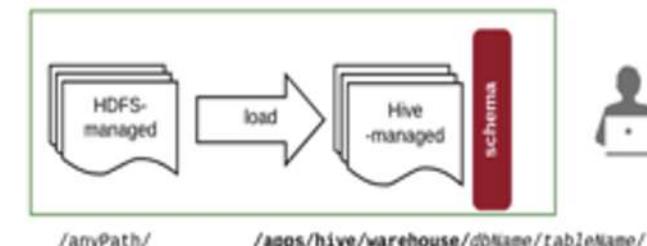


```
CREATE EXTERNAL TABLE myTable  
(name STRING, age INT)  
LOCATION '/anyPath';
```

- No load data step required.
  - Data must simply reside in the path
- DROP Table removes only metadata.

## Internal Managed Table

- Data life cycle and access is isolated to Hive.



```
CREATE TABLE myTable  
(name STRING, age INT);  
LOAD DATA INPATH 'anypath'  
[OVERWRITE] INTO TABLE myTable
```

- DROP Table removes metadata and data.

# Hive 3.0 ACID Transactions

---

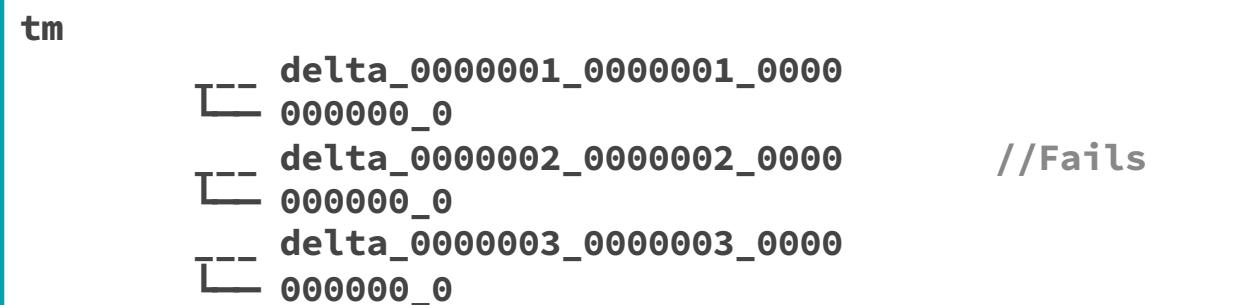
- Hive supports single-table / single-statement transactions and are the default table type in HDP 3.0
- Transactional tables perform as well as other tables.
- Hive atomic operations extended operations to support:
  - Writing to multiple partitions
  - Using multiple insert clauses in a single INSERT statement.

# Hive ACID Insert-only Tables

- Insert-only transactions, the transaction manager gets a transaction ID.
- Transaction Manager allocates a write ID and determines a path to write the data.
- Hive creates a delta directory to which the transaction manager writes data files.
- In the Read process, the transaction manager maintains the state of every transaction.
- On read, the snapshot information is represented by a high watermark.  
**(Watermark identifies the highest transaction ID.)**

```
CREATE TABLE tm (a int, b int) TBLPROPERTIES  
('transactional'='true',  
 'transactional_properties'='insert_only');
```

```
INSERT INTO tm VALUES(1,1);  
INSERT INTO tm VALUES(2,2); // Assume this fails  
INSERT INTO tm VALUES(3,3);
```



# Hive Atomicity and Isolation in CRUD tables

- **No in-place updates. Hive uses a row ID (struct) to achieve atomicity and isolation that contains:**
  - Write ID maps transactions that creates the row.
  - Bucket ID, a bit-backed integer with several bits of information, of the physical writer that created the row.
  - Row ID, numbers rows as they are written to a data file.
- **No in-place deletes. Hive appends changes to a table when a deletion occurs. Deleted data becomes unavailable during compaction process.**

```
CREATE TABLE acidtbl (a INT, b STRING)
    STORED AS ORC TBLPROPERTIES
    ('transactional'='true');
```

Metadata Columns	original_write_id bucket_id row_id current_write_id	ROW_ID
User Columns	col_1: a : INT col_2: b : STRING	

# Hive Transactional Operations

## Insert Operation

```
INSERT INTO acidtbl (a,b) VALUES  
(100, "oranges"),  
(200, "apples"),  
(300, "bananas");
```

ROW_ID	a	b
{1,0,0}	100	oranges
{1,0,1}	200	apples
{1,0,2}	300	bananas

## Delete Operation

```
DELETE FROM acidTbl where a = 200;
```

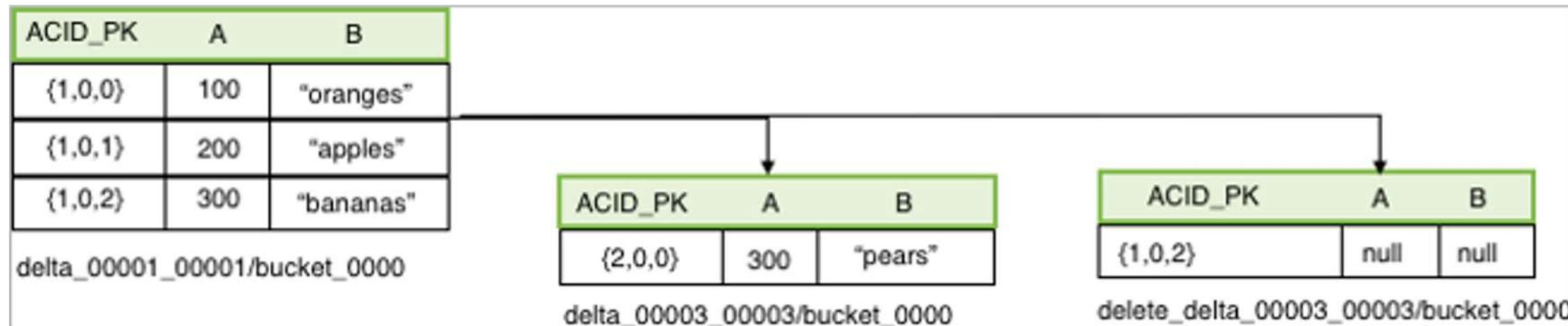
This operation generates a directory and file, delete\_delta\_00002\_00002/bucket\_0000.

ROW_ID	a	b
{1,0,1}	null	nul

# Hive Transactional Operations (Continued)

## Update Operation

```
UPDATE acidTbl SET b = "pears" where a = 300;
```



## Knowledge Check

---

- 1. What element within the Hive architecture do clients make xDBC connections to?**
  
- 2. List the execution engines that can be used when running your queries.**
  
- 3. What is the primary difference between an external table and a managed one?**
  
- 4. List at least two of the improvements that were added to the initial design**

## Essential Points

---

- Apache Hive gives a semantic SQL Layer on top of Hadoop
- Hive has evolved from its simple beginning to become a complex and feature rich database system
- CDP supports Hive 3.1
- Users can connect to Hive using Beeline to connect to client
- Hue and Apache Zeppelin provide User Interfaces to Hive



# Hive and Spark Integration

---

## Chapter 9

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- **Hive and Spark Integration**
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Chapter Topics

---

## Hive and Spark Integration

- **Hive and Spark Integration**
- **Exercise: Spark Integration with Hive**

# Hive and Spark integration (Spark Version < 2.x)

- Spark SQL also supports reading and writing data stored in Apache Hive.
- Hive support is enabled by adding the -Phive and -Phive-thriftserver flags to Spark's build.
- Configuration of Hive is done by placing your `hive-site.xml`, `core-site.xml` (for security configuration), `hdfs-site.xml` (for HDFS configuration) file in `conf/`.
- When working with Hive one must construct a `HiveContext`, which inherits from `SQLContext`, and adds support for finding tables in the MetaStore and writing queries using `HiveQL`.

```
// sc is an existing SparkContext.  
val sqlContext = new org.apache.spark.sql.hive.HiveContext(sc)  
  
sqlContext.sql("CREATE TABLE IF NOT EXISTS src (key INT, value STRING)")  
sqlContext.sql("LOAD DATA LOCAL INPATH 'examples/src/main/resources/kv1.txt' INTO TABLE src")  
  
// Queries are expressed in HiveQL  
sqlContext.sql("FROM src SELECT key, value").collect().foreach(println)
```

# Hive Metadata Configurations (Spark Version < 2.x)

Property Name	Default	Meaning
spark.sql.hive.metastore.version	1.2.1	Version of the Hive metastore. Available options are 0.12.0 through 1.2.1.
spark.sql.hive.metastore.jars	builtin	<p>Location of the jars that should be used to instantiate the HiveMetastoreClient. This property can be one of three options:</p> <ol style="list-style-type: none"><li>1. builtin</li><li>2. Use Hive 1.2.1, which is bundled with the Spark assembly jar when -Phive is enabled.</li><li>3. maven</li><li>4. Use Hive jars of specified version downloaded from Maven repositories. This configuration is not generally recommended for production deployments.</li><li>5. A classpath in the standard format for the JVM. This classpath must include all of Hive and its dependencies, including the correct version of Hadoop. These jars only need to be present on the driver, but if you are running in yarn cluster mode then you must ensure they are packaged with your application.</li></ol> <p>When this option is chosen, spark.sql.hive.metastore.version must be either 1.2.1 or not defined.</p>
spark.sql.hive.metastore.sharedPrefixes	com.mysql.jdbc, org.postgresql, com.microsoft.sqlserver, oracle.jdbc	A comma separated list of class prefixes that should be loaded using the classloader that is shared between Spark SQL and a specific version of Hive. An example of classes that should be shared is JDBC drivers that are needed to talk to the metastore. Other classes that need to be shared are those that interact with classes that are already shared. For example, custom appenders that are used by log4j.
spark.sql.hive.metastore.barrierPrefixes	(empty)	A comma separated list of class prefixes that should explicitly be reloaded for each version of Hive that Spark SQL is communicating with. For example, Hive UDFs that are declared in a prefix that typically would be shared (i.e. org.apache.spark.*).

# Hive and Spark Integration (Spark Version >2.x and < 3.x)

---

- **SparkSession has merged SQLContext and HiveContext in one object.**
- **The `hive.metastore.warehouse.dir` property in `hive-site.xml` is deprecated since Spark 2.0.0.**  
Instead, use `spark.sql.warehouse.dir` to specify the default location of database in warehouse.
- **On Hive table creation, the read/write data from/to file system needs to be defined.**  
( e.g. `CREATE TABLE src(id int) USING hive OPTIONS(fileFormat 'parquet')`)

```
val spark = SparkSession  
    .builder()  
    .appName("SparkSessionZipsExample")  
    .config("spark.sql.warehouse.dir", warehouseLocation)  
    .enableHiveSupport()  
    .getOrCreate()
```

# Hive 3.0 ORC Integration with Spark

- Since Spark 2.3, Spark supports a vectorized ORC reader with a new ORC file format for ORC files.
- Vectorized reader is used for native ORC tables.

Property Name	Default	Meaning
spark.sql.orc.impl	hive	The name of ORC implementation. It can be one of native and hive. native means the native ORC support that is built on Apache ORC 1.4.1. `hive` means the ORC library in Hive 1.2.1.
spark.sql.orc.enableVectorizedReader	true	Enables vectorized orc decoding in native implementation. If false, a new non-vectorized ORC reader is used in native implementation. For hive implementation, this is ignored.

## Example

```
spark.read.format("orc").load (path)  
df.write.format("orc").save(path)
```

# Hive Warehouse Connector Examples (2.x < Spark Version < 3.x)

## Create HiveWarehouse session

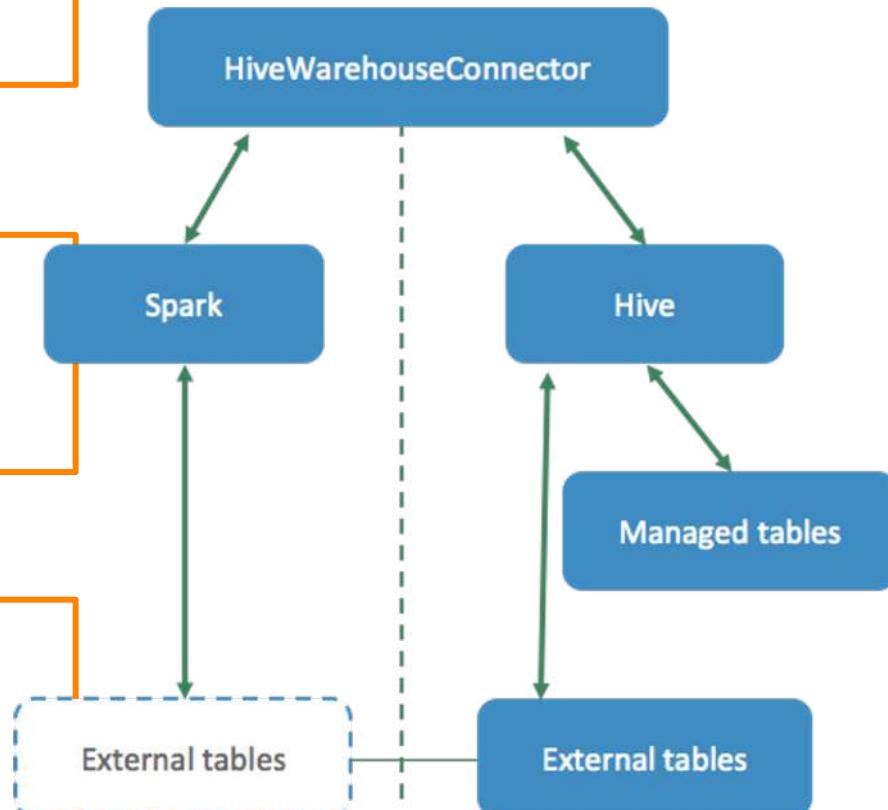
```
import com.hortonworks.hwc.HiveWarehouseSession  
import com.hortonworks.hwc.HiveWarehouseSession._  
val hive = HiveWarehouseSession.session(spark).build()
```

## Execute query

```
//Execute Hive Query  
hive.executeQuery("select * from web_sales")  
  
// Execute Hive update  
hive.executeUpdate("ALTER TABLE old_name RENAME TO new_name")
```

## Write Spark Dataframe and Stream to Hive

```
//Dataframe write to Hive  
  
df.write.format(HIVE_WAREHOUSE_CONNECTOR).option("table",<tableName>).save()  
  
//DataStream write to Hive  
  
stream.writeStream.format(STREAM_TO_STREAM).option("table", "web_sales").start()
```



## Essential Points

---

- Before Spark 2.0 integration with Hive was provided by a Hive context
- With Spark 2.0 the Hive context was built in the Spark session
- Since Spark 2.3, Spark supports the ORC format
- Access to transactional tables in Hive 3 via Spark 2.\* required a Hive Warehouse Connector

# Chapter Topics

---

## Hive and Spark Integration

- Hive and Spark Integration
- **Exercise: Spark Integration with Hive**



# Data Visualization with Zeppelin

---

Chapter 10

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- **Data Visualization with Zeppelin**
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Chapter Topics

---

## Data Visualization with Zeppelin

- **Introduction to Data Visualization with Zeppelin**
- Zeppelin Analytics
- Zeppelin Collaboration
- Exercise: AdventureWorks

# Visualizations Are Essential

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

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

age	balance	marital
58	2,143	married
44	29	single
33	2	married
47	1,506	married
33	1	single
35	231	married
28	447	single
42	2	divorced
59	121	married



# Tables Visualizations

**Visualize Transactions**

```
%sql  
select * from trans limit 200
```

FINISHED

accountNumber	accountType	amount	currency	ipAddress	isCardPresent	latitude	longitude	merchantId	merchantType	transactionId	transactionTimeStamp
19123	VISA	152	USD		true	39.9404246	-75.0254382	ChIJJoSGITK3LxokRK...	clothing_store	ChIJJoSGITK3LxokRK...	1362124800000
19123	VISA	145	USD		true	39.95096099999999	-75.170273	ChIJCx6IijDGxokR_...	restaurant	ChIJCx6IijDGxokR_...	1362128400000
19123	VISA	119	USD		true	40.067832	-74.854028	ChIJqyZSY5JOWYkRV...	grocery_or_supermarket	ChIJqyZSY5JOWYkRV...	1362132000000
19123	VISA	47	USD		true	39.91442370000001	-75.1731870999999	ChIJ8VtuebFxokRf...	entertainment	ChIJ8VtuebFxokRf...	1362135600000
19123	VISA	47	USD		true	39.91442370000001	-75.1731870999999	ChIJ8VtuebFxokRf...	entertainment	ChIJ8VtuebFxokRf...	1362139200000
19123	VISA	168	USD		true	39.9404246	-75.0254382	ChIJJoSGITK3LxokRK...	clothing_store	ChIJJoSGITK3LxokRK...	1362142800000
19123	VISA	104	USD		true	39.9223459	-75.1856254	ChIJ8X8rPXbGxokRj...	electronics_store	ChIJ8X8rPXbGxokRj...	1362146400000
19123	VISA	113	USD		true	40.067832	-74.854028	ChIJqyZSY5JOWYkRV...	grocery_or_supermarket	ChIJqyZSY5JOWYkRV...	1362150000000

**Account Distribution**

FINISHED

17439 18675 19123

**Merchant Distribution**

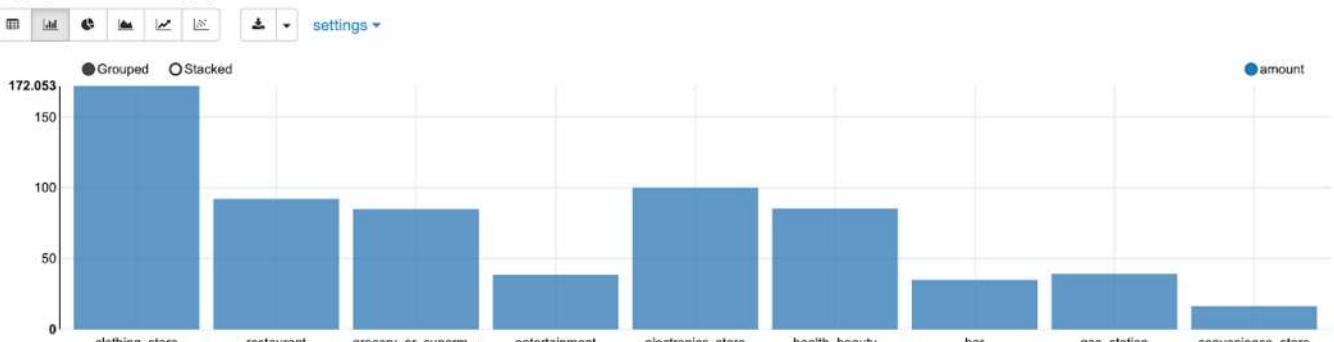
FINISHED

Grouped Stacked

amount

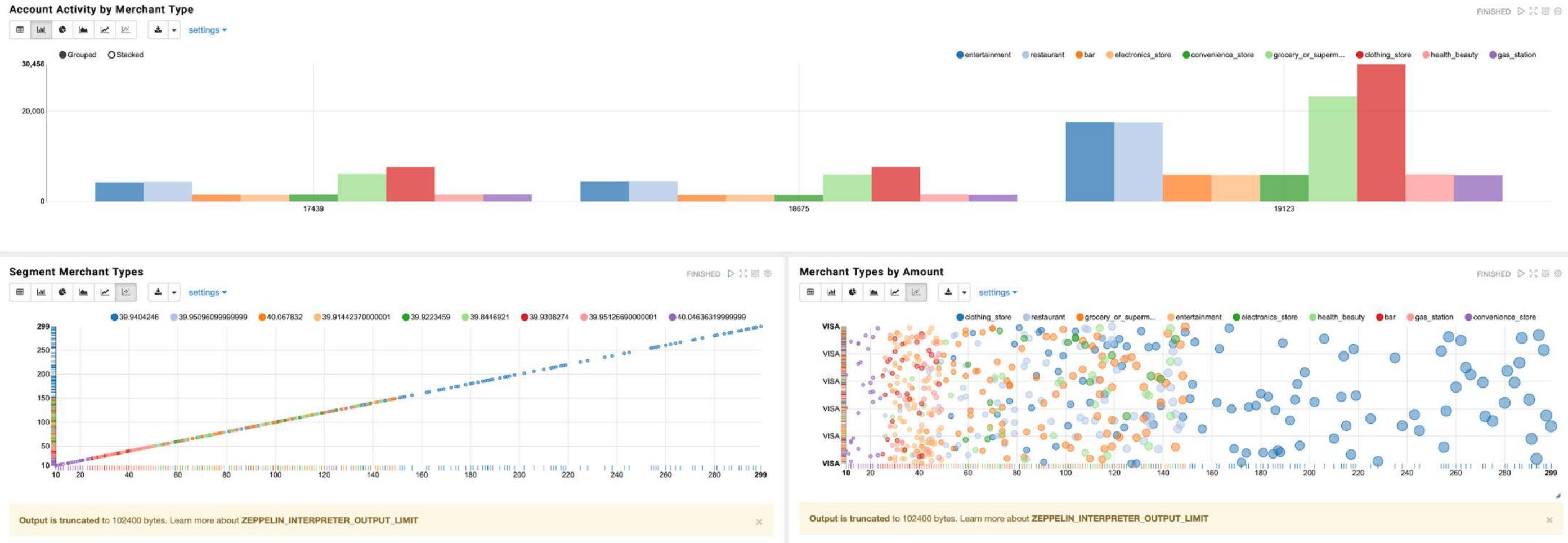
**Avg Spend Per Category**

FINISHED



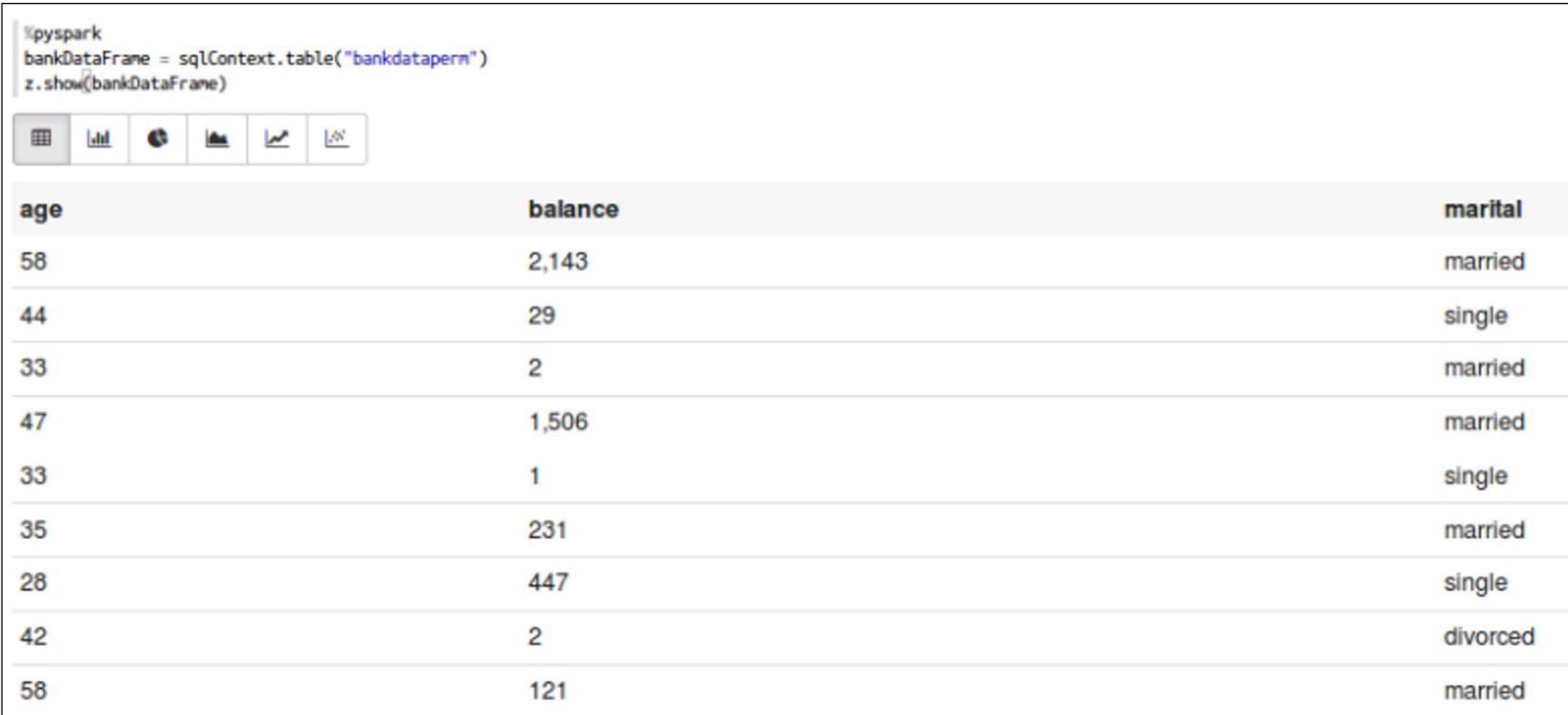
Output is truncated to 102400 bytes. Learn more about ZEPPELIN\_INTERPRETER\_OUTPUT\_LIMIT

# Tables Visualizations



# DataFrame Visualizations

- z.show(DataFrameName)



The screenshot shows a Jupyter Notebook cell with the following content:

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

Below the code, there is a toolbar with six icons: grid, chart, globe, histogram, line graph, and scatter plot.

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

## Formatted Data Visualizations

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

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

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

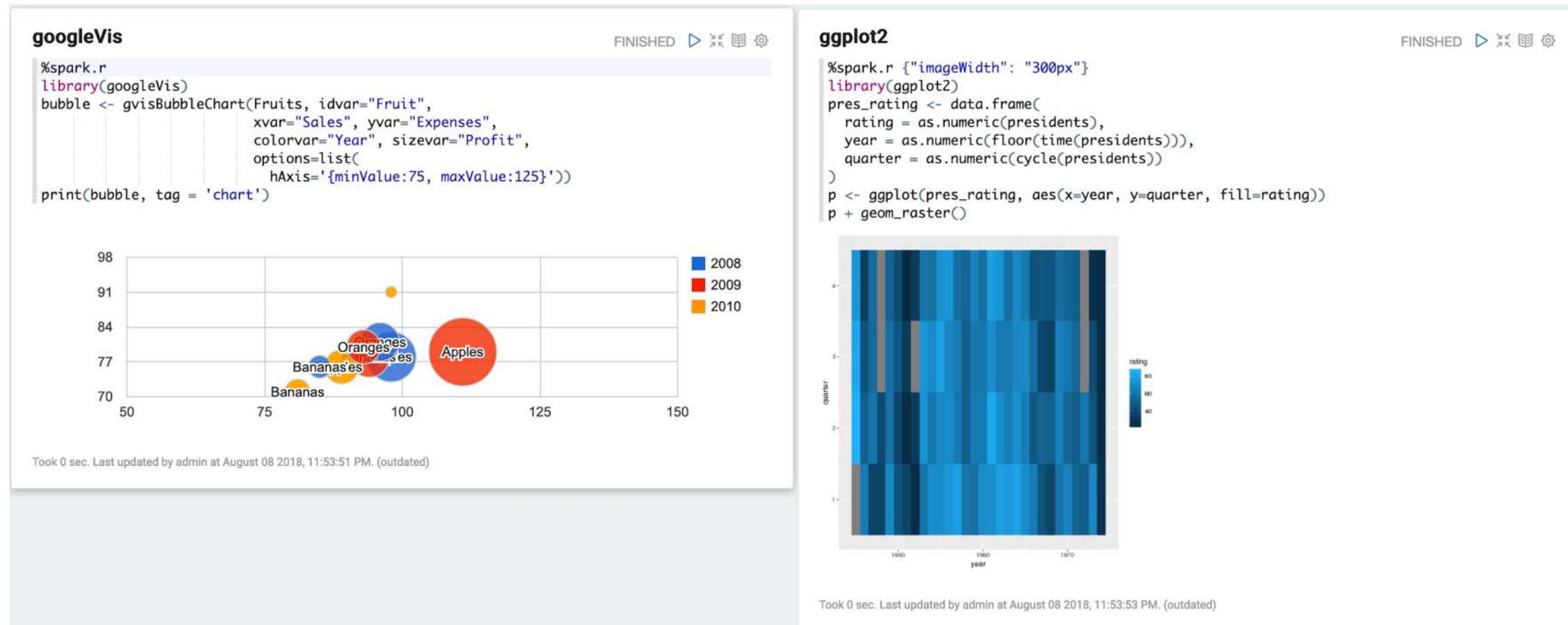
code	value
AA	150000
BB	80000

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

code	value
AA	150,000
BB	80,000

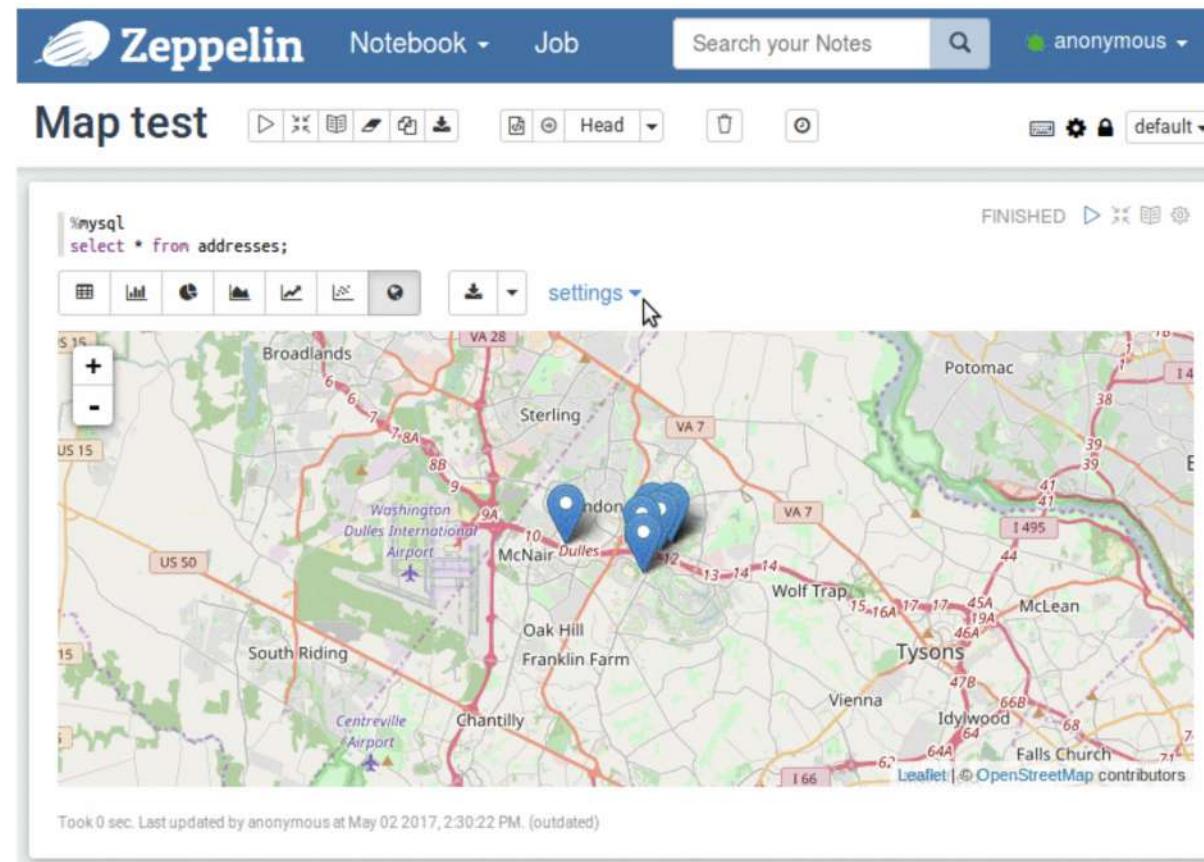
# Python and R Visualizations Libraries

- Python and R visualization frameworks can be used if appropriately installed and configured
- These frameworks will require data to be regrouped locally and are not suitable for visualizing large datasets



# The Helium Project

- The [Helium Project](#) seeks to increase Zeppelin's pluggability by leveraging a packaging system
- Some of the packages already available add additional visualization features like geospatial visualization
- The list of available packages can be consulted [here](#)



# Chapter Topics

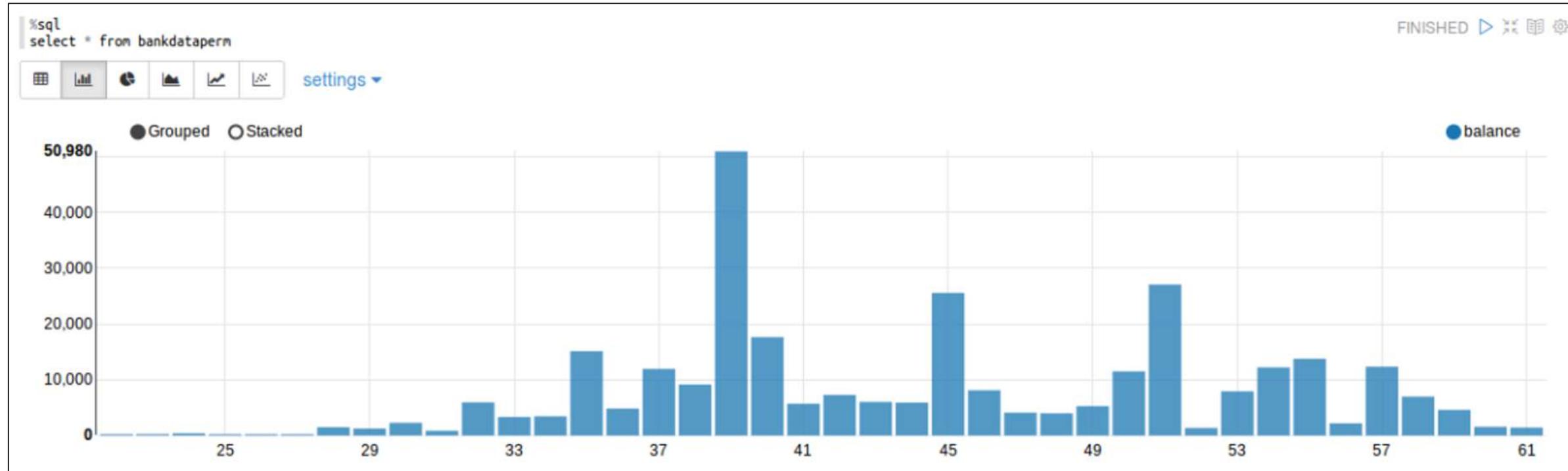
---

## Data Visualization with Zeppelin

- Introduction to Data Visualization with Zeppelin
- **Zeppelin Analytics**
- Zeppelin Collaboration
- Exercise: AdventureWorks

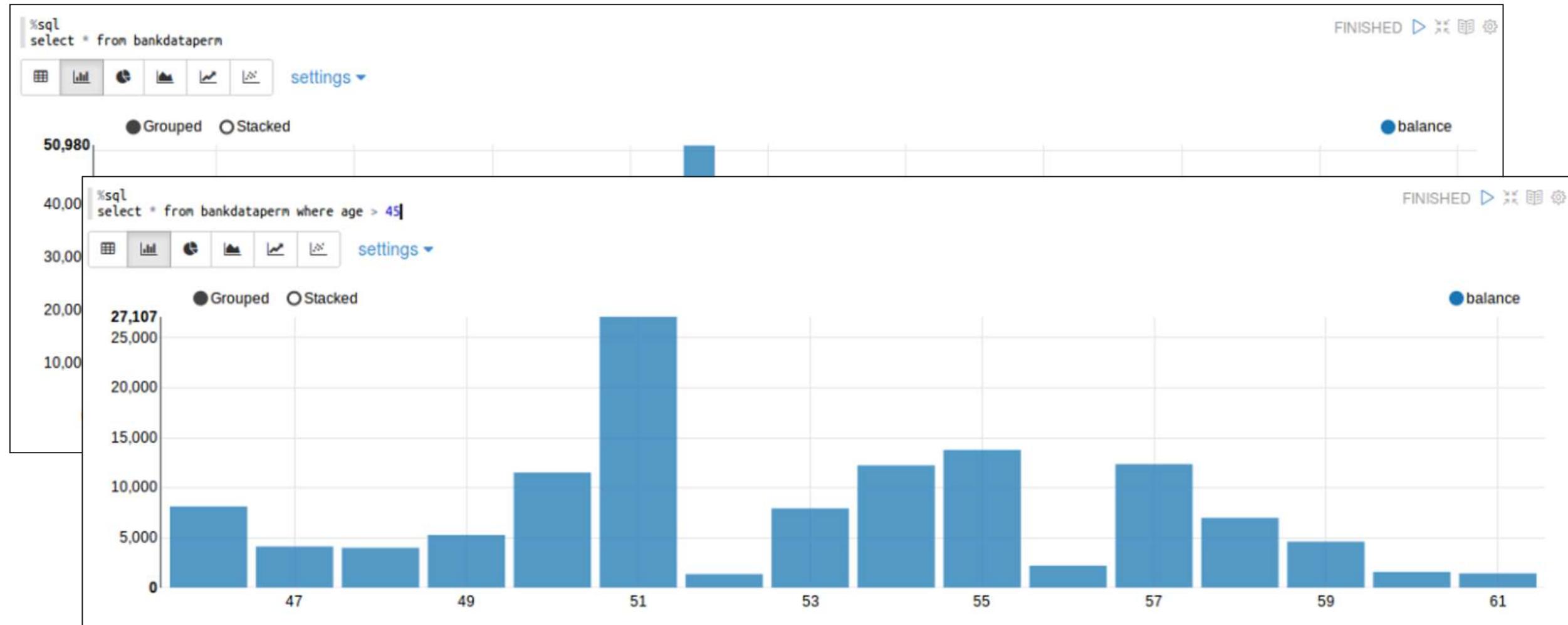
# Interactive Visualizations - Programmatic

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



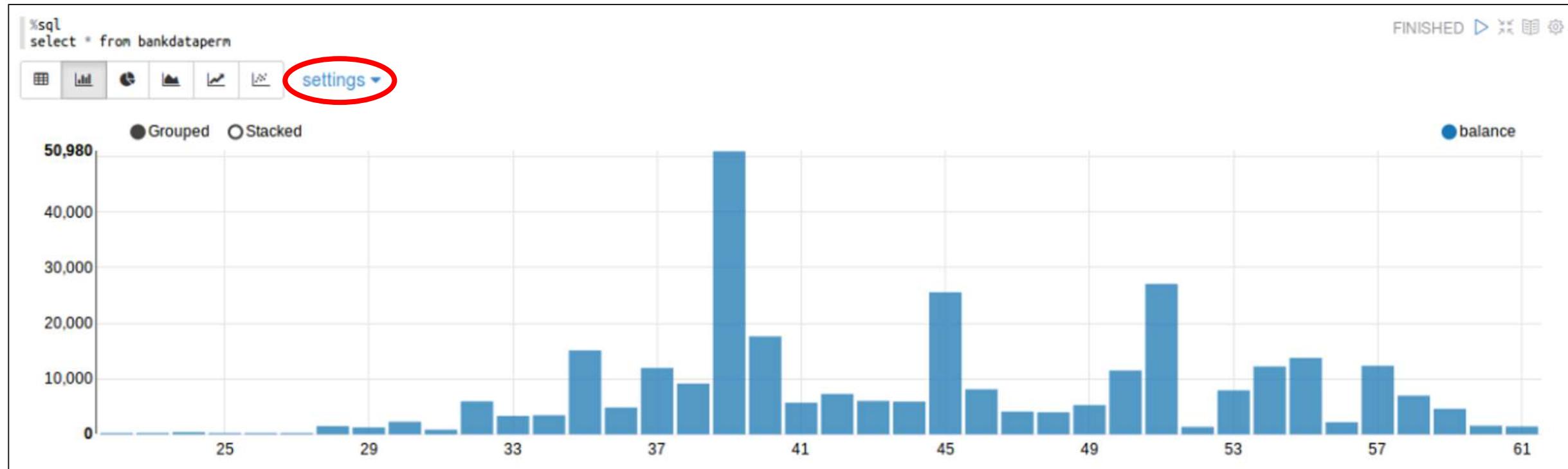
# Interactive Visualizations - Programmatic

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



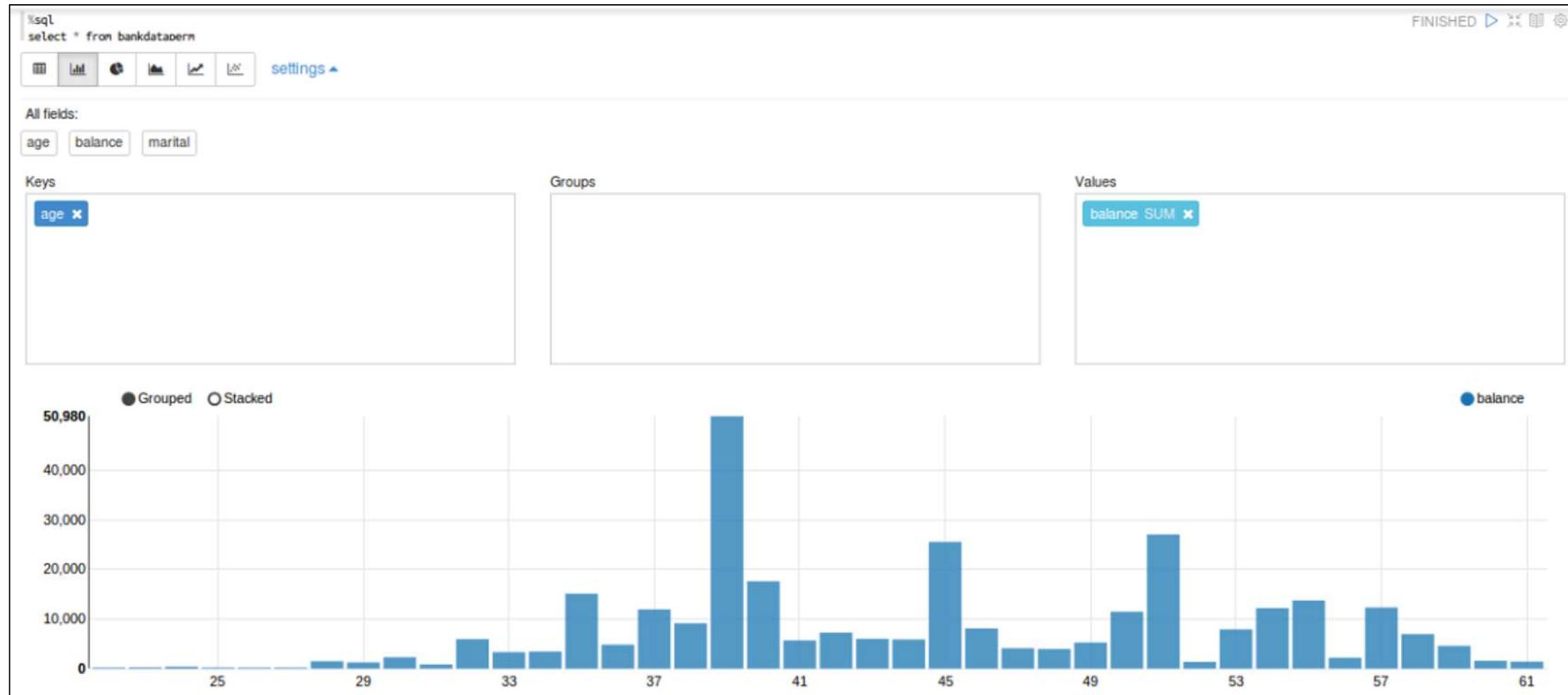
## Interactive Visualizations - Pivot Charts

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



# Interactive Visualizations - Pivot Charts

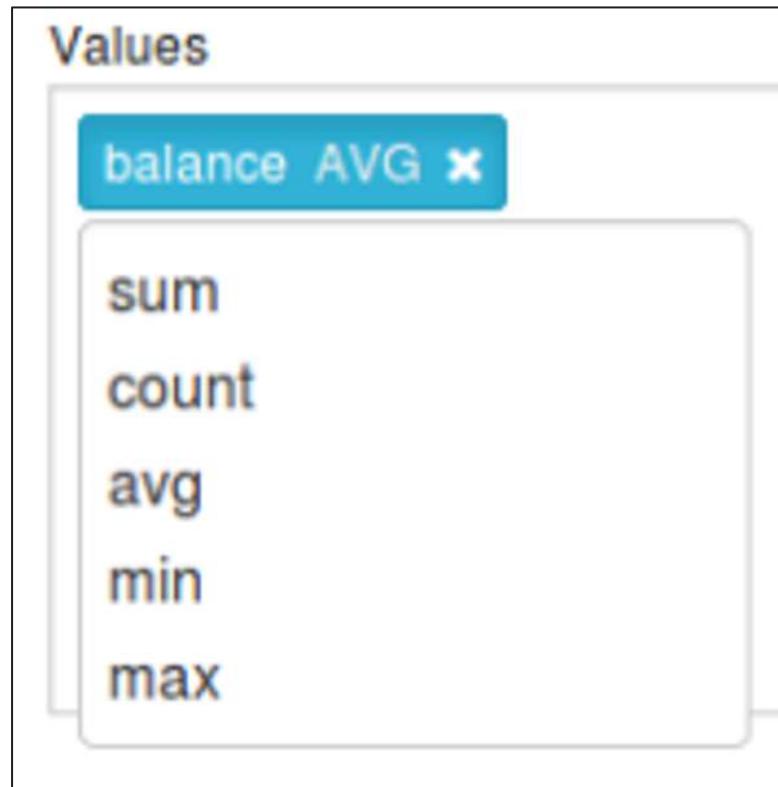
- In addition, Zeppelin provides a Pivot Chart capability under Settings in which additional data manipulations can be performed without changing the original query or command



## Pivot Chart - Value Options

---

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



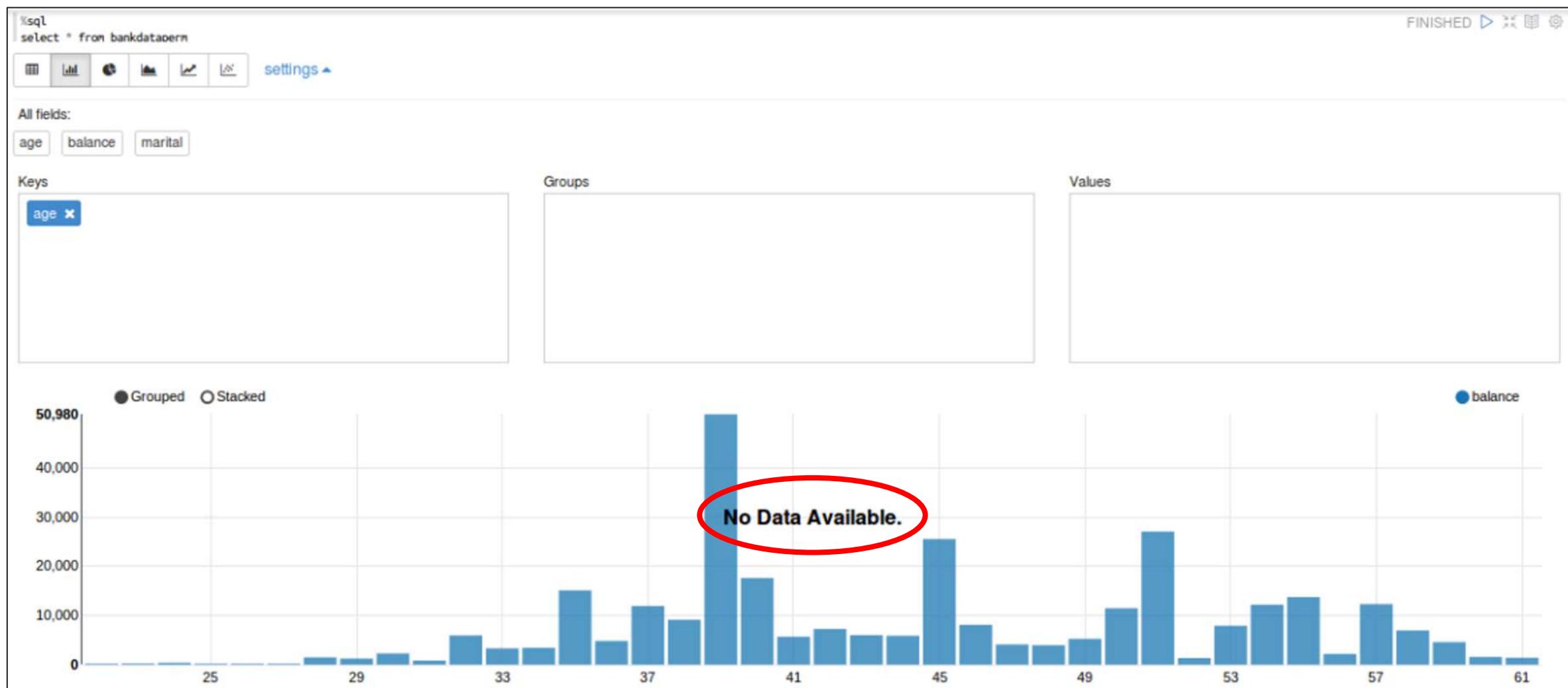
## Pivot Charts - Change Values or Keys

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



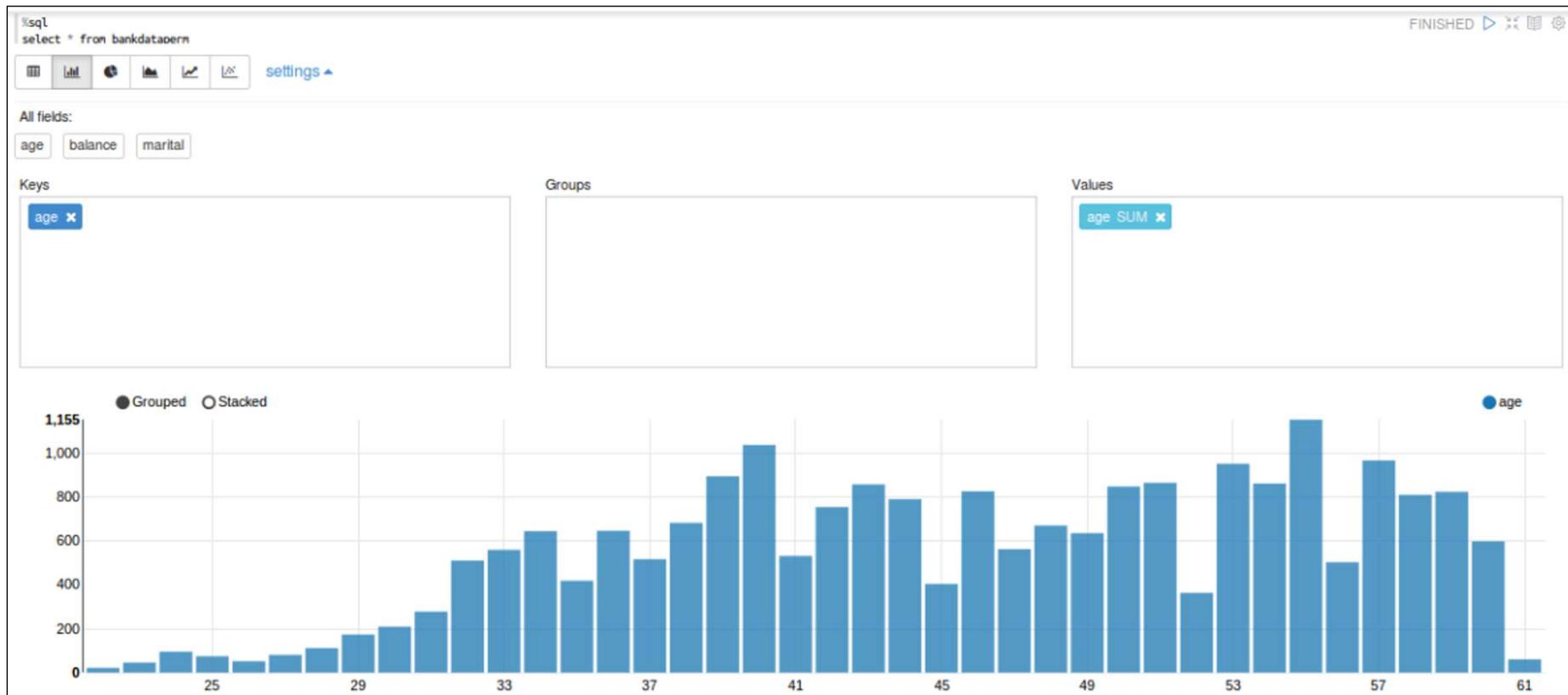
## Pivot Charts - Change Values or Keys

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



## Pivot Charts - Change Values or Keys

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



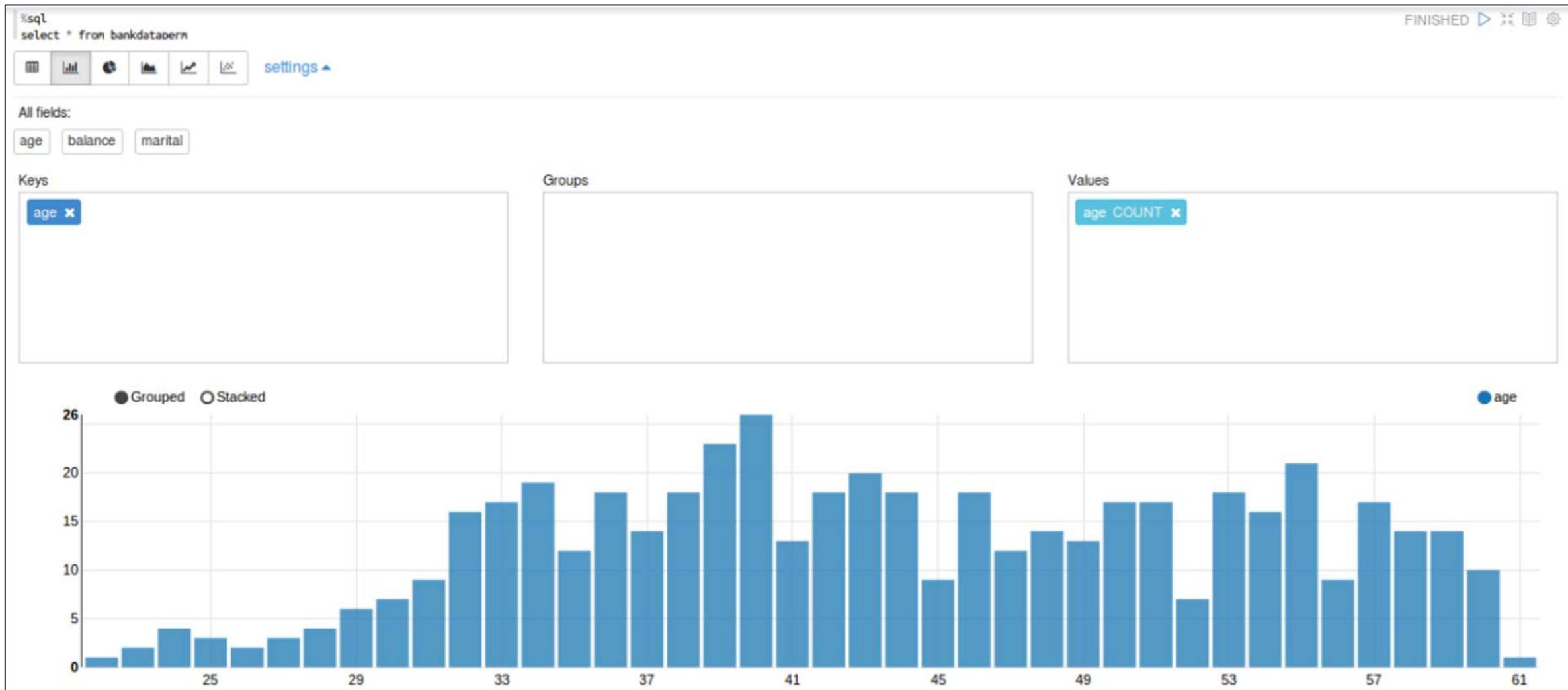
## Pivot Charts - Change Values or Keys

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



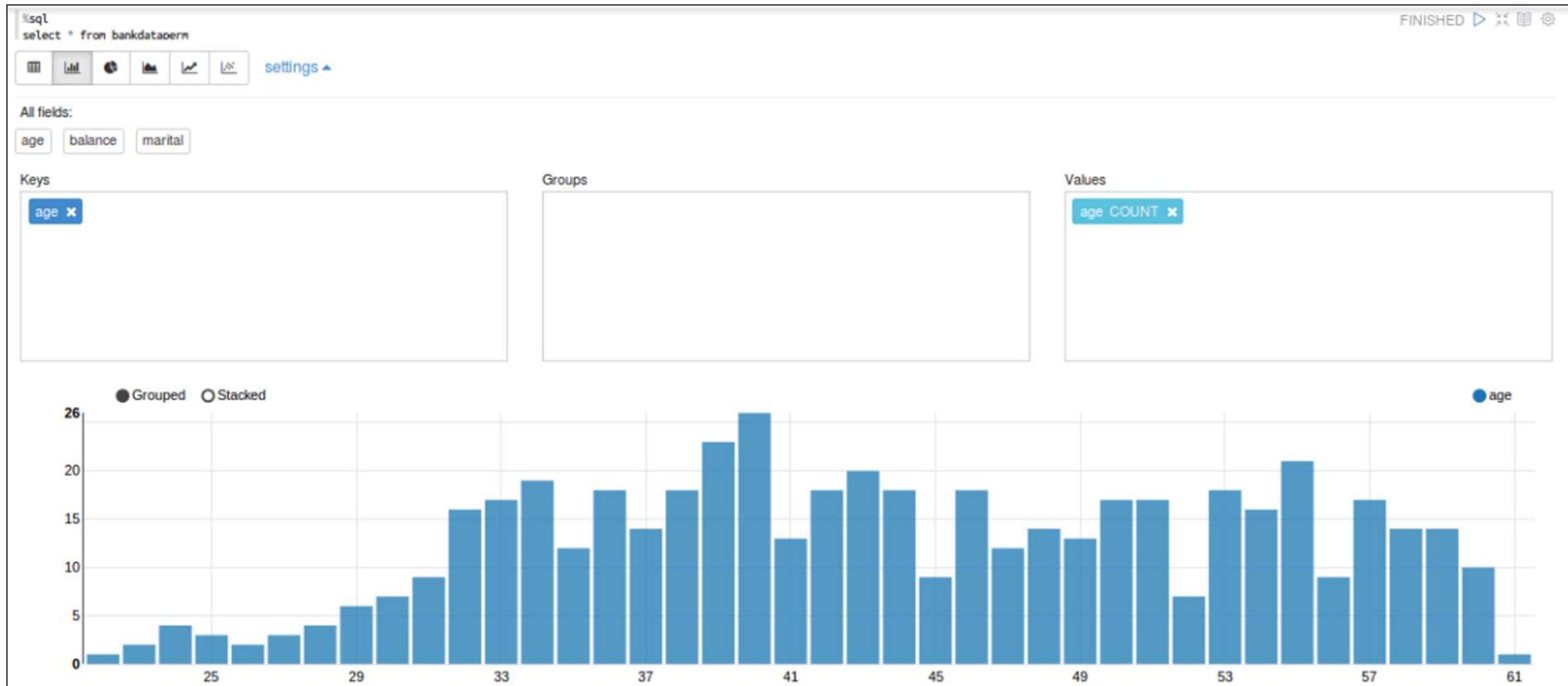
## Pivot Charts - Change Values or Keys

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



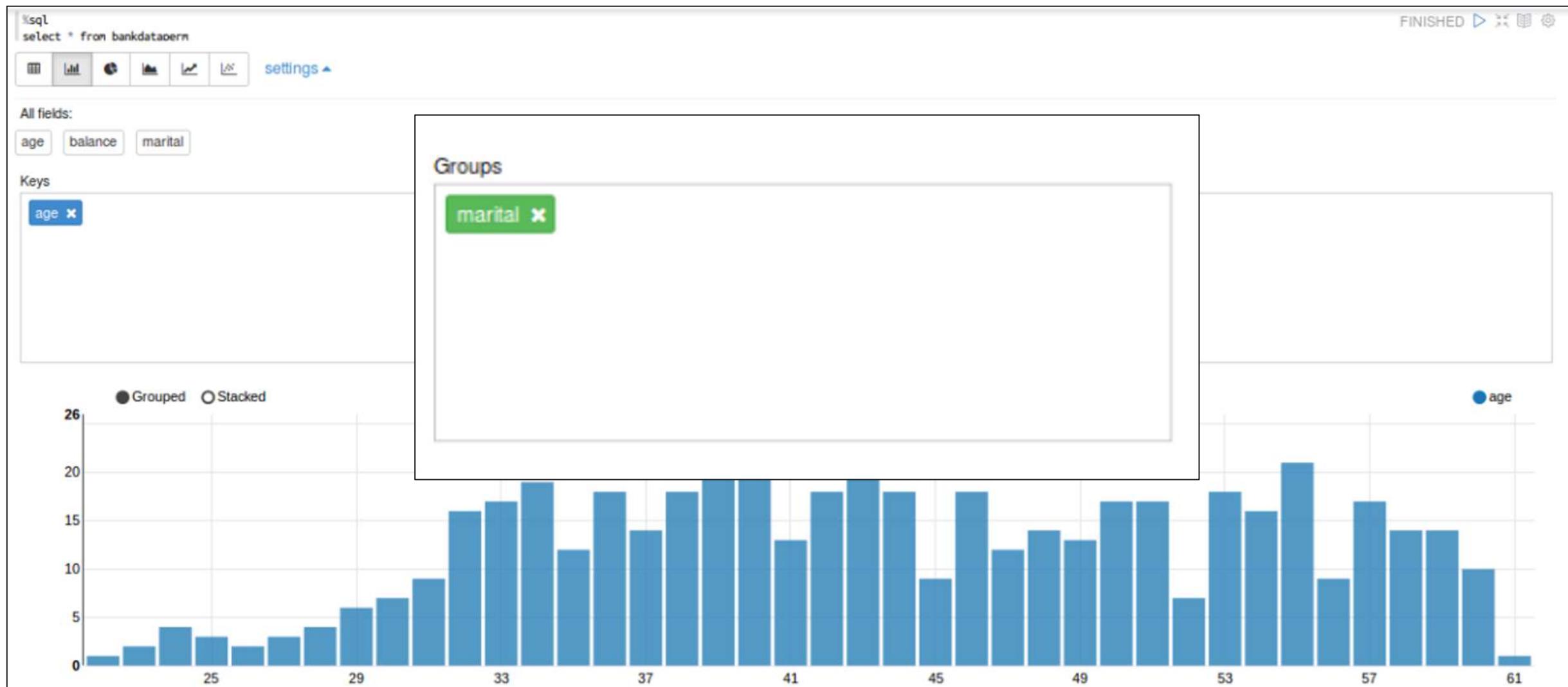
## Pivot Charts - Add Groups

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



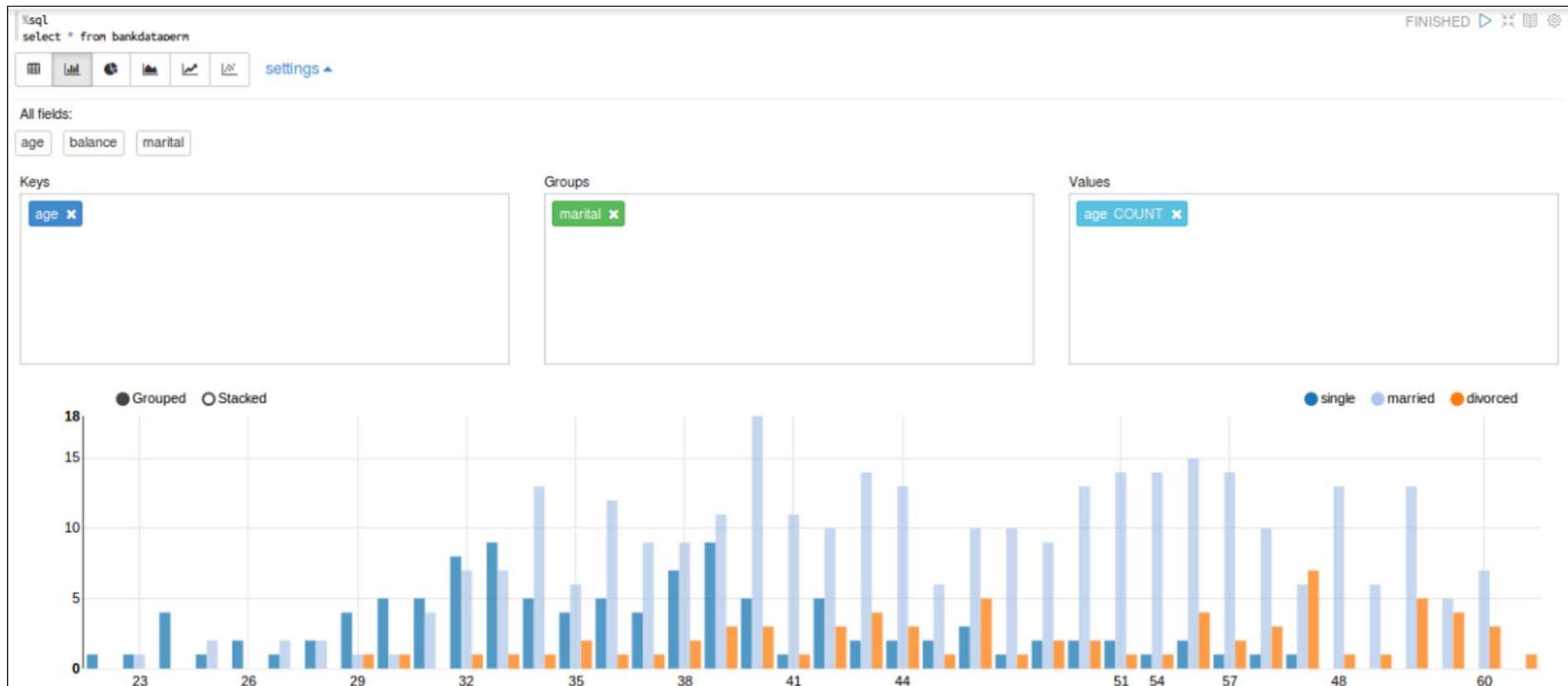
## Pivot Charts - Add Groups

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



## Pivot Charts - Add Groups

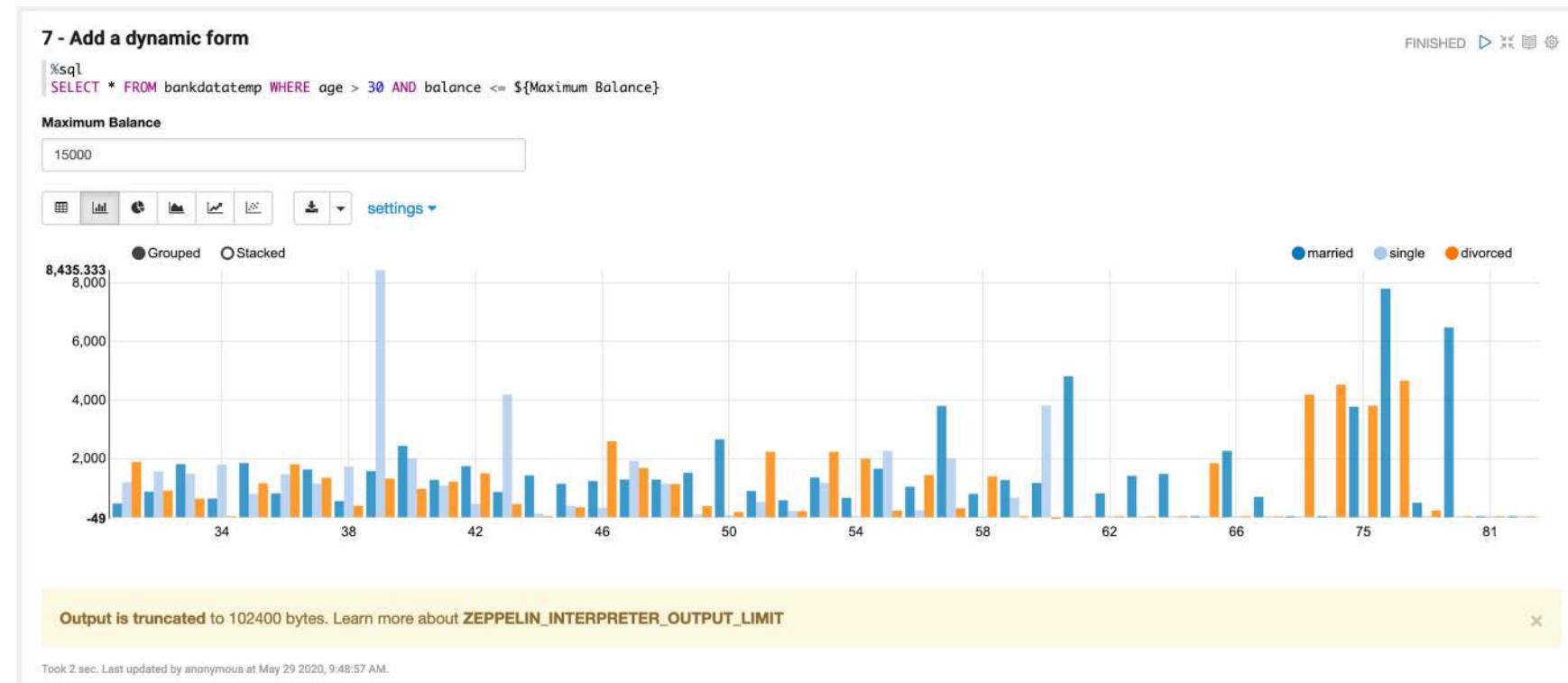
- Drag and drop the appropriate grouping category from the list of options to see the data further broken down



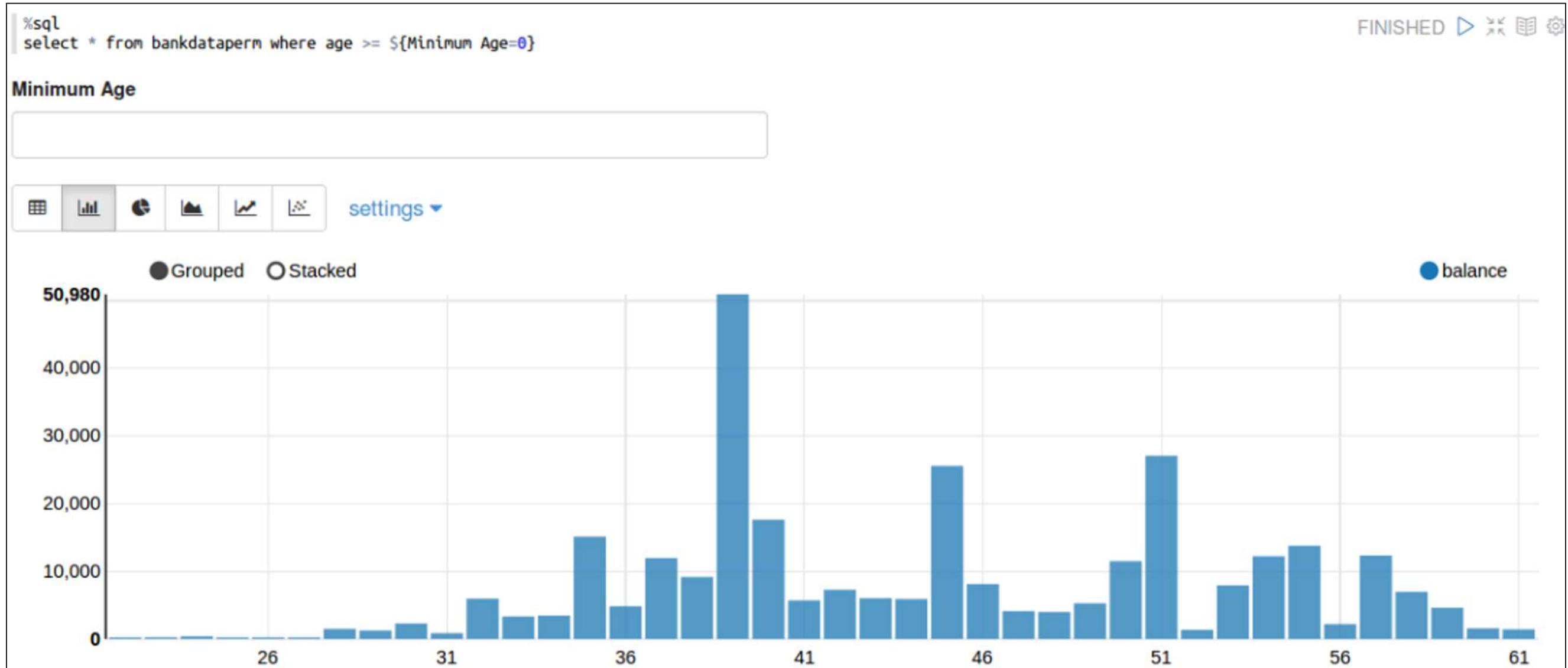
# Dynamic Forms

- Sometimes Pivot Charts don't provide the flexibility needed when interacting with data
- In these instances, Dynamic Forms can be implemented in the query / command to provide parameters for WHERE clauses

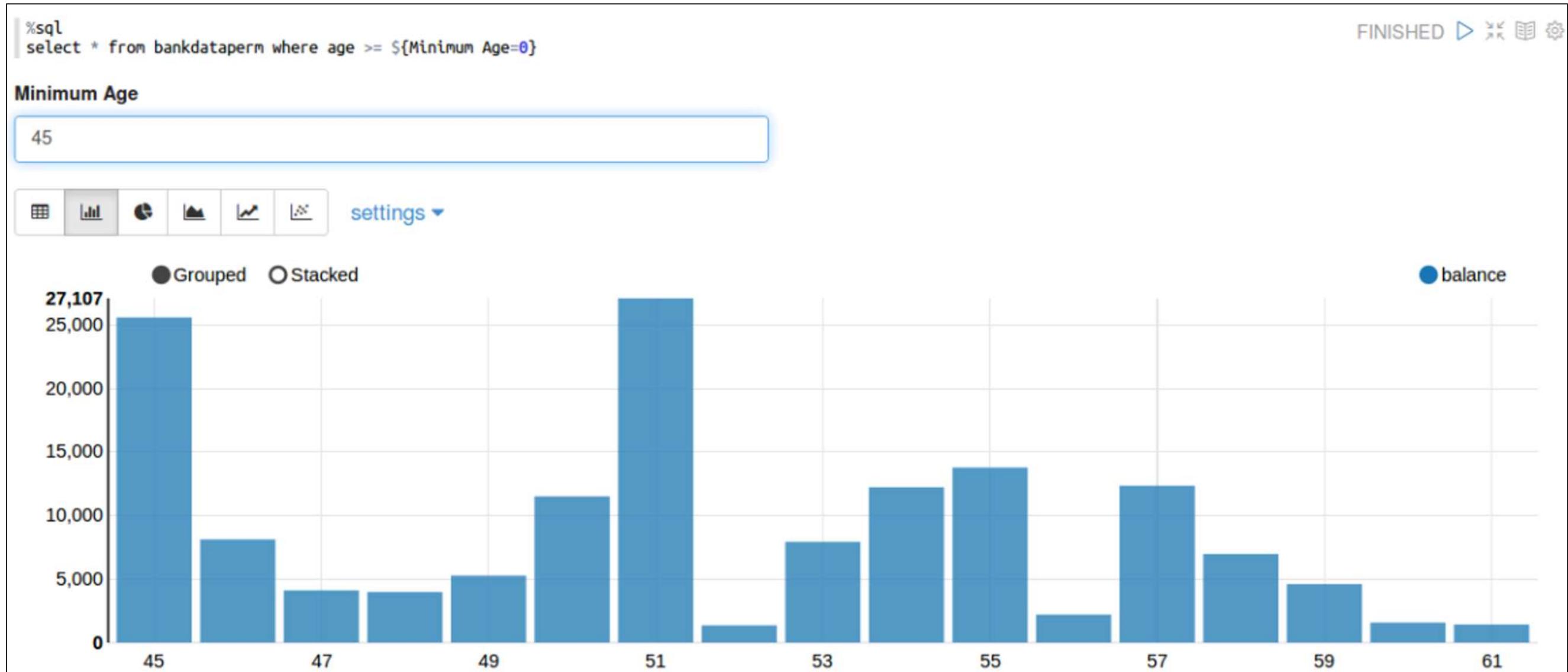
```
SELECT * FROM table WHERE colName [mathOp] ${LabelName=DefValue}
```



# Dynamic Forms

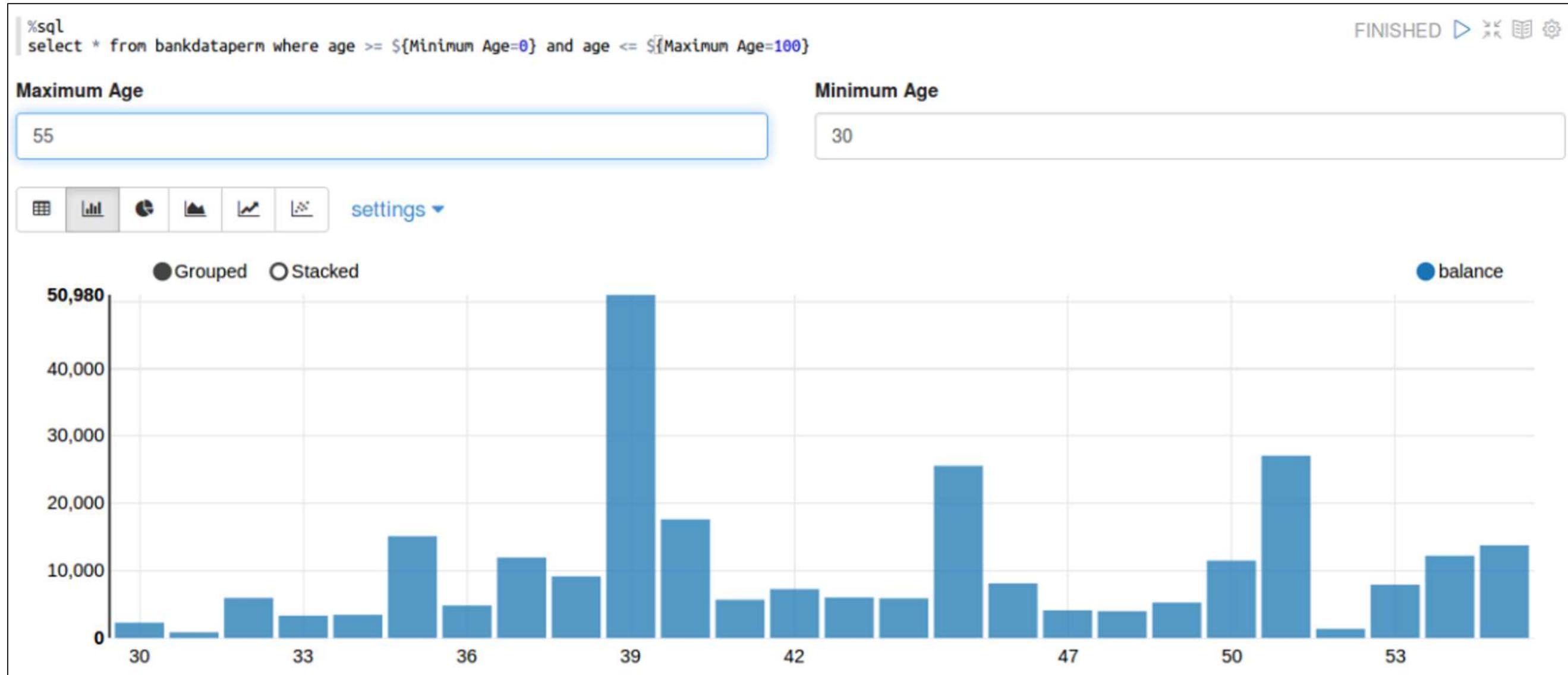


# Dynamic Forms



# Dynamic Forms - Multiples

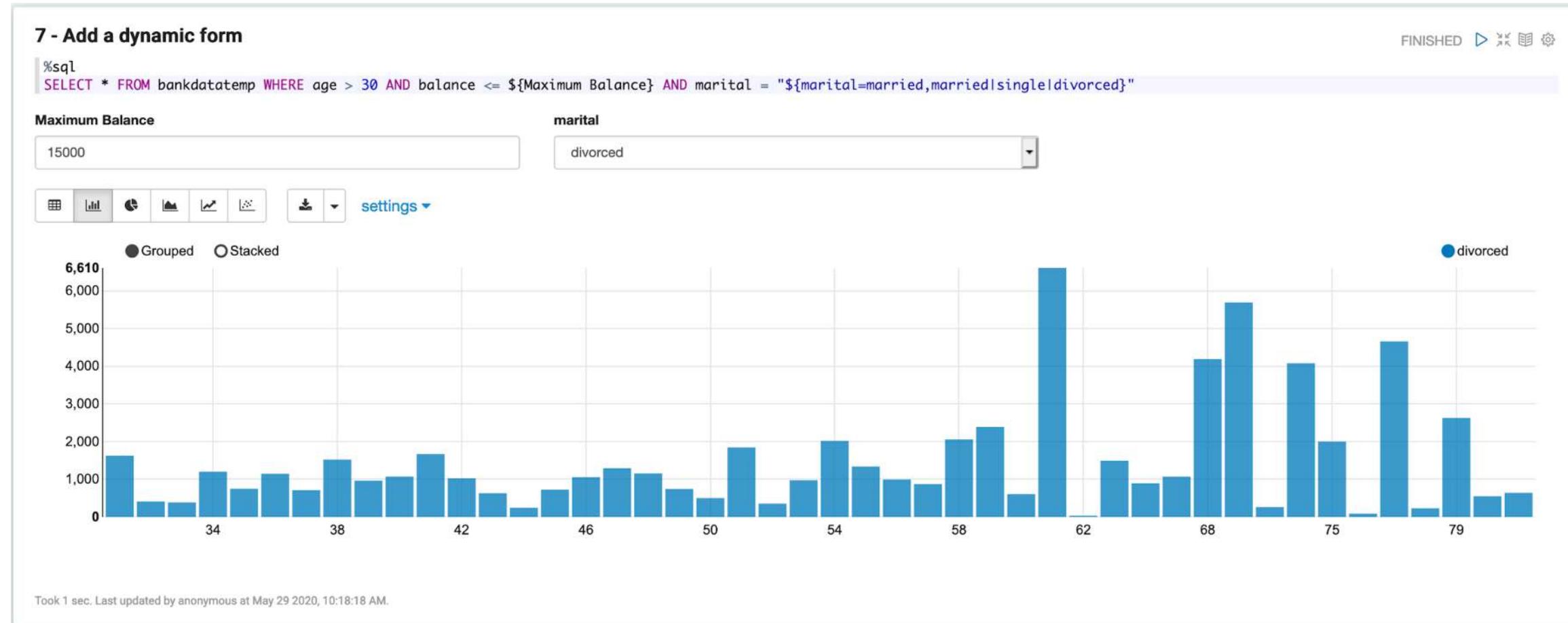
- Multiple variables can be included as Dynamic Forms



# Dynamic Forms - Select

- Select forms (drop-down menus) can be created as Dynamic Forms as well.

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



# Dynamic Forms - Select

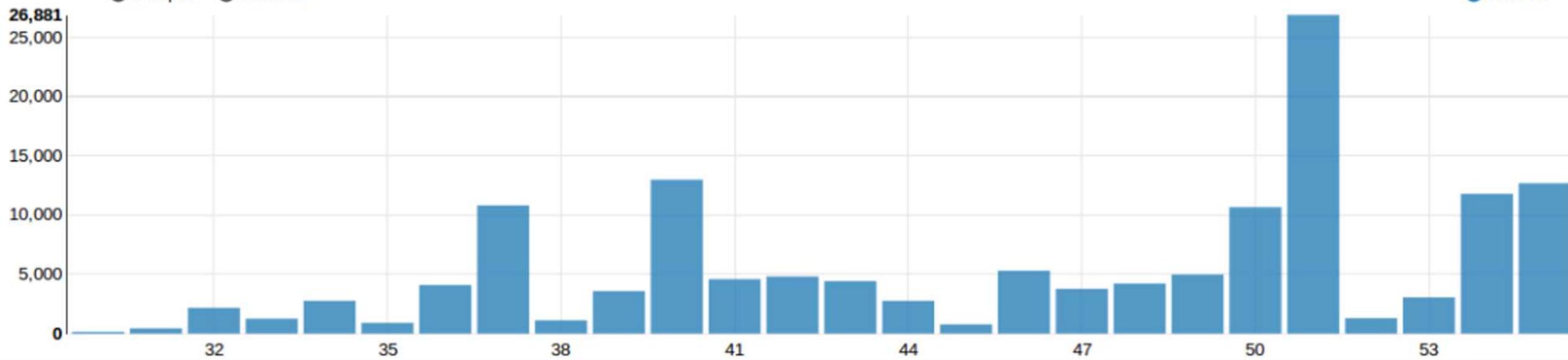
```
select * from bankdataperm where age >= ${Minimum Age=0} and age <= ${Maximum Age=100} and marital = "${marital=married,married|single|divorced}"
```

marital

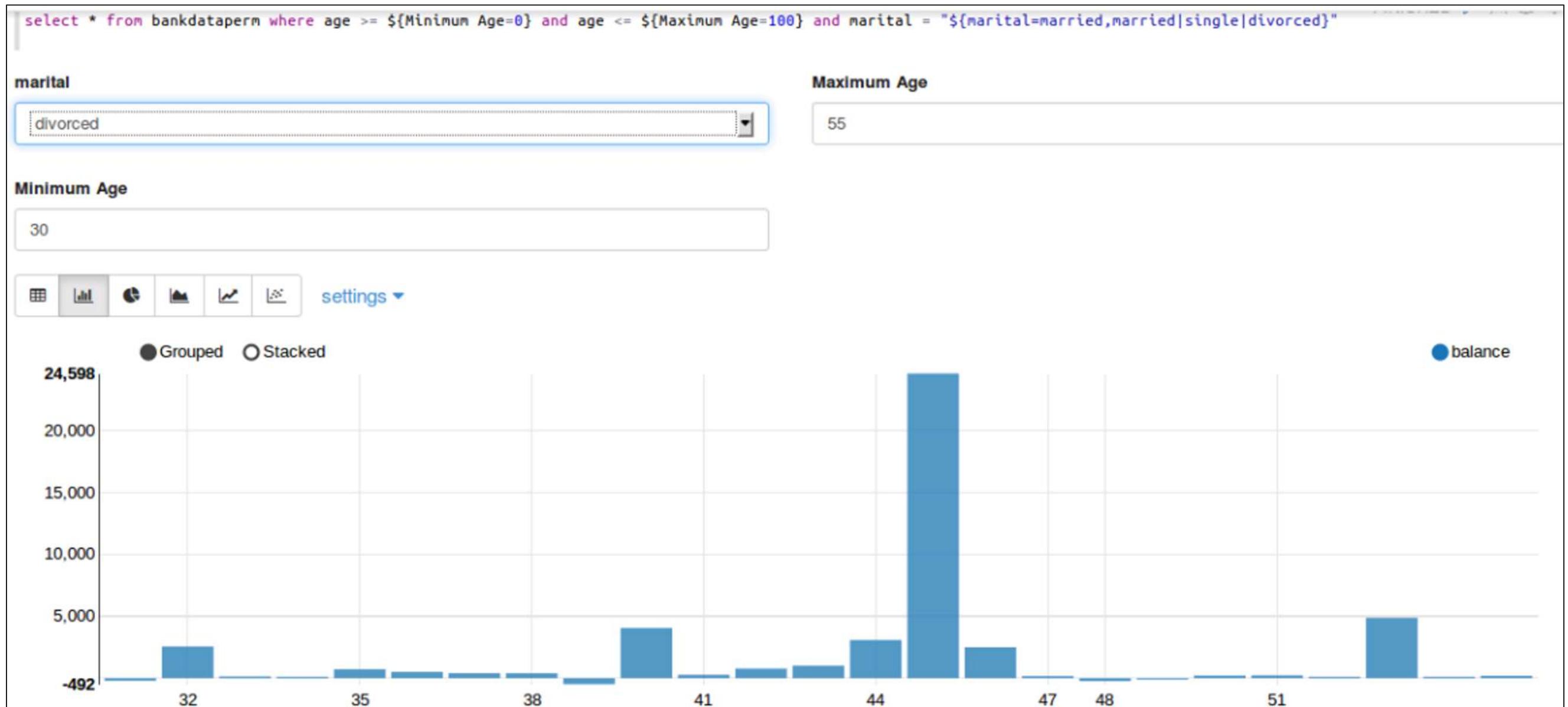
married  
single  
divorced

● Grouped   ○ Stacked

● balance



# Dynamic Forms - Select



# Chapter Topics

---

## Data Visualization with Zeppelin

- Introduction to Data Visualization with Zeppelin
- Zeppelin Analytics
- **Zeppelin Collaboration**
- Exercise: AdventureWorks

## Clone and Export a Note

---

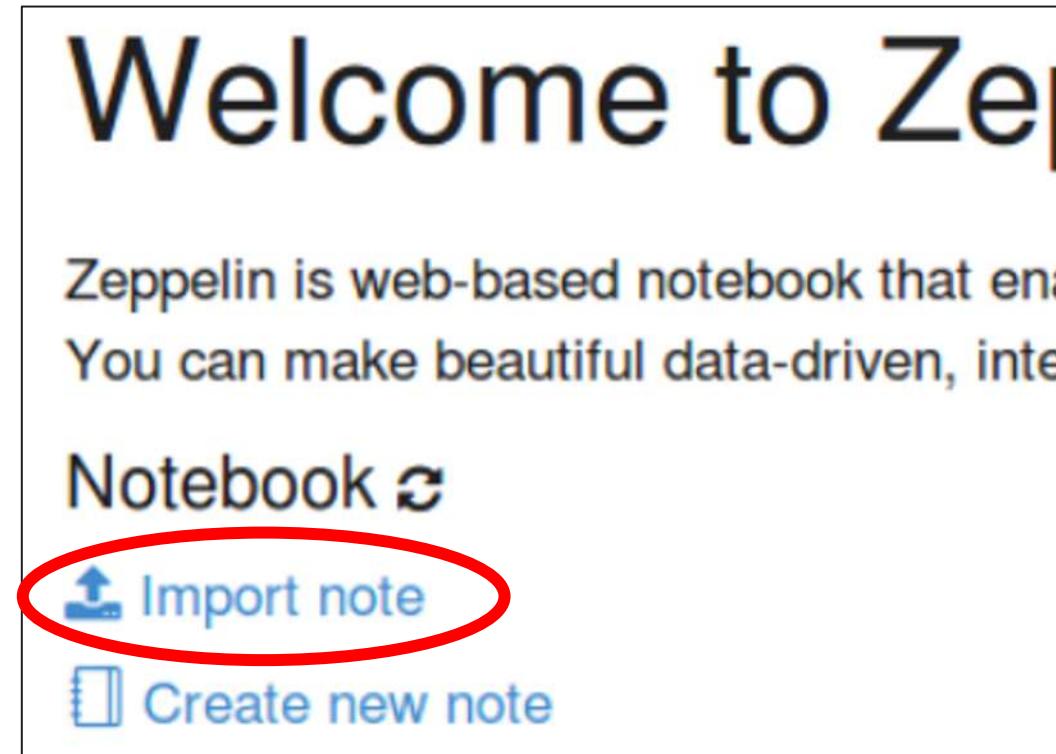
- Before sharing a note with others, it may be a good idea to make a copy of it
- Two ways to do this:
  - Clone: make a copy of the note in Zeppelin
  - Export: save a copy of the note in JSON format



## Import a Note

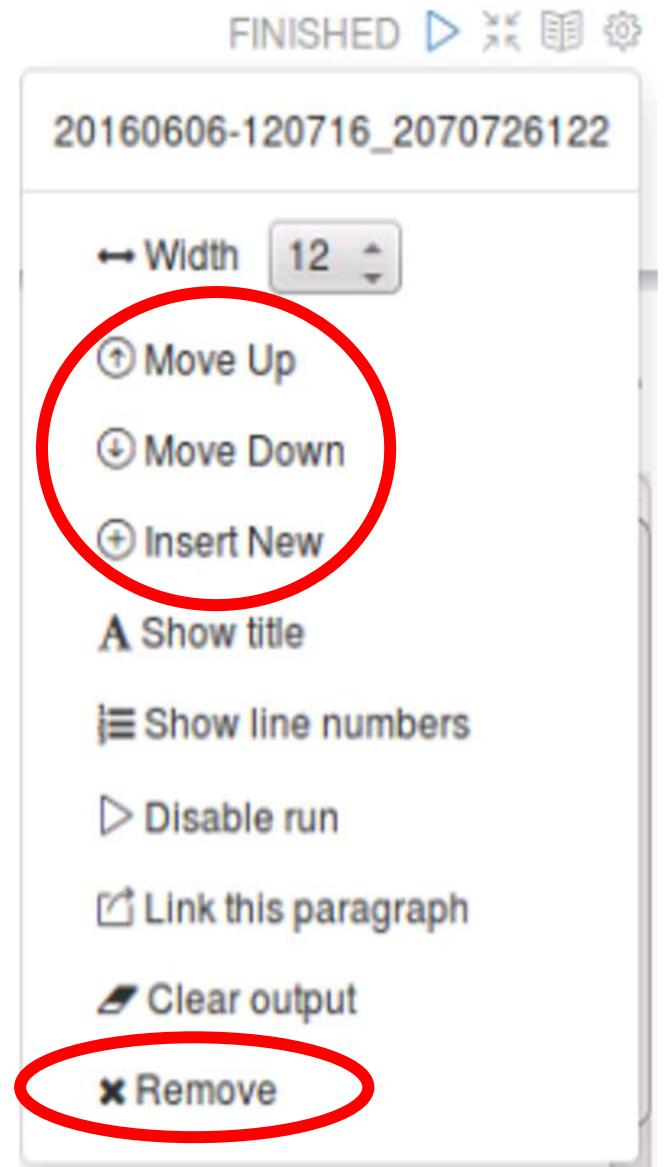
---

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



## Note Cleanup

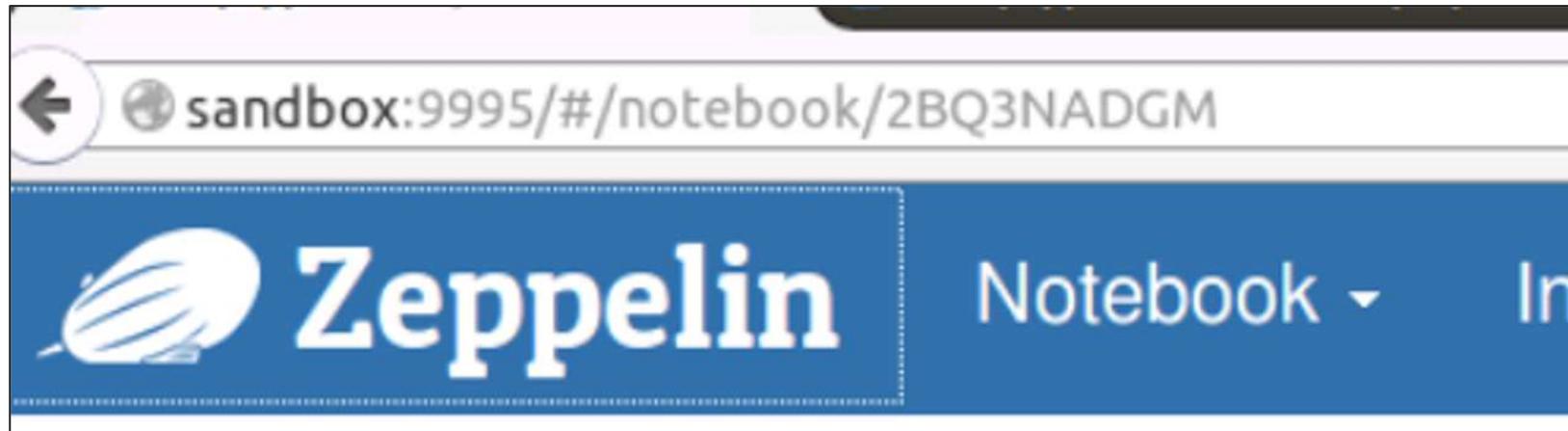
- In-process notes can be messy and contain unnecessary duplicate code or alternatives
- Individual paragraphs that are no longer needed can be deleted from the note
- Paragraphs can also be reordered and new paragraphs can be inserted
  - For example, to add Markdown comments



# Interactive Note Sharing

---

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



## Note Access Control

- By default, anyone with the note link can completely control the note
- To control access, click the Note Permissions (padlock) icon at the top-right corner of the note and set permissions accordingly

Note Permissions (Only note owners can change)

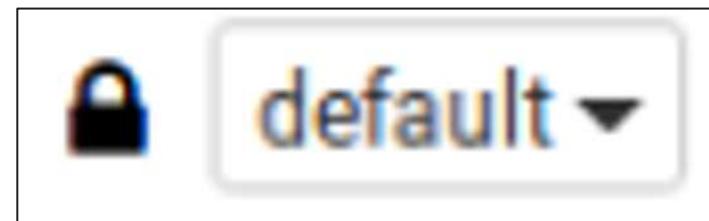
Enter comma separated users and groups in the fields.  
Empty field (\*) implies anyone can do the operation.

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

Readers : \*  Readers can only read the note.

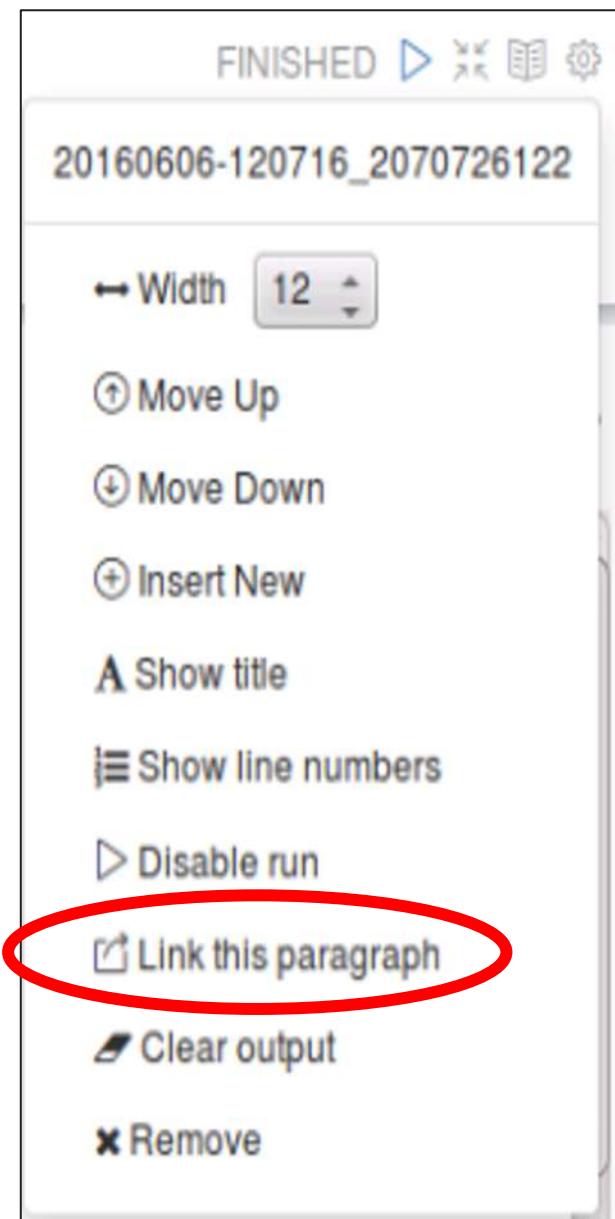
Writers : \*  Writers can read and write the note.

**Save** **Cancel**

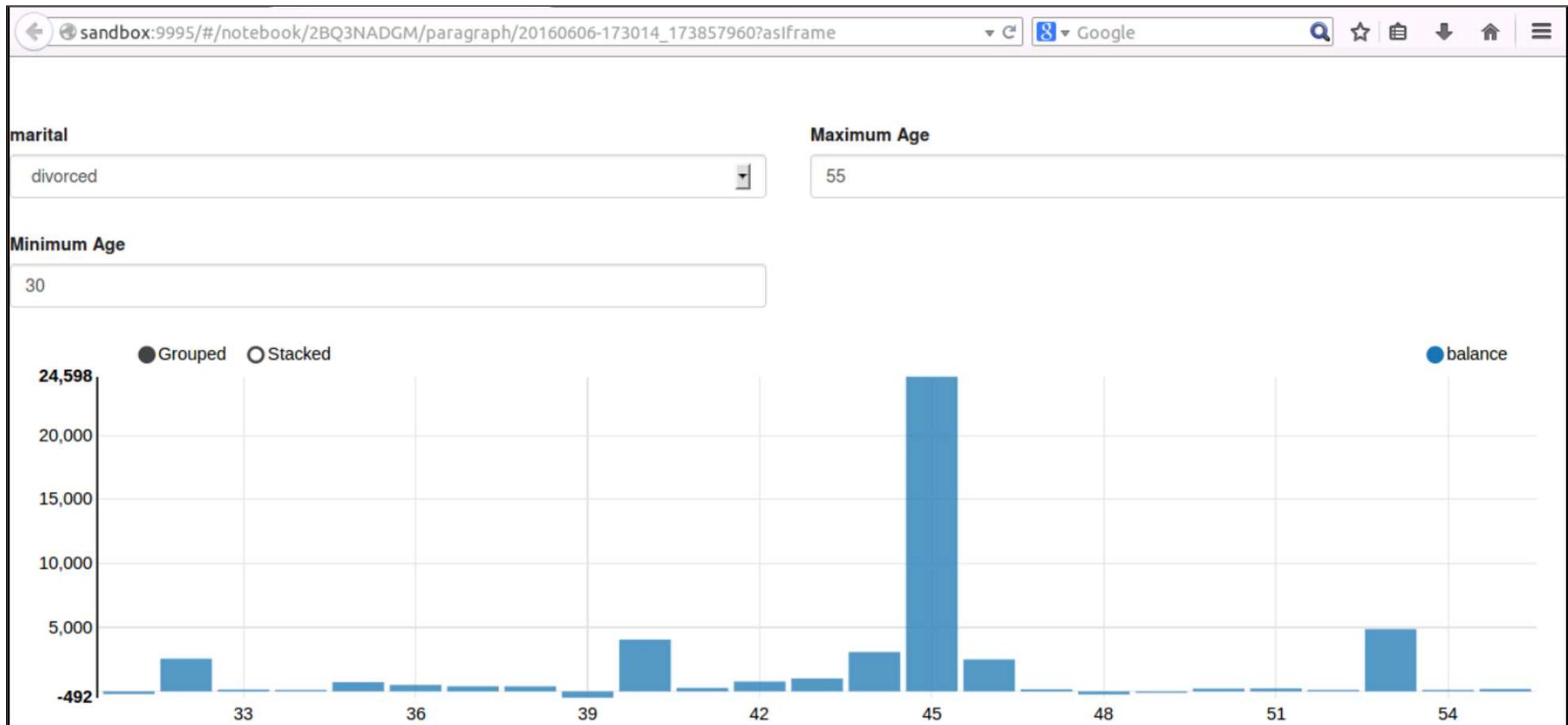


## Paragraph Sharing

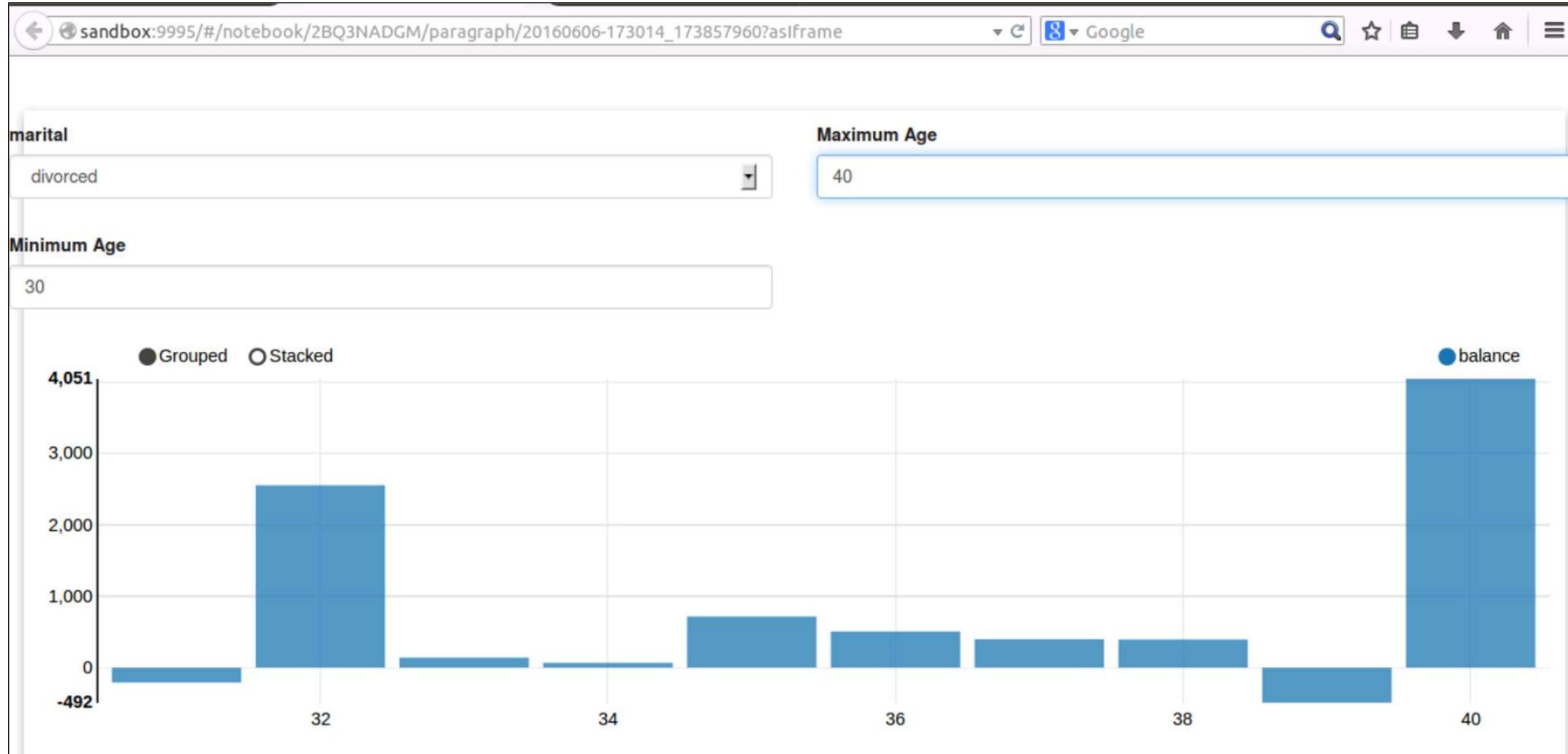
- Individual paragraphs can be shared via links which can be sent to users or embedded in reports generated by other tools
- If Dynamic Forms have been implemented in the paragraph, users will be able to interact with the data, even though they do not have access to the code



# Paragraph Sharing



# Paragraph Sharing



## Knowledge Check

---

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

## Essential Points

---

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

# Chapter Topics

---

## Data Visualization with Zeppelin

- Introduction to Data Visualization with Zeppelin
- Zeppelin Analytics
- Zeppelin Collaboration
- **Exercise: AdventureWorks**



# Distributed Processing Challenges

---

Chapter 11

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- **Distributed Processing Challenges**
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Chapter Topics

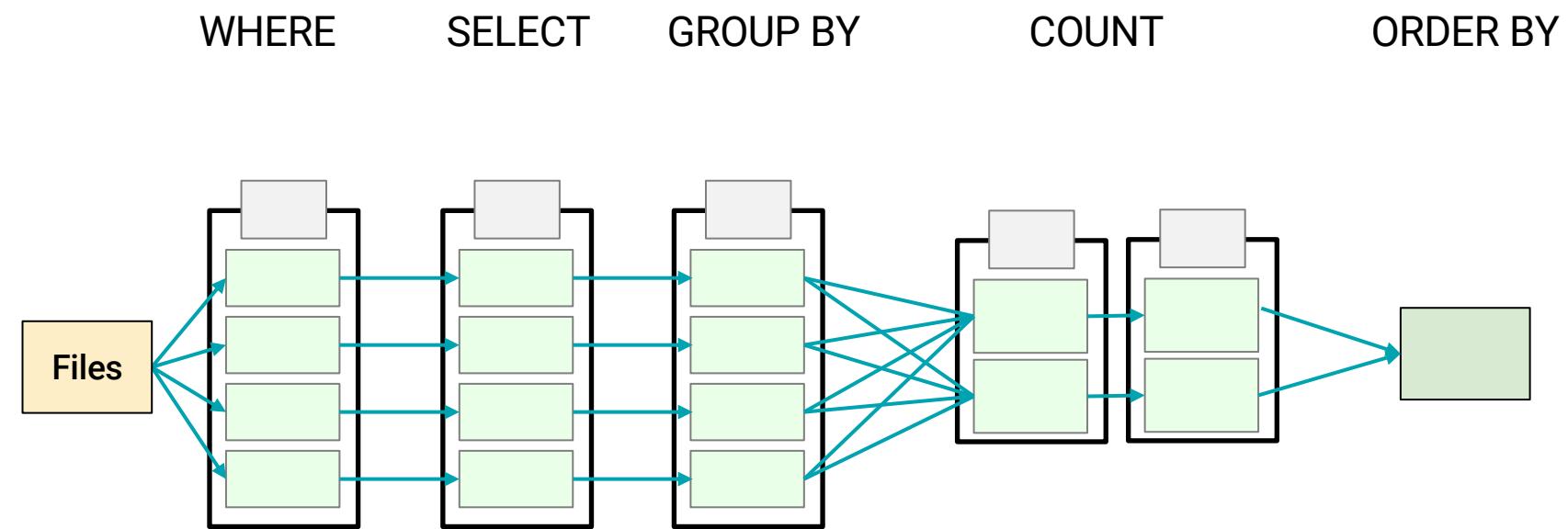
---

## Distributed Processing Challenges

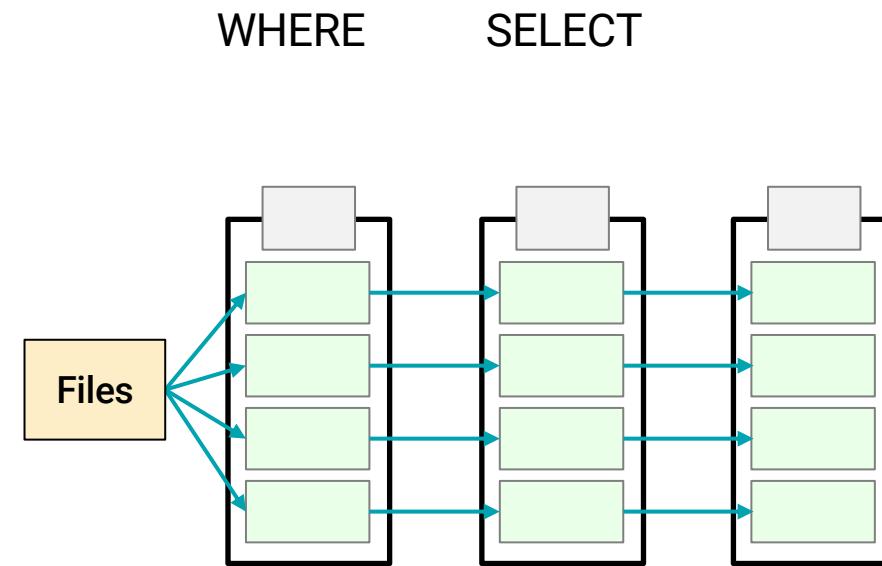
- **Shuffle**
- Skew
- Order

# SELECT COUNT(\*) FROM WHERE GROUP BY ORDER BY

- Let's take a closer look at how distributed processing works with a simple SQL query:



# SELECT WHERE



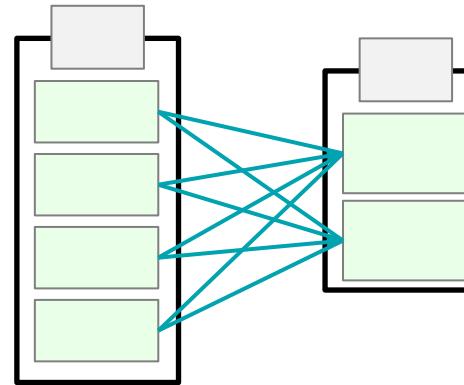
This is the fast part, distributed processing is efficient

- Rule #1: don't read what you don't need ([filters can be pushed down if the file format supports this feature](#))
- Rule #2: filter early
- Rule #3: project early

# GROUP BY (OR JOIN)

---

## GROUP BY



This is a shuffle, it's messy

- Data needs to be materialized in **memory**
- Before spilling to **disk**
- And being shuffled across the **network**

# Chapter Topics

---

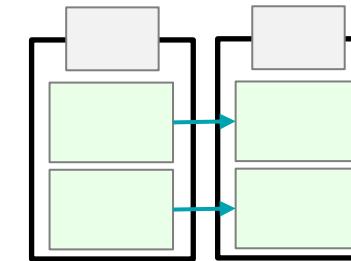
## Distributed Processing Challenges

- Shuffle
- **Skew**
- Order

# SELECT COUNT(\*) FROM WHERE GROUP BY ORDER BY

---

COUNT

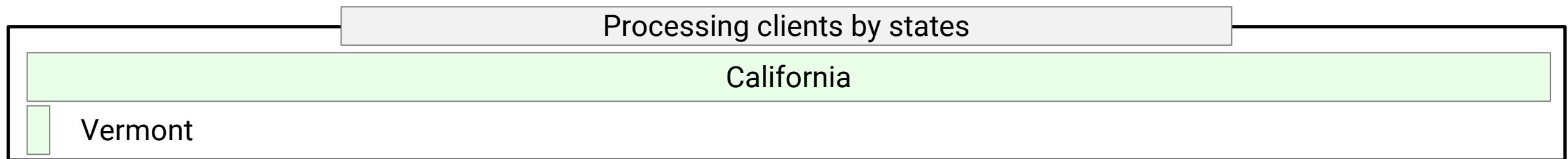


This can also be a problem:

- How many partitions should be used?
- What if the data is skewed for this group by key?

# Skew is Kryptonite for Distributed Processing

- Business data is always skewed (more or less)
- Skew degrades the performance of distributed processing proportionally
  - If you have 50 times more clients in California than in Vermont then if you perform some processing after a group by state then the partition for California will have 50 times more work to do than the one for Vermont



- In case of extreme skew then one of your worker node can be overwhelmed and stall the whole job

Duration	Tasks: Succeeded/Total
27.4 h	199/200

# Chapter Topics

---

## Distributed Processing Challenges

- Shuffle
- Skew
- Order

# **SELECT COUNT(\*) FROM WHERE GROUP BY ORDER BY**

---

ORDER BY



This can also be a problem:

- We are using a 'share nothing' model so JVMs cannot communicate with one another
- Therefore all the data that needs to be sorted should be in the same JVM
- What if the data to be sorted cannot fit in a single JVM?

# Distributed Processing and Implicit Order

- Assume you want to process a text file. The records in this text file have a implicit order: their offset.
- In order to process this file in parallel, the content of this file must be hash partitioned between the machines that will perform the processing
  - The implicit order is shuffled
- Then at the end of the distributed processing you want to save your work. Each machine involved in the process will dump its partition of the result set in the same HDFS folder as soon as they are done.
  - The order is shuffled again.

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

Three executors were involved in saving devDF to HDFS.

```
$ hdfs dfs -ls devices

Found 4 items
-rw-r--r-- 3 training training 2149 ... devices/_SUCCESS
-rw-r--r-- 3 training training 2119 ... devices/part-00000-e0fa63811-....csv
-rw-r--r-- 3 training training 2202 ... devices/part-000011-e0fa63811-....csv
-rw-r--r-- 3 training training 2333 ... devices/part-00002-e0fa63811-....csv
```

# Essential Points

---

- Spark SQL does a great job at hiding the complexity of distributed processing
- But the challenges of distributed processing are still there and will hurt you if you are not wary of them
  - Shuffle
  - Skew
  - Order
- Spark will do its utmost to minimize the impact of these
  - Newer versions will do a better job
- Optimizations lead to more optimizations
  - and non optimizations lead to more non optimizations
- In order to avoid being disappointed it is critical to understand
  - the challenges above
  - as well as what are your options given your version of Spark



# Spark Distributed Processing

---

Chapter 12

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- **Spark Distributed Processing**
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# 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

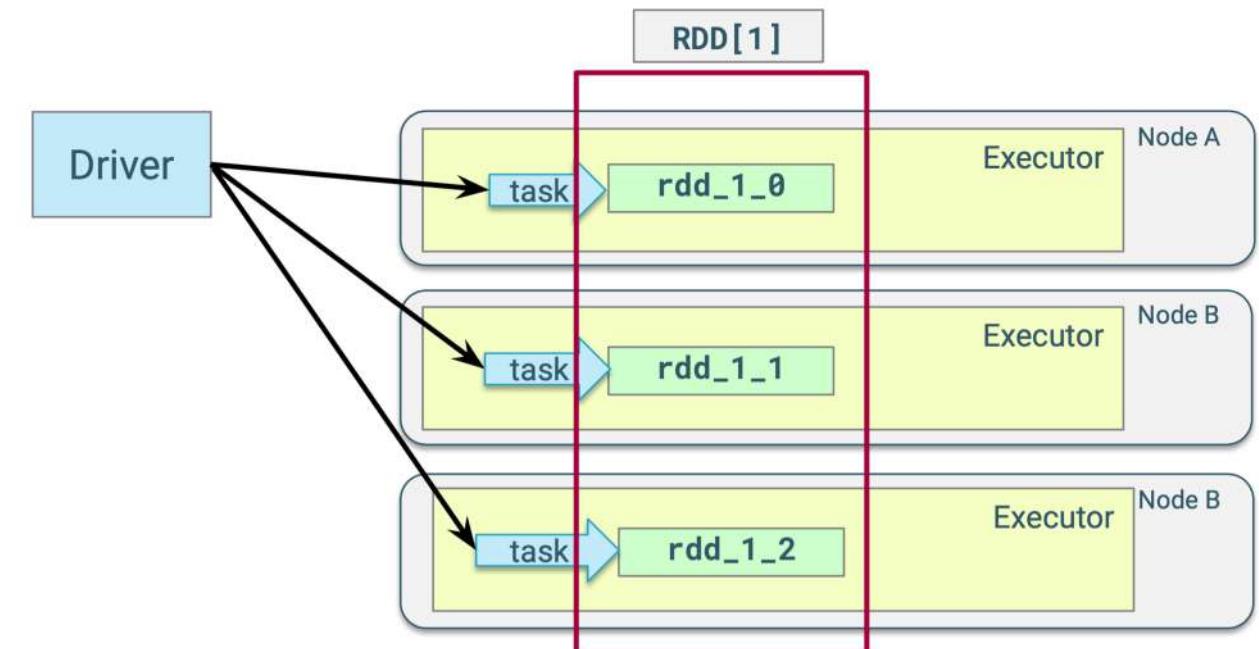
---

## Spark Distributed Processing

- **Spark Distributed Processing**
- **Exercise: Explore Query Execution Order**

# 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
  - More partitions = more parallelism
  - In some cases, you can control how many partitions are created



## 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
- **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

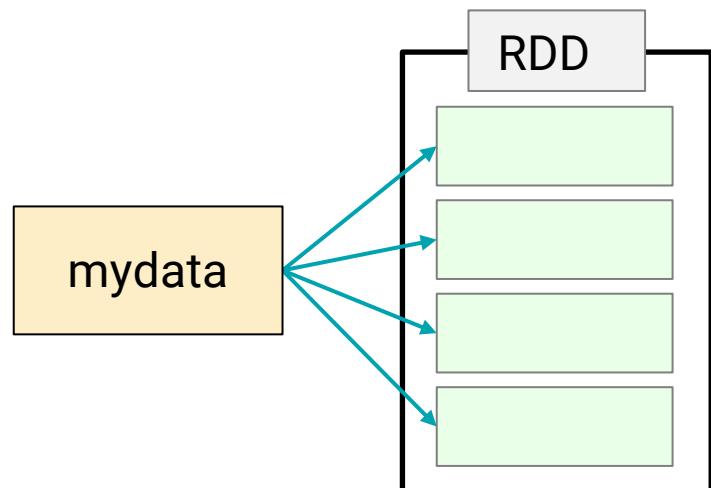
```
> myRDD.getNumPartitions()
```

Language: Python

## Example: Average Word Length by Letter (1)

Language: Python

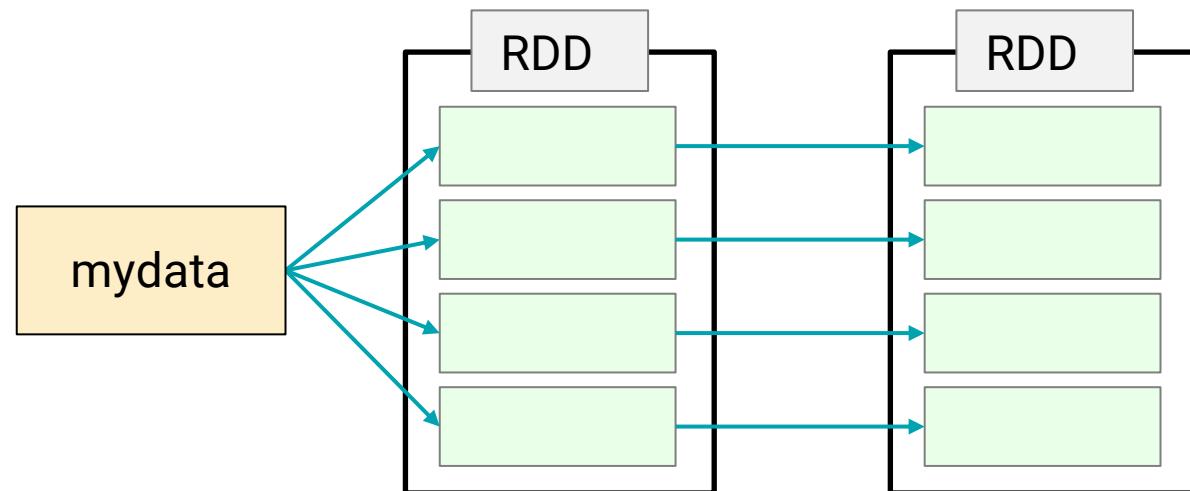
```
> avglens = sc.textFile(file)
```



## Example: Average Word Length by Letter (2)

Language: Python

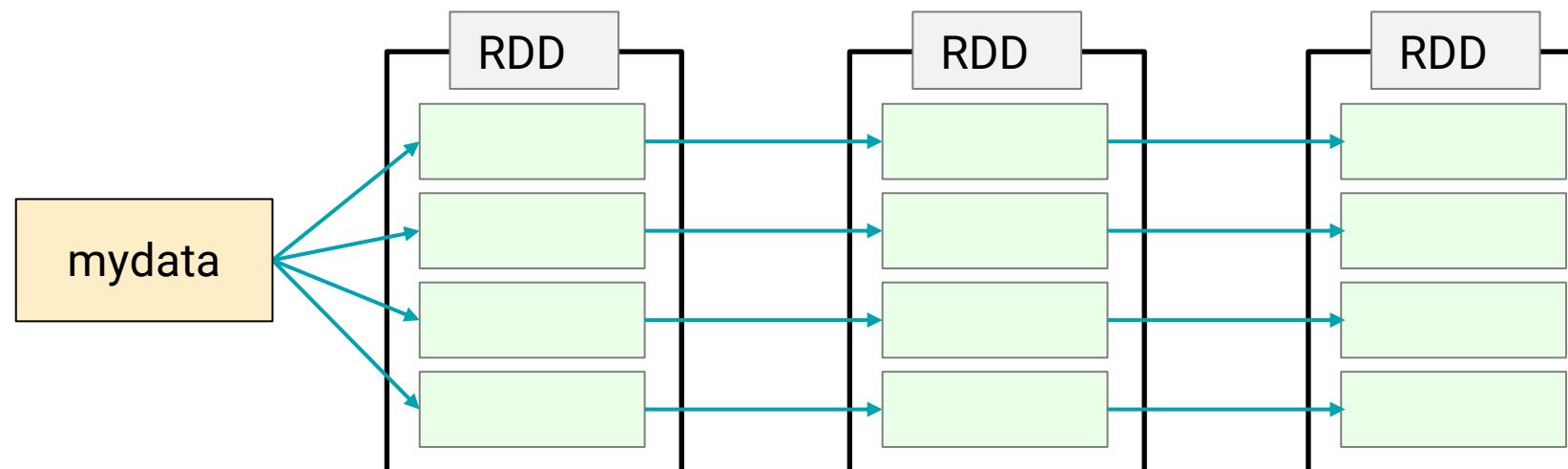
```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split(' '))
```



## Example: Average Word Length by Letter (3)

Language: Python

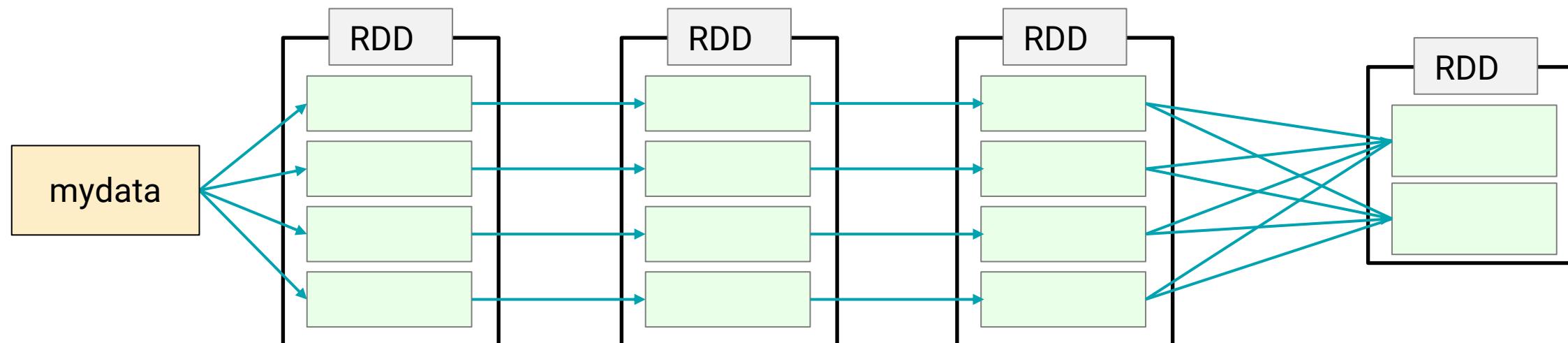
```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split(' ')) \
    .map(lambda word: (word[0],len(word)))
```



## Example: Average Word Length by Letter (4)

Language: Python

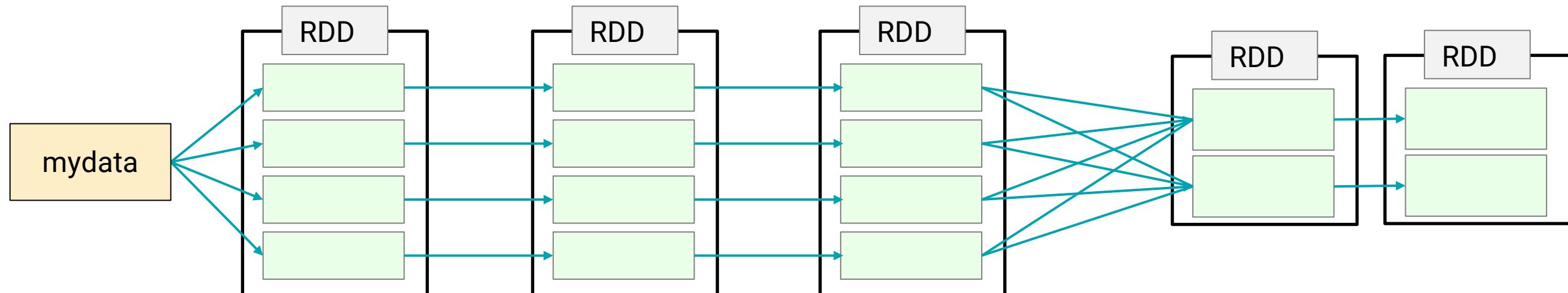
```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split(' ')) \
    .map(lambda word: (word[0],len(word))) \
    .groupByKey(2)
```



# Example: Average Word Length by Letter (5)

Language: Python

```
> avglens = sc.textFile(file) \
    .flatMap(lambda line: line.split(' ')) \
    .map(lambda word: (word[0],len(word))) \
    .groupByKey(2) \
    .map(lambda wordKVP: (wordKVP[0], sum(wordKVP[1])/len(wordKVP[1])))
```



# Stages and Tasks

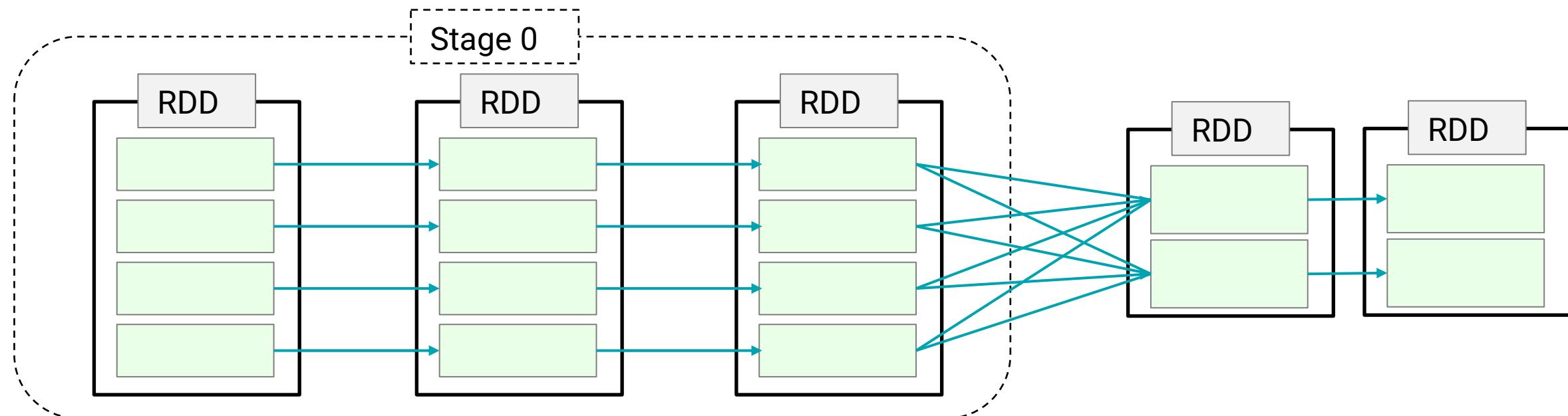
---

- 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

# Spark Execution: Stages (1)

Language: Scala

```
> val avglen = sc.textFile(myfile).  
    flatMap(line => line.split(' ')).  
    map(word => (word(0), word.length)).  
    groupByKey(2).  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```

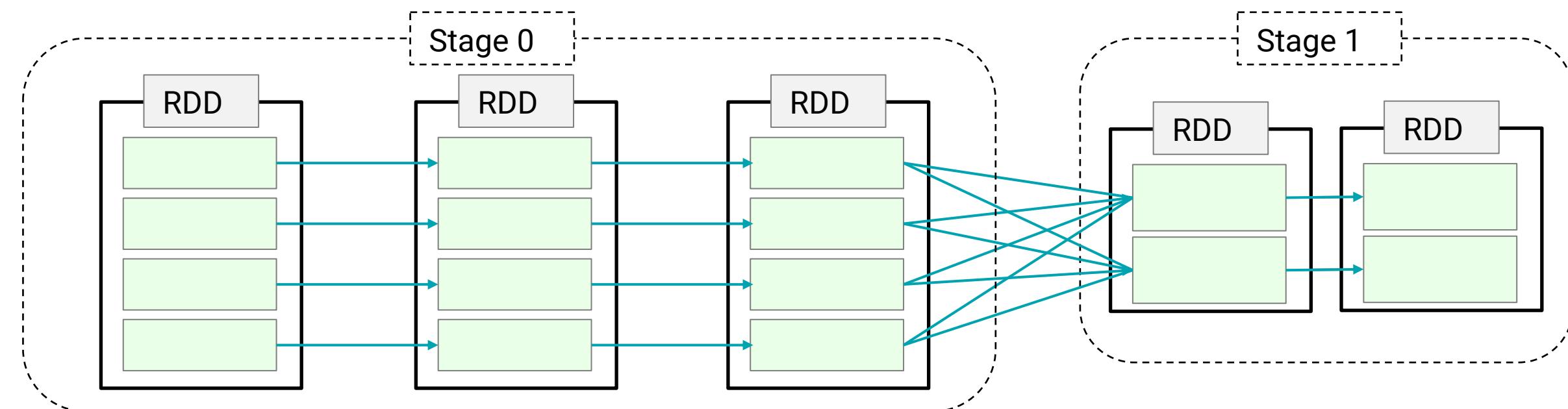


## Spark Execution: Stages (2)

Language: Scala

```
'> val avglen = sc.textFile(myfile).
  flatMap(line => line.split(' ')).
  map(word => (word(0), word.length)).
  groupByKey(2).
  map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))

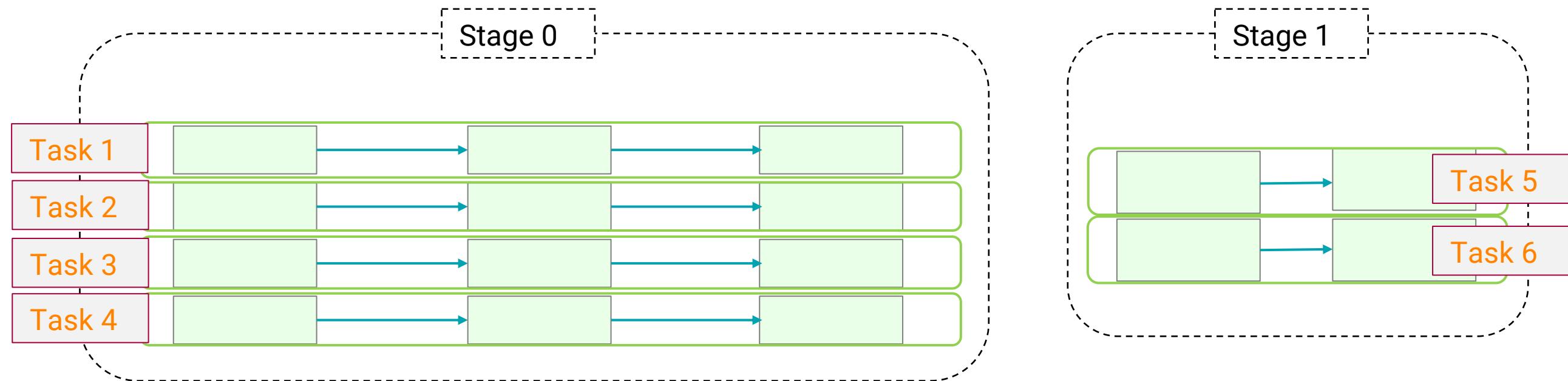
'> avglen.saveAsTextFile("avglen-output")
```



# Spark Execution: Stages (3)

Language: Scala

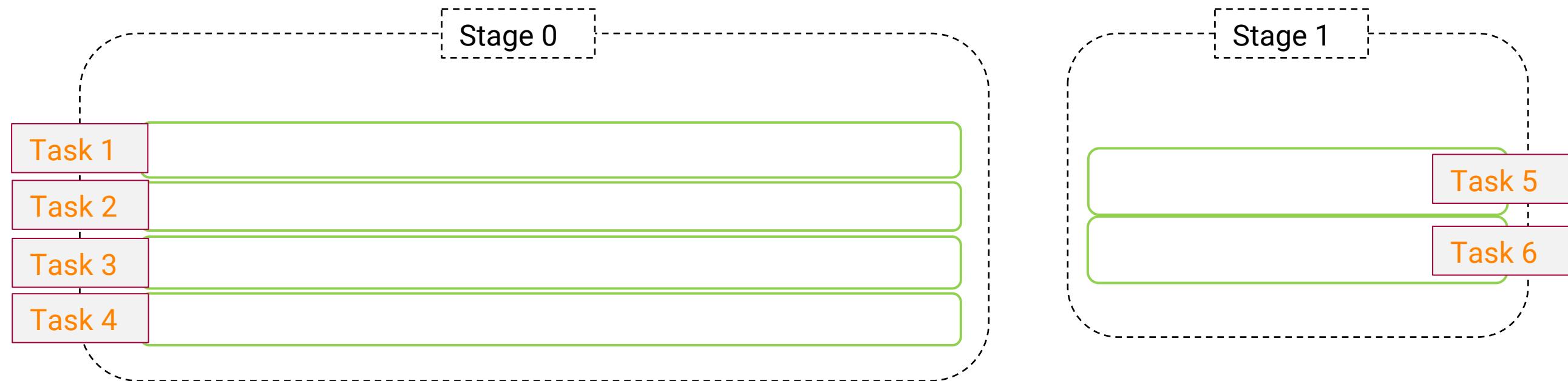
```
> val avglen = sc.textFile(myfile).  
    flatMap(line => line.split(' ')).  
    map(word => (word(0), word.length)).  
    groupByKey(2).  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```



# Spark Execution: Stages (4)

Language: Scala

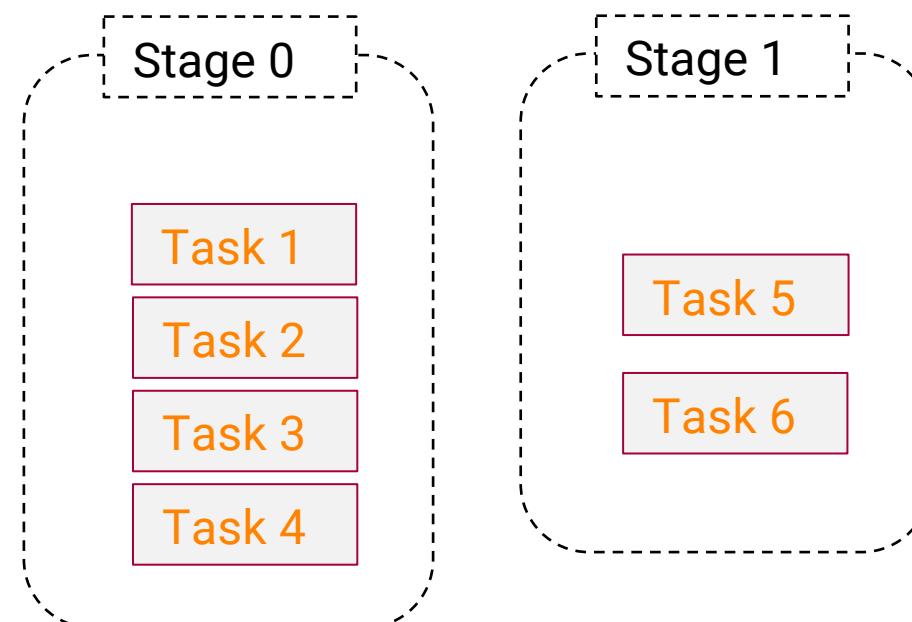
```
> val avglen = sc.textFile(myfile).  
    flatMap(line => line.split(' ')).  
    map(word => (word(0), word.length)).  
    groupByKey(2).  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```



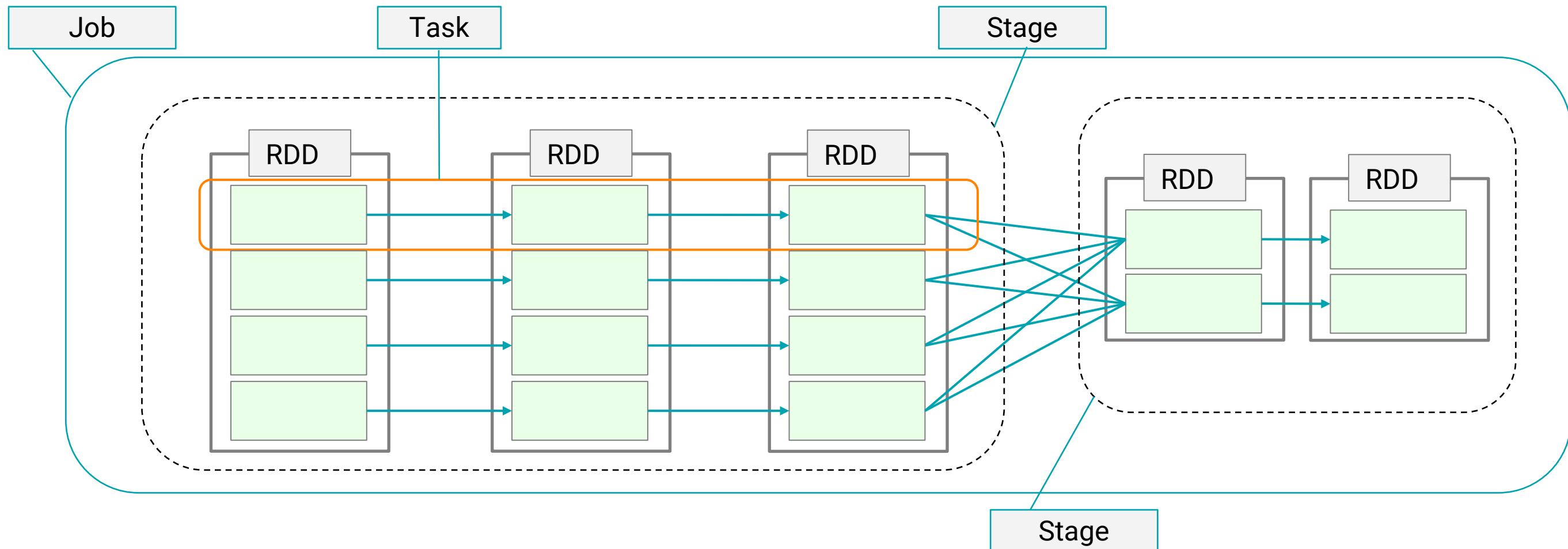
# Spark Execution: Stages (5)

Language: Scala

```
> val avglen = sc.textFile(myfile).  
    flatMap(line => line.split(' ')).  
    map(word => (word(0),word.length)).  
    groupByKey(2).  
    map(pair => (pair._1, pair._2.sum/pair._2.size.toDouble))  
  
> avglen.saveAsTextFile("avglen-output")
```



# Summary of Spark Terminology



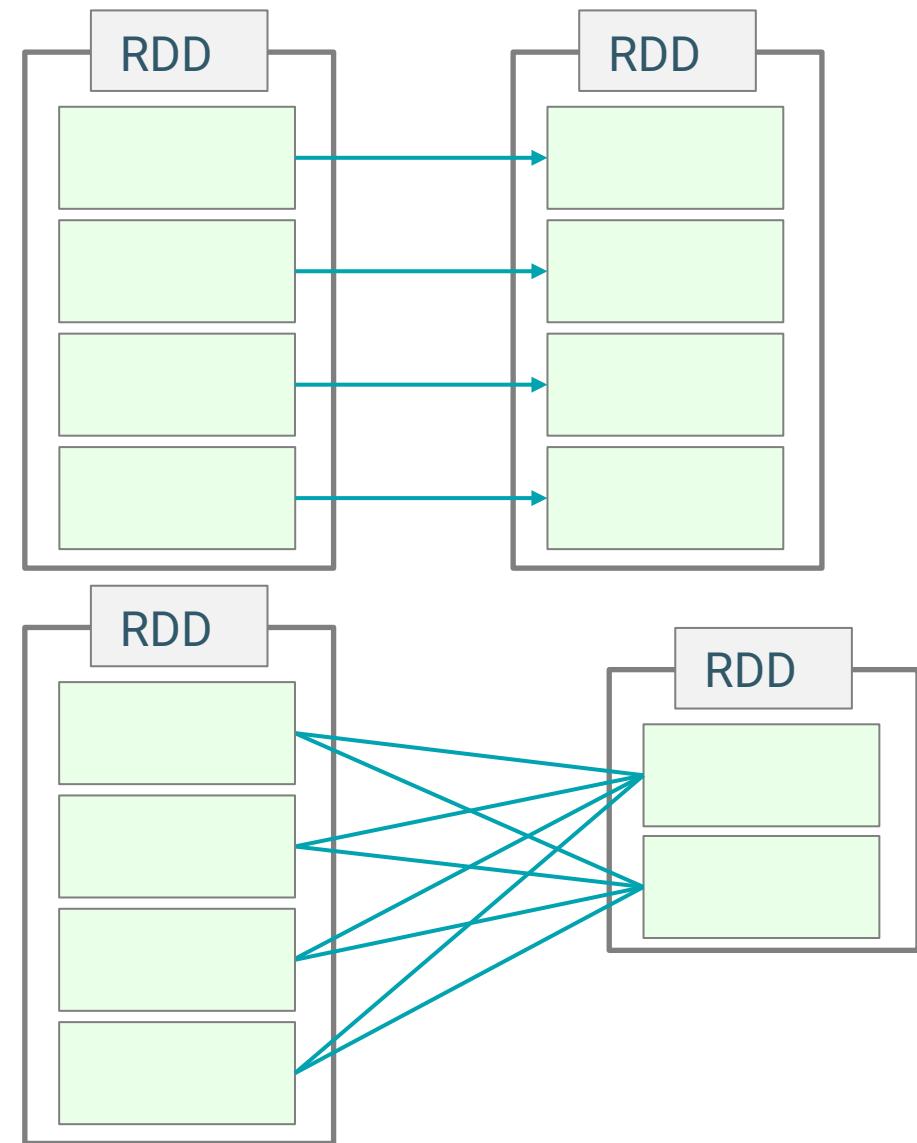
# Execution Plans

---

- **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

- 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



# Catalyst Optimizer

---

- **Catalyst improves 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**
- **Note:** Catalyst relies on the transformations names (select, filter,...) to infer their logic and apply optimizations.  
This system is broken if you introduce lambda functions with unknown behaviours in your DataFrames processing.

# 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

# 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

== 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
```

Language: Python

## Example: Catalyst Execution Plan (2)

```
== 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

## 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)

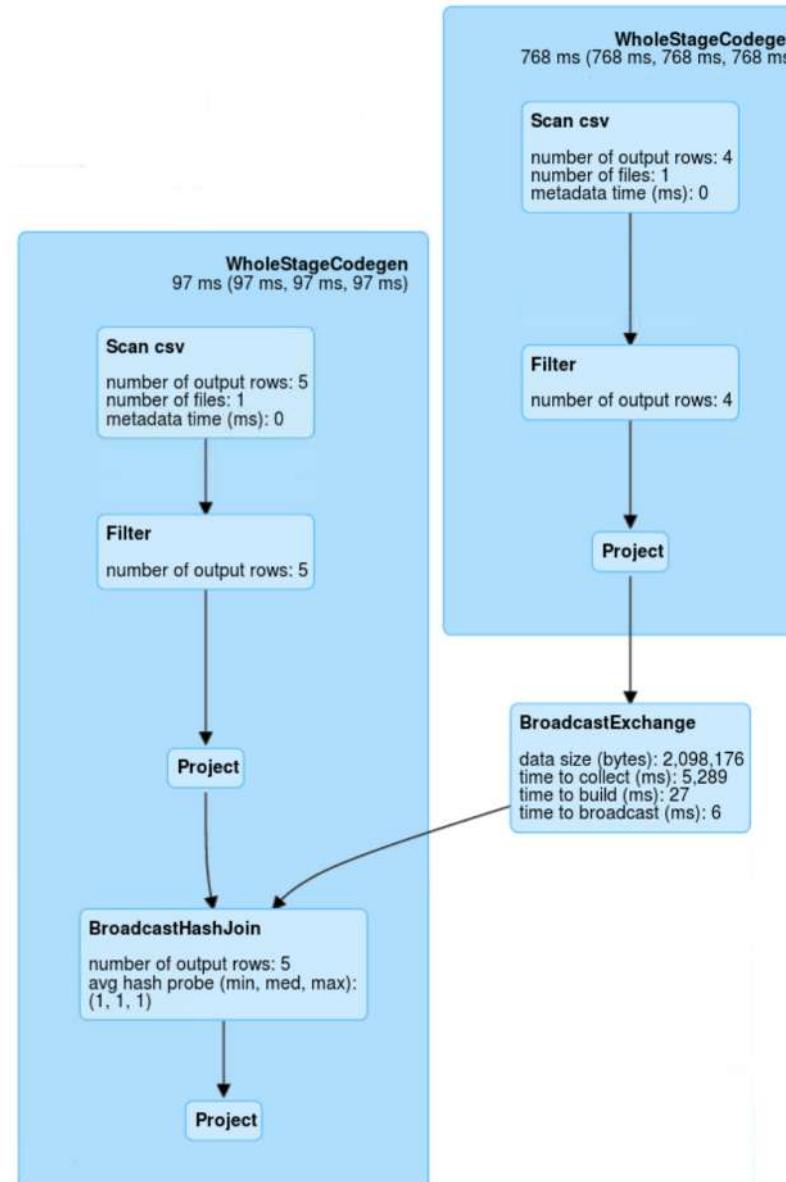
### SQL

Completed Queries: 3

#### Completed Queries (3)

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

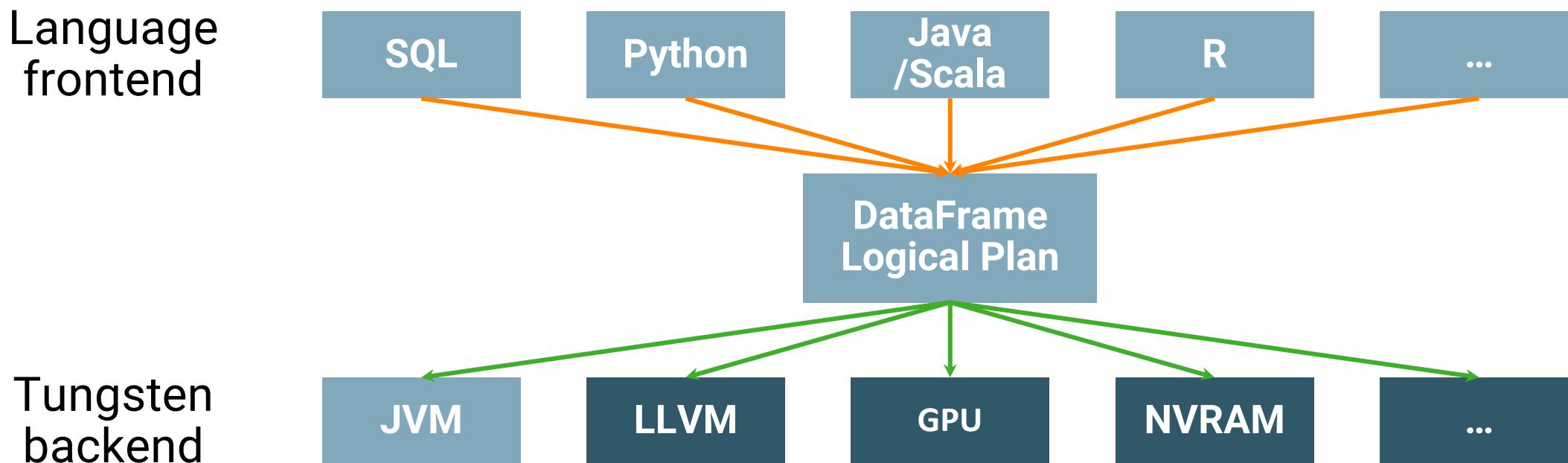
## Example: Catalyst Execution Plan (5)



# Tungsten Overview

- Improve Spark execution by optimizing CPU / memory usage

- Understands and optimizes for hardware architectures
- Tunes optimizations for Spark's characteristics



# Tungsten's Binary Format

---

- **Binary representation of Java objects (Tungsten row format)**
  - Different from Java serialization and Kryo
- **Advantages include:**
  - Much smaller size than native Java serialization
  - Supports off-heap allocation, **default is on-heap**
  - Structure supports Spark operations **without deserialization**
    - e.g. You can sort data while it remains in the binary format
  - Avoids GC overhead if using off-heap
- **Result:**
  - Much faster, less memory, less CPU
  - Can process much larger datasets

## Knowledge Check

---

- 1. True or False, The more partitions the better.**
- 2. True or False, a Job and an Application are the same thing.**
- 3. What are the names of the two optimizers through which DataFrame processing go?**
- 4. True or False, it's cool to use Lambda functions to process DataFrames.**
- 5. True or False, you can monitor the execution of your RDD based code in the SQL tab of the Spark UI.**
- 6. True or False, RDD based code can be difficult to read but is usually more efficient**
- 7. Scala based DataFrame processing is 4 times faster than its Pyspark equivalent.**

## 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

- Spark Distributed Processing
- **Exercise: Explore Query Execution Order**



# Spark Distributed Persistence

---

Chapter 13

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence**
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

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

---

## Spark Distributed Persistence

- **DataFrame and Dataset Persistence**
- Persistence Storage Levels
- Viewing Persisted RDDs
- Exercise: Persisting DataFrames

# Persistence

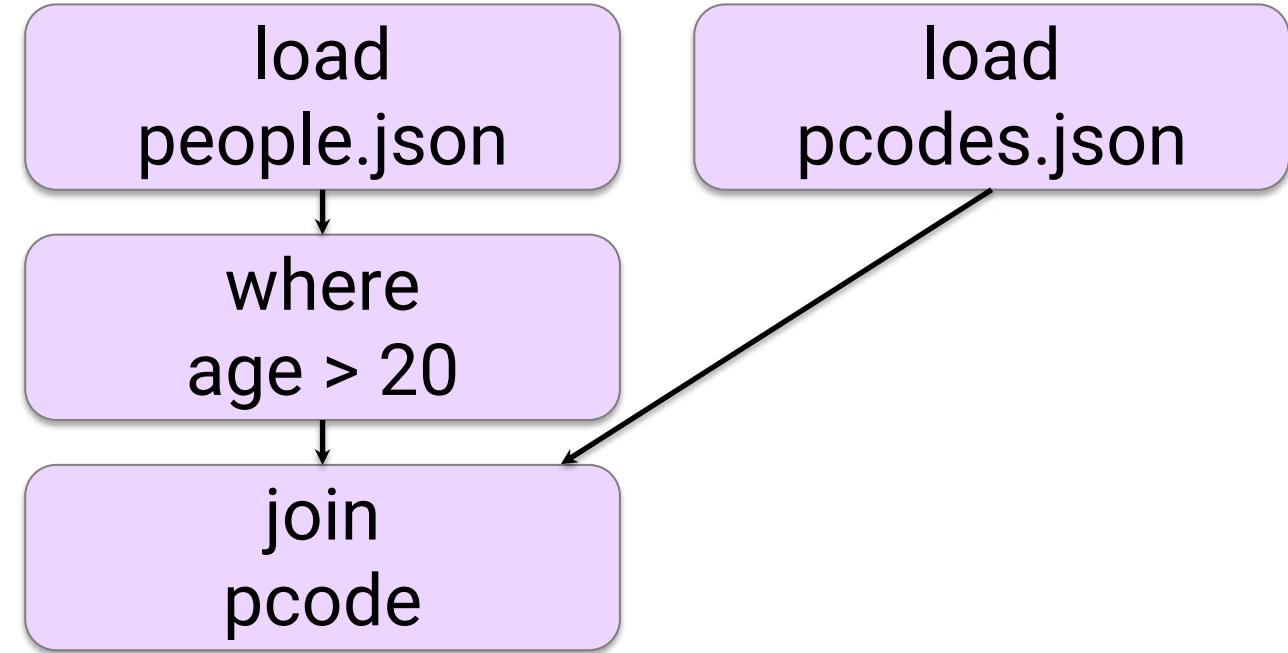
---

- 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.\n    json("people.json").\n    where("age > 20")\n\npcodesDF = spark.read.\n    json("pcodes.json").\n    persist()\n\njoinedDF = over20DF.\n    where("PCODE = 94020").\n    count()
```

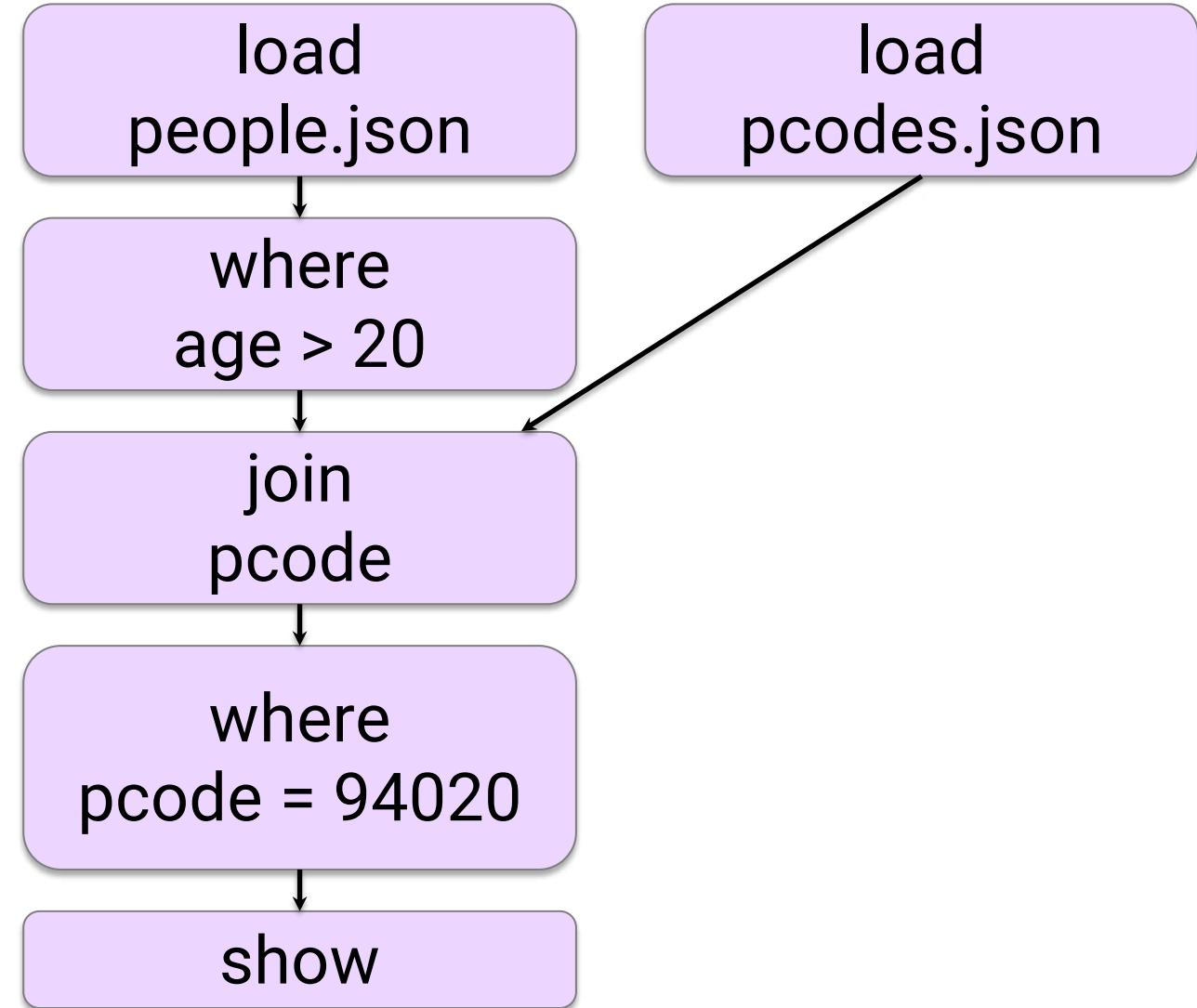
Language: Python



## Example: DataFrame Persistence (2)

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

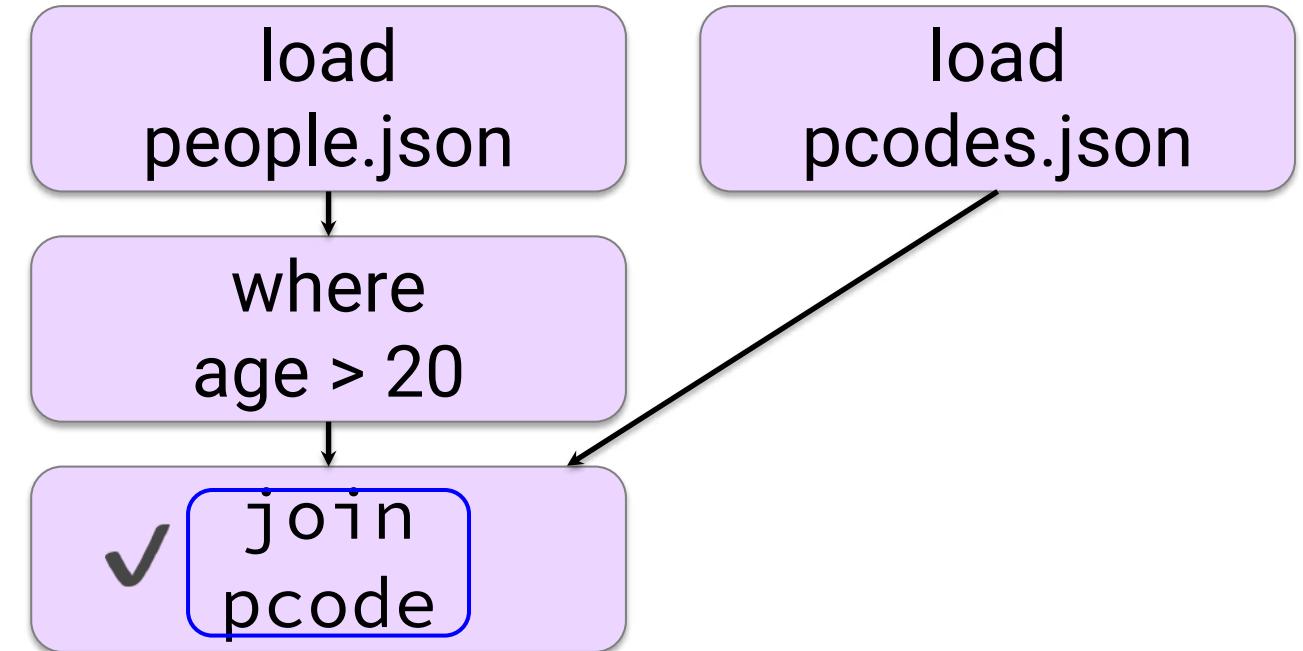
Language: Python



## Example: DataFrame Persistence (3)

```
> over20DF = spark.read.\n    json("people.json").\n    where("age > 20")\n\npcodesDF = spark.read.\n    json("pcodes.json")\n\njoinedDF = over20DF.\n    where("pcode = 94020").\n    persist()
```

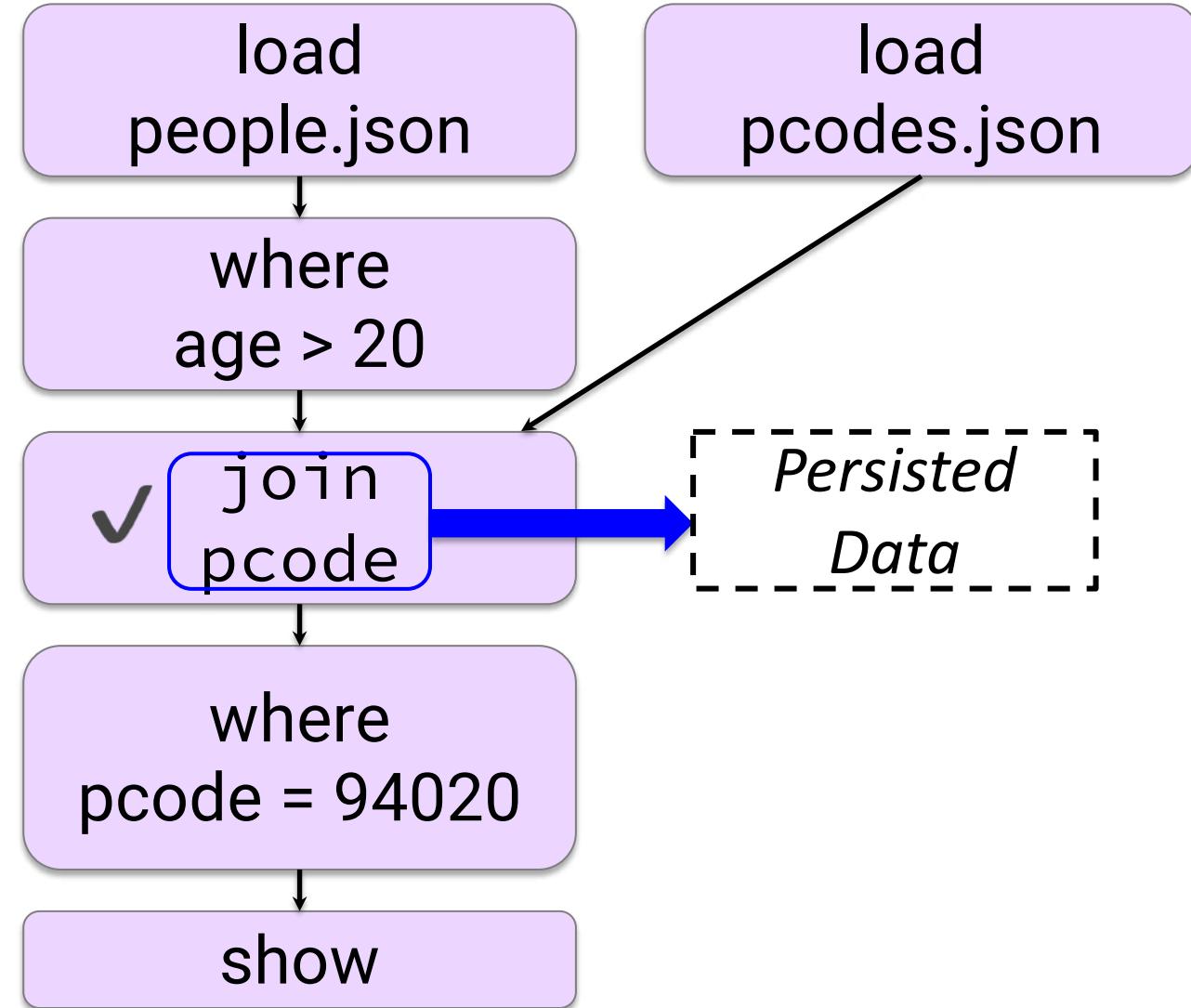
Language: *Python*



## Example: DataFrame Persistence (4)

```
> over20DF = spark.read.\n    json("people.json").\n    where("age > 20")\n\npcodesDF = spark.read.\n    json("pcodes.json")\n\njoinedDF = over20DF.\n    where("PCODE = 94020").\n    persist()\n\njoinedDF = over20DF.\n    where("PCODE = 94020").\n    show()
```

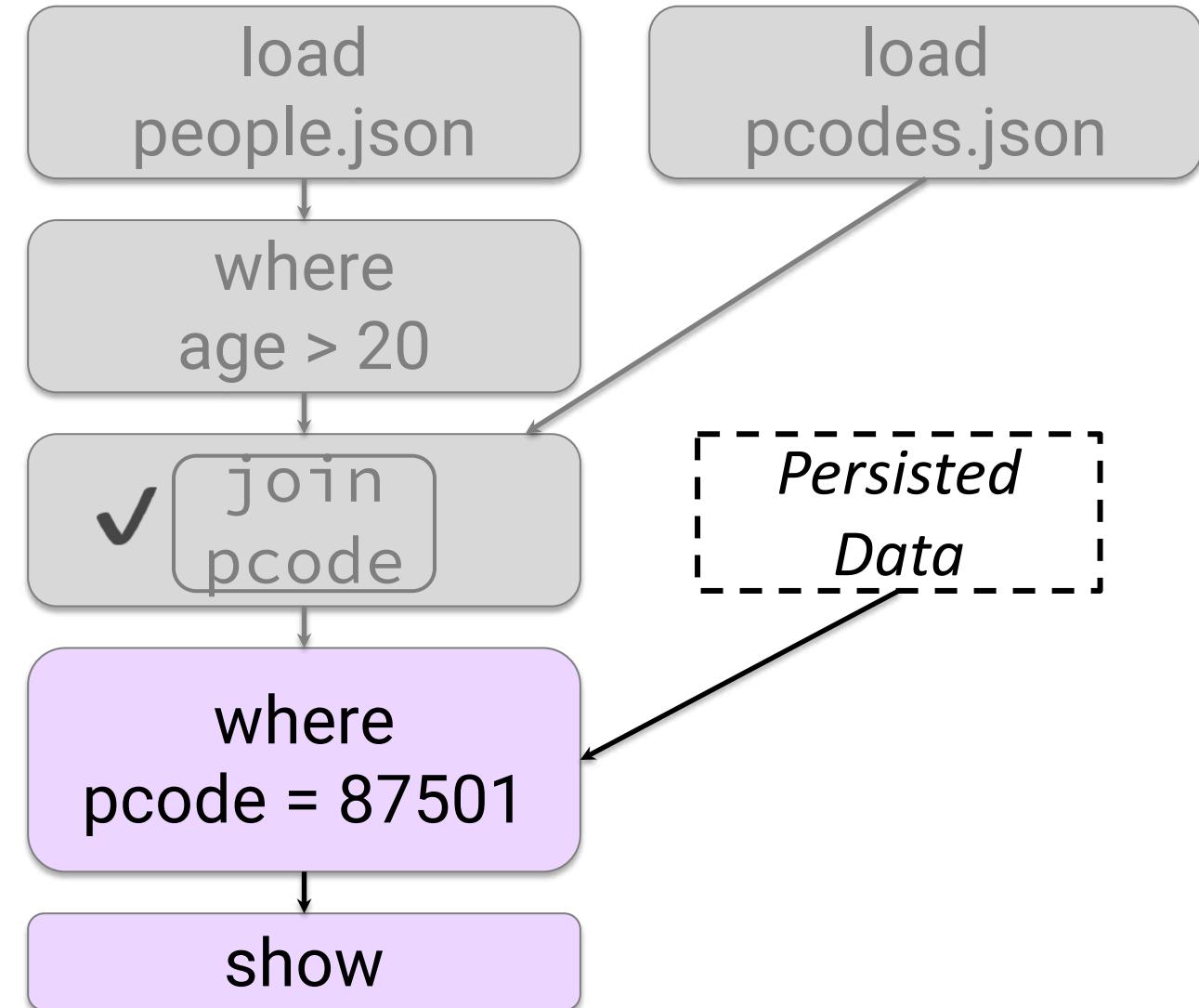
Language: Python



## Example: DataFrame Persistence (5)

```
> over20DF = spark.read.\n    json("people.json").\n    where("age > 20")\n\npcodesDF = spark.read.\n    json("pcodes.json")\n\njoinedDF = over20DF.\n    where("PCODE = 94020").\n    persist()\n\njoinedDF = over20DF.\n    where("PCODE = 94020").\n    show()\n\njoinedDF.\n    where("PCODE = 87501").\n    count()
```

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

---

## Spark Distributed Persistence

- DataFrame and Dataset Persistence
- **Persistence Storage Levels**
- Viewing Persisted RDDs
- 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

---

## Spark Distributed Persistence

- DataFrame and Dataset Persistence
- Persistence Storage Levels
- **Viewing Persisted RDDs**
- 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
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

## 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**
- **Persistence is most useful in iterative applications or when executing a complicated query is very expensive**

# Chapter Topics

---

## Spark Distributed Persistence

- DataFrame and Dataset Persistence
- Persistence Storage Levels
- Viewing Persisted RDDs
- **Exercise: Persisting DataFrames**



# Writing, Configuring and Running Spark Applications

---

## Chapter 14

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- **Writing, Configuring and Running Spark Applications**
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# 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
- Exercise: Writing, Configuring, and Running a Spark Application

# Spark Interpreters and Spark Applications

---

- **Spark interpreters allow interactive exploration and manipulation of data**
  - REPL using Python or Scala
  - Python, Scala or R notebooks
- **Spark applications run as independent programs**
  - For jobs such as ETL processing, streaming, and so on
  - Python, Scala, R, 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
- Interpreters create 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 `SparkContext` 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: 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
- Exercise: Writing, Configuring, and Running a Spark Application

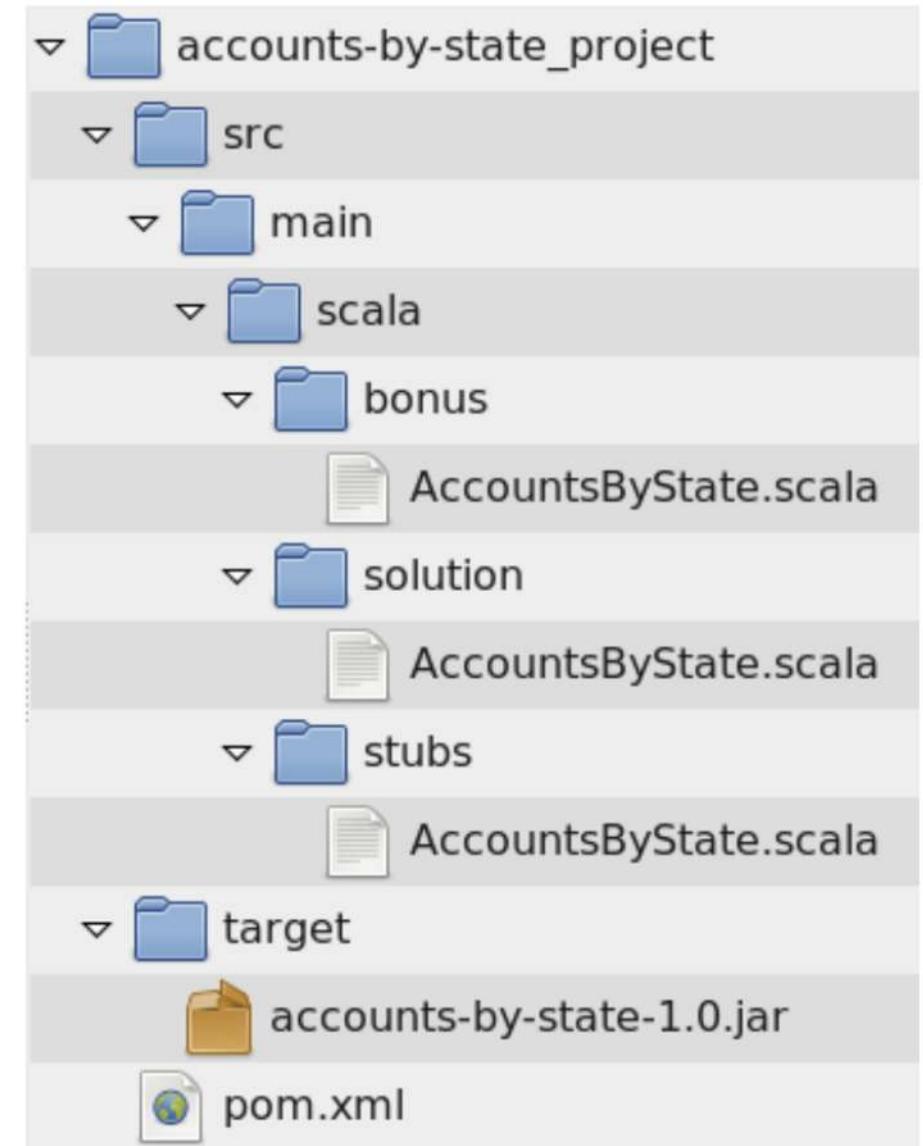
# Building an Application: Scala or Java

---

- **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, Kubernetes, 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
- Exercise: Writing, Configuring, and Running a Spark Application

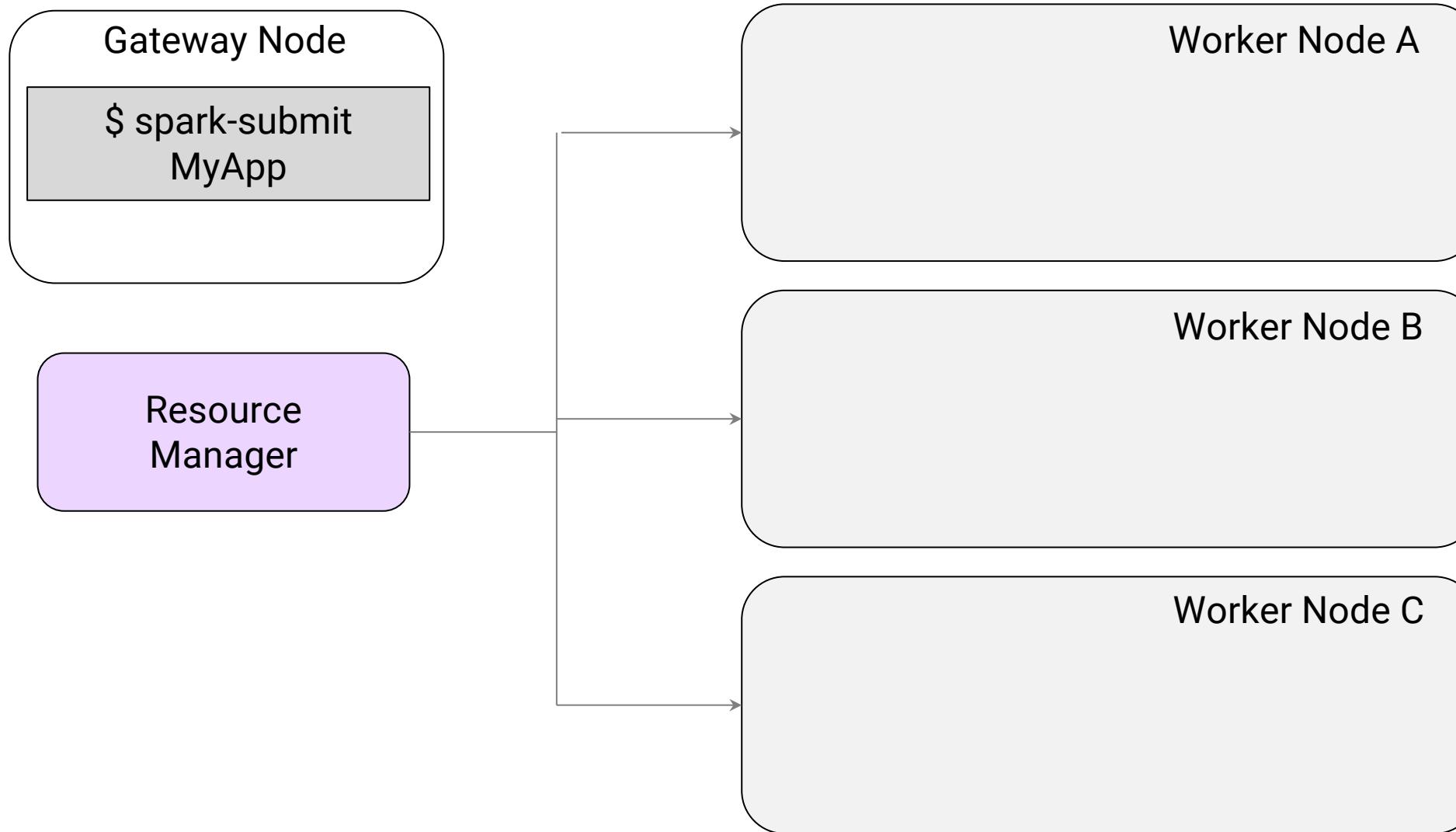
# Application Deployment Mode

---

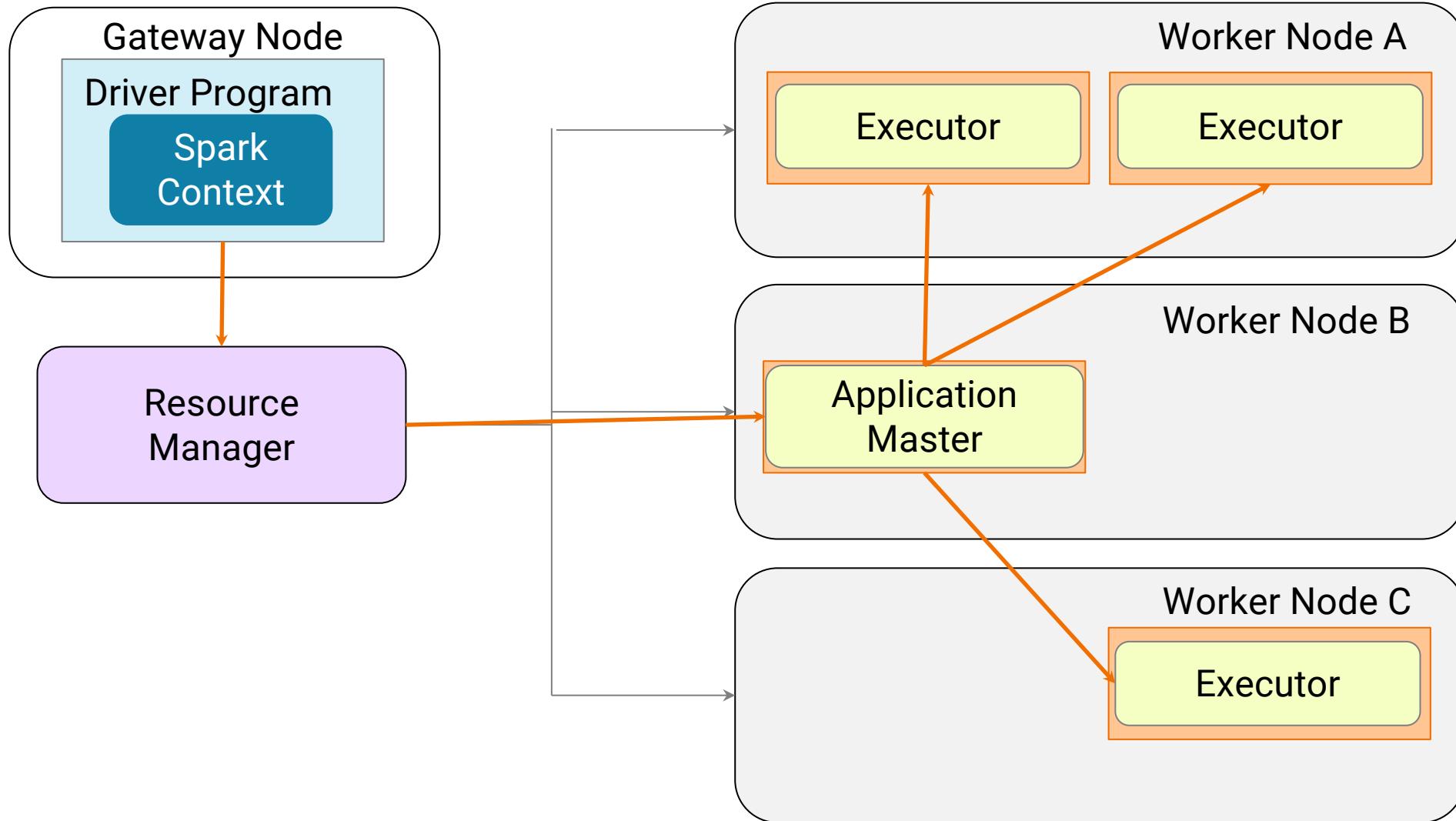
- **Spark applications can run**
  - Locally with one or more threads
  - On a cluster
- **In client mode (default), the driver runs locally on a gateway node**
  - Requires direct communication between driver and cluster worker nodes
- **In cluster mode, the driver runs in the application master on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

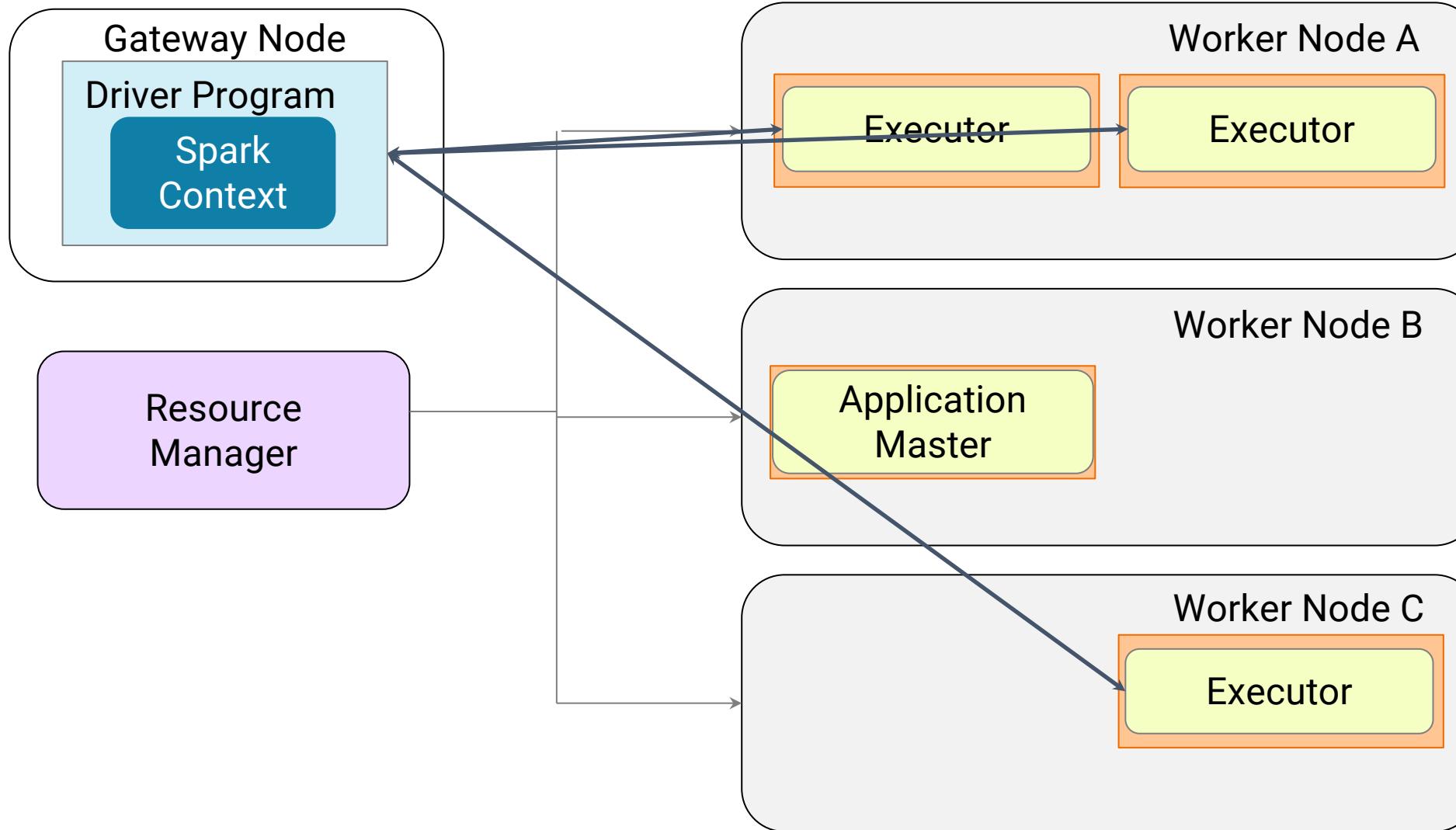
# Spark Deployment Mode on YARN: Client Mode (1)



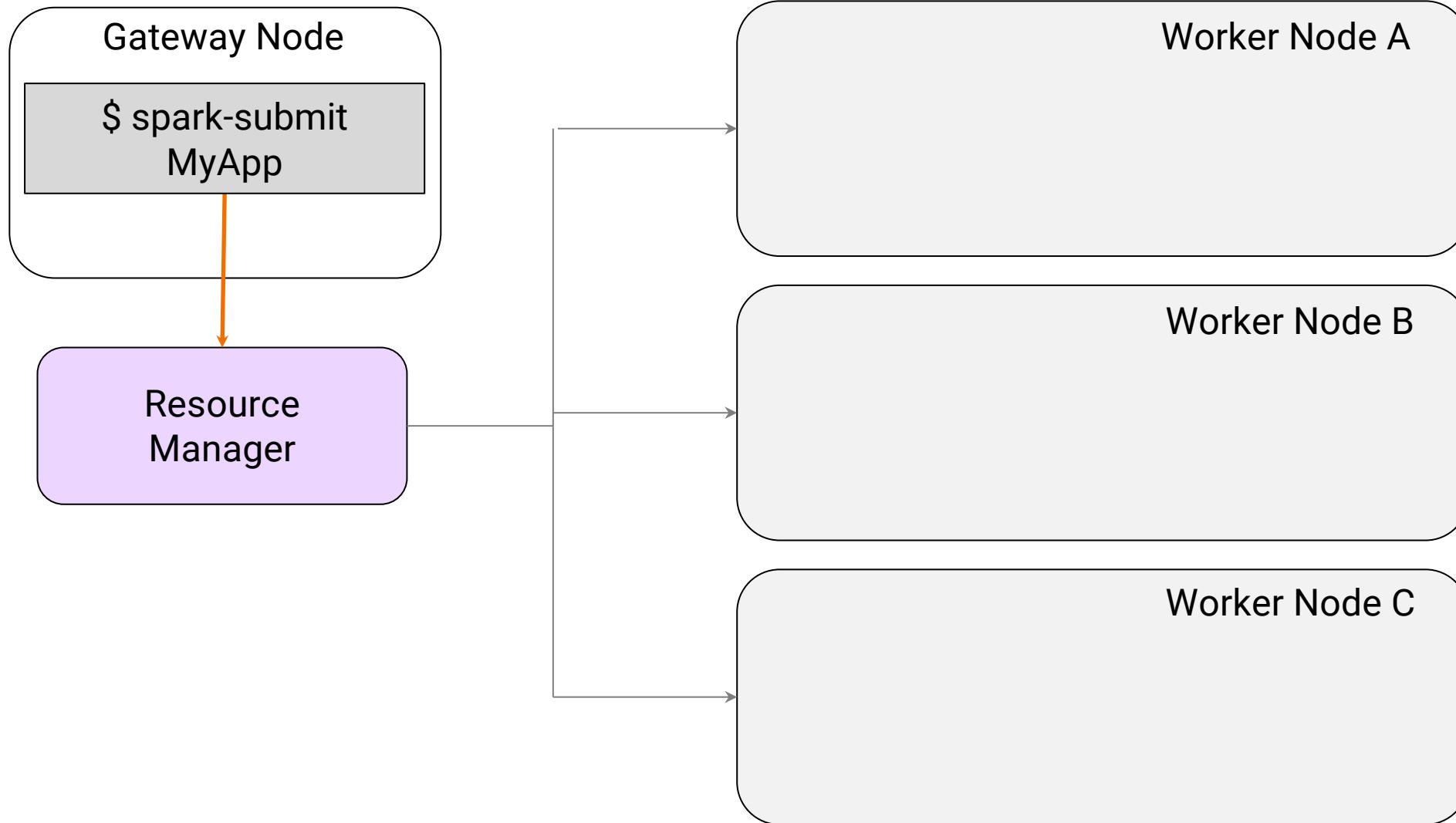
# Spark Deployment Mode on YARN: Client Mode (2)



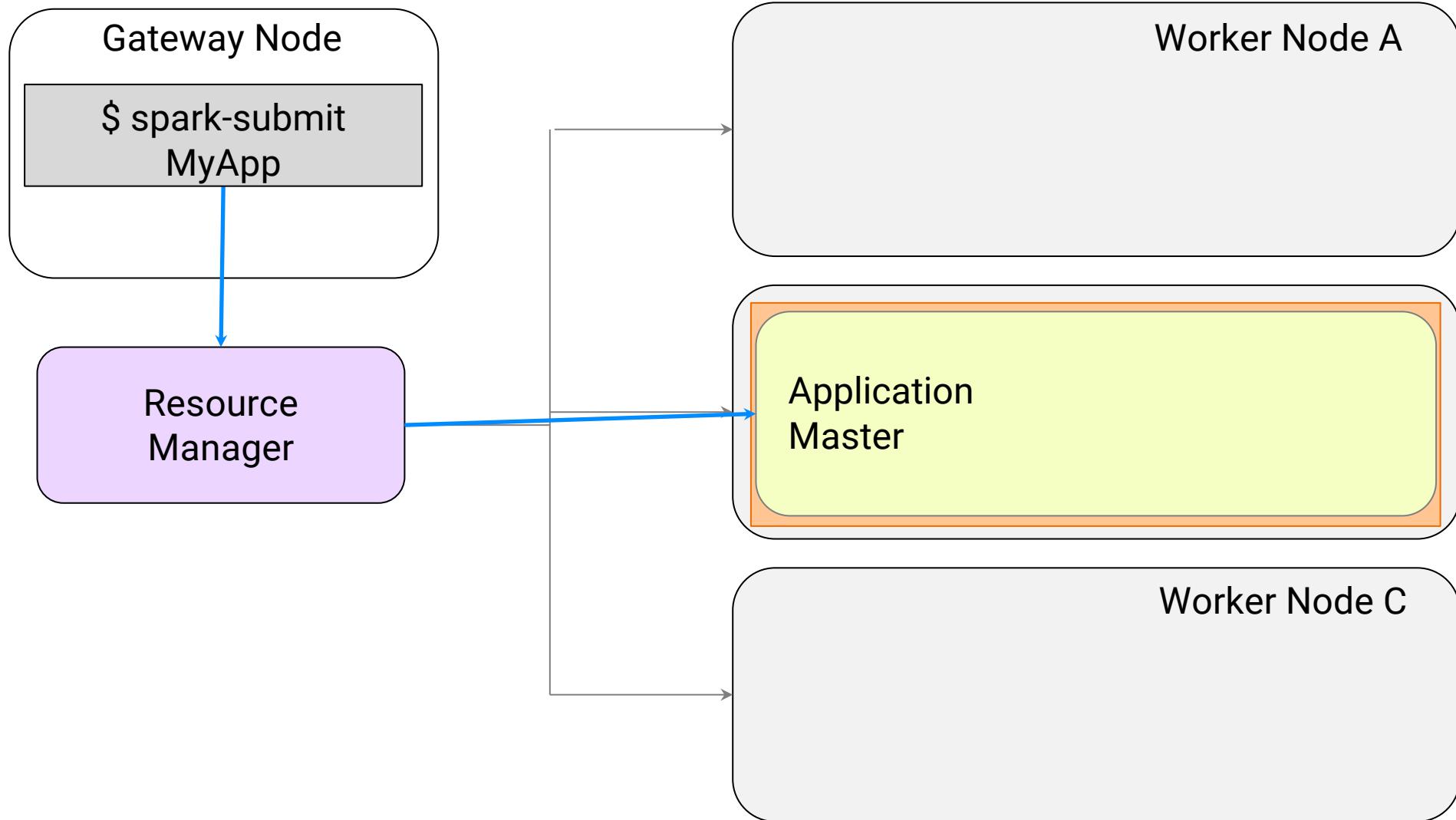
# Spark Deployment Mode on YARN: Client Mode (3)



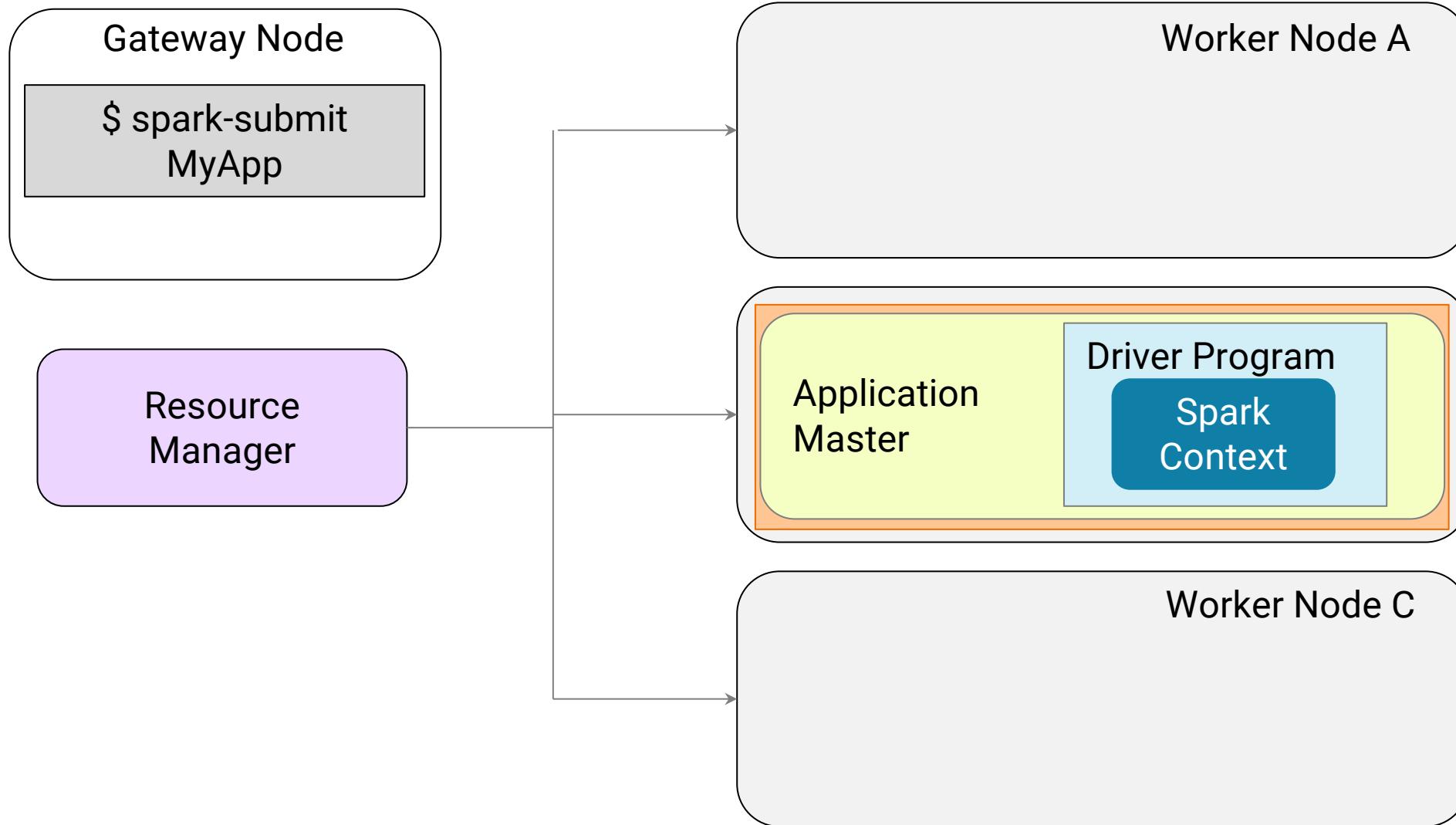
# Spark Deployment Mode on YARN: Cluster Mode (1)



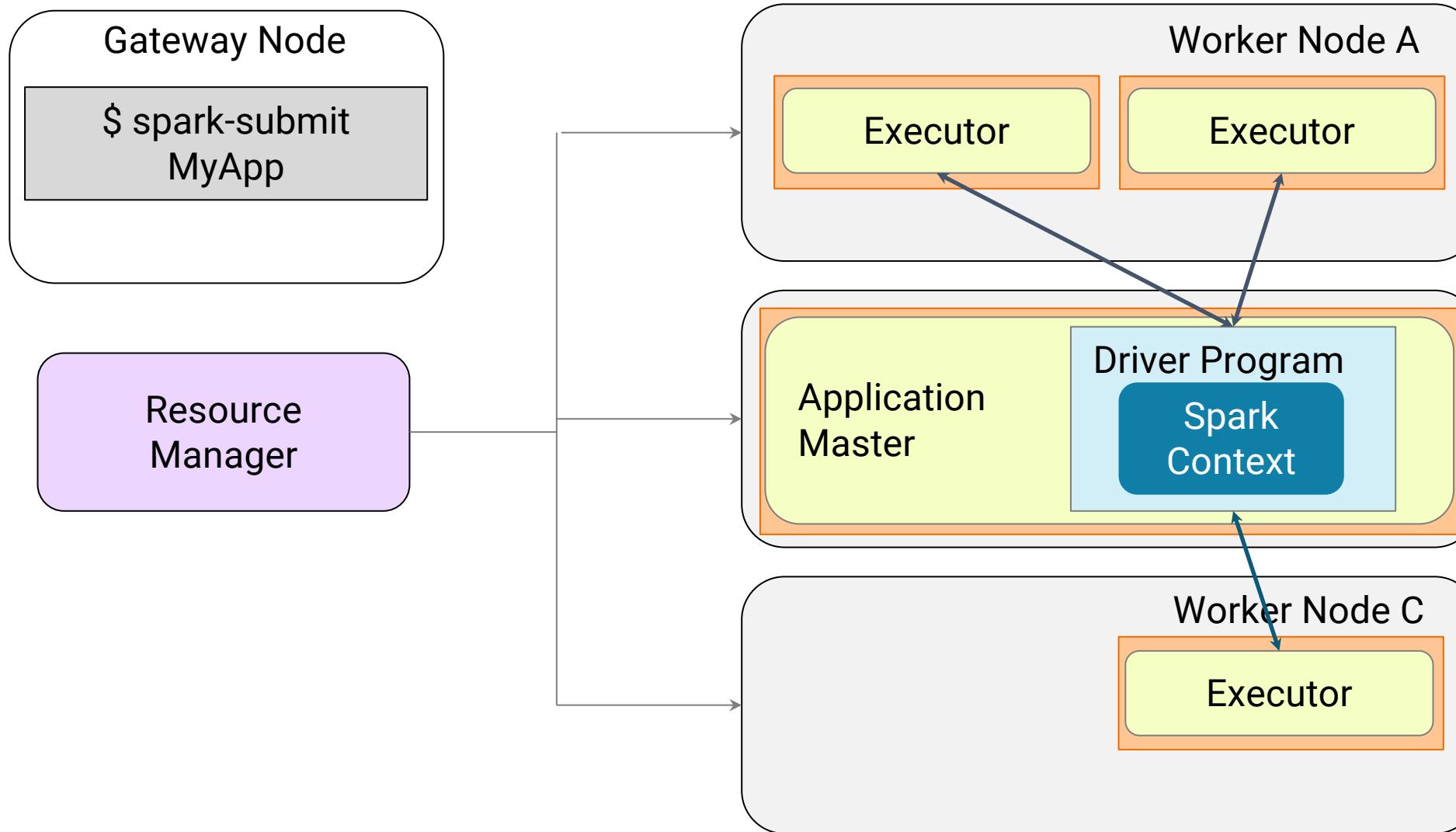
# Spark Deployment Mode on YARN: Cluster Mode (2)



# 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
- Exercise: Writing, Configuring, and Running a Spark Application

# The Spark UI Application Web UI

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

The screenshot displays two main sections of the Apache Spark Web UI:

- Jobs Tab:** Shows the "Spark Jobs" section with the following details:
  - User: training
  - Total Uptime: 50 min
  - Scheduling Mode: FIFO
  - Completed Jobs: 4
- Executors Tab:** Shows the "Executors" summary table and a detailed table of executor metrics.

**Summary Table Headers:**

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
------------	----------------	-----------	-------	--------------	--------------	----------------	-------------	---------------------	-------	--------------	---------------	-------------

**Summary Data:**

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	179.6 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

**Executor Metrics Table Headers:**

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
-------------	---------	--------	------------	----------------	-----------	-------	--------------	--------------	----------------	-------------	---------------------	-------	--------------	---------------	------	-------------

**Executor Metrics Data:**

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 stderr	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 stderr	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 stderr	Thread Dump

**Completed Jobs Table Headers:**

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
--------	-------------	-----------	----------	-------------------------	---

**Completed Jobs Data:**

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

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	<input type="button" value="History"/>	<a href="#">History</a>	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	<input type="button" value="History"/>	<a href="#">History</a>	0
N/A	RUNNING	UNDEFINED	1	1	1024	0	0	25.0	25.0	<input type="button" value="ApplicationMaster"/>	<a href="#">ApplicationMaster</a>	0

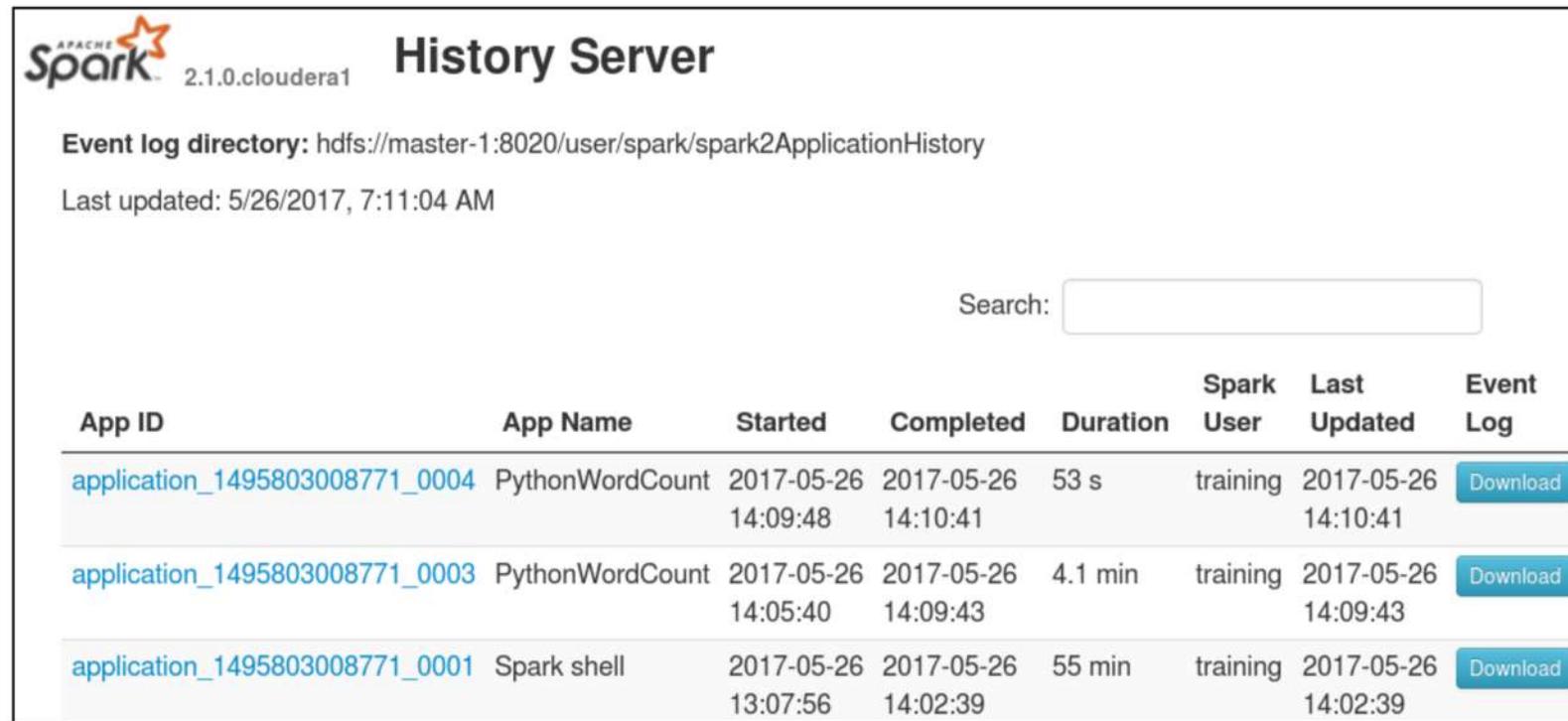
# Spark Application History UI

- 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
Thu May 2 07:53:47 -0700 2019	FINISHED	SUCCEEDED	N/A	N/A	N/A	N/A	N/A	0.0	0.0	<a href="#">History</a>	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	<a href="#">History</a>	0
N/A	RUNNING	UNDEFINED	1	1	1024	0	0	25.0	25.0	<a href="#">ApplicationMaster</a>	0

# Viewing the Applications 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, it displays the Apache Spark logo (2.1.0.cloudera1) and the title "History Server". Below this, it shows the event log directory: "Event log directory: hdfs://master-1:8020/user/spark/spark2ApplicationHistory" and the last update time: "Last updated: 5/26/2017, 7:11:04 AM". A search bar is present above the table. The table lists three applications with columns: App ID, App Name, Started, Completed, Duration, Spark User, Last Updated, and Event Log. Each row includes a "Download" button. The applications listed are:

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	<a href="#">Download</a>
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	<a href="#">Download</a>
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	<a href="#">Download</a>

# 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**
- 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
- **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 is a navigation bar with tabs: Jobs, Stages, Storage, Environment (which is currently selected), Executors, SQL, PySparkShell, and application UI. Below the navigation bar, the page title is "Environment". Under "Runtime Information", there is a table with columns "Name" and "Value". The table contains three rows: Java Home (/usr/java/jdk1.7.0\_67-cloudera/jre), Java Version (1.7.0\_67 (Oracle Corporation)), and Scala Version (version 2.11.8). Under "Spark Properties", there is another table with columns "Name" and "Value". This table contains eight rows: 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), and spark.driver.port (50884).

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

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

## Essential Points

---

- Use the Spark interpreters 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
- **Exercise: Writing, Configuring, and Running a Spark Application**



# Introduction to Structured Streaming

---

Chapter 15

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- **Introduction to Structured Streaming**
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# 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

- **Introduction to Structured Streaming**
- **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**
  - 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
- **DStreams API**
  - RDD-based
  - Lower level API
  - Best for unstructured data

# Why Structured Streaming?

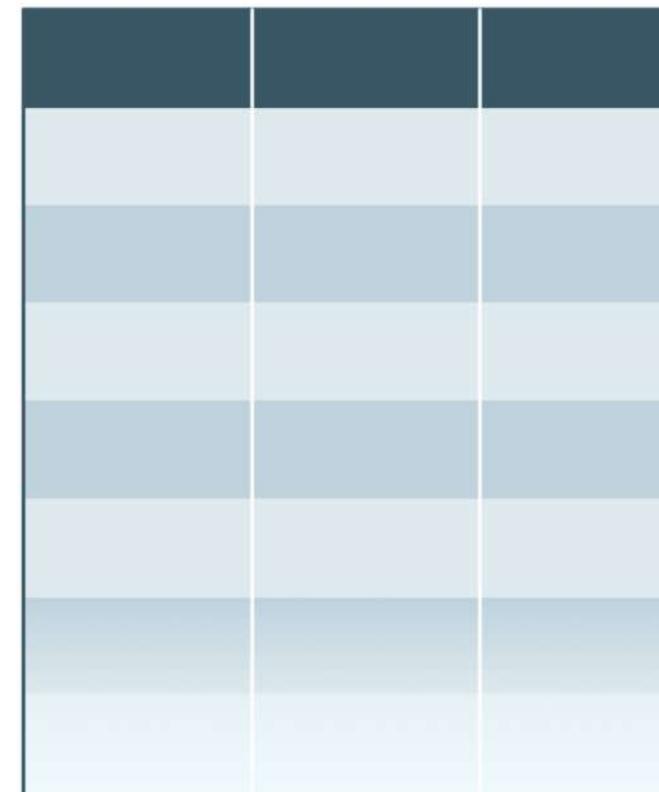
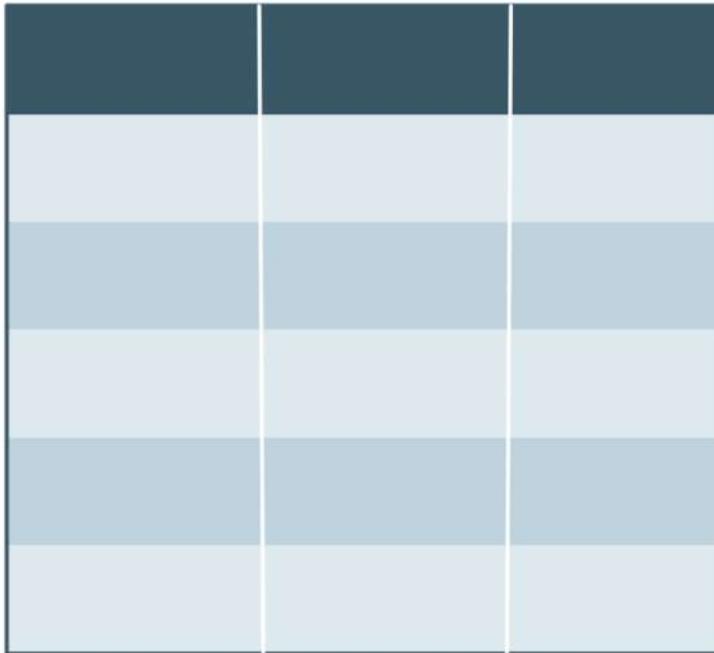
---

- **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

---

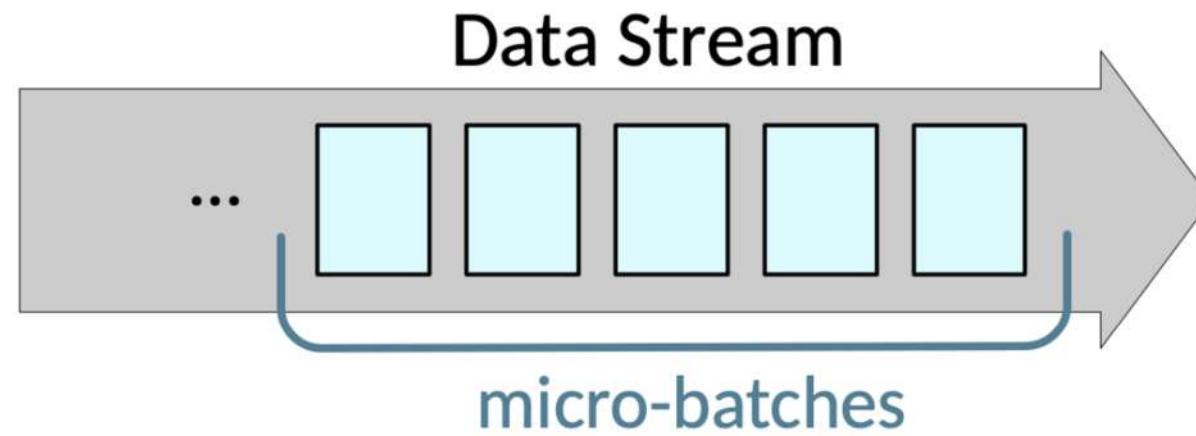
- Regular DataFrames model data as a **bounded table**
  - Data is immutable
- Streaming DataFrames model data as an **unbounded table**
  - Data grows over time



# Micro-batches

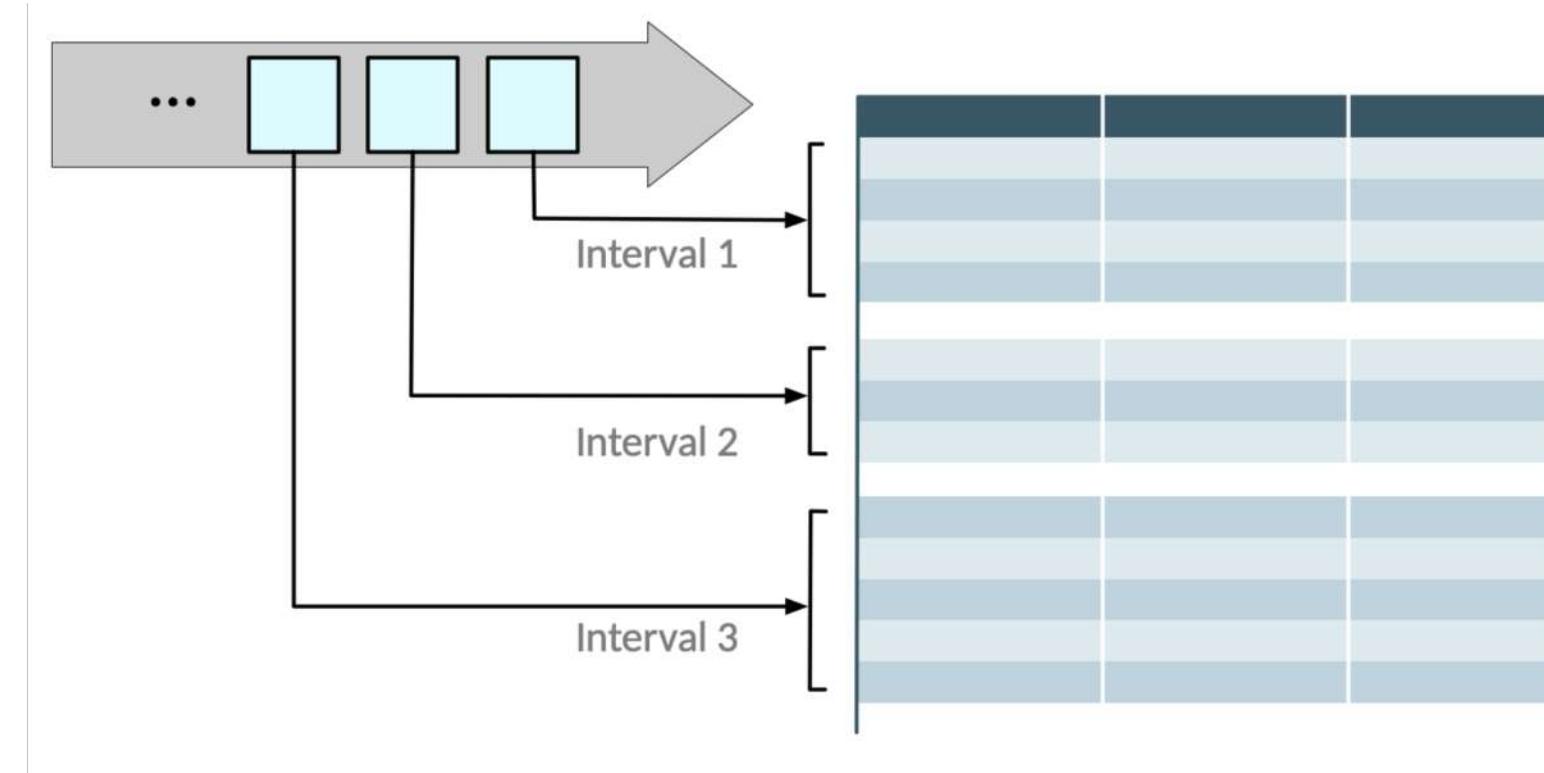
---

- 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 Applications

---

- Typical steps in writing a structured streaming application
  - a. Create a streaming DataFrame from a streaming data source
  - b. Define a query including one or more transformations on the DataFrame
  - c. 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

# 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 Sources 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

# 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

```
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")). \
...
statusFilterDF = statusDF. \
    select("dev_id","dev_temp","signal"). \
    where("model='Sorrento F41L'")
```

# 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")...
```

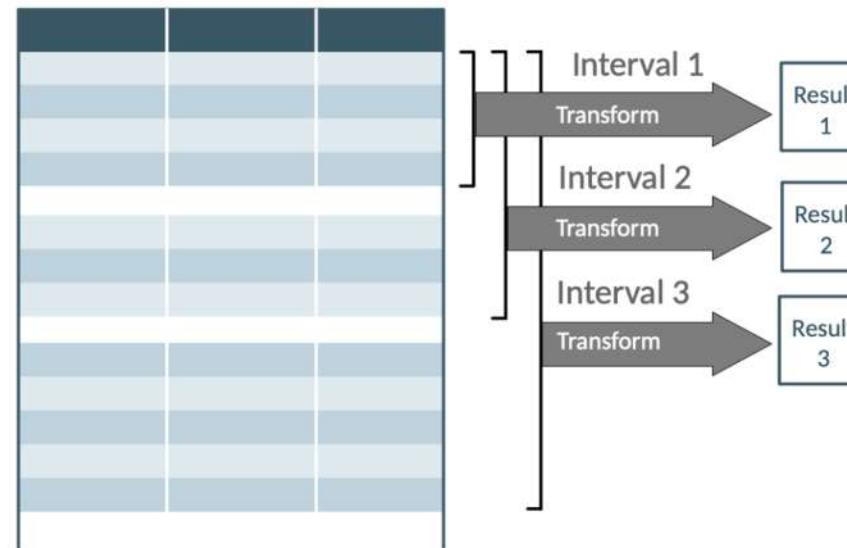
Language: Python

- 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

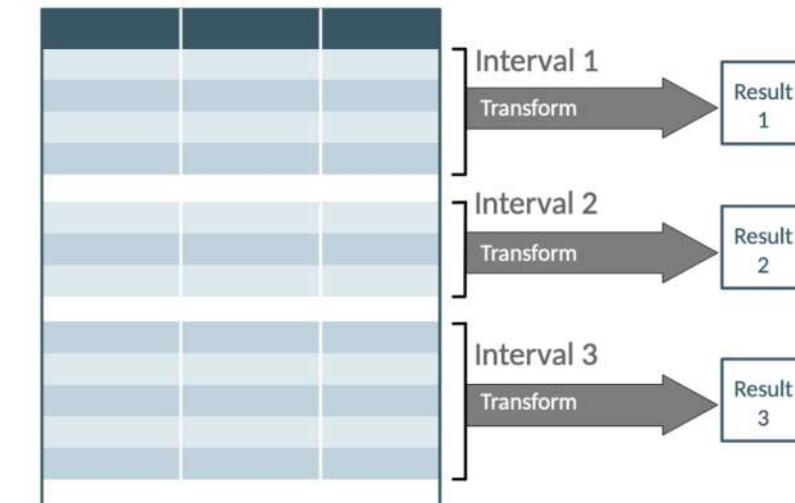
# Output Mode

- **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

```
> 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
a98febcd-683c-4b96-9e50-9615418ce1a2	27	38
5b42e782-f0ab-4428-aea4-cbeaf77032ff	13	28
7836c1a6-365b-4ec2-a402-65cdbc03a171	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

...

## 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

- Introduction to Structured Streaming
- **Exercise: Processing Streaming Data**



# Message Processing with Apache Kafka

---

## Chapter 16

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- **Message Processing with Apache Kafka**
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Message Processing with Apache Kafka

---

In this chapter, 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

# 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
    - using modest hardware, with messages of a typical size
- **Low latency**
  - Data is delivered in a fraction of a second
- **Flexible**
  - Decouples the production of data from its consumption

# Kafka Use Cases

---

- Kafka is used for a variety of use cases, such as
  - Log aggregation
  - Messaging
  - Website activity tracking
  - Stream processing
  - Event sourcing

# Chapter Topics

---

## Message Processing with Apache Kafka

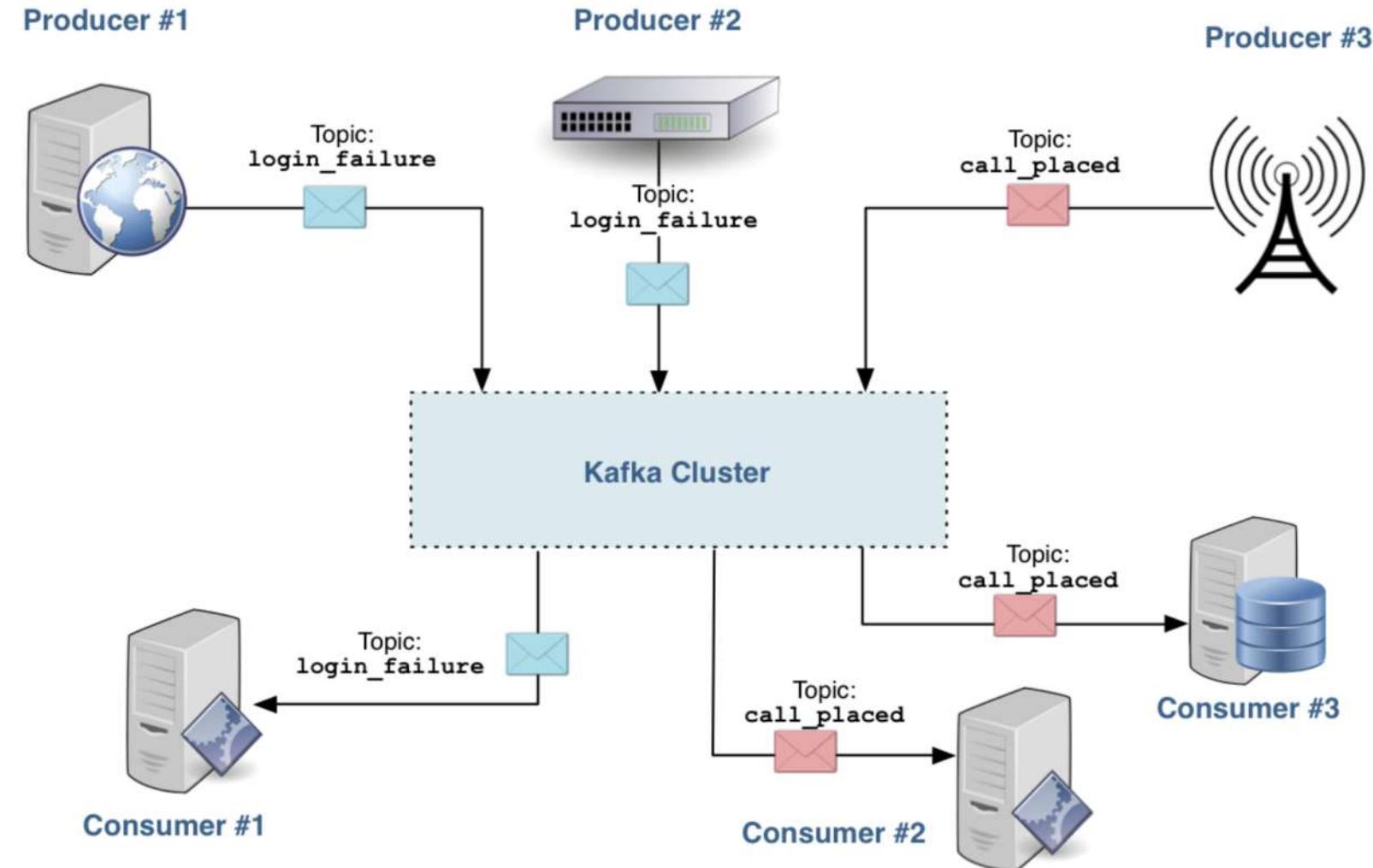
- What is Apache Kafka?
- **Apache Kafka Overview**
- Scaling Apache Kafka
- Apache Kafka Cluster Architecture
- Apache Kafka Command Line Tools

# 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

---

- **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
- **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

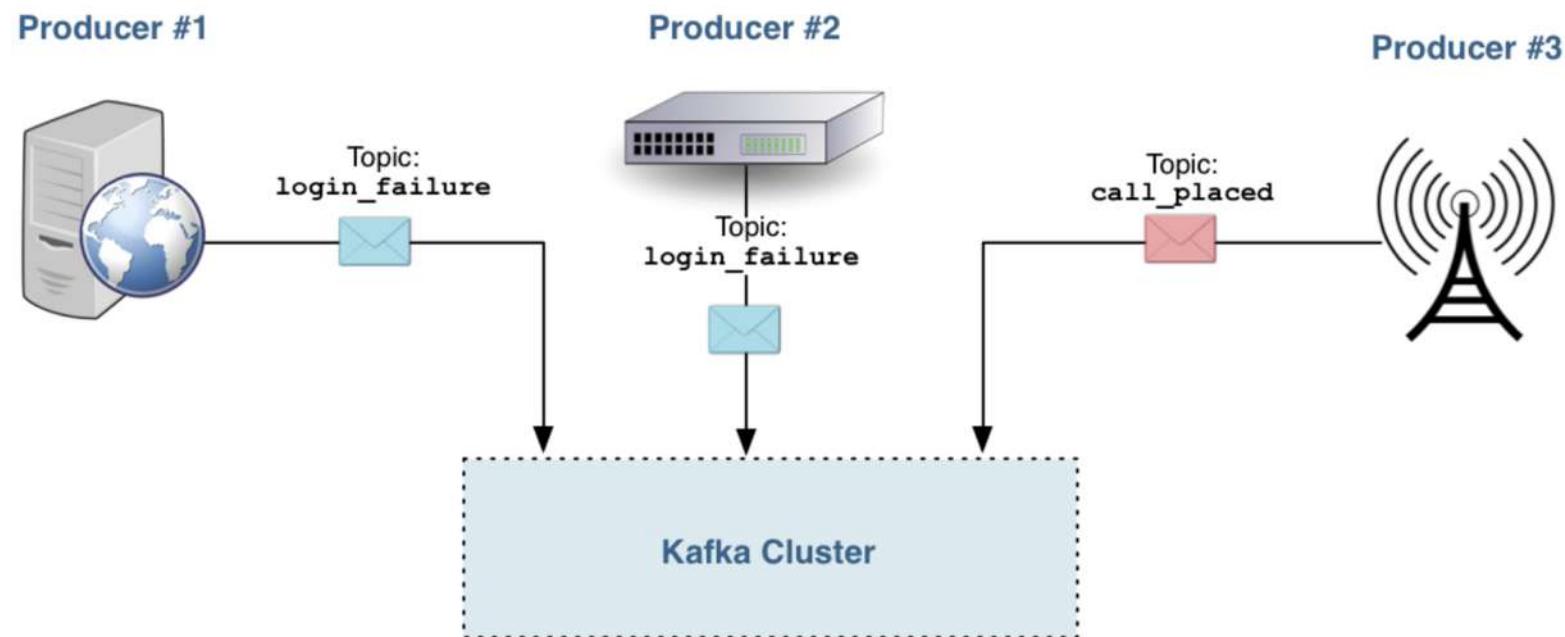
# Topics

---

- **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

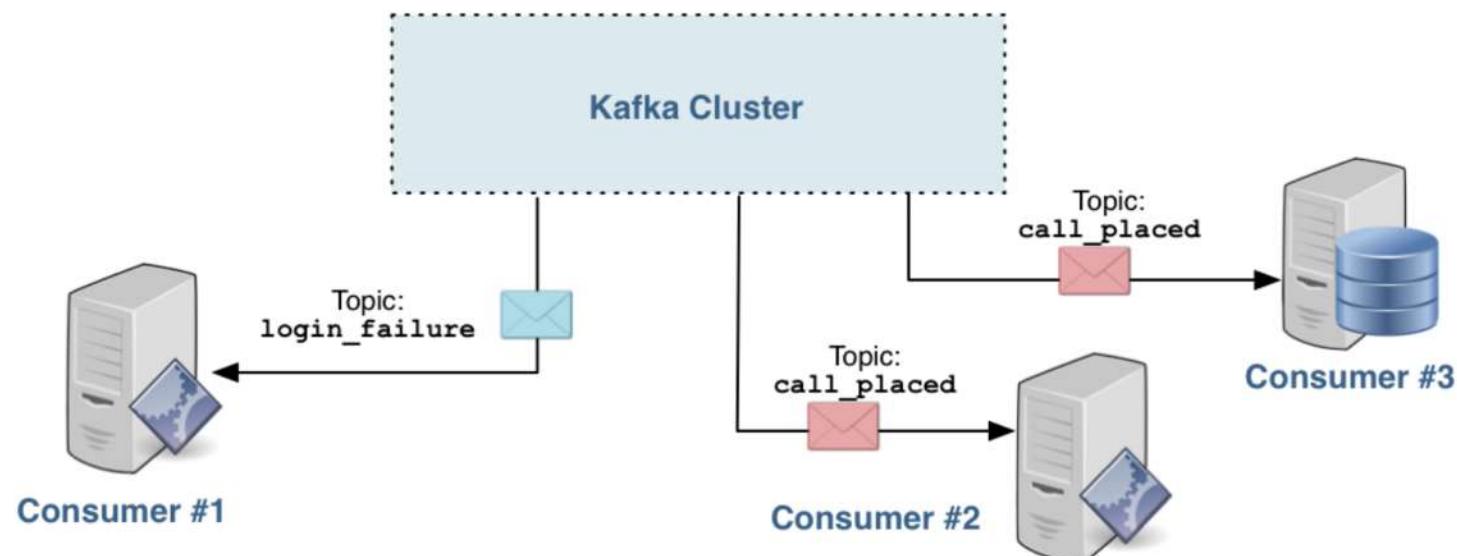
# Producers

- 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 NiFi
  - 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

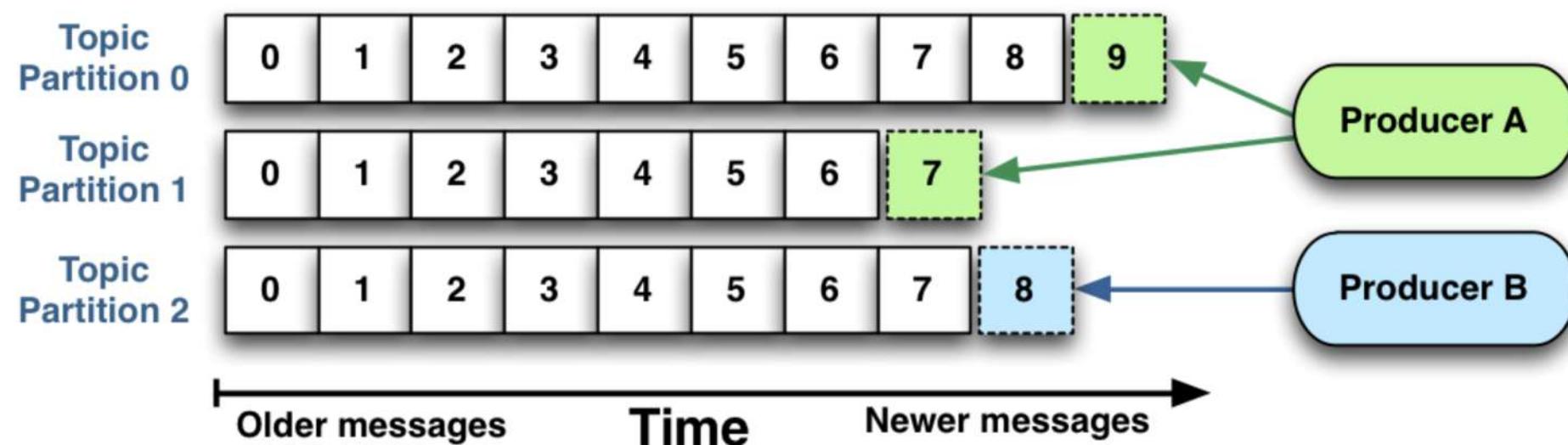
# Scaling Kafka

---

- **Scalability is one of the key benefits of Kafka**
- **Two features let you scale Kafka for performance**
  - Topic partitions
  - Consumer groups

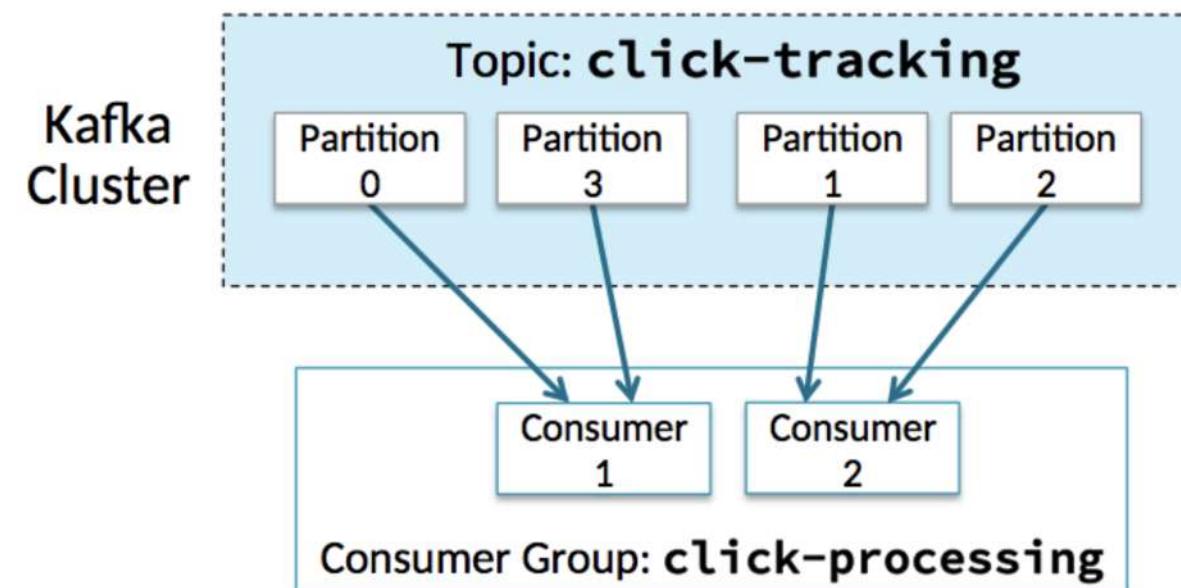
# Topic Partitioning

- Kafka divides each topic into some number of partitions (Note that this is unrelated to partitioning in HDFS or Spark)
  - 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



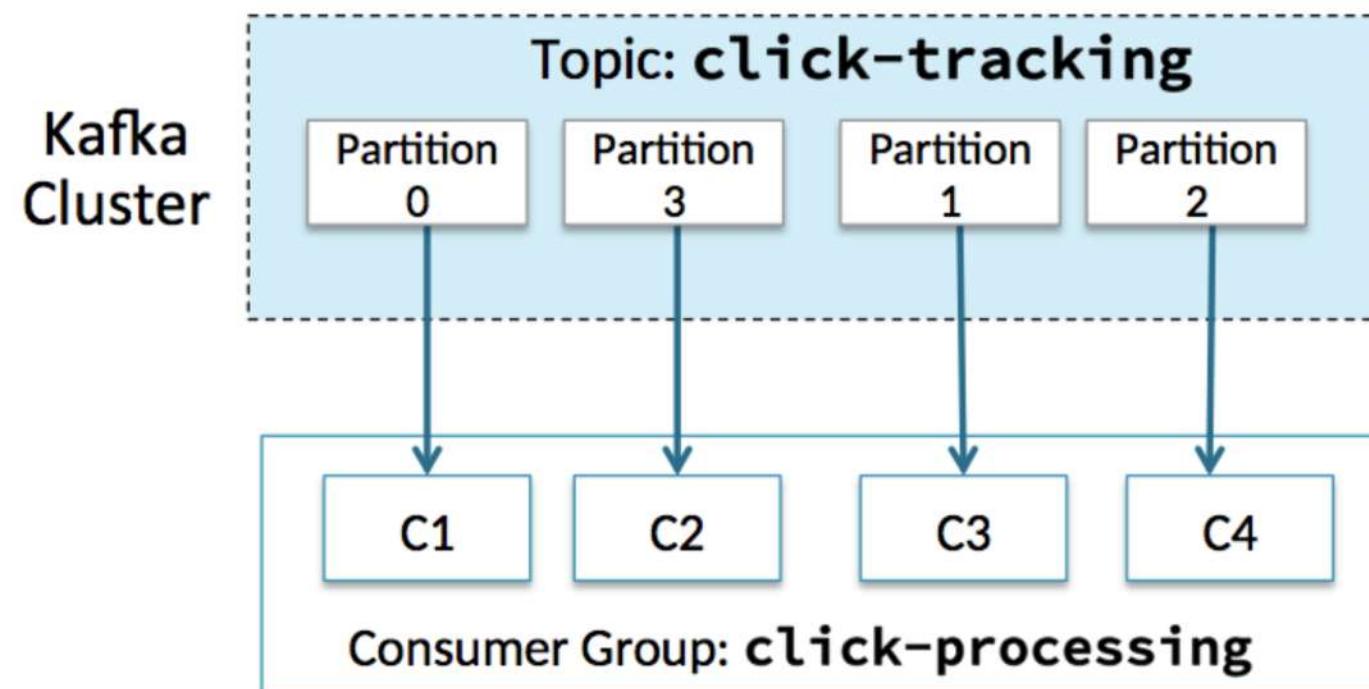
# Consumer Groups

- 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



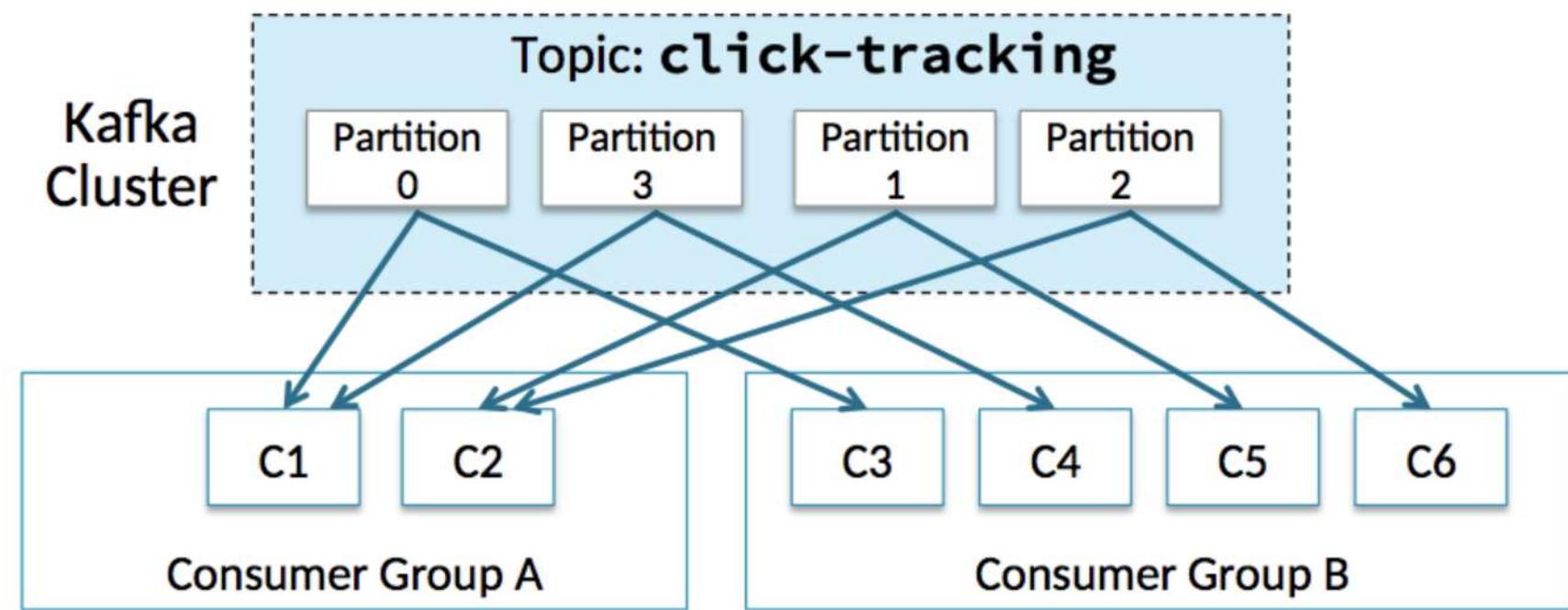
# Increasing Consumer Throughput

- 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



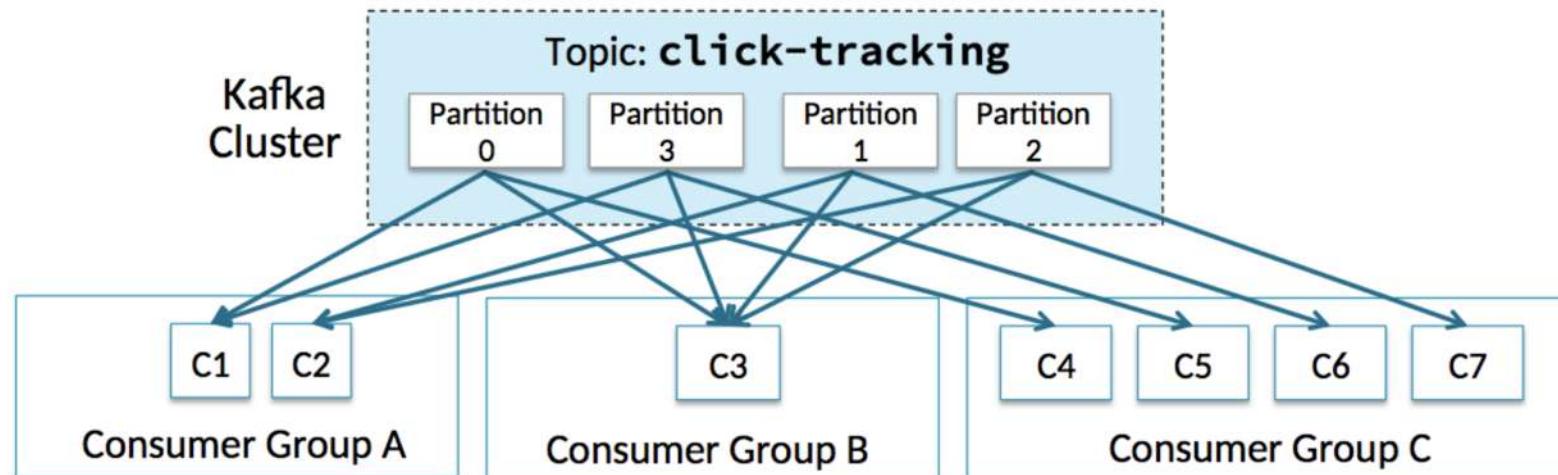
# Multiple Consumer Groups

- 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 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

---

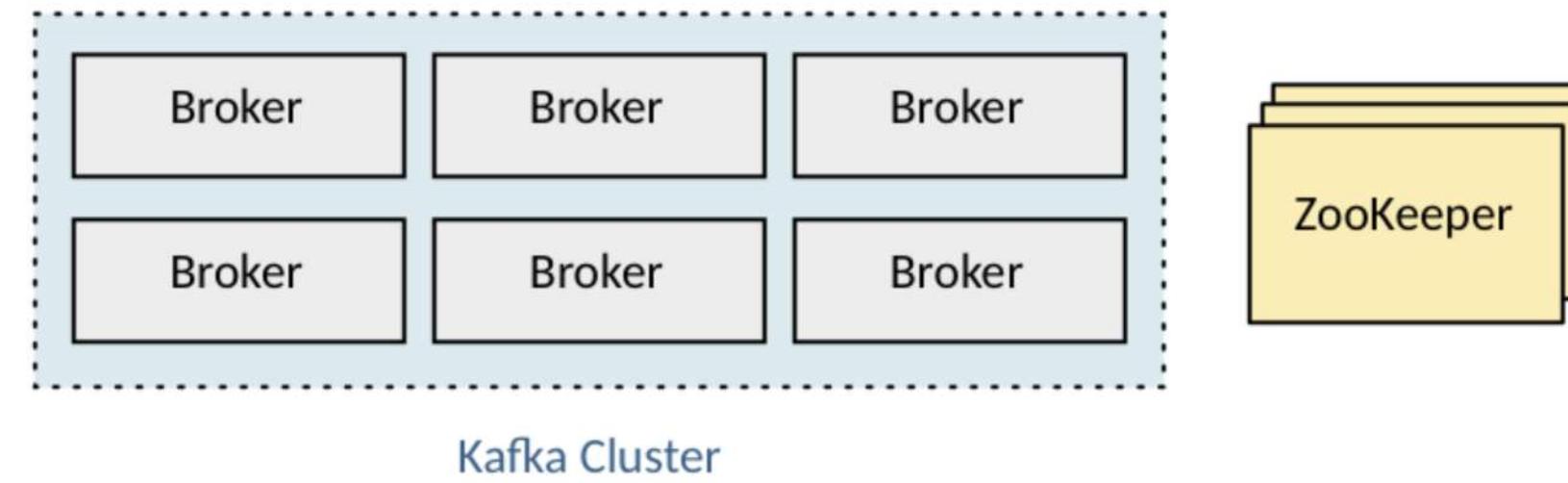
## Message Processing with Apache Kafka

- What is Apache Kafka?
- Apache Kafka Overview
- Scaling Apache Kafka
- **Apache Kafka Cluster Architecture**
- Apache Kafka Command Line Tools

# 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



# 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

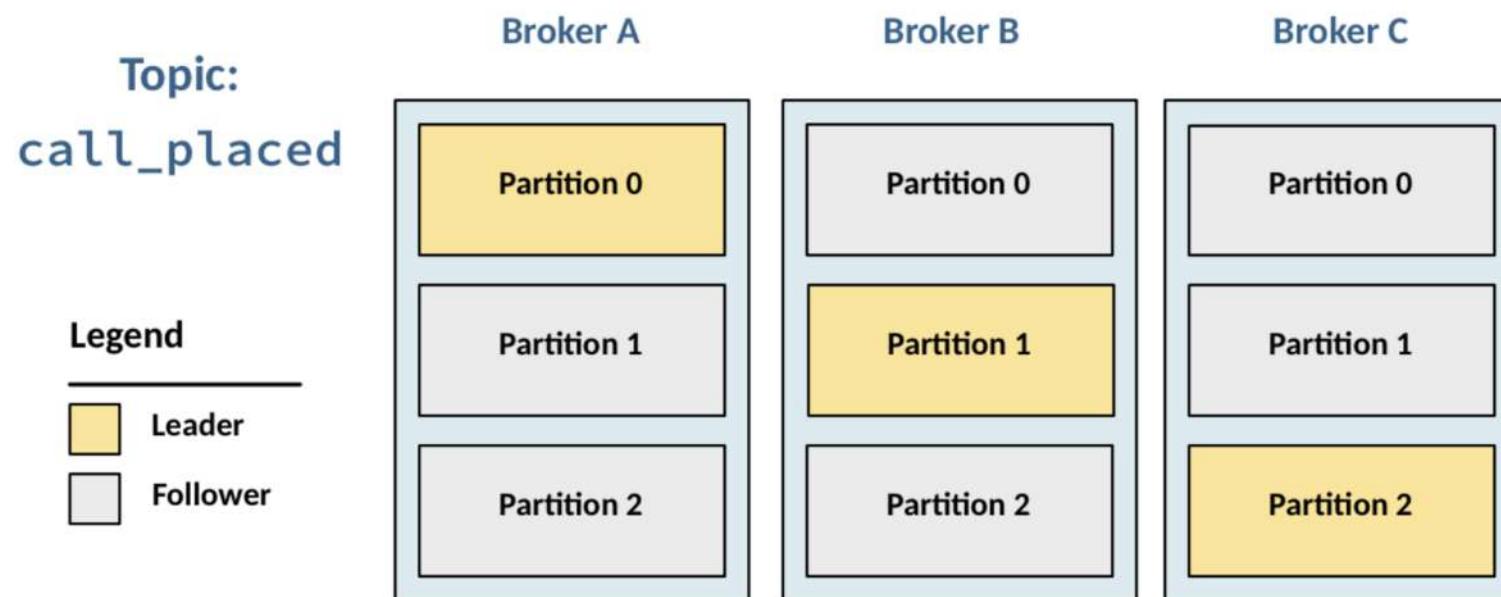
# Kafka Brokers

---

- **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

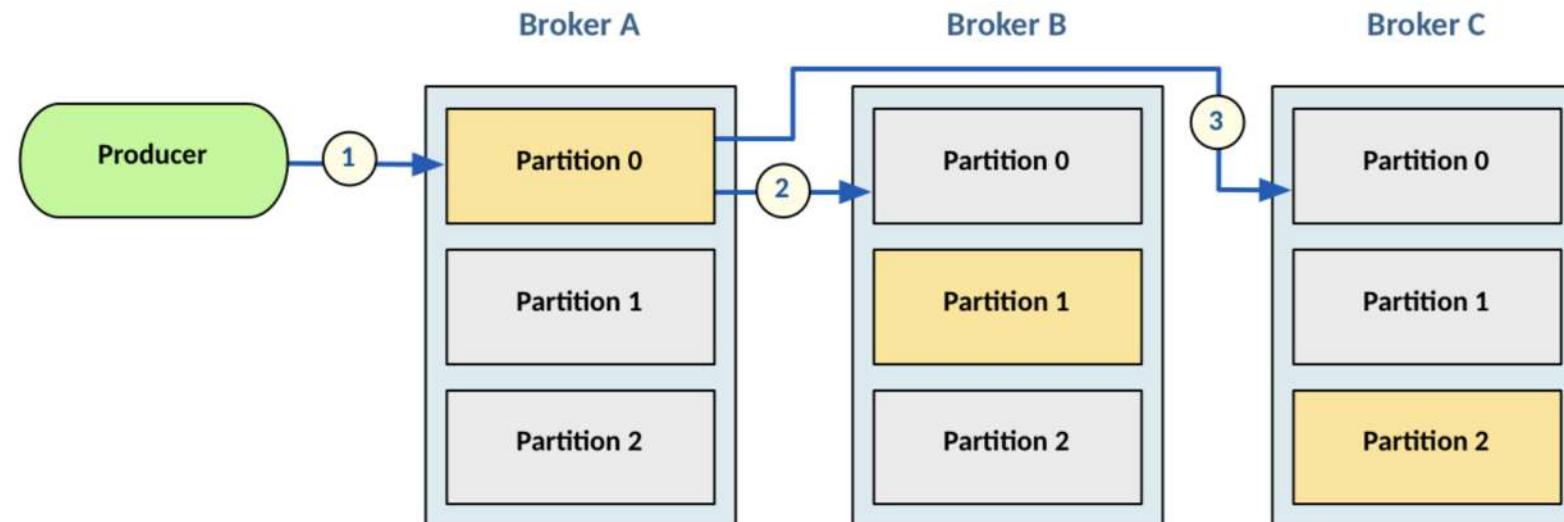
# Topic Replication

- 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**

# 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
```

## 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>



# Structured Streaming with Apache Kafka

---

Chapter 17

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- **Structured Streaming with Apache Kafka**
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# 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

## Overview

---

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

# Chapter Topics

---

## Structured Streaming with Apache Kafka

- **Receiving Kafka Messages**
- **Sending Kafka Messages**
- **Exercise: Working with 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.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
  - **maxOffestsPerTrigger**—maximum number of messages in each micro-batch

# Kafka Input DataFrame Schema

---

- 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 partition the message was in
  - **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.format("kafka"). \
    option("kafka.bootstrap.servers","broker-host1:9092,broker-host2:9092"). \
    option("subscribe","status").load()

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

from pyspark.sql import functions
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

- Receiving Kafka Messages
- **Sending Kafka Messages**
- Exercise: Working with Kafka Streaming Messages

# Sending Data to a Kafka Streaming Sink

---

- **Create a DataFrame containing output data**
  - Each DataFrame element is sent as a single Kafka message
  - Elements must be in the correct format
- **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

## 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

- Receiving Kafka Messages
- Sending Kafka Messages
- **Exercise: Working with Kafka Streaming Messages**



# Aggregating and Joining Streaming DataFrames

---

Chapter 18

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames**
- Conclusion
- Appendix: Working with Datasets in Scala

# 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**
- **Exercise: Aggregating and Joining Streaming DataFrames**

# Streaming Aggregation

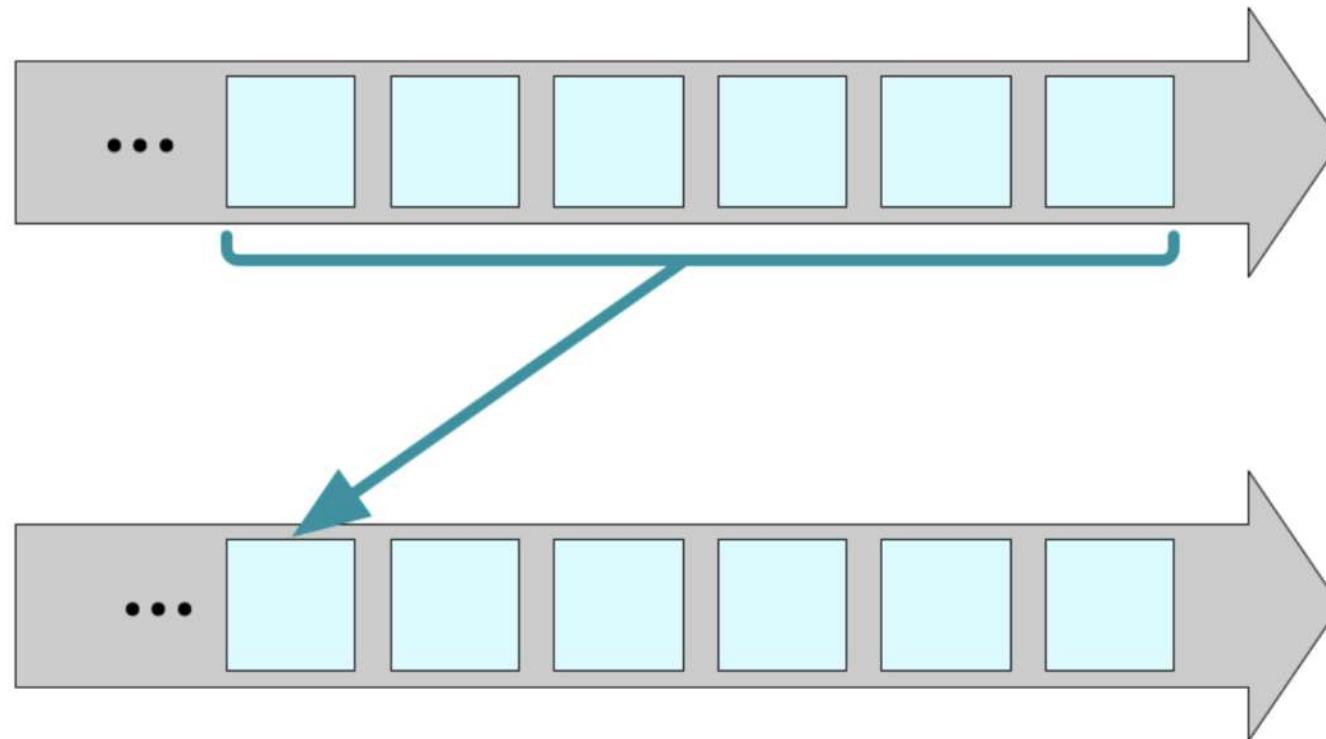
---

- 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

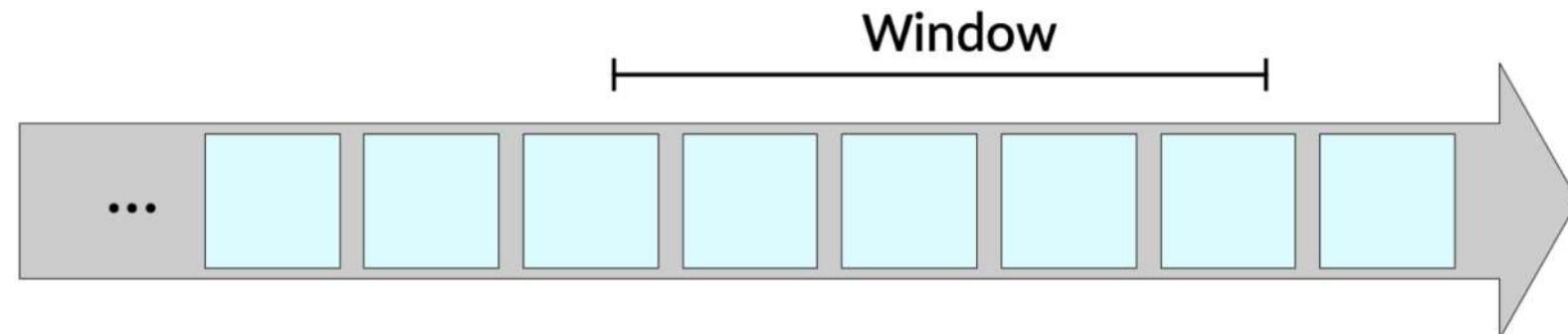
model	count
Ronin S3	2
Ronin S2	2
Ronin S1	4

...

# Sliding Window Aggregation (1)

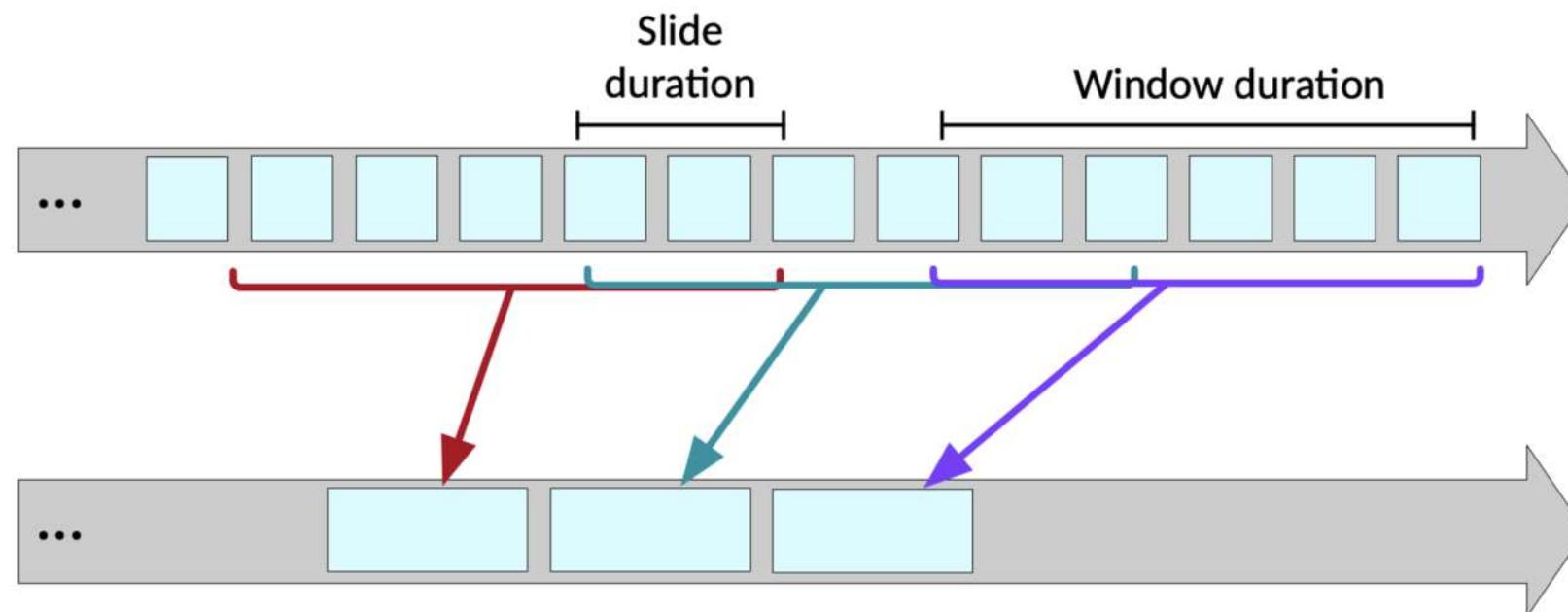
---

- 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 Total Status Messages by Model per Window

- Count status messages for selected models for every 10 minutes 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 Mode
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**
- 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

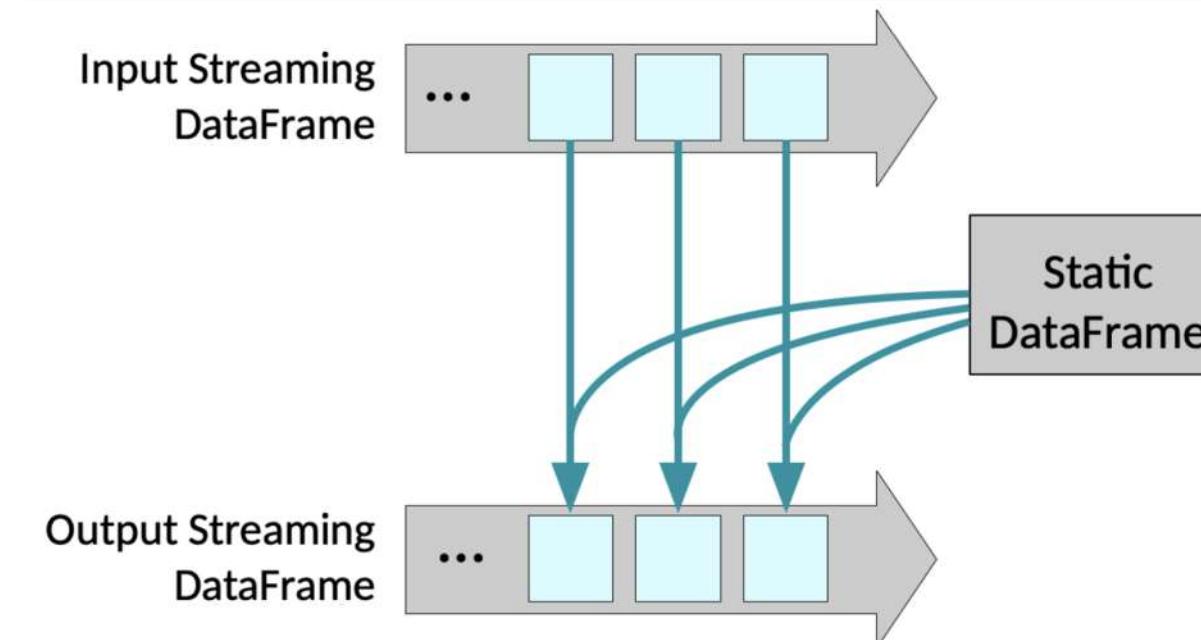
# Supported Types of Joins

---

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

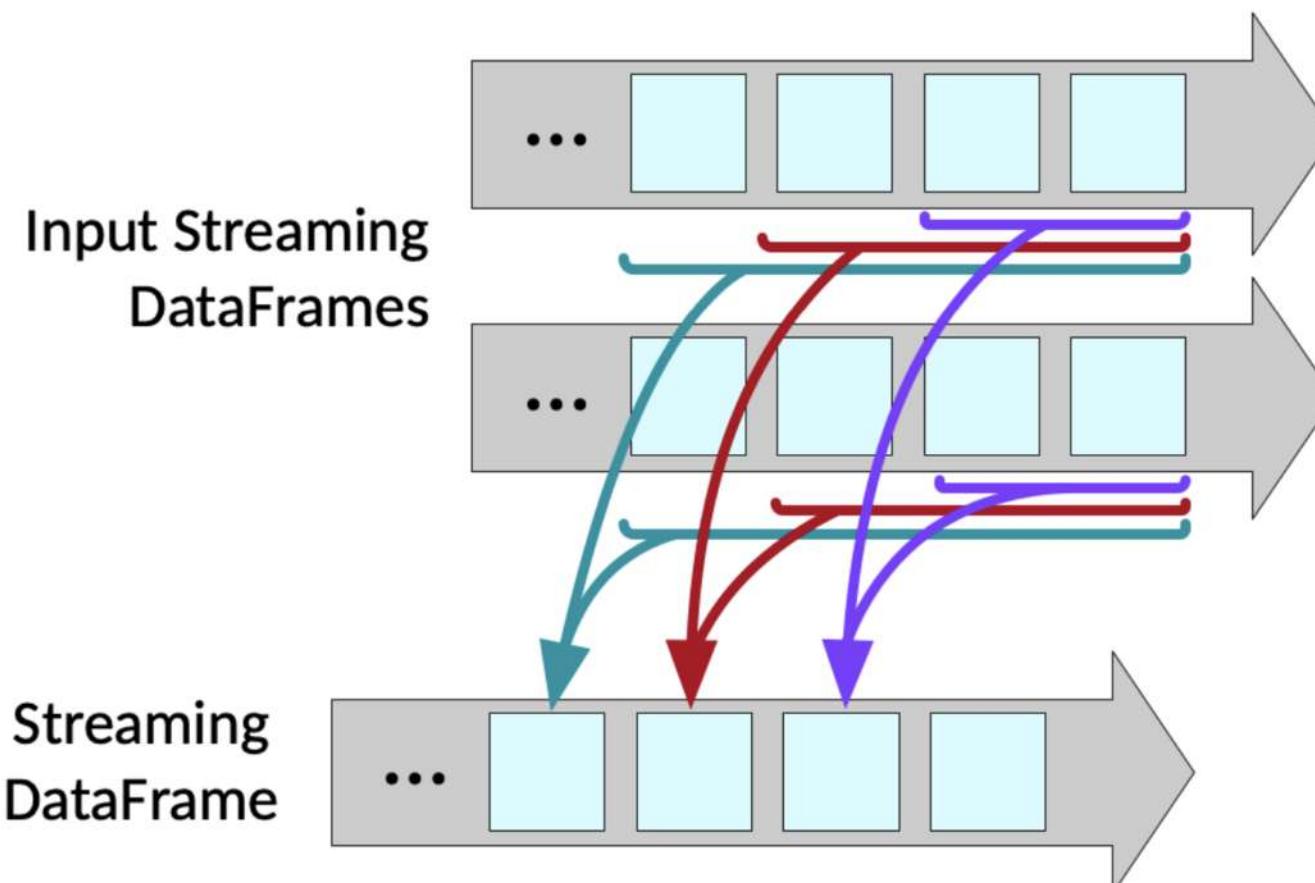
- **statusStreamDF**—streaming status messages including device IDs
- **accountDevStaticDF**—static data containing account numbers with IDs of the account's devices
- **Output: status messages with account number of device**

```
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

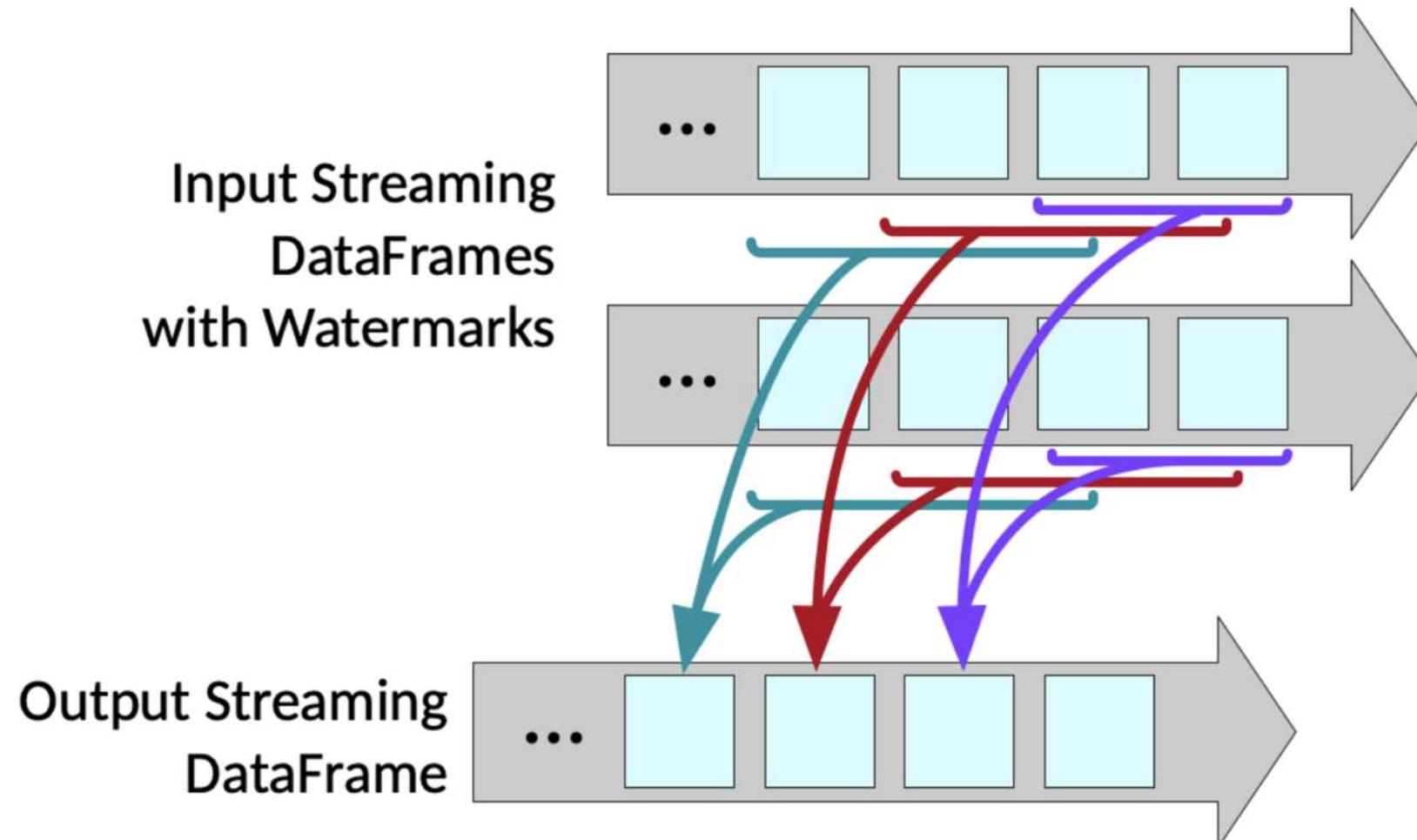
- Join activation messages with status messages with the same device ID
  - Inner join (default)
  - Status messages with no activation record will be excluded

```
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

## 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
- **Exercise: Aggregating and Joining Streaming DataFrames**

## Conclusion

---

### Chapter 19

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- Appendix: Working with Datasets in Scala

# Conclusion

---

**During this class 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 interpreters 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 Training for Apache Kafka
  - Cloudera DataFlow: Flow Management with Apache NiFi
  - Spark Application Performance Tuning
  - Cloudera Streaming Analytics: Using Apache Flink and SQL Stream Builder on CDP
- **For system administrators**
  - Administrator Training: CDP Private Cloud Base
- **For data analysts and data scientists**
  - Cloudera Data Analyst Training
  - Cloudera Data Scientist Training



## Working with Datasets in Scala

---

### Appendix A

# Course Chapters

---

- Introduction
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working with RDDs
- Working with DataFrames
- Introduction to Apache Hive
- Hive and Spark Integration
- Data Visualization with Zeppelin
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Writing, Configuring and Running Spark Applications
- Introduction to Structured Streaming
- Message Processing with Apache Kafka
- Structured Streaming with Apache Kafka
- Aggregating and Joining Streaming DataFrames
- Conclusion
- **Appendix: Working with Datasets in Scala**

# Working with Datasets in Scala

---

After completing this chapter, you'll 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

- **Working with Datasets in Scala**
- **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

# 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

## 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
```

# Loading and Saving Datasets

---

- You cannot load a Dataset directly from a structured source
  - Create a Dataset by loading a DataFrame 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

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

Data File: names.json

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

# 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]
```

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

## 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

- Working with Datasets in Scala
- **Exercise: Using Datasets in Scala**