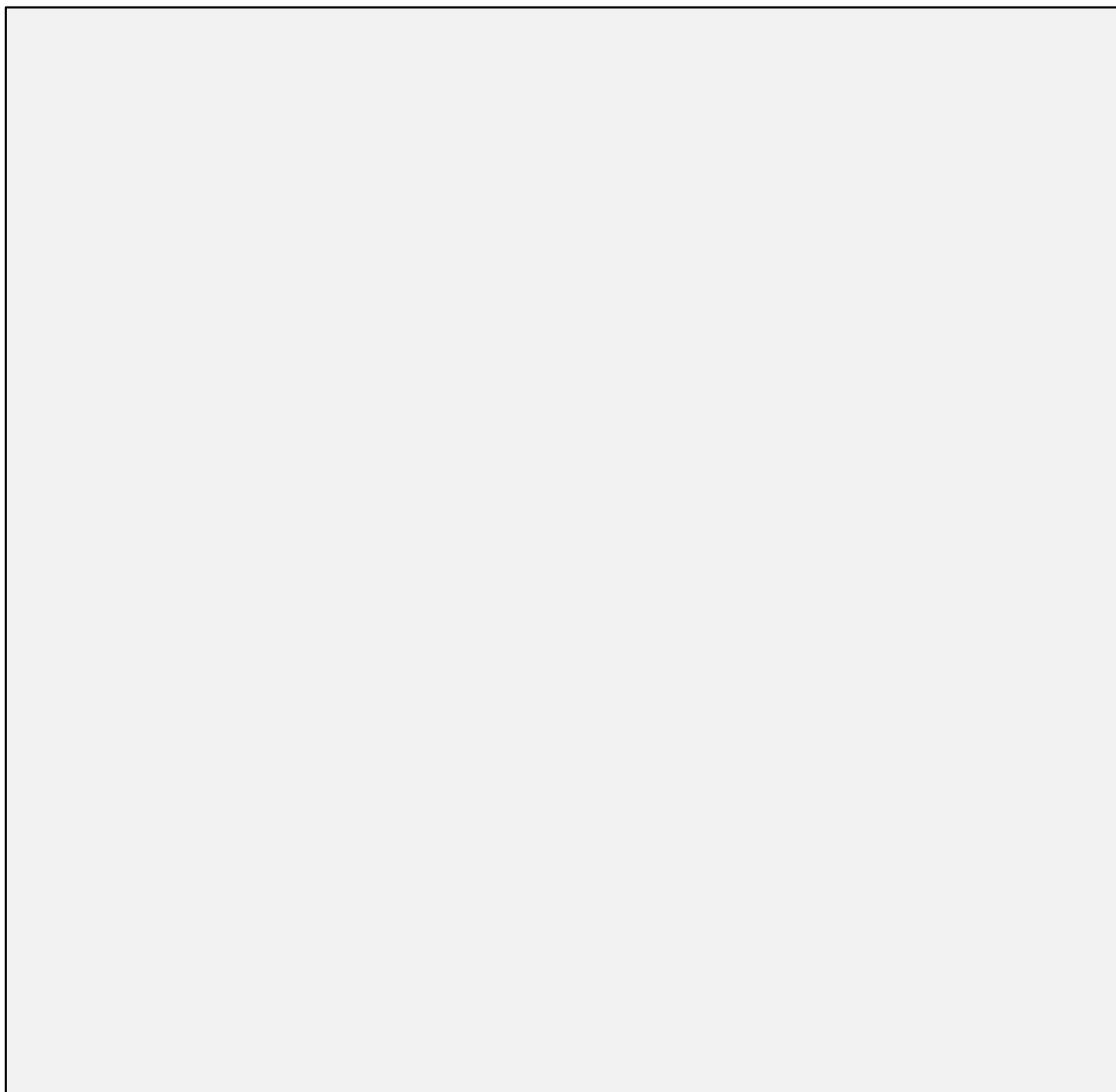


Preparing with Cloudera Data Engineering

DENG-254 220331



Introduction

Chapter 1

The goal of this chapter is to introduce Cloudera, Cloudera's products, educational services, and the training environment.

Course Chapters

- **Introduction**
- Why Data Engineering
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

This slide is intended to clearly convey to students that, while we may sometimes refer to products like Hadoop, Hive, and Impala later in the course, these are simply shorthand for the longer and more formal names. Apache Hadoop, as well as many related software projects that Cloudera helps to develop and distribute, is owned by the Apache Software Foundation (ASF). The slide is not intended to introduce these products; that will be done later, if needed.

Throughout its history, Cloudera has been strongly committed to a community-driven, Hadoop-based platform based on open standards that meets the highest enterprise expectations for stability and reliability. Cloudera's Chief Architect, Doug Cutting, served as Director of the ASF for more than five years. He has been an Apache committer for more than 15 years, serving alongside dozens of other Clouderans who also help to work on many of the open source projects. In fact, Cloudera employees have founded more than 20 successful Hadoop ecosystem projects, including Apache Hadoop itself. Cloudera is a Platinum-level sponsor of the ASF.
<http://apache.org/foundation/thanks.html>.

Additionally, course material may make occasional and incidental references to other product names covered by trademark, such as commercial software from partner companies or brand names for hardware on which one might deploy a cluster. Reference to any products, services, processes or other information, by trade name, trademark, manufacturer, supplier, or otherwise does not necessarily constitute or imply endorsement, sponsorship, or recommendation by Cloudera.

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 [Data Engineering](#) service
- How to use the Spark and/or Hive to explore, process, and analyze distributed data
- How to query data using Spark SQL, DataFrames, and Datasets
- How to chain your Spark and Hive applications using [AirFlow](#)
- How to monitor and troubleshoot your applications using the [Workload Manager](#)

Agenda

Day 1	Day 2	Day 3	Day 4
Class Introduction	Spark DataFrames	Apache Hive Introduction	Distributed Processing
Zeppelin Introduction	Reading DataFrames	Transforming data with Hive	Distributed Persistence
HDFS Introduction	Working with Columns	Data Engineering with Hive	Working the Data Engineering Service
YARN Introduction	Transforming DataFrames	<ul style="list-style-type: none">• Partitions• Buckets• Skewed Tables	Working with Airflow
Distributed Processing History	Working with UDFs	<ul style="list-style-type: none">• Text Data• Complex Types	Working with WXM
Spark RDDs	Working with Windows	Spark Integration with Hive	(*) Appendix: Scala Datasets

(*) 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?

Establish your credibility and enthusiasm here. You'll likely want to mention your experience as an instructor, plus any relevant experience as a developer, system administrator, DBA, or business analyst. If you can relate this to the audience (because you're from the area or have worked in the same industry), all the better.

This is also an opportunity to get to know the students, so you can tailor your explanations and analogies to the experience that they already have. It's a good idea to draw out a grid corresponding to the seat layout and write students' names down as they introduce themselves, allowing you to remember someone's name a few days later based on where they're sitting.

The outline for all our courses are available online, so you should be familiar with them and will know whether a student's expectations for the course are reasonable.

The next section will be "About Cloudera," and the background notes begin here, and continue on the "About Cloudera" slide.

Google, at the turn of the millennium, realized that their vision of leveraging planet-scale search data couldn't be completed with the available technologies. Rejecting expensive enterprise class servers, they chose large numbers of industry standard hardware instead. They got more cheap servers for the same price as a few expensive servers. But cheap servers fail more frequently—thus the importance of handling failures in software. Google *also* rejected standard database technologies, because they don't scale to the extent of need they projected. Google created their own technologies: The Google file system (GFS), a distributed map-reduce implementation, and later Big Table.

In the 2000s, there was an open source effort to create a web-scale search engine for others to use. The small team was basically trying to recreate Google. Google did not contribute source code to this effort, but they started to publish academic papers about what they had done. They published a paper on GFS, on their map-reduce implementation, and on Big Table. The search engine open source team read those papers and essentially created alternate implementations of those technologies. They realized that web search was only one application for the technologies. So they created them as a separate project—Hadoop was born.

Chapter Topics

Introduction

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

CLOUDERA

THE HYBRID DATA COMPANY

We believe that **data** can make what is impossible today, possible tomorrow

We empower people to transform complex **data anywhere** into actionable insights faster and easier

We deliver a hybrid data platform with secure **data management** and portable cloud-native **data analytics**

We help with a hybrid data cloud platform with an integrated suite of secure cloud-native data services to **make it easier to manage and secure the entire data lifecycle** – from ingest to experimentation, Edge to AI – in any cloud or data center ---- a hybrid data cloud for your company...

- Our WHY is that we genuinely believe that data can make things previously deemed impossible, possible. Today, we talk to our smartphones and other devices and they talk back - providing real-time insights into everything from sports scores to understanding gene sequencing. Cloudera powers that capability for the largest smartphone maker in the world.
- Our HOW is that we EMPOWER PEOPLE in organizations to build these powerful data apps. So we don't make these apps ourselves, our customers do. And our customers include the world's top banks, telecommunications companies and healthcare providers. So next time your credit card company sends you a fraud alert for a potential misuse of your card in real time - You're Welcome. It's likely that Cloudera helped provide the technology that made that data app possible.
- And lastly, WHAT we provide our customers is a Hybrid Data Platform and an integrated suite of Data Services that make all of this data magic possible on any cloud!

“The **future data ecosystem** should leverage distributed data management components — which may **run on multiple clouds and/or on-premises** — but are **treated as a cohesive whole** with a high degree of automation. Integration, **metadata and governance capabilities** glue the individual components together.”



Strategic Roadmap for Migrating Data Management to the Cloud
Published 21 March 2022 - ID G00746011, Analyst(s): Robert Thanaraj, Adam Ronthal, Donald Feinberg

The industry and the analysts agree... the future data ecosystem is a hybrid one..... Multi-cloud and on-premises... but with common security governance and control plane. Make no mistake... Cloudera is the only company currently positioned to deliver this hybrid data platform.

A More Specific Definition of Hybrid

Hybrid = Freedom to move existing and future applications, data, and users bi-directionally between the data center and multiple public clouds



Applications include streams, pipelines, workloads, workspaces, and other data products



Data considerations include formats, policies, metadata, replication etc.



Users need to leverage existing skills and processes, while continuously modernizing

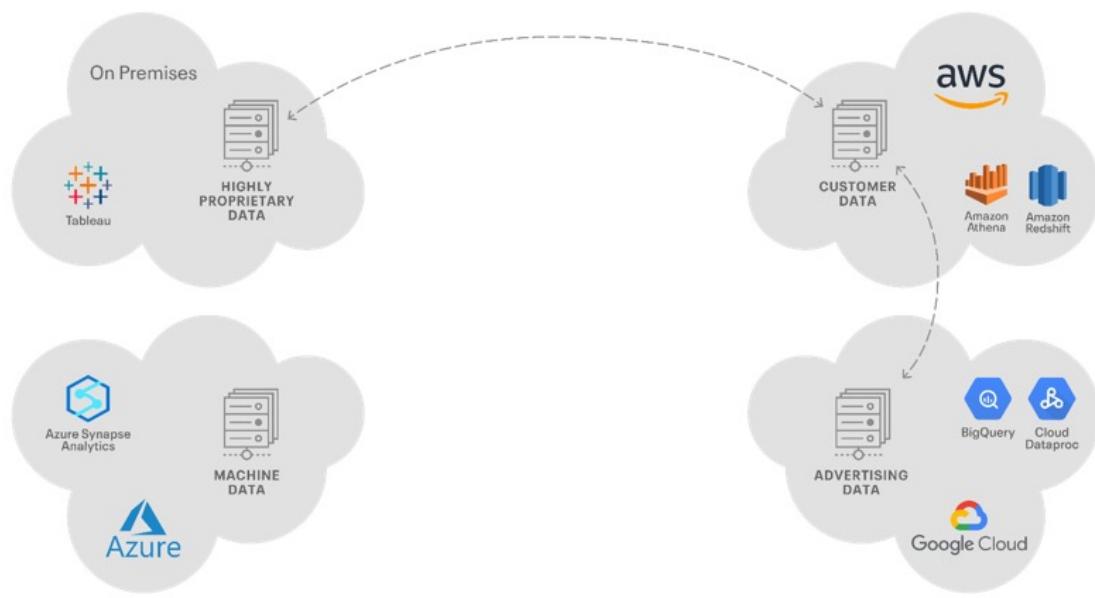
A definition from Cloud Standards Customer Council ISO17788.

Long version: “a cloud deployment model using at least two different cloud deployment models” where the deployment models can be IaaS, PaaS, SaaS, and/or XaaS and the deployments involved remain unique entities but are bound together by appropriate technology that enables interoperability, data portability and application portability.

Shorter version: using at least two different cloud deployments where the deployments remain unique entities but enable interoperability, data portability and application portability

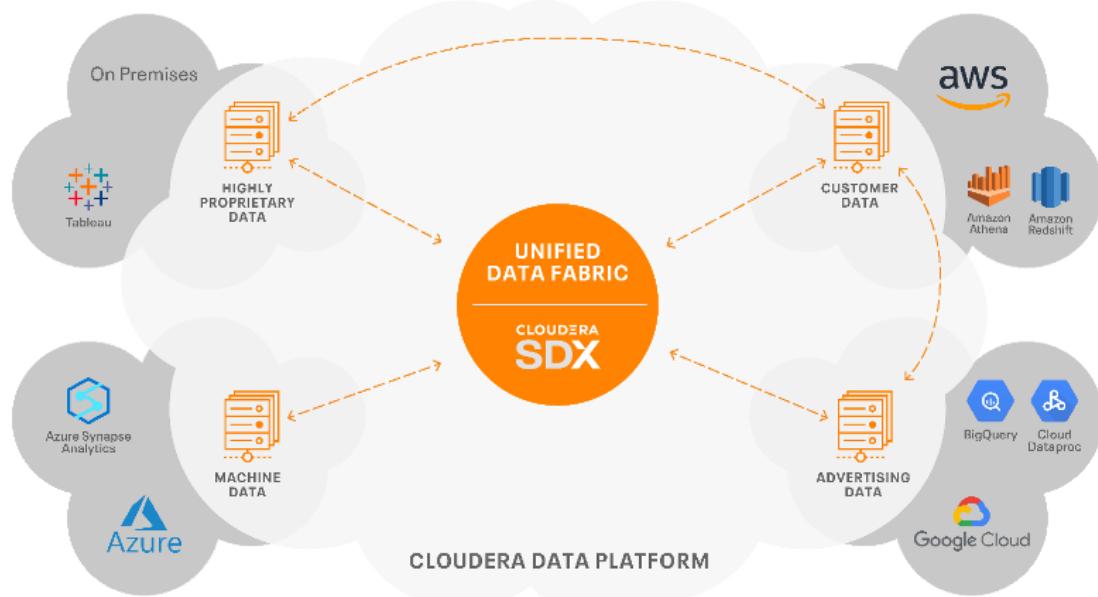
Cloudera's Hybrid Data Platform

Portable – Interoperable – Open – Secure



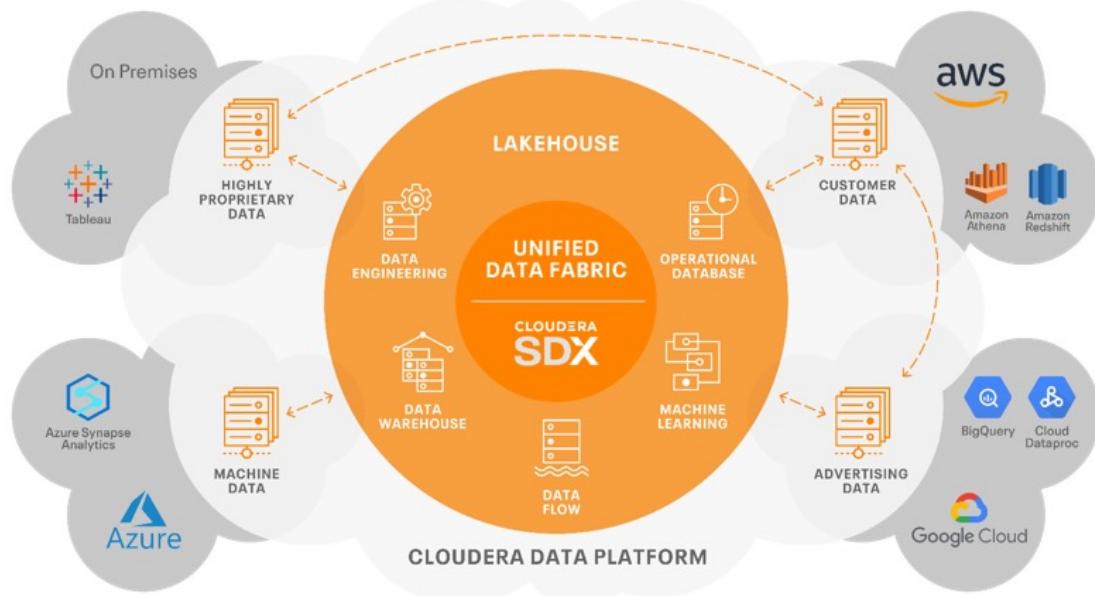
Cloudera's Hybrid Data Platform (2)

Portable – Interoperable – Open – Secure



Cloudera's Hybrid Data Platform (3)

Portable – Interoperable – Open – Secure



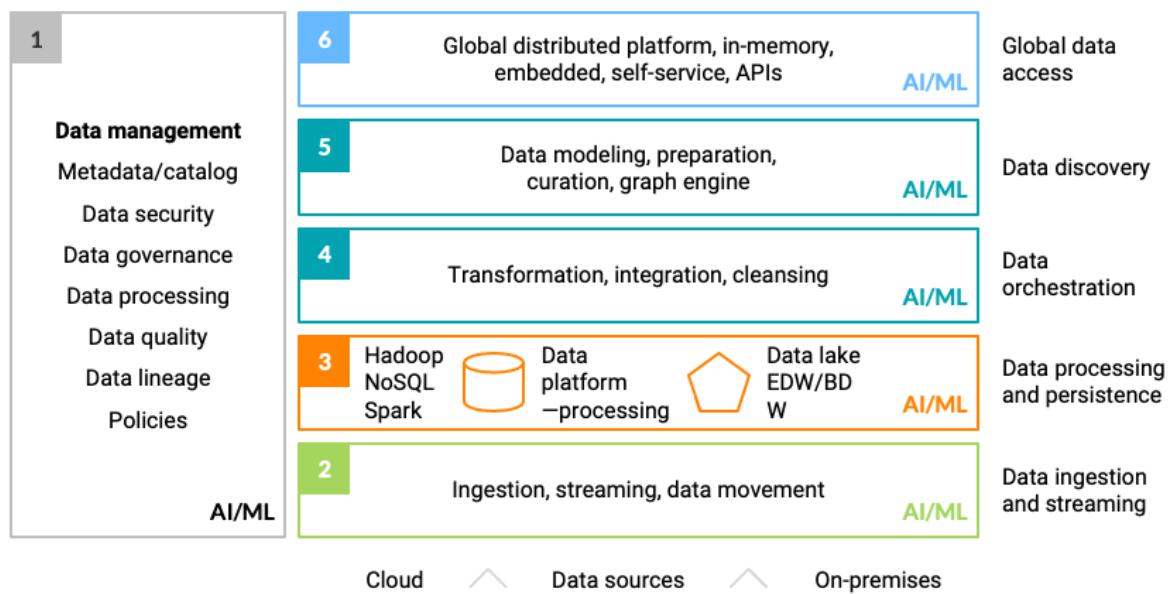
Forrester's Definition

A **Data Fabric** orchestrates disparate data sources intelligently and securely in a self-service manner, leveraging data platforms such as data lakes, data warehouses, NoSQL, translytical, and others to deliver a unified, trusted, and comprehensive view of customer and business data across the enterprise to support applications and insights.

The term “translytical” is quite a new term in the IT industry and can be defined as a single data platform that can support workloads that are diverse including transactional, operational, and analytical information in real-time. Translytical data platforms integrate in-memory tools to support SSD, flash, and DRAM.

CDP Hybrid Data Platform – Enabling Data Fabric (2)

Forrester's Definition



By Definition:

It Orchestrates disparate data sources intelligently and securely in a self-service manner, leveraging data platforms such as data lakes, datawarehouses, NoSQL, translytical, and others to deliver a unified, trusted, and comprehensive view of customer and business data across the enterprise to support applications and insights.

Gartner's Definition

Data lakehouses integrate and unify the capabilities of data warehouses and data lakes, aiming to support AI, BI, ML and data engineering (“multifunction analytics”) on a single platform.

The “Lakehouse” – Best of Both Worlds

When ALL data needs to come together to answer critical business challenges



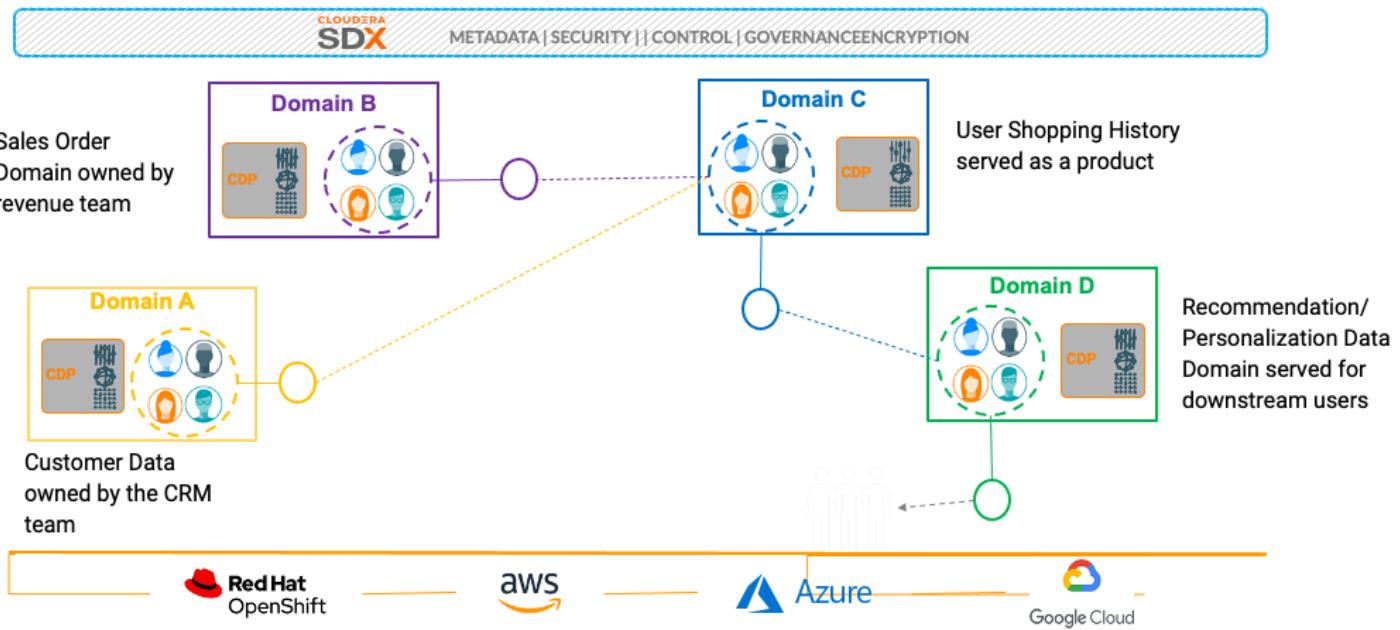
Data as a Product

Distributed data products oriented around domains and owned by independent cross-functional teams who have embedded data engineers and data product owners, using common data infrastructure as a platform to host, prep and serve their data assets.

Here is what the Gartner says:

- A data fabric is a technology-enabled implementation capable of many outputs, only one of which is data products.
- A data mesh is a solution architecture for the specific goal of building business-focused data products.

CDP Hybrid Data Platform – Enabling Data Mesh (2)



See the Cloudera Blog: How Cloudera DataFlow Enables Successful Data Mesh Architectures:
<https://blog.cloudera.com/how-cloudera-data-flow-enables-successful-data-mesh-architectures/>.

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

You can see a list of customers that we can reference on our website <https://www.cloudera.com/about/customers.html>. Note that Cloudera also has many customers who do not wish us to refer to them, and it is essential that we honor this. The only exception to this important rule is that you may refer to something that was intentionally made available to the public in which Cloudera or that customer has disclosed that they are a Cloudera customer. For example, it is permissible to mention an article in a reputable trade publication in which Cloudera's CEO mentions a specific customer or the keynote address that the customer's CTO gave at the Strata conference talking about the benefits they've experienced as a Cloudera customer.

Cloudera Education Catalog

■ A broad portfolio across multiple platforms

- Not all courses shown here

■ See [our website](#) for the complete catalog

Administrator	Administrator	Administrator Private Cloud Ba sic	Administrator CDP Public Clou d	Security Administration	Upgrading HDP/CDH to CDP			
Data Steward	Data Governance							
Data Analyst	Analyst Hive/Impala	CDP Data Visualization						
Developer Data Engineer	Spark Development	Spark Performance Tuning	NiFi Flow Management	Kafka/Stream Processing	Architecture	HBase	Flink	
Data Scientist	CDSW	Data Science	CML					
General	OnDemand Library CDP CDF	"CDP" Essentials, AWS Fundamentals	"Just Enough" Python, GIT, Scala	Tech Overviews		Live OnDemand		

CLOUDERA
Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

1-24

Note that the course “titles” here are not their true, full titles. The intent of this slide is to give a quick view of the breadth of courses in our catalog, not to go over each one. These short titles are meant to convey the content of the courses, in limited space, rather than provide a full reference for students to follow.

Feel free to emphasize any courses that could be of particular interest to your students. You also might use the link to show the full course catalog, as we may have added new courses since this was produced.

Cloudera OnDemand

■ Our OnDemand catalog includes

- Courses for developers, data analysts, administrators, and data scientists
- Exclusive OnDemand-only courses, such as those covering CDP Public Cloud Administration and Cloudera Data Science Workbench
- Free courses

■ Features include

- Video lectures and demonstrations
- Hands-on exercises through a browser-based virtual environment

■ Purchase access to a library of courses or individual courses

- [Full OnDemand Library subscription](#)

Accessing Cloudera OnDemand

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

The screenshot shows the Cloudera OnDemand dashboard. At the top, there's a search bar and navigation links for 'Dashboard', 'My Courses' (which is selected), 'My Live Sessions', and 'Download Training History'. A large orange banner at the top left says 'Welcome to Cloudera University' and describes the service as offering video-based instruction led by experts and access to 100 hours of cloud-based, ready-to-run lab environments. It features a video thumbnail of a person in a white spacesuit. Below the banner, it shows 'Total Number of Courses' with 9 Enrolled Courses, 5 Completed Courses, and 1 Learning Path. Two course cards are displayed: 'Just Enough Git' (In Progress, 6 Modules, 33% completed) and 'Developer Training for Apache Spark and Hadoop' (In Progress, 20 Modules, 0% completed). The bottom section shows 'Recent Activity' with three entries: 'You visited course details' 2 days ago, 'You visited course details' 2 days ago, and 'You visited course details' 2 days ago.

CLOUDERA
Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

1-26

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

A CCA (Cloudera Certified Associate) exam is our entry level exam. To achieve CCA certification, candidates complete a set of core tasks on a working cluster instead of answering multiple-choice questions. Each exam maps to a Cloudera course:

- CCA Spark and Hadoop Developer (CCA175) → Developer Training for Spark and Hadoop
- CCA HDP Administrator (CCA121) → Cloudera HDP Administrator Training
- CCA CDH Administrator (CCA131) → Cloudera Administrator Training
- CCA Data Analyst (CCA159) → Data Analyst Training

We also have a CCP (Cloudera Certified Professional) exam.

- CCP Data Engineer (DE575)

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**
- **Exercise Environment**

“Exercise Environment” is a cue for the instructor to explain briefly how to perform hands-on exercises in this class. This is also a good time to verify that the students can log into Keycloak and access CDP.

Downloading the Course Materials

1. Log in using <https://education.cloudera.com/>

- Click **Sign In** on the top right
- If necessary, create an account
- If you have forgotten your password, use the **Forgot your password** link

2. Locate the course

- Find the course in your list of enrollments, or enter the course title in the search bar
- Click on the course title

3. Access the materials

- Click on **Content** across the top
- Click on the module to view or download the contents
- If there is more than one module, use the Course Contents panel on the left to navigate

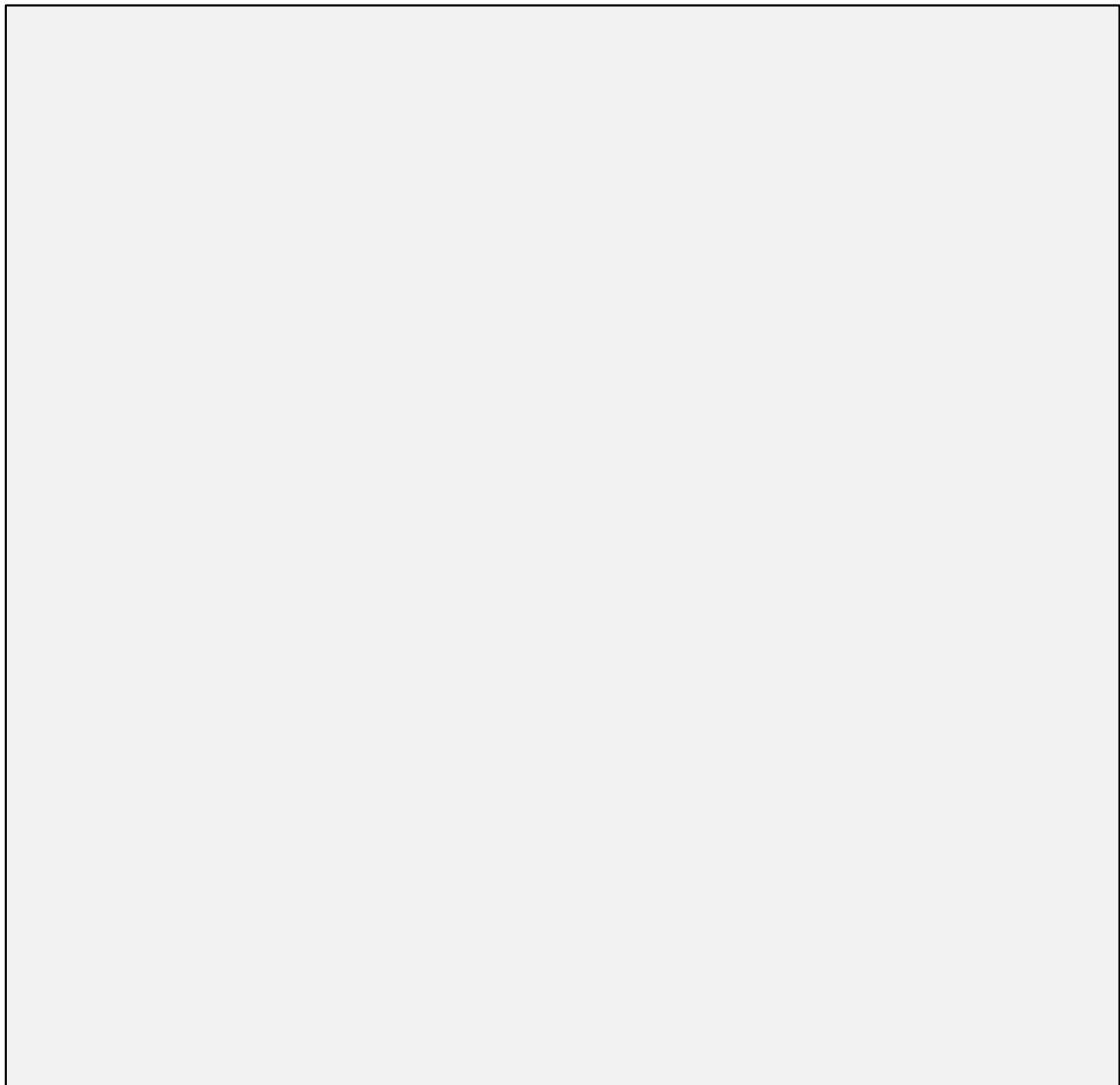
The screenshot shows the 'Course Contents' panel from the Cloudera DE platform. At the top, there are three icons: a grid, a mail icon, and a user profile icon. Below them is a 'Return to Dashboard' button. The main title is 'ILT - Cloudera DE: Developing Applications with Apache Spark - 2764504'. A progress bar indicates 'My Progress' at 33%. The content area lists three modules:

- 1. Developing Applications Notebooks (210831) (marked with a green checkmark)
- 2. Developing Applications Student Slides (210831) (marked with a grey square)
- 3. ILT - Cloudera DE: Developing Applications with Apache Spark - 2764504 (marked with a grey square)

Emphasize that students must, at the very least, download the exercise instructions.

Why Data Engineering?

Chapter 2



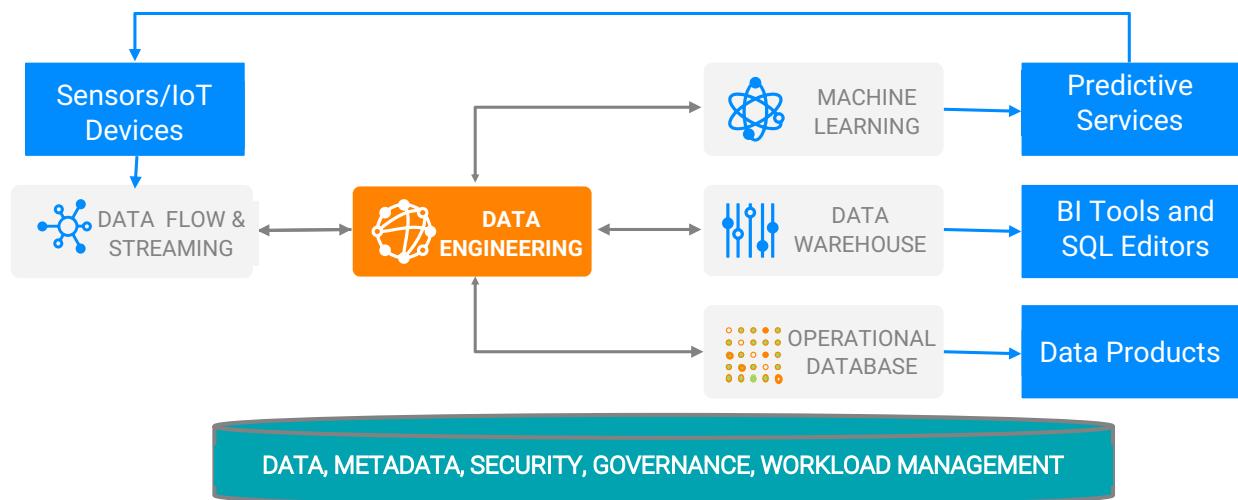
Course Chapters

- Introduction
- **Why Data Engineering?**
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- Conclusion
- Appendix: Working with Datasets in Scala

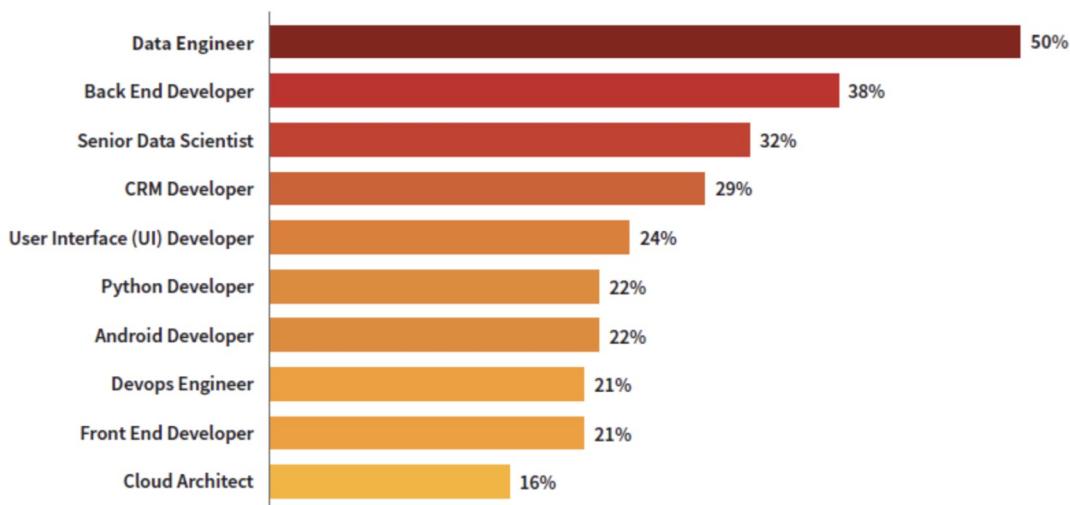
Well defined and managed datasets are the backbone of Machine Learning, Data Warehousing, and Data Cataloging

Data Engineering is Central

To Powering Business Analytics, ML, and Data Products



The Fastest Growing, In-Demand Need for Enterprise Businesses



1 - DICE - 2020 Tech Job Report

2 & 3 - Fior Markets, the global big data and data engineering services market report

The Challenges with Data Engineering



Managing Spark Resources



Orchestrating Complex Pipelines



Visibility & Troubleshooting

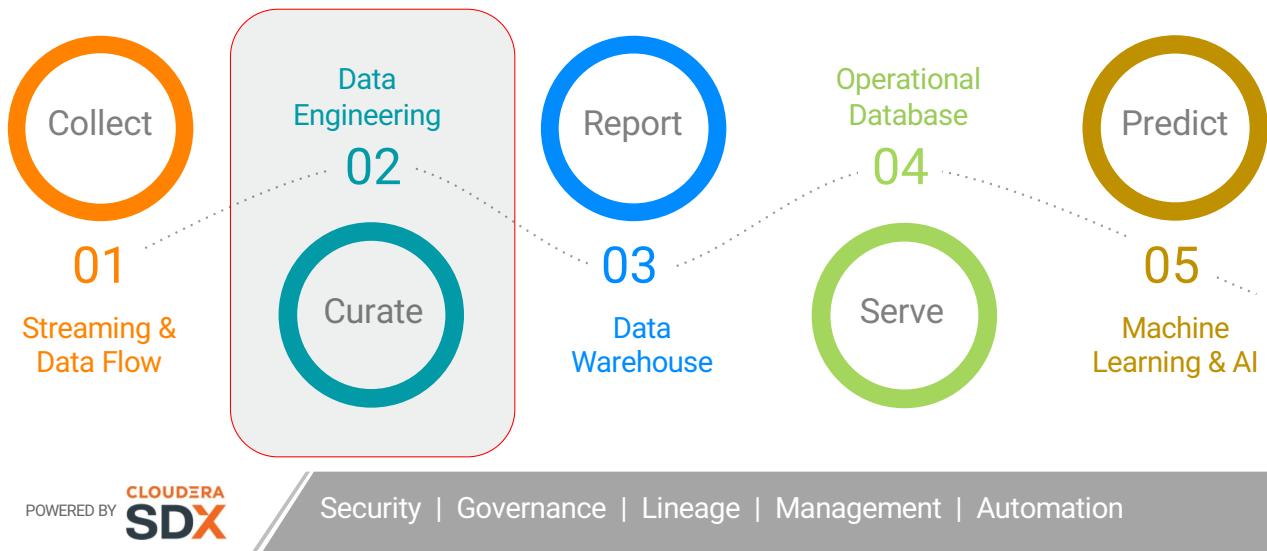


Secure & Fast Delivery

In talking with customers to understand their data engineering pain point we see some common themes

Should consider scheduling?

Data Engineering within the Data Lifecycle



An Integrated, Purpose-Built Experience for Data Engineers



OPTIMIZED FOR DATA ENGINEERS

Streamlined service for scheduling, monitoring, debugging, and promoting data pipelines quickly & securely.



EVERYTHING YOU NEED TO POWER ANALYTICS

- Inherited governance
- Deliver data pipelines to CDW, CML, or COD easily
- Portable and flexible



COMPLETE DATA PIPELINE MANAGEMENT

- Monitoring & alerting for catching issues early
- Visual troubleshooting
- Governed and secure workflows with SDX

Not only about adoption, but driving consumption -reduce barriers to moving workloads to CML

An Integrated, Purpose-Built Experience for Data Engineers



CONTAINERIZED, MANAGED SPARK SERVICE

- Autoscaling compute
- Governed & secure with Cloudera SDX
- Mix version deployments

APACHE AIRFLOW SCHEDULING

- Open preferred tooling
- Orchestrate complex data pipelines
- Manage & schedule dependencies easily

TUNING & VISUAL TROUBLESHOOTING

- Resolve issues fast with real-time visual performance profiling
- Complete monitoring & alerting capabilities

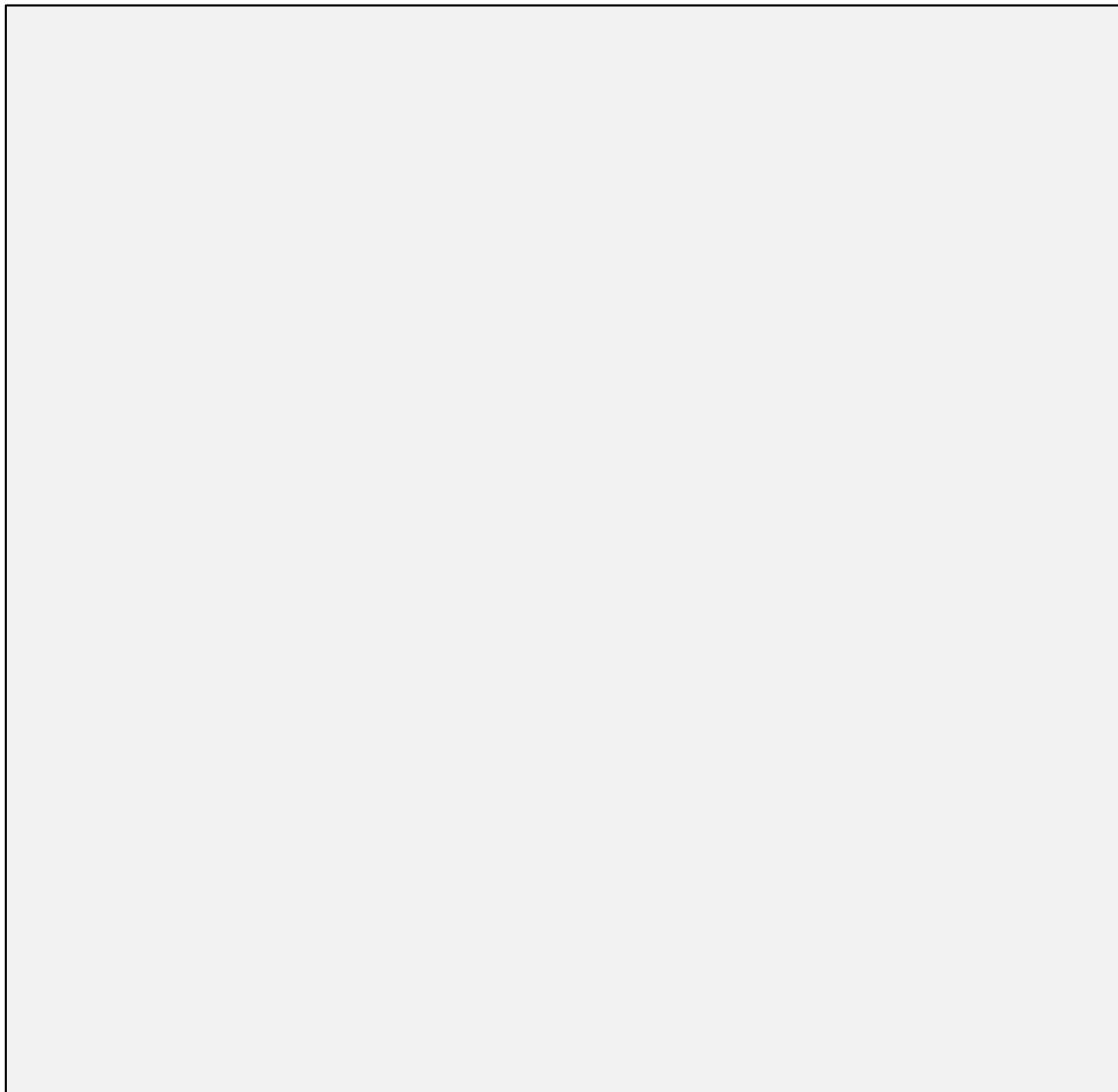
SIMPLIFIED JOB MANAGEMENT & APIS

- Full lifecycle mgmt.
- API-driven pipeline automation for any service
- Any language: SQL, Java, Scala, Python

Not only about adoption, but driving consumption -reduce barriers to moving workloads to CML

Introduction to Zeppelin

Chapter 3



Course Chapters

- Introduction
- Why Data Engineering
- Introduction to Zeppelin**
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

Hello and welcome to this series of short videos about Zeppelin.
My name is François Reynald and I am a senior training engineer with Cloudera.

But before we dive into Zeppelin let's take a step back and think about what are the unique features of notebook systems that make them so pervasive in our new data driven world.

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

Notebook systems were implemented to provide data scientists the tools to be able to perform reproducible data analysis with paragraphs that could either be text, code or visualizations.

Today there are many notebook systems available, the most popular ones are often associated with a particular language:

- RStudio for R,
- Jupyter for Python,
- Zeppelin for Spark.

The good news is that all of the above can be used as IDEs within our Cloudera Data Science Workbench.

Chapter Topics

Introduction to Zeppelin

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

In the Zeppelin documentation, notebooks are called notes but everywhere else they are referred to as notebooks so I will try and use the official term but may occasionally slip and use the common term.

What is Apache Zeppelin?

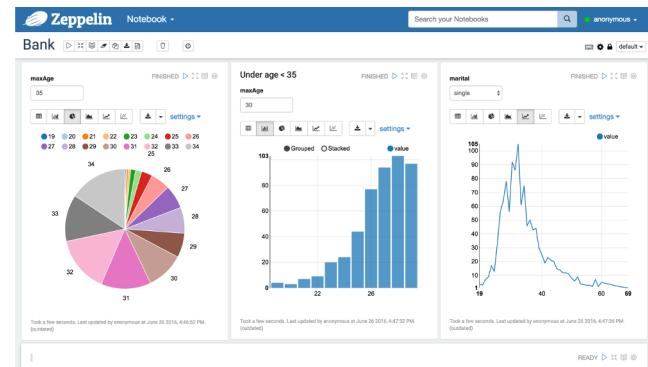


■ An open source project from the Apache Software Foundation

- 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



As many Apache Software projects, the Zeppelin project was initiated in a private company in 2013 before being incubated in 2014 and becoming a Top Level project in 2016.

Today Zeppelin is has reached version 0.9 and defines itself as 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

The screenshot shows the 'Interpreter binding' section of the Zeppelin note settings. It displays a list of available interpreters: spark (%spark (default), %sql, %pyspark, %ipyspark, %r, %ir, %shiny, %kotlin), md (%md), sh (%sh, %sh.terminal), and angular (%angular, %angular.ng). The 'spark' entry is highlighted with a blue background, indicating it is the default interpreter. Below the list are 'Save' and 'Cancel' buttons.

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 of 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

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



The latest version of Zeppelin contains over 30 interpreters some of which are shown on the right hand side of this slide

In CDP DC the following interpreters are available by default:

- angular
- livy
- md

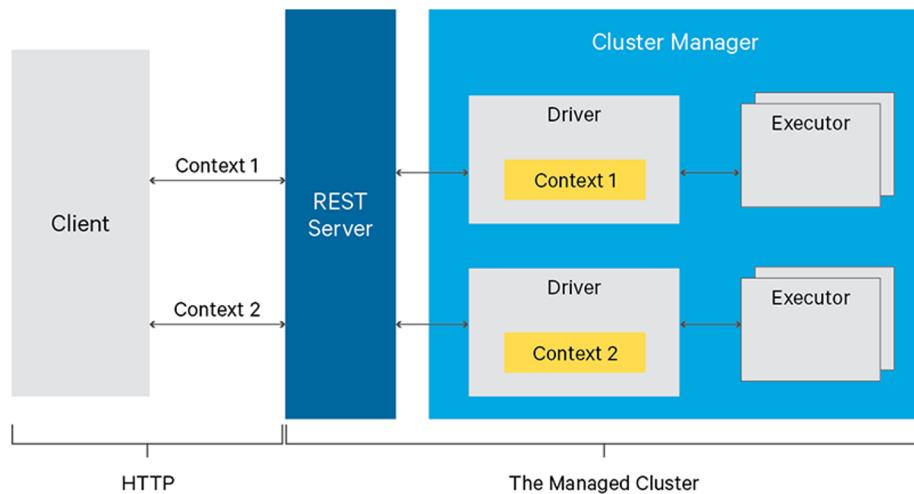
The default one is livy

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



So what is Livy?

This is what the livy project home page will tell you.

Apache Livy is a service that enables easy interaction with a Spark cluster over a REST interface. It enables easy submission of Spark jobs or snippets of Spark code, synchronous or asynchronous result retrieval, as well as Spark Context management, all via a simple REST interface or an RPC client library.

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

Bear in mind that CDP DC is secured by design so it makes sense to use Livy as a broker between Zeppelin and the Spark cluster.

Note Formatting

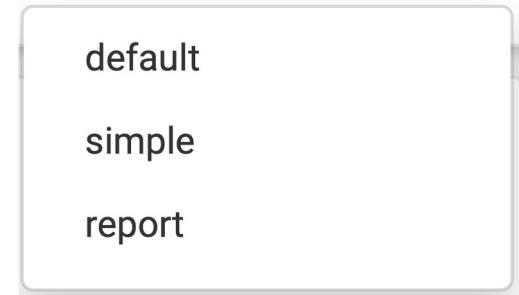
- Note owners can control all paragraphs at the note level, including:

- 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



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

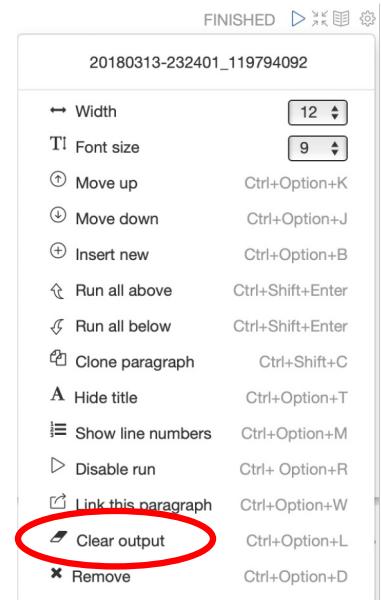
- Hide/Show all code via the button labeled "Show/hide the code,"
- Hide/Show all output via the button labeled "Show/hide the output" (which changes from an open book to a closed book icon based on the current setting), and
- Clear all output via the eraser icon button labeled "Clear output."

There are also two additional note views. The Simple view removes the note-level controls at the top of the note. The Report view removes all note-level controls, as well as hides all code in the note, resulting in a series of outputs. These views can be selected by clicking the button labeled "default" at the top-right corner of the note and then choosing the appropriate option from the resulting drop-down menu.

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)

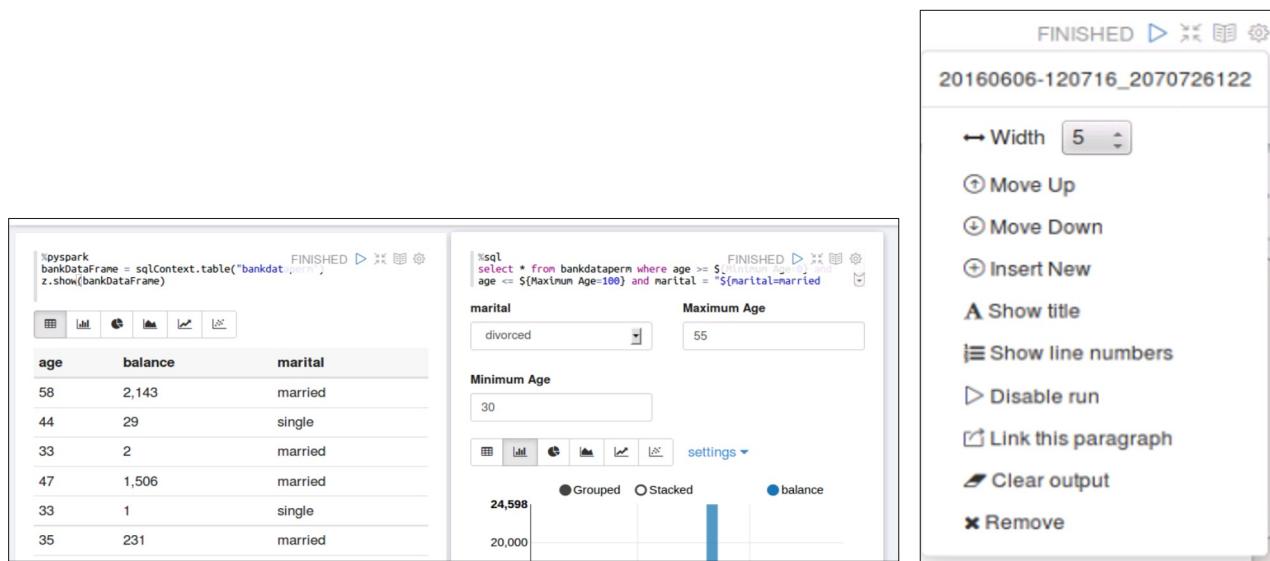


Paragraphs can also be formatted prior to distribution on an individual basis. These settings are available in the buttons menu at the top right of each paragraph, as well as underneath the settings menu (gear icon) button. Formatting options that were also available at the note level include:

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

Paragraph Enhancement - Width

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



The visual appearance of paragraphs can also be enhanced to support various collaboration goals. For example, assume you want to create a dashboard within a Zeppelin note, showing multiple views of the same data on the same line. This can be accomplished by modifying the Width setting, found in paragraph settings. By default, the maximum width is used per paragraph, however, this can be modified so that two or more paragraphs will appear on the same line.

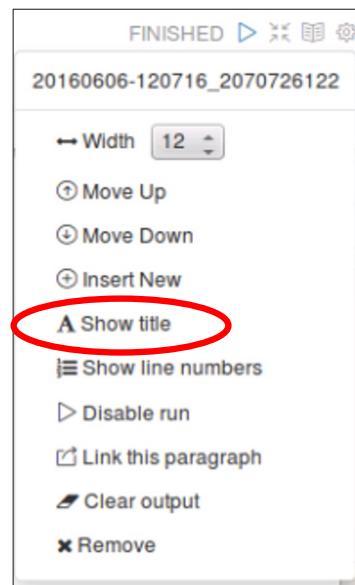
Paragraph Enhancement - Show Title

- Paragraph titles can be added for clarity

Untitled FINISH

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

marital	Maximum Age
Married	100

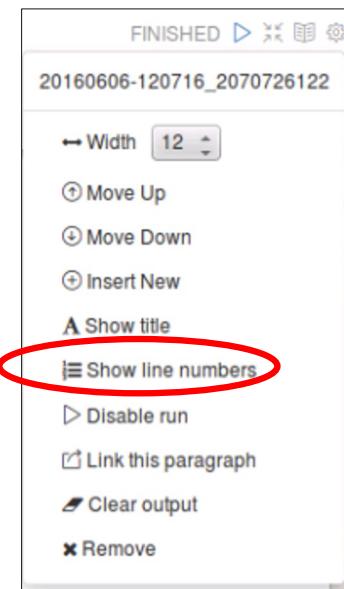


Paragraphs can be given titles for added clarity when viewing output. To set a title, under paragraph settings select Show title. The default title is "Untitled." Click on the title to change it, type the new title, and press the Enter key to set it.

Paragraph Enhancement - Line Numbers

- Line numbers can be added to paragraph code

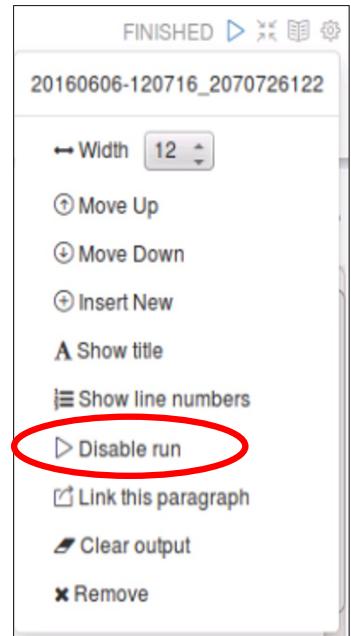
```
1 %sql  
2 select * from bankdataperm where age >= ${Min Age}  
and age <= ${Maximum Age=100} and marital =  
  
marital                         Maximum Age
```



Paragraphs displaying code can also be enhanced by showing the line numbers for each line of code. To turn this feature on, under paragraph settings select Show line numbers. The numbers will appear to the left of the code lines. Lines that are wrapped based on the width of the paragraph will only be given a single number, even though on the screen they will appear as multiple lines.

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



In some cases, you may want to freeze a paragraph's output in its current form. To do so, under paragraph settings select the Disable run option. Any changes to the code, as well as changes to dynamic forms input, will not change the output presented as long as the Disable run option is selected.

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**

- 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

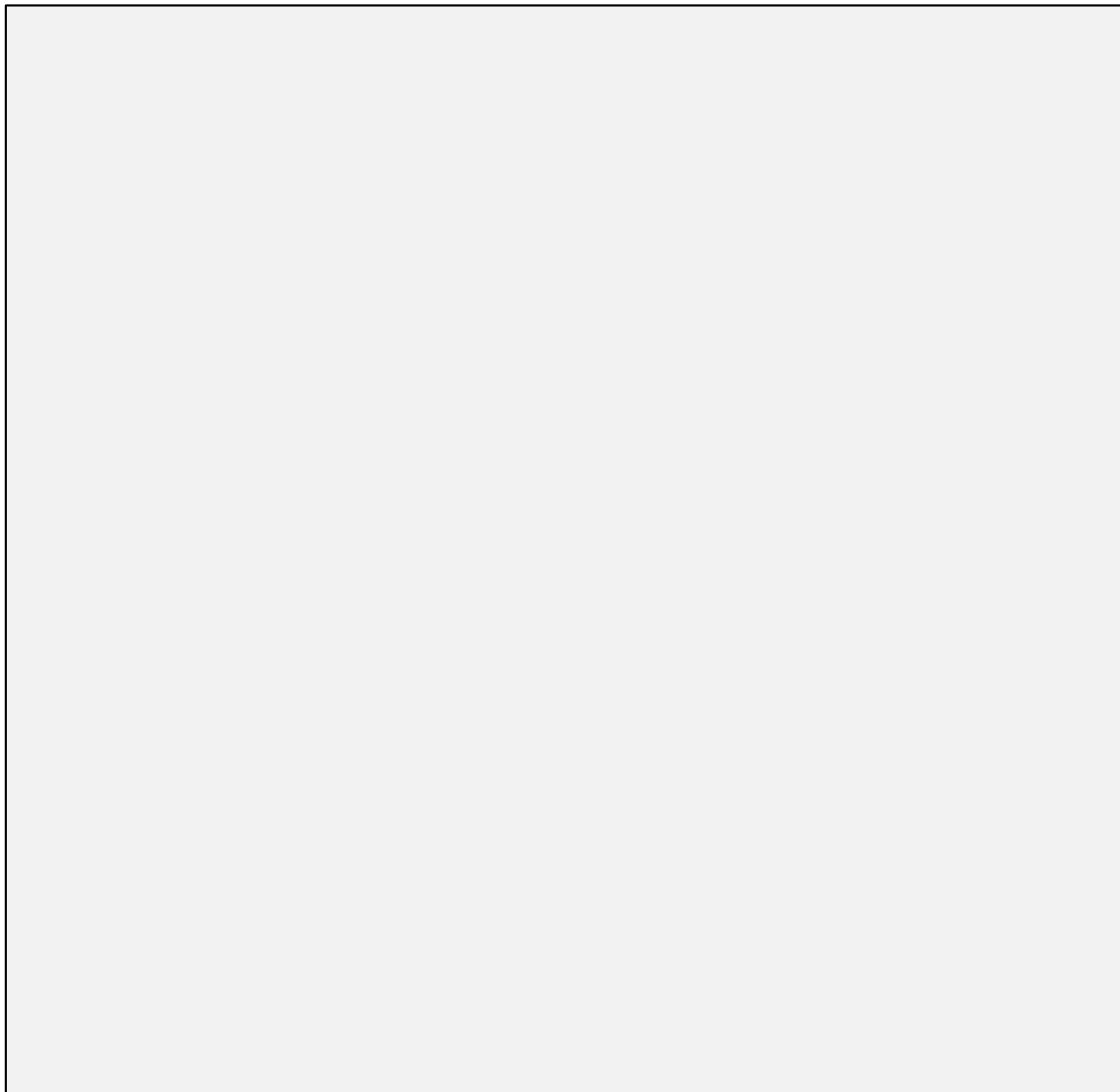
Chapter Topics

Introduction to Zeppelin

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

HDFS Introduction

Chapter 4



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction**
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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?



Hadoop
Client

"I have a 200 TB file
that I need to store."



HDFS

What is HDFS?



Hadoop
Client

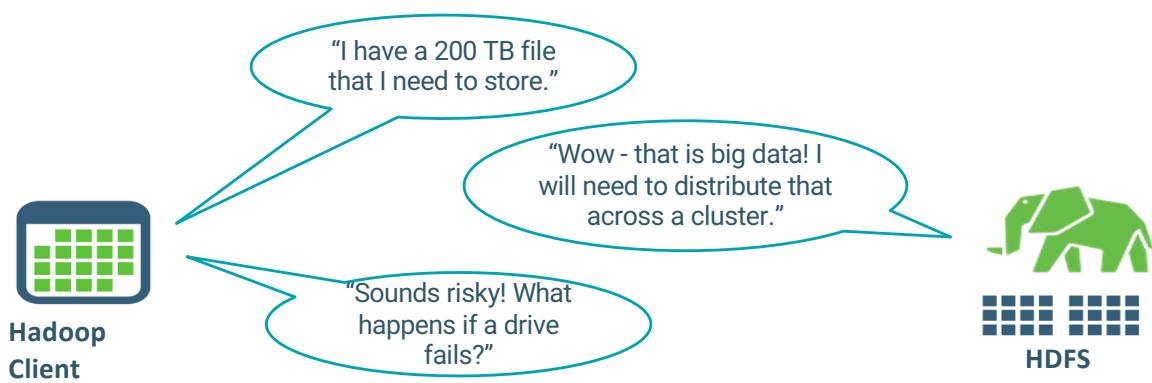
"I have a 200 TB file
that I need to store."

"Wow - that is big data! I
will need to distribute that
across a cluster."



HDFS

What is HDFS?



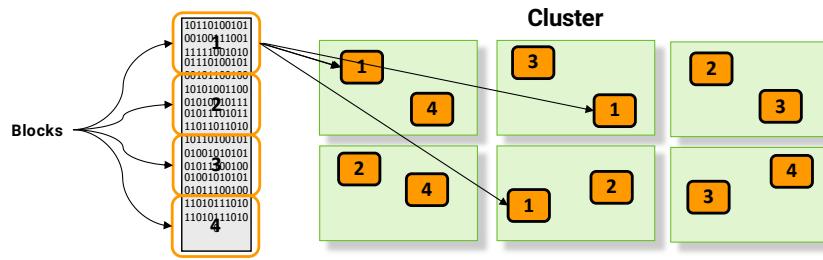
What is HDFS?



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.
- Search: Search File Names
- Table Headers: Name, Size, Last Modified, Owner, Group, Permission
- Table Data:
 - ..
 - 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--)
- Terminal Output:

```
lmartin — it1@sandbox:~ — ssh root@127.0.0.1 -p 2222 — 106x24
[[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 ~]$ ]$
```

Just pointing out that common tools such as a web-based file system browser and basic CLI represent the basic expectations of a file system

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

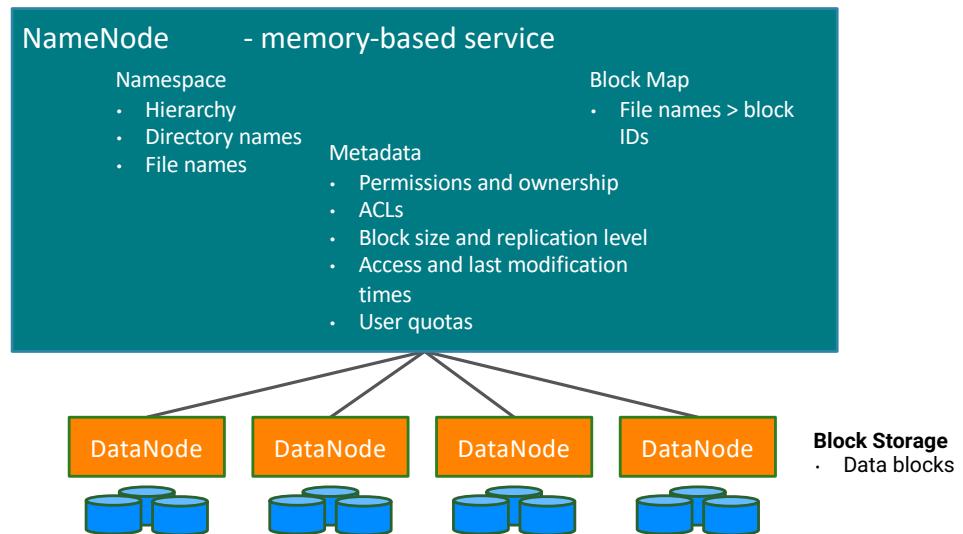
- **NameNode**
 - Is the master service of HDFS
 - Determines and maintains how the chunks of data are distributed across the DataNodes

- **DataNode**
 - Stores the chunks of data, and is responsible for replicating the chunks across other DataNodes

- A NameNode represents a single namespace
 - Data never resides on the NN, only on the DNs
 - Default block size in HDP is 128MB
 - Default replication factor is 3

HDFS Architecture

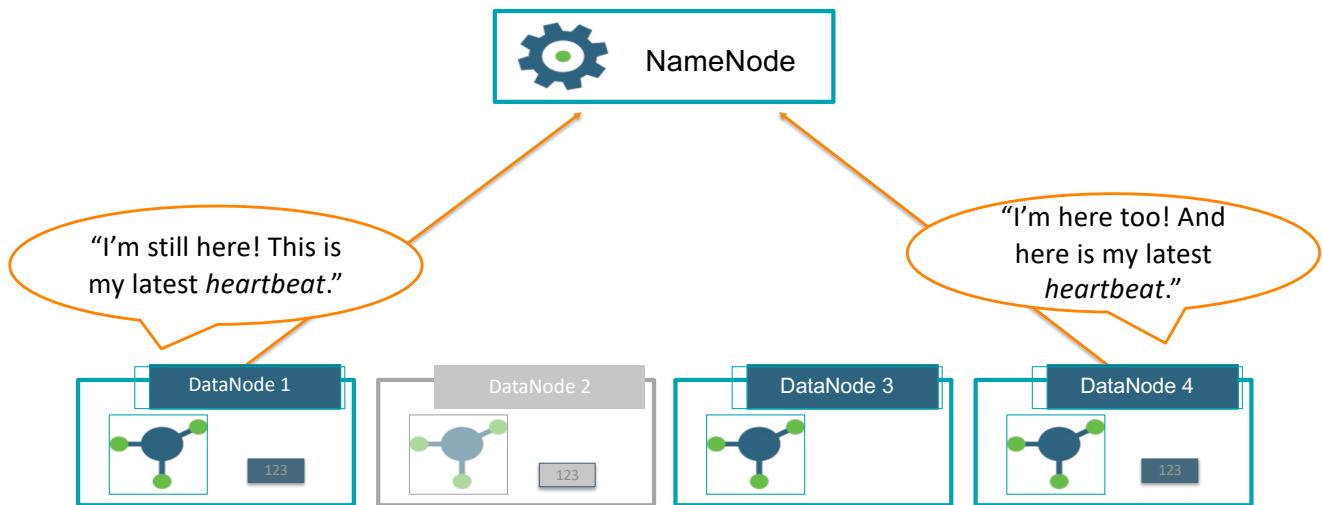
- The NameNode (master node) and DataNodes (worker nodes) are daemons running in a Java virtual machine.



The NameNode and DataNode are components of the HDFS service. The NameNode is an HDFS master component while a DataNode is an HDFS worker component. The NameNode and DataNode are implemented as daemons running inside a Java virtual machine.

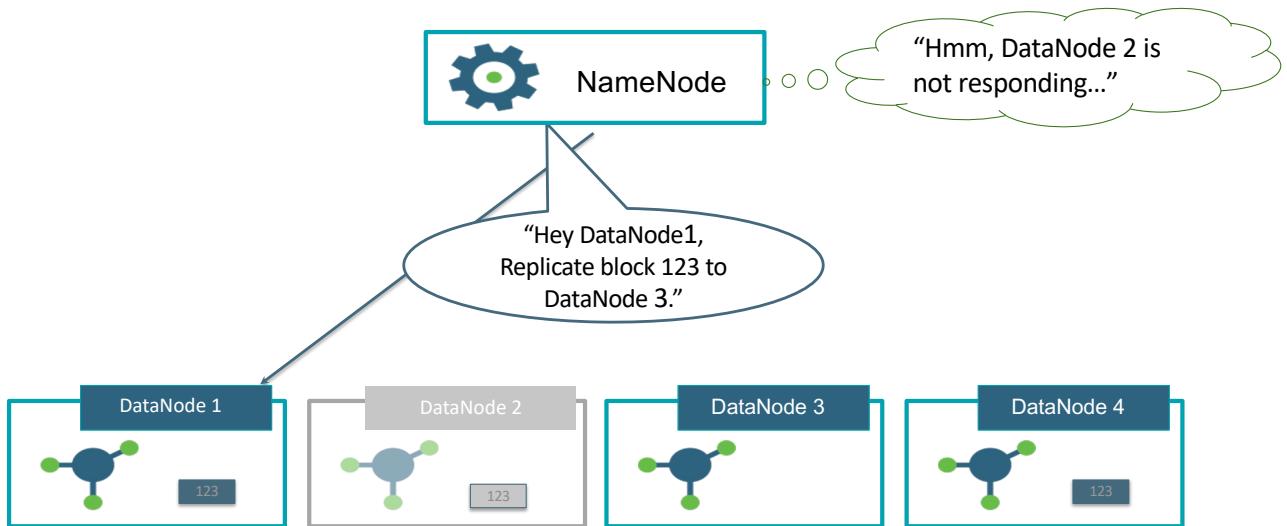
The NameNode maintains critical HDFS information. To enhance HDFS performance, it maintains and serves this information from memory. The memory-based information includes namespace information, metadata information, journaling information, and a block map information. Because the NameNode maintains all file system state information in memory, it is critical to ensure that the NameNode has sufficient memory.

The DataNodes



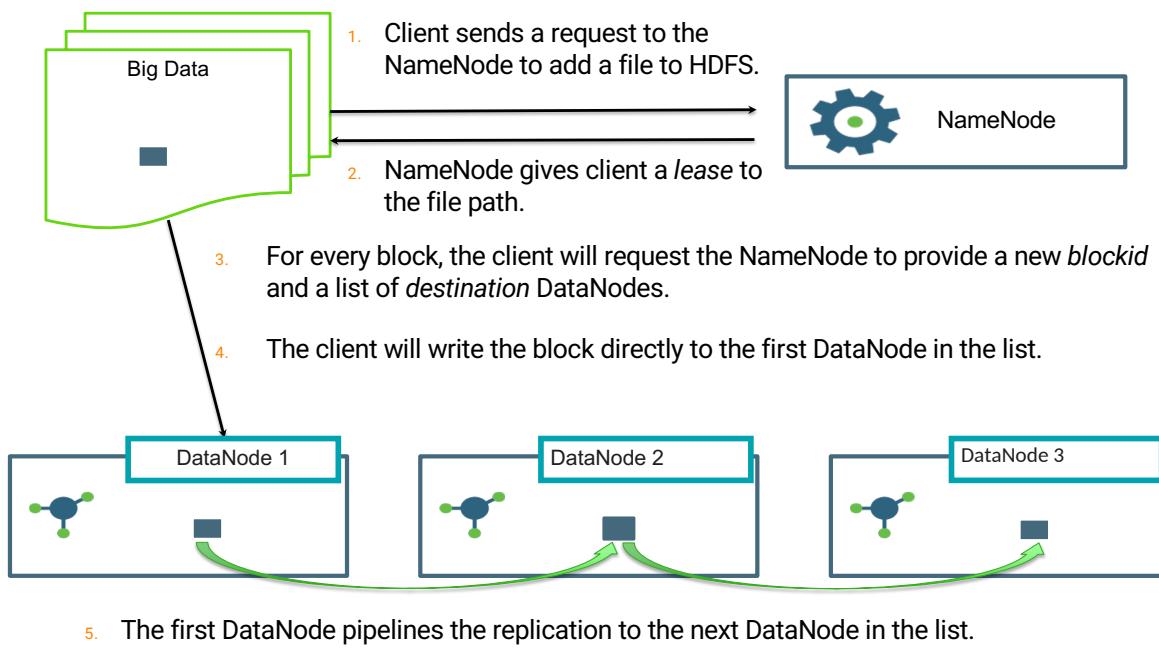
NN is responsible for resolving any missing or corrupted blocks

The DataNodes



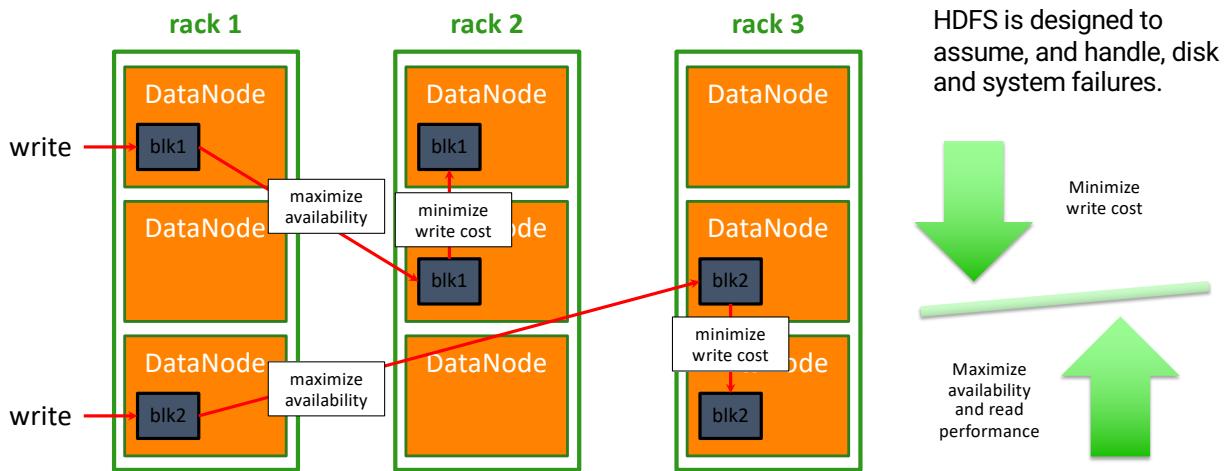
NN is responsible for resolving any missing or corrupted blocks

Writing to HDFS



Client does NOT send actual data through the NN to allow it to scale.

Replication and Block Placement



HDFS is designed to assume that disk, system, and network failures will occur. As a result, HDFS is also designed to automatically and transparently handle disk failures. It does this by automatically replicating data across different DataNodes.

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**

In Hadoop prior to version 2.0, the NameNode was a single point of failure. The entire cluster would become unavailable if the NameNode failed or became unreachable. Even maintenance events such as software or hardware upgrades on the NameNode machine would result in periods of cluster downtime.

The HDFS NameNode High Availability (HA) feature eliminates the NameNode as a single point of failure. It enables a cluster to run redundant NameNodes in an Active/Standby configuration.

NameNode HA enables fast failover to the Standby NameNode in response to a failure, or a graceful administrator-initiated failover for planned maintenance.

There are two ways of configuring NameNode HA. Using the Cloudera Manager Web UI is the easiest way. Manually editing the configuration files and starting or restarting the necessary daemons is also possible. However, manual configuration of NameNode HA is not compatible with Cloudera Manager administration. Any manual edits to the `hdfs-site.xml` file would be overwritten by information in the Cloudera Manager database when the HDFS service is restarted.

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

NOTE: Ranger discussed in a future module

Chapter Topics

HDFS Introduction

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

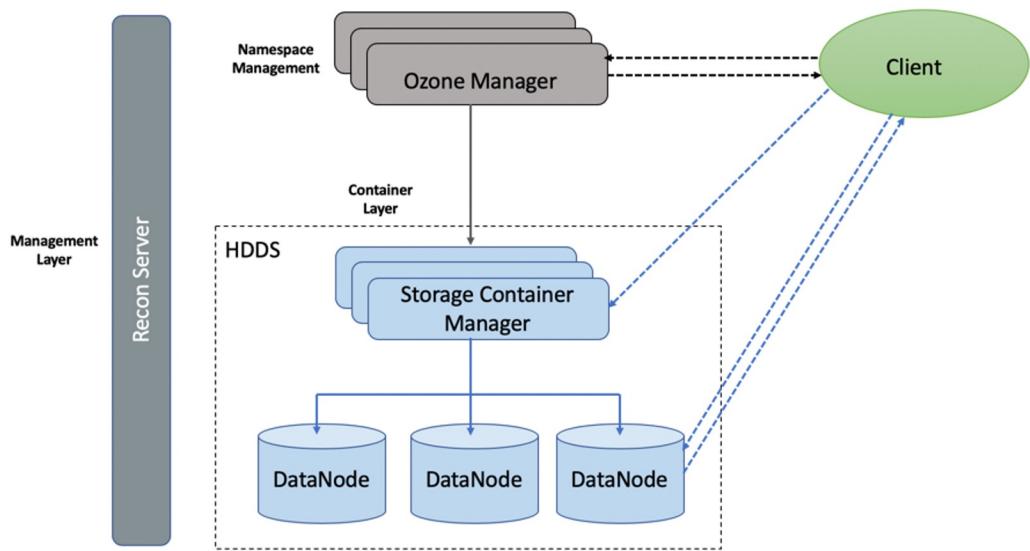
Ozone Overview

- **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

BlockName 1	... BlockData ...
BlockName 2	... BlockData ...
BlockName N	... BlockData ...

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.

Answers

- 1 - Blocks, replicas/copies, reliability/robustness, data locality
- 2 - NameNode
- 3 - DataNode
- 4 - True. This allows the NameNode to be so highly-available
- 5 - False. Clients write to the first DataNode in the list created by the NameNode and the DN's then "pipeline" the data writing to the additional DN's

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

This summary page lists some of the main points from this lesson.

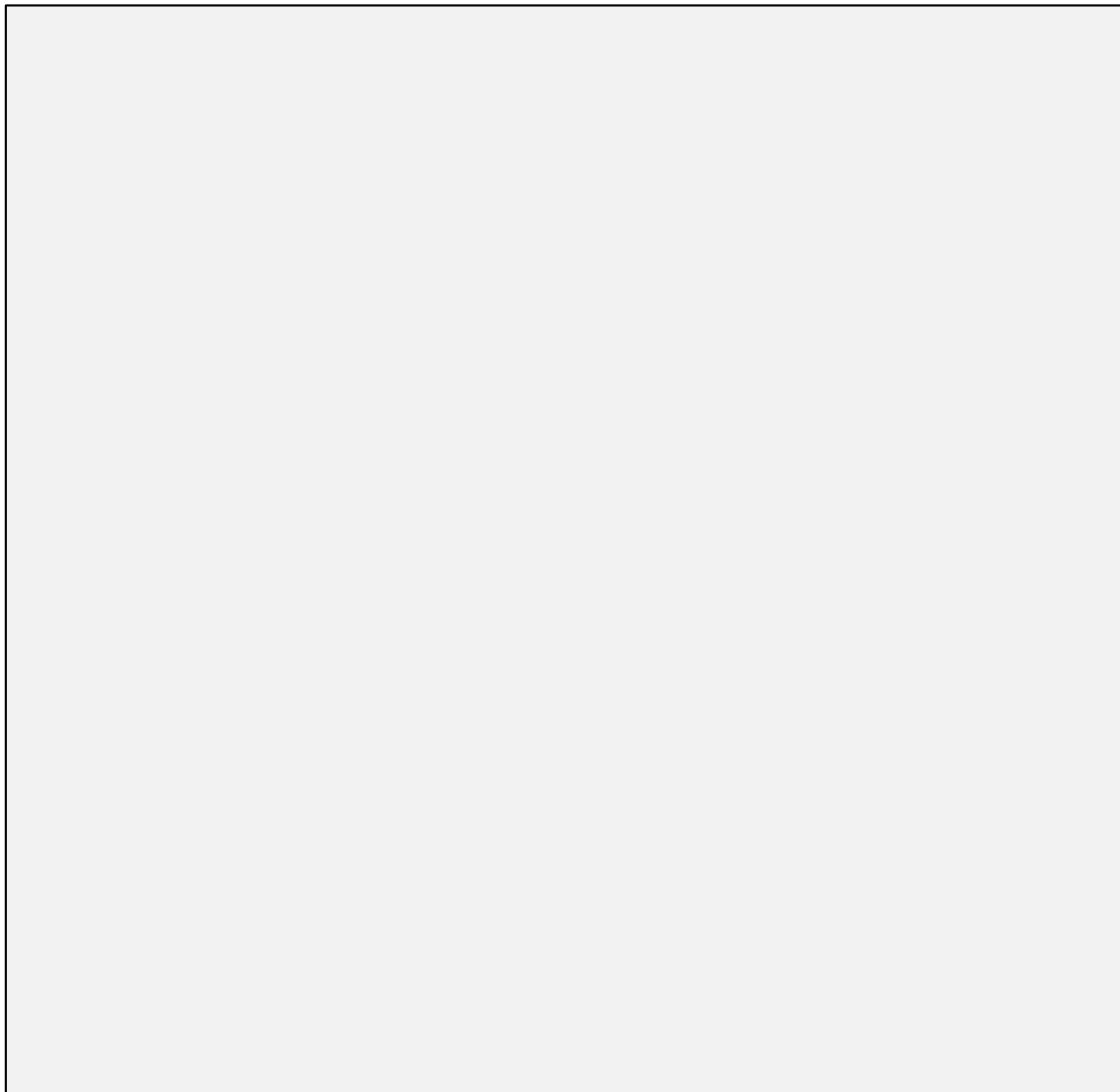
Chapter Topics

HDFS Introduction

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

YARN Introduction

Chapter 5



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- **YARN Introduction**
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

BACKGROUND INFO FOLLOWS...

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

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

Definition and Discussion – Multitenancy: Software multitenancy is achieved when a single instance of an application serves multiple groups of users, or “tenants.” Each tenant shares common access to an application, hardware, and underlying resources (including data), but with specific and potentially unique privileges granted by the application based on their identification. This is in contrast with multi-instance architectures, where each user gets a unique instance of an application, and the application then competes for resources on behalf of its tenant. A typical example of a multitenant application architecture would be SaaS cloud computing, where multiple users and even multiple companies are accessing the same instance of an application at the same time (for example, Salesforce CRM). A typical example of a multi-instance architecture would be applications running in virtualized or IaaS environments (for example, applications running in KVM virtual machines).

Note: In prior versions of Hadoop, resource management was part of the MapReduce process. In this scenario, you had a single application handling both job scheduling and running data processing jobs at the same time. Starting with Hadoop 2.0, MapReduce is simply another data processing application running on top of the YARN framework.

YARN Architectural Components

▪ Resource Manager

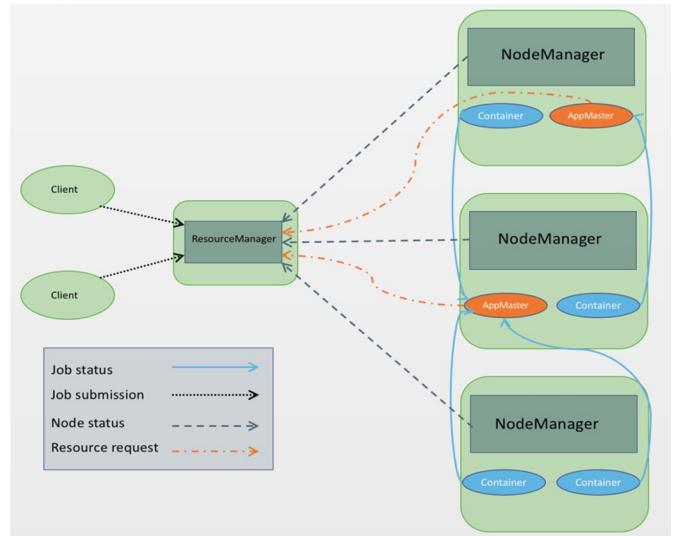
- Global resource scheduler
- Hierarchical queues

▪ 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



We will review these components more in the coming slides, two key points from this slide and the last

- M/R framework was decomposed to generalize resource management from a single workload type (MR)
- The ApplicationMaster allows clusters to scale beyond 2000-4000 node range due to the decentralization of keeping track of independent jobs

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

Container

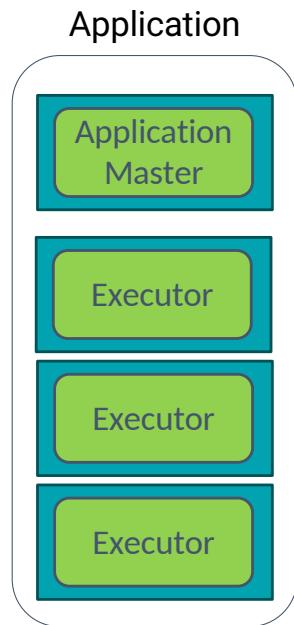
▪ ApplicationMaster (AM)

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

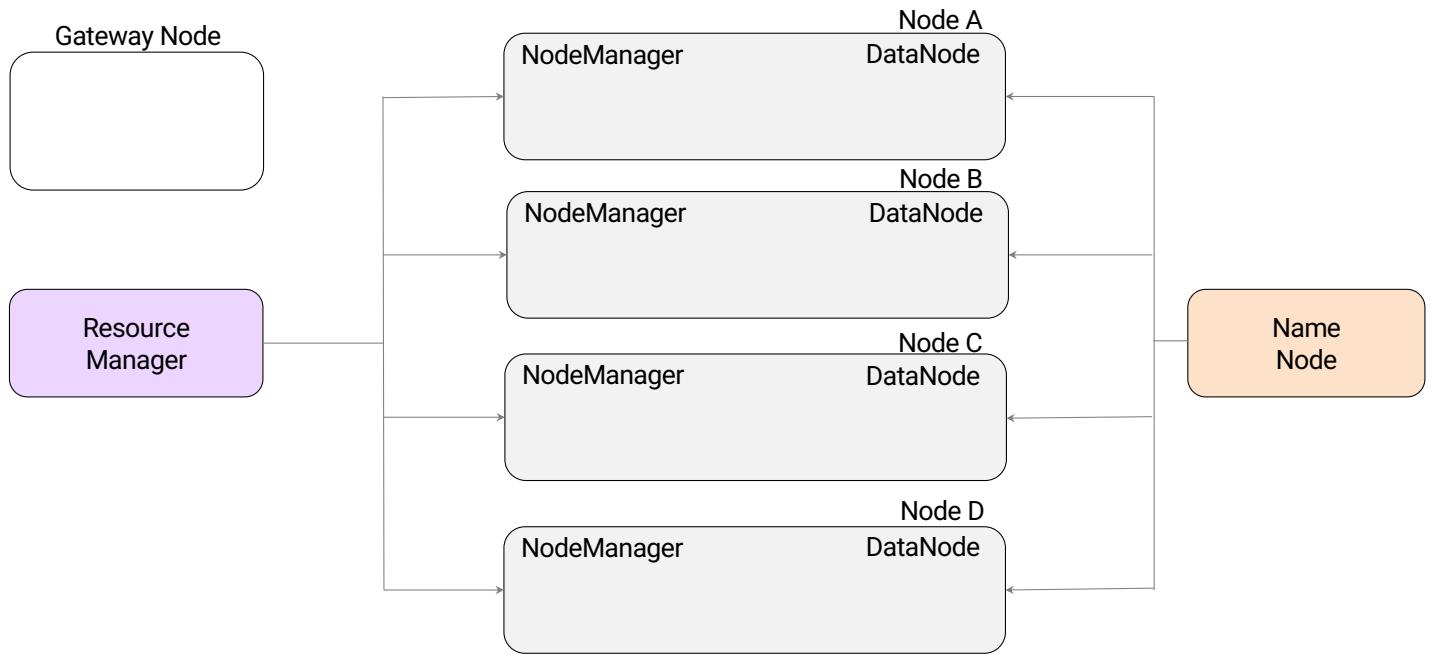
Application
Master

Applications on YARN (2)

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



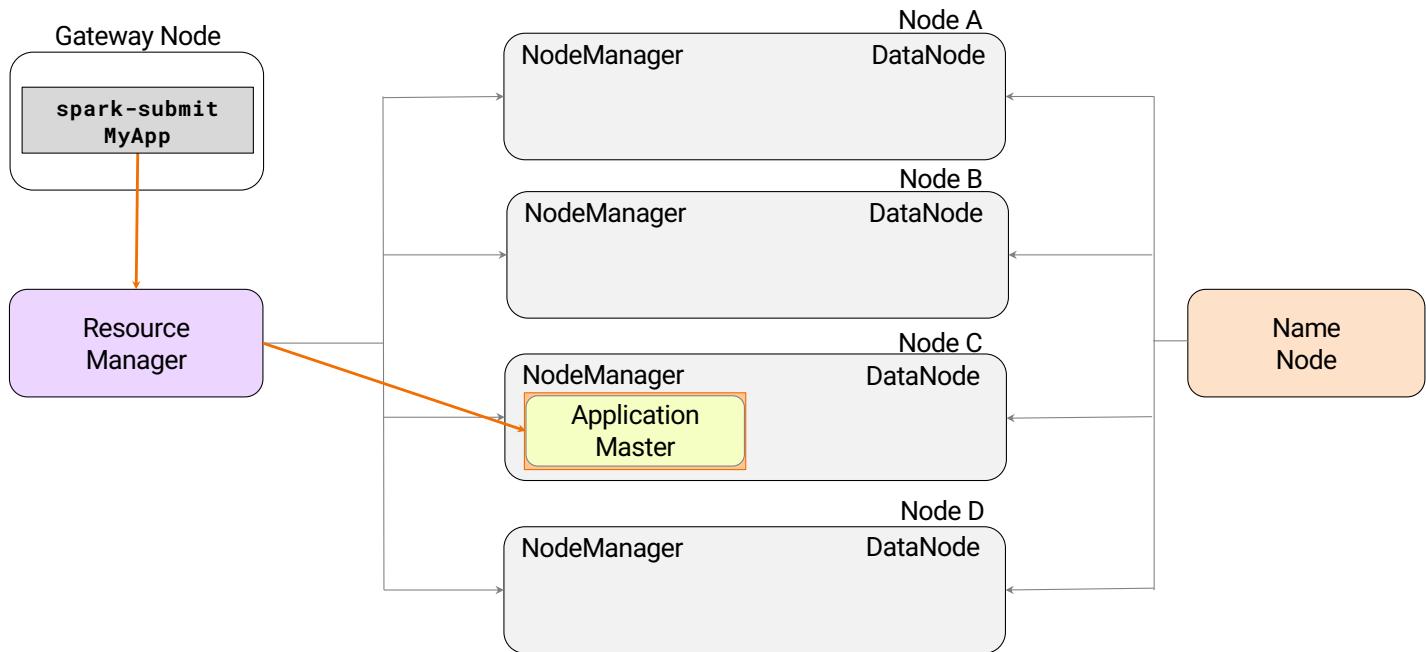
Running an Application on YARN (1)



START BUILD

Here's a typical YARN cluster.

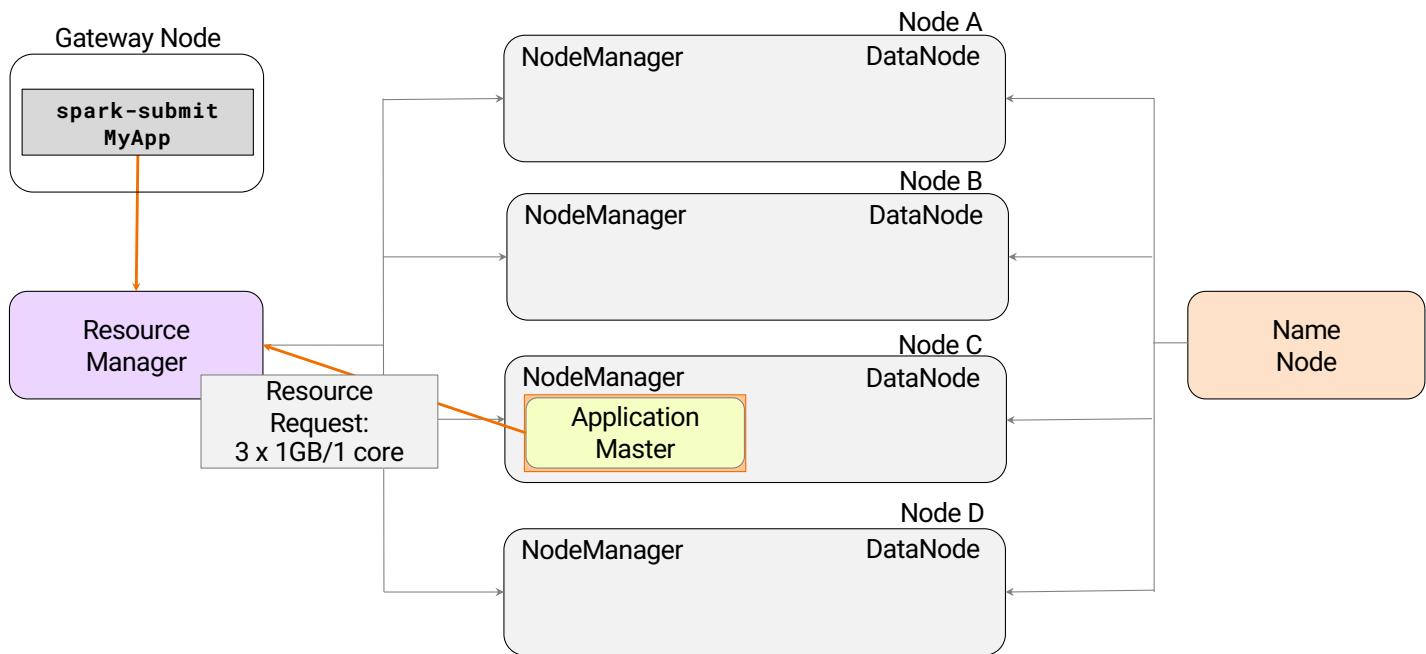
Running an Application on YARN (2)



Let's look at how the YARN application lifecycle applies works.

1. When the user submits an application, the Hadoop client will do the following:
 - a. Copy job information to HDFS—application jar file, configuration, and anything for the distributed cache
 - b. Submit the job to the ResourceManager using the YARN Client Protocol submitApplication()
2. When the ResourceManager gets the submitApplication call, it will do the following:
 - a. Tell the scheduler to allocate a container for the ApplicationMaster
 - b. Launch the new container to run the Application Master (AM) (MRAppMaster or SparkAppMaster)

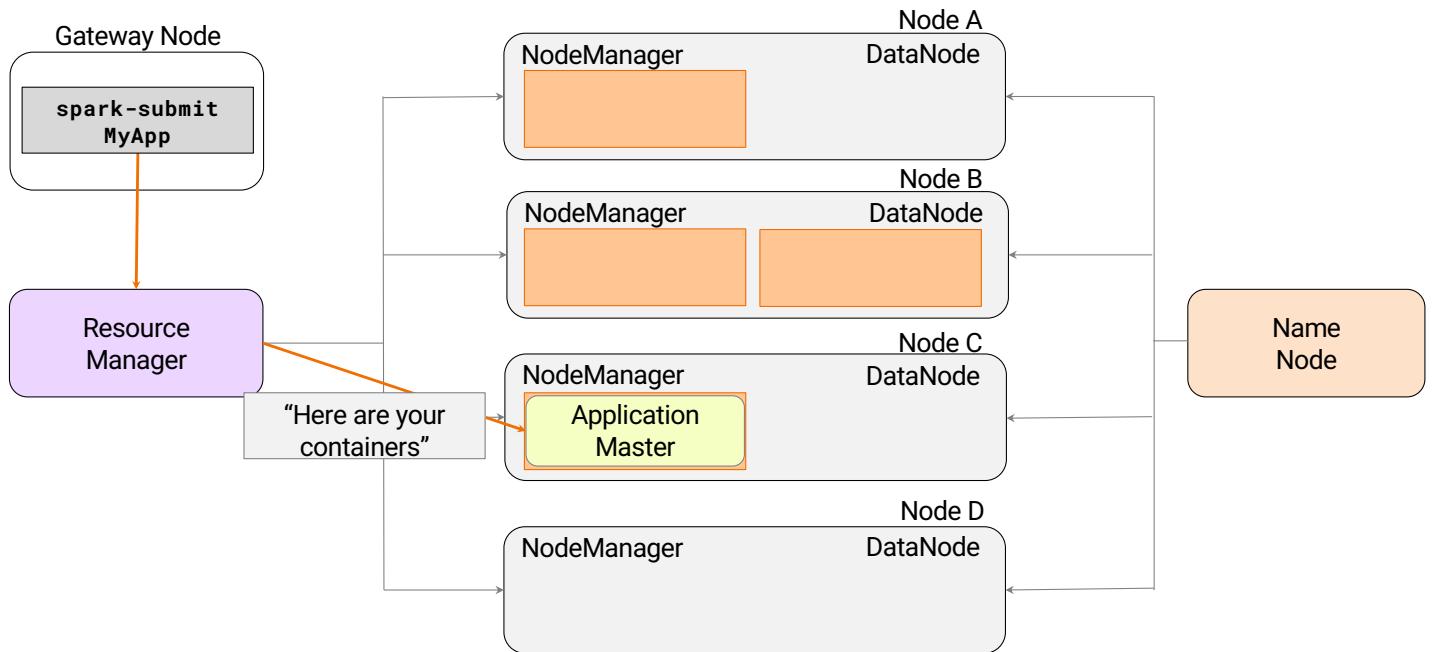
Running an Application on YARN (3)



How many containers the AM asks for depends on the specifics of the application. In this example, perhaps our application is Spark, and the `spark.executor.instances` property has been set to three. If it's a MapReduce application, it might be reading a file with three HDFS blocks. In any case, the AM must know how many containers to request. The resources for the containers are also configurable.

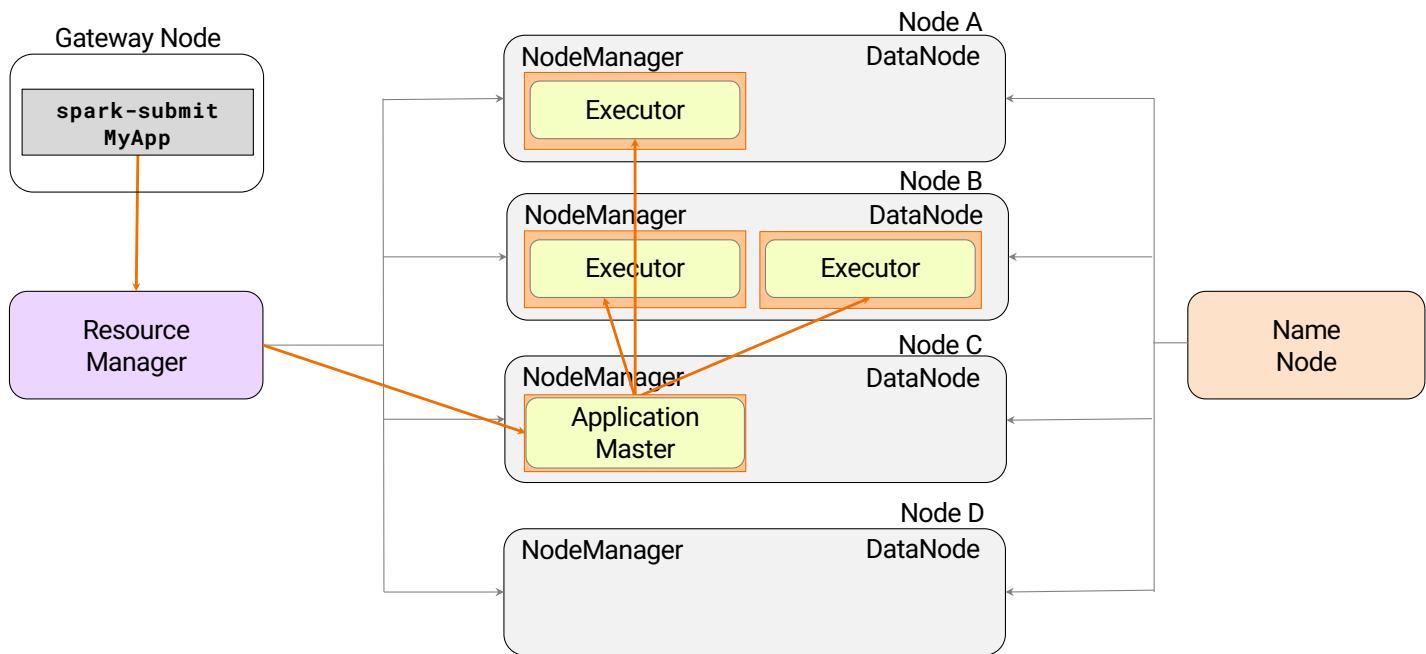
Note that this diagram shows containers being requested in order to run tasks in a distributed way, which is usually how it works.

Running an Application on YARN (4)



The ResourceManager will allocate the number of containers requested.

Running an Application on YARN (5)



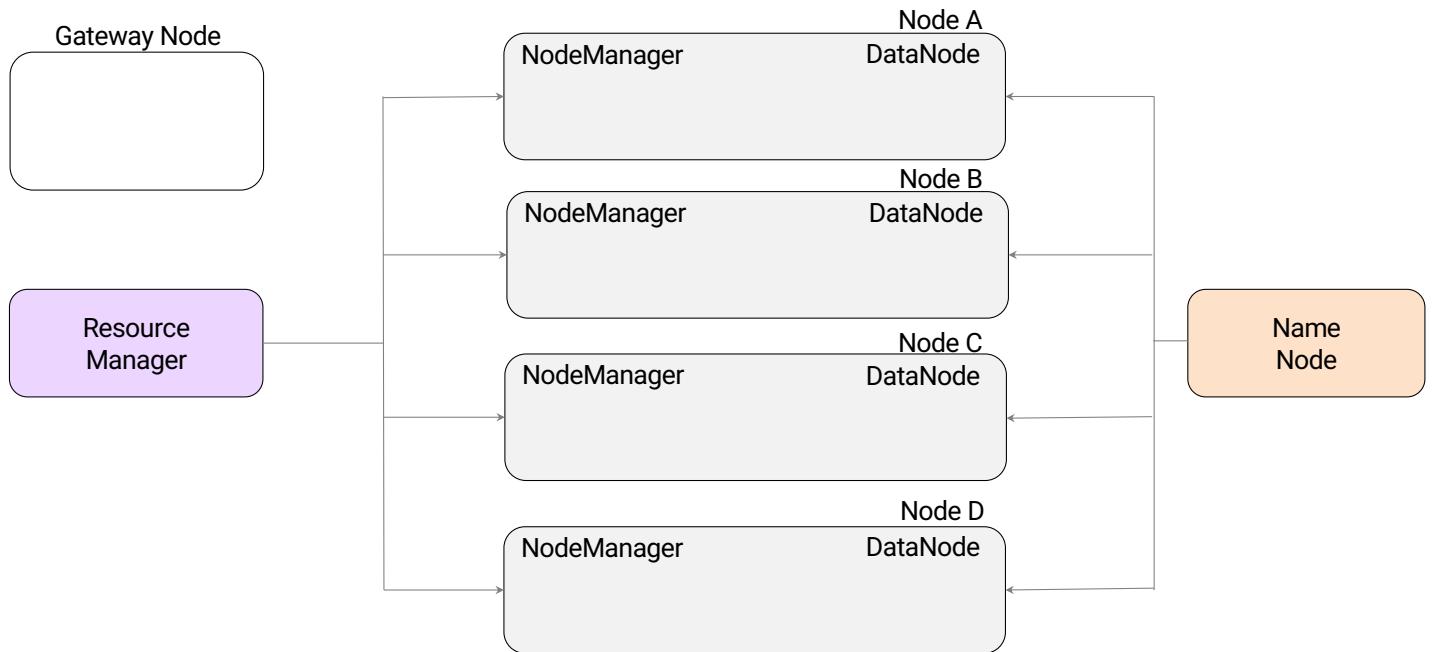
The ApplicationMaster will instruct the NodeManager to launch tasks in the containers by contacting the NodeManager and passing a LaunchContext.

The NodeManager will start a new JVM. (JVM reuse is not supported in YARN.) While running, the application task will communicate its status back to the AM.

The AM aggregates the status from the various tasks.

As we will see later, the AM can provide an application-specific web UI (in addition to the generic YARN ones provided by RMs and NMs). Both Spark ApplicationMaster and MapReduce ApplicationMaster provide web UIs with information specific to that framework. The Spark ApplicationMaster provides the Spark Application web UI which will be covered later in the course.

Static Resource Allocation (6): Application Terminates



END BUILD

When an application is complete, the ApplicationMaster notifies the RM, which frees the containers.

Chapter Topics

YARN Introduction

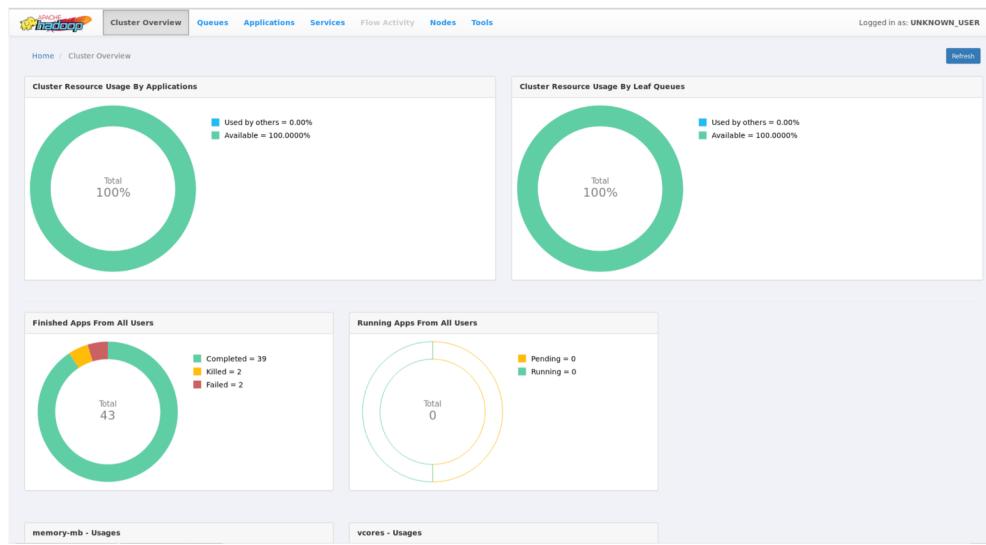
- YARN Overview
- YARN Components and Interaction
- **Working with YARN**
- Exercise: Working with YARN

Working with YARN

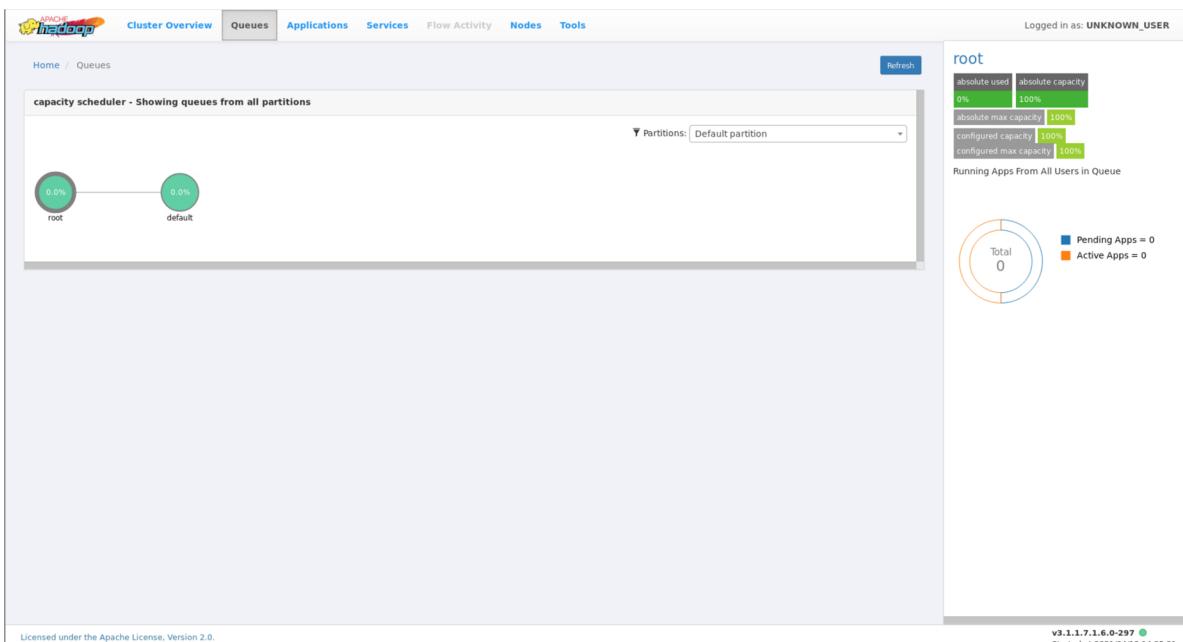
- **Developers need to be able to**
 - Submit jobs (applications) to run on the YARN cluster
 - Monitor and manage jobs
- **YARN tools for developers**
 - The YARN web UI
 - The YARN command line interface

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

This screenshot shows the Cloudera YARN Web UI Applications page. The page header includes the Cloudera logo, Cluster Overview, Queues, Applications, Services, Flow Activity, Nodes, Tools, and a user logged in as UNKNOWN_USER. The main content area displays a table of completed applications (status FINISHED). The table columns are Application ID, Application Type, Application Name, User, State, Queue, Progress, Start Time, and Elapsed Time. The table contains 37 rows of data, each representing a completed SPARK application named 'livy-session-n' for various session numbers (11 to 271). The progress for all applications is 100%, and the start time ranges from April 14, 2021, to April 15, 2021. The elapsed times are mostly under 1 minute, with one notable entry at 3m 35s 419ms.

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

The YARN Web UI - Nodes

The screenshot shows the YARN Web UI - Nodes page. At the top, there is a navigation bar with links: Cluster Overview, Queues, Applications, Services, Flow Activity, Nodes (which is highlighted in grey), and Tools. To the right of the navigation bar, it says "Logged in as: UNKNOWN_USER". Below the navigation bar, there is a breadcrumb trail: Home / Nodes. On the right side of the header, there is a "Refresh" button.

The main content area has three tabs: Information (which is selected and highlighted in blue), Node Status, and Nodes Heatmap Chart. Below the tabs is a search bar with a "Search" button and a dropdown menu set to "25 Rows".

On the left, there are two filter panels:

- Node Label (1)**: Shows one entry: "default" with a checked checkbox and a count of 1. There is also a "More" link.
- Node State (1)**: Shows one entry: "RUNNING" with a checked checkbox and a count of 1. There is also a "More" link.

Below the filters is a "Apply" button and a "Clear" button.

The central part of the page is a table with the following columns: Node Label, Rack, Node State, Node Address, Node HTTP Address, Containers, Mem Used, Mem Available, and Vcores Used. The table contains one row:

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

At the bottom of the page, there is a footer with the text "Licensed under the Apache License, Version 2.0." and "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`

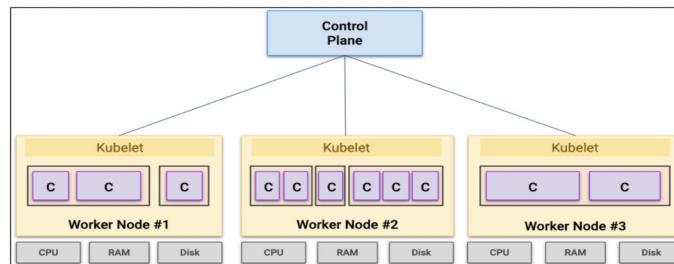
Kubernetes - K8s

- 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.

Answers

1 - master = ResourceManager and worker = NodeManager
2 - Memory
3 - False, they run on worker nodes and control a specific job
4 - The ApplicationMaster must decide what to do which is not always to spin up a replacement
5 - False, queues are given a certain percentage of the whole. Bonus answer is that "node labels" could be assigned to particular worker nodes which then can be configured for a given queue, but we didn't discuss this in the preso.

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

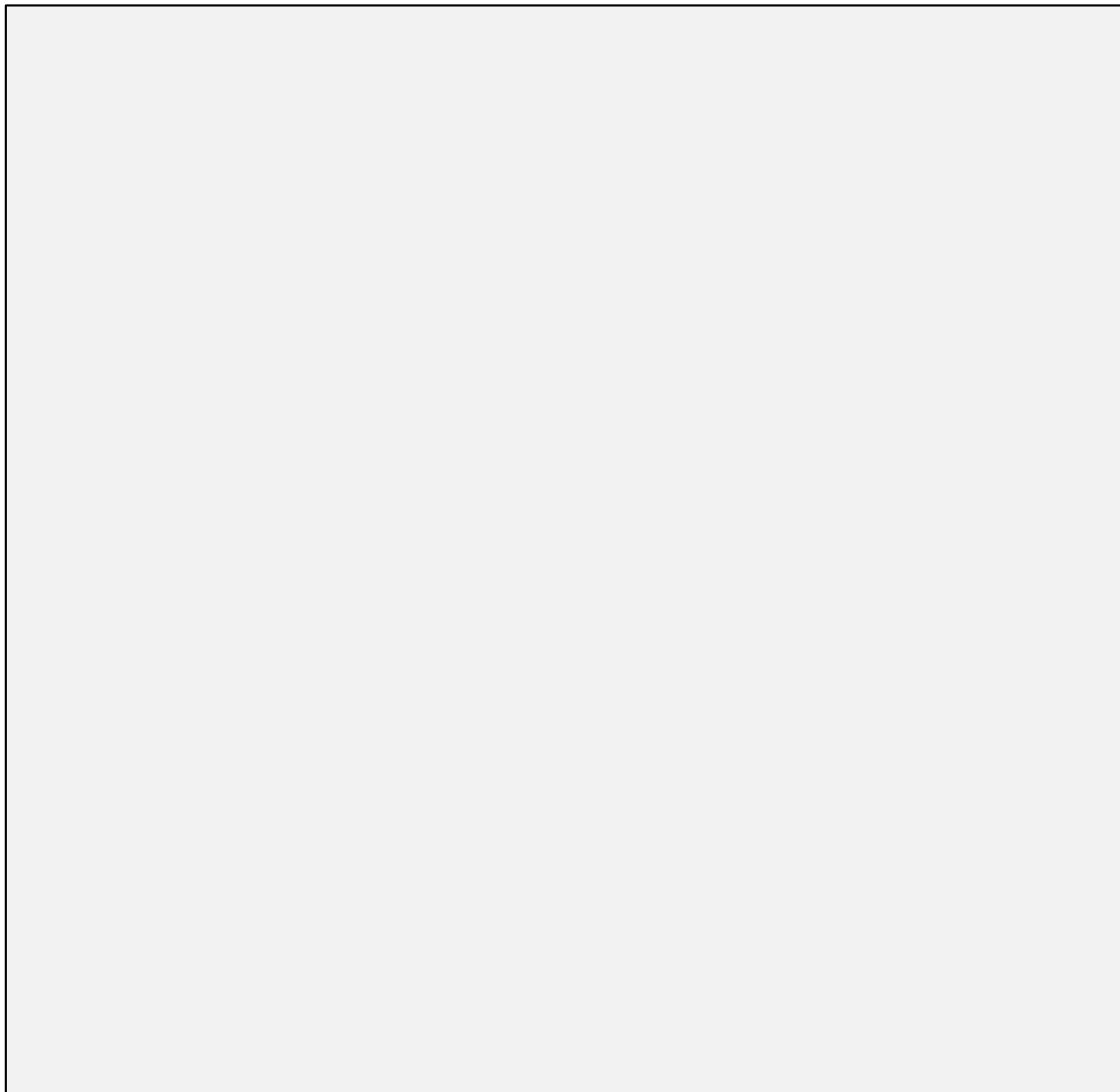
Chapter Topics

YARN Introduction

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

Distributed Processing History

Chapter 6



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- **Distributed Processing History**
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

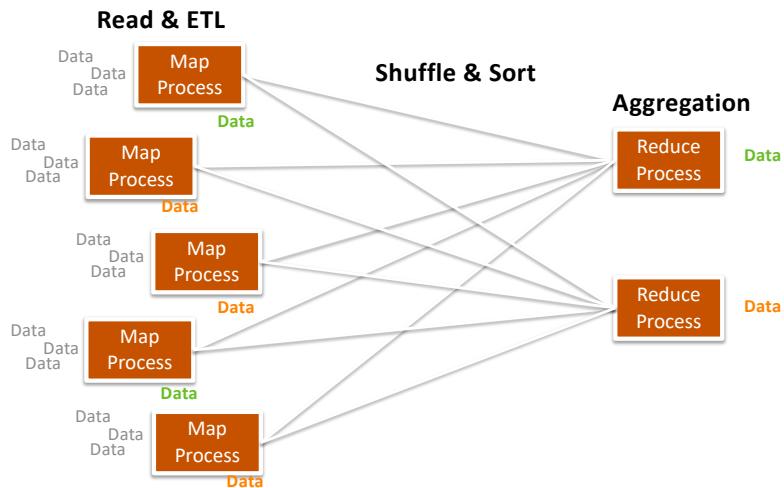
Distributed Processing History

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

“Everything fails, all of the time.” — Werner Vogels, CTO, Amazon.com

What is MapReduce?

Breaking a large problem into sub-solutions



The number of mappers is aligned with the number of blocks that the input data takes up.

The number of reducers is set by the application developer.

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!

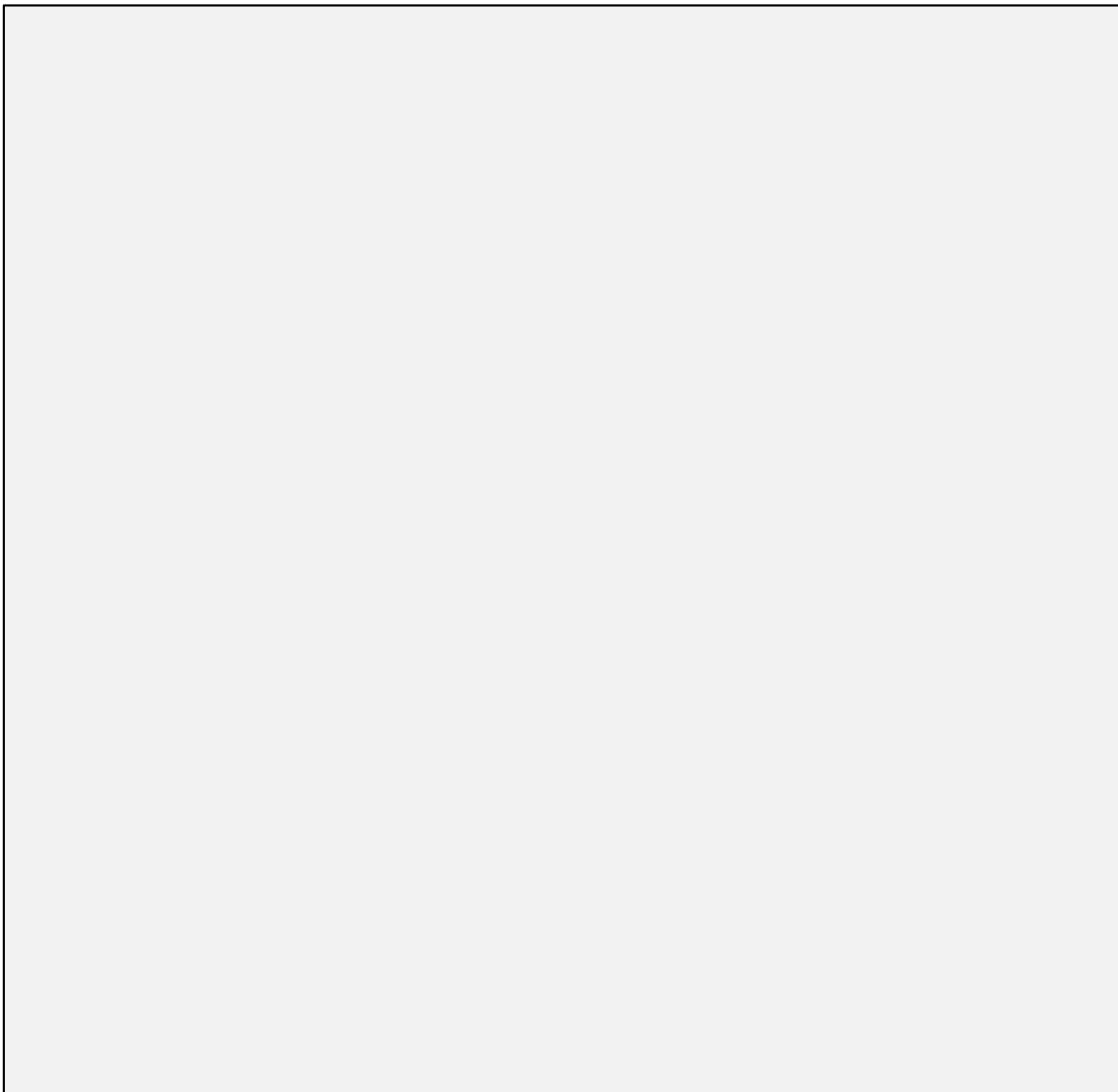


Simple Algorithm

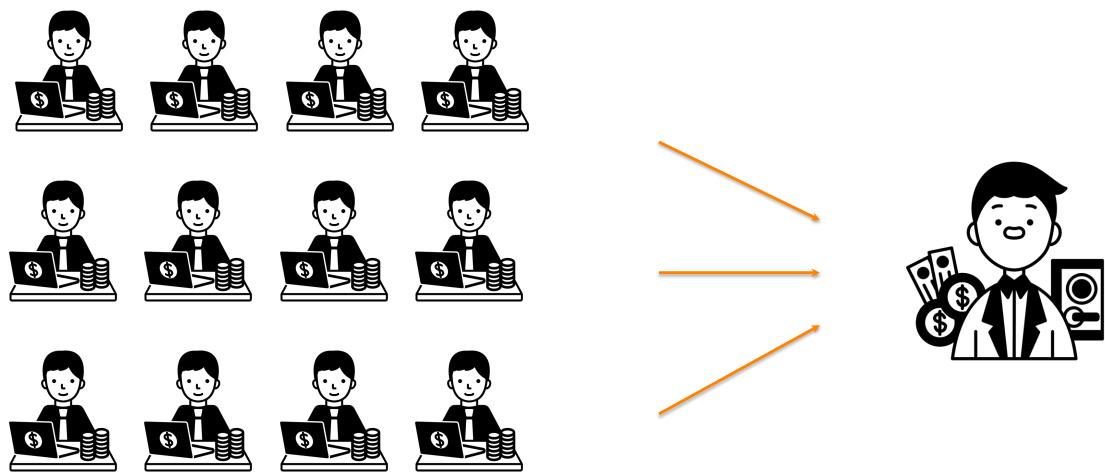
- 1. Review stack of quarters**
- 2. Count each year that ends in an even number**



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

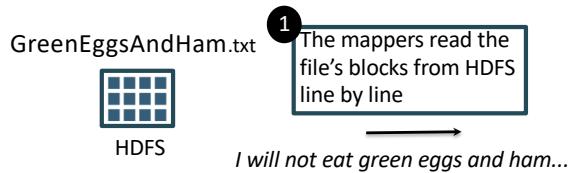
```
('I', 3)  
('Sam', 1)  
('am', 1)  
('and', 1)  
('eat', 2)  
('eggs', 1)  
('green', 1)  
('ham', 1)  
('not', 2)  
('them', 1)  
('will', 2)
```

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

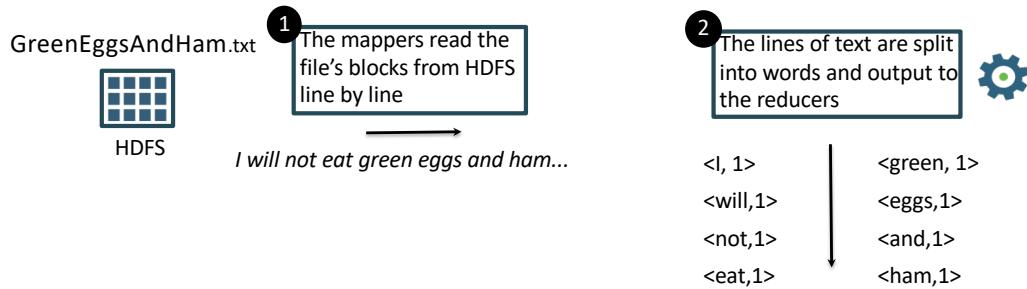
The Reducer

- 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

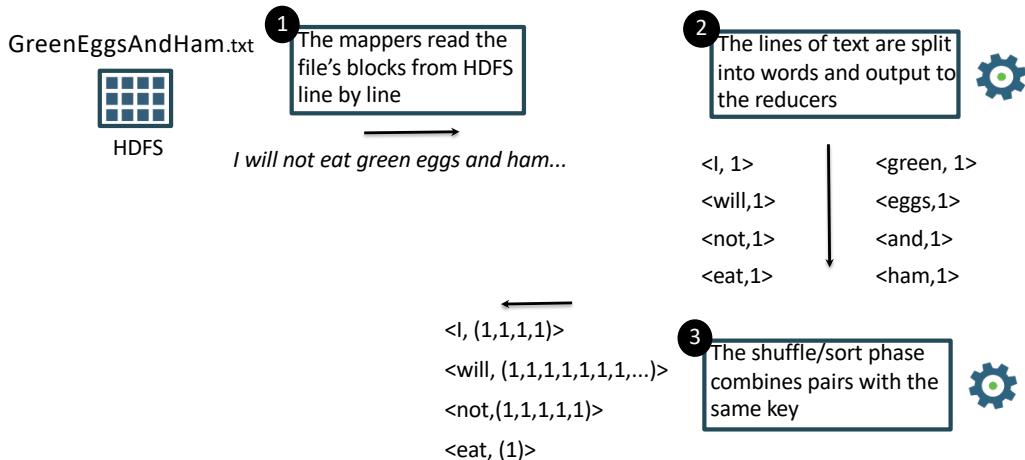
Recap of Word Count



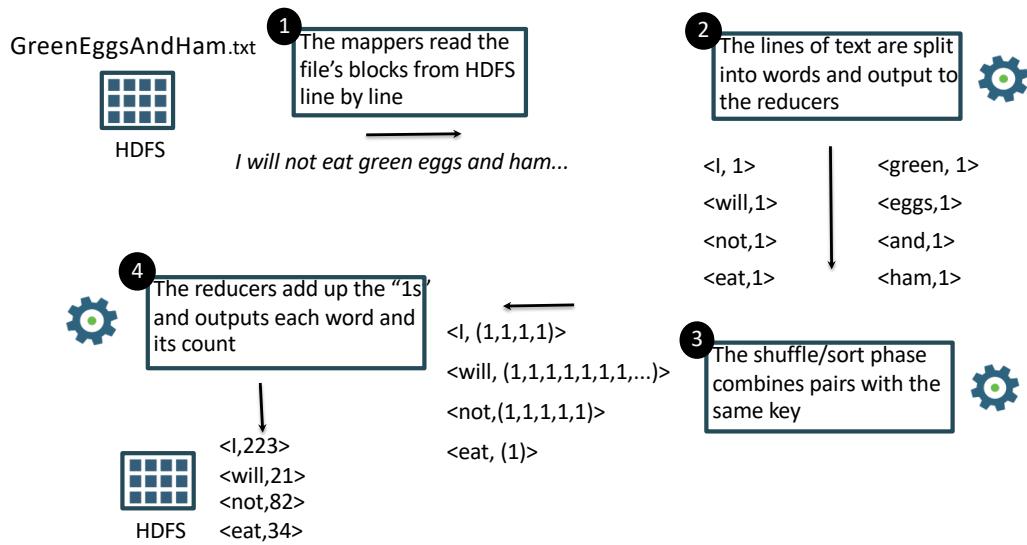
Recap of Word Count



Recap of Word Count



Recap of Word Count



Consider adding a demo of MRViewer: <https://github.com/msiddalingaiah/MRViewer>

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);  
            }  
        }  
    }  
}
```

Source : https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Source_Code

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);
    }
}
```

Source : https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Source_Code

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);  
}
```

Source : https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html#Source_Code

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

Apache spark started as a research paper in 2009 by a graduate student at Berkeley and former Cloudera trainee.
With spark, they began to create the BDAS, or Berkeley Data Analytics Stack.
Spark was created to be a general purpose data processing engine, focused on in-memory distributed computing use-cases. The grad paper started because of the struggles current users we're having with certain use cases in the MapReduce framework. Spark took many concepts from MapReduce and implemented them in a new way.

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 was built with the developer in mind. Spark has very elegant API's which seek to minimize the "plumbing" that developers traditionally had to worry about. Spark provides APIs that allow developers to focus on the business logic, and not the internals. Spark has brought forward in-memory computation for hadoop. This allows large amounts of data to be stored in memory, and quickly access them. Some applications have seen a 100x speed increase due to these new abilities. Spark also provides a single framework for most data processing needs. Spark took concepts from MapReduce, identified bottlenecks and inefficiencies, along with developer feedback and created an elegant solution attempting to address most of the concerns. That being said, spark allow for a wide variety of user types, to do a wide variety of data operations. MLLib was integrated for data scientists. Spark SQL was created to enable traditional sql analysts a familiar interface for interacting with large amounts of data. Spark also introduced spark stream to cover micro batch uses cases. Spark allows users to switch between frameworks with minimal differences in coding, to increase the flexibility of applications users can develop. Spark is fully integrated with Hadoop, and an open source too.

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.mapred.*;  
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 Mapper 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 {  
            StringTokenizer tokenizer = new StringTokenizer(value.toString());  
            while (tokenizer.hasMoreTokens()) {  
                word.set(tokenizer.nextToken());  
                context.write(word, one);  
            }  
        }  
        public static class Reducer 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.setMapperClass(TextMapper.class);  
        job.setMapOutputClass(IntMapper.class);  
        job.setReducerClass(TextReducer.class);  
        job.setOutputClass(TextOutputFormat.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);  
    }  
}
```



"Up to" does NOT mean "guaranteed" and really depends on the use case

Pictured are the WordCount implementation in Spark using Python API and Java MapReduce. We can see immediately how much plumbing is removed when we implement a simple word count. Spark can perform WordCount in ~3 lines of code, where as MapReduce requires over 50.

In addition, when taking advantage of in memory storage, Spark performs over 100x faster than tradition MapReduce when doing a logistic regression.

Slide shows a comparison with Java MapReduce, but a more "fair" comparison is with Pig as seen at <http://hortonworks.com/hadoop-tutorial/word-counting-with-apache-pig/>

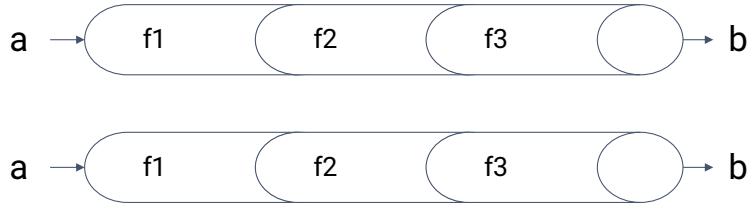
Why Functional Programming?

- **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

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

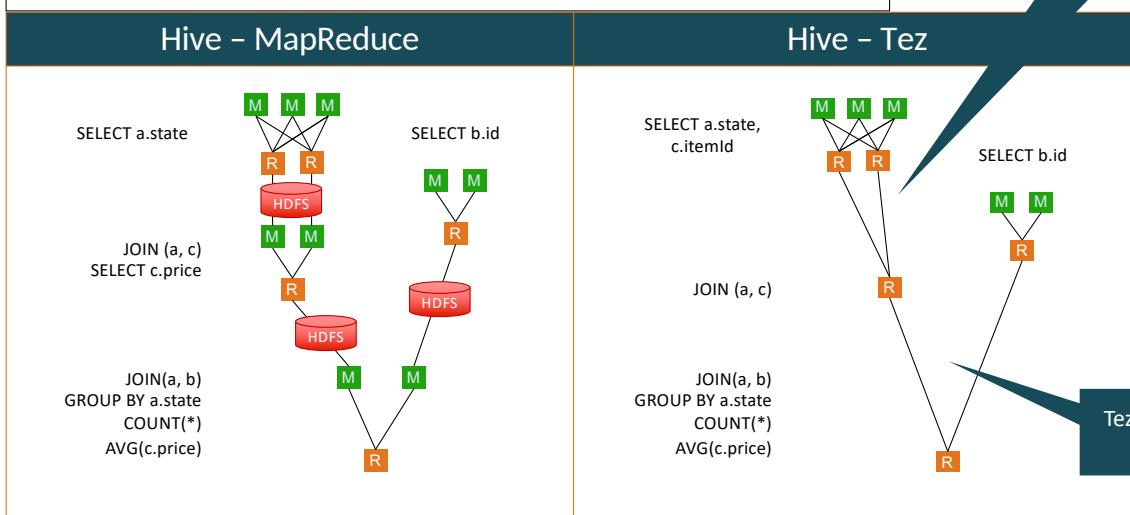
Immutability + Functional Programming = Distributed Processing!

- 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



Spend as much time as you want on this slide. Tez is a huge improvement in Hive execution.

Tez provides a general-purpose, highly customizable framework that simplifies data-processing tasks across both small-scale (low-latency) and large-scale (high-throughput) workloads in Hadoop. It generalizes the MapReduce paradigm to a more powerful framework by providing the ability to execute a complex DAG of tasks for a single job.

<http://tez.apache.org>

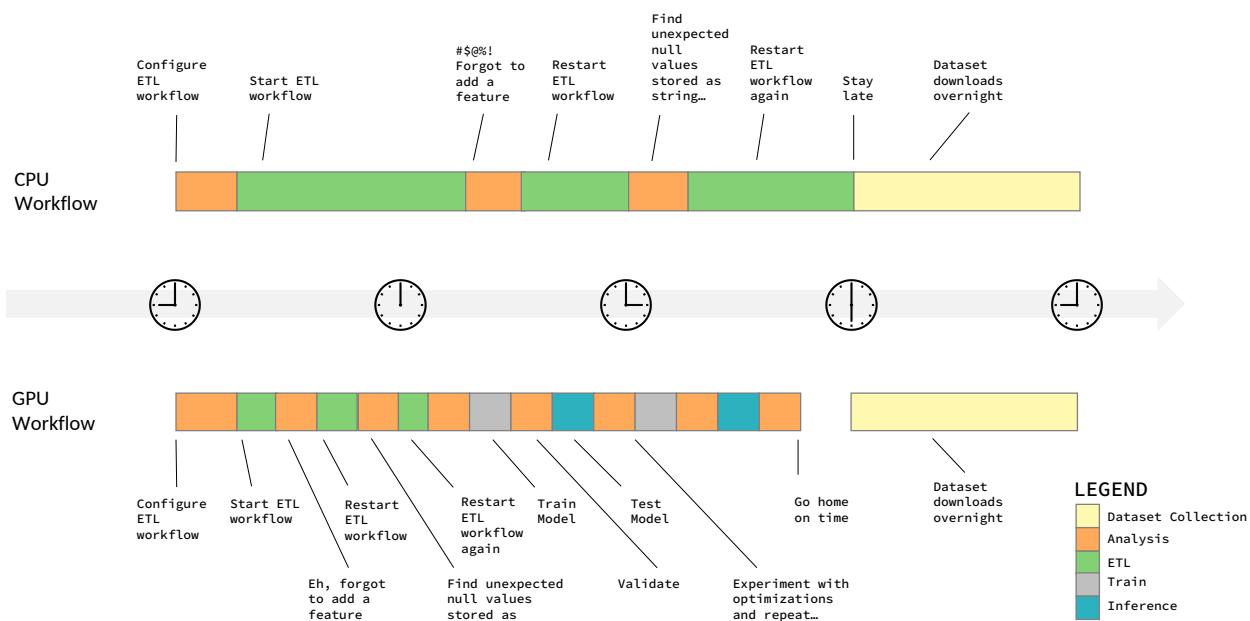
Chapter Topics

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!

- Requirements:

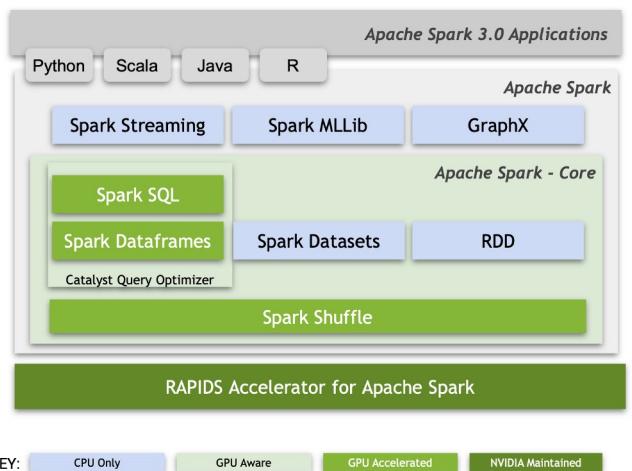
- Apache Spark - Version 3.0 or later
- RAPIDS Accelerator for Apache Spark
- Hardware with NVIDIA GPUs

- 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



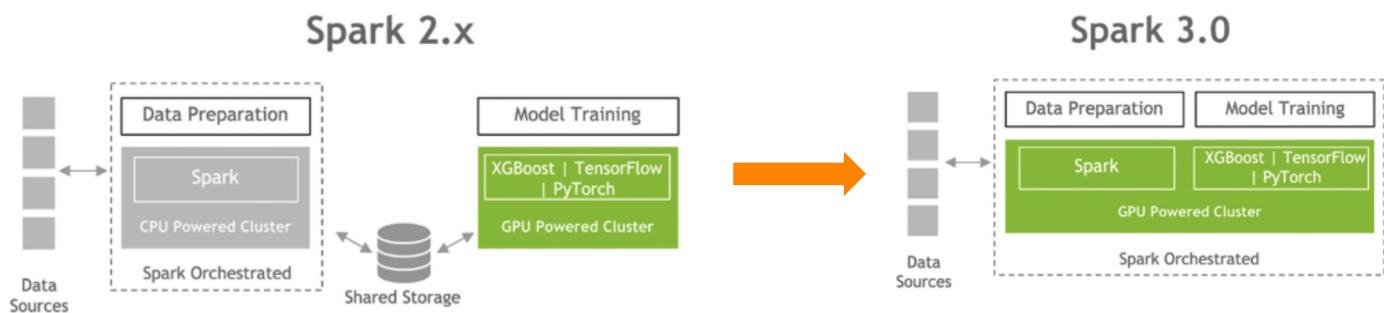
End to End GPU Acceleration for Spark Workloads

CLOUDERA

NVIDIA

APACHE
Spark™

RAPIDS



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

Feedback from early adopters is 5 to 10x performance increase

Mention this blog about the collaboration between Cloudera and NVIDIA:

<https://blog.cloudera.com/cloudera-supercharges-the-enterprise-data-cloud-with-nvidia/>

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

Answers

- 1 – Map and Reduce
- 2 – The number of blocks the input data is persisted to on HDFS. User can supply the number of reducers.
- 3 – Mappers are called with a single KVP and can return 0..m. Reducers receive a single KVP (the value is a list of values) and can also return 0..m.
- 4 – False, but you can have a Map-only job which would be the quickest possible job due to the lack of shuffle/sort and Reducer phases.
- 5 – Because they are “easier” for most developers and analysts
- 6 – False, Spark took many concepts from MapReduce and implemented them in a new way.
- 7 – False, no code change is required the GPU library mirrors the existing CPU one

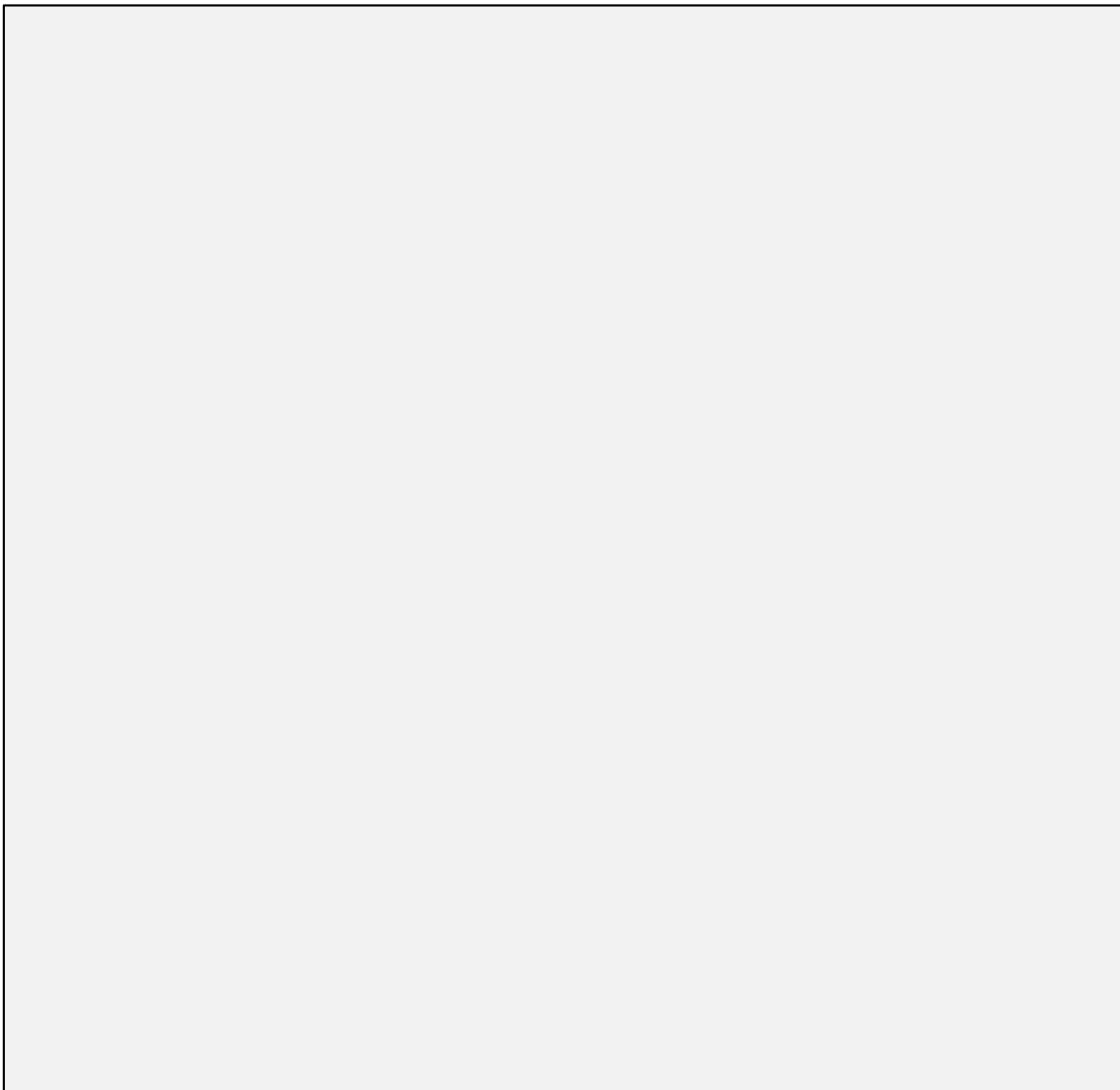
Essential Points

- 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

This summary page lists some of the main points from this lesson.

Working with RDDs

Chapter 7



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- **Working With RDDs**
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

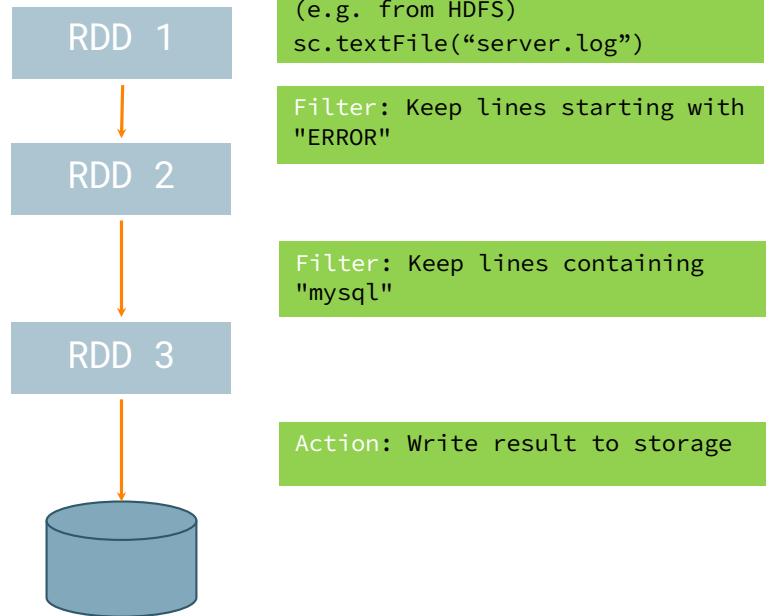
- **RDD is transformed:**

- 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**



Transformations are operations on an RDD that create a new RDD.

- The new RDD contains the result of the transformation on the old RDD(s).
- Recall that an RDD is read-only, so is never affected by a transformation — it's just the input to creating a new RDD.

An action is an operation that externalizes a value.

- Either returning a value to the driver program, or storing something in external 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)

DAG = Directed Acyclic Graph.

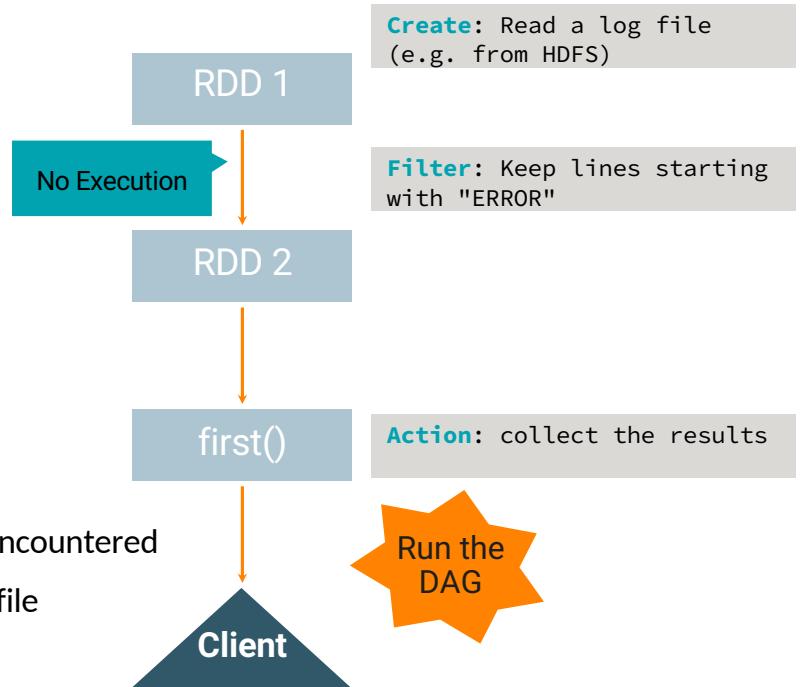
DAG is a programming framework for distributed systems – as the data is spread across multiple workers, conventional programming doesn't work.

A DAG organization is an alternative to MapReduce's organization

Unlike MR, Spark's DAG can have multiple levels that can form a tree structure, as well as doing Map Reduce.

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



We haven't yet talked about how data is partitioned across the Spark cluster, and how this job may involve tasks executing on multiple nodes.

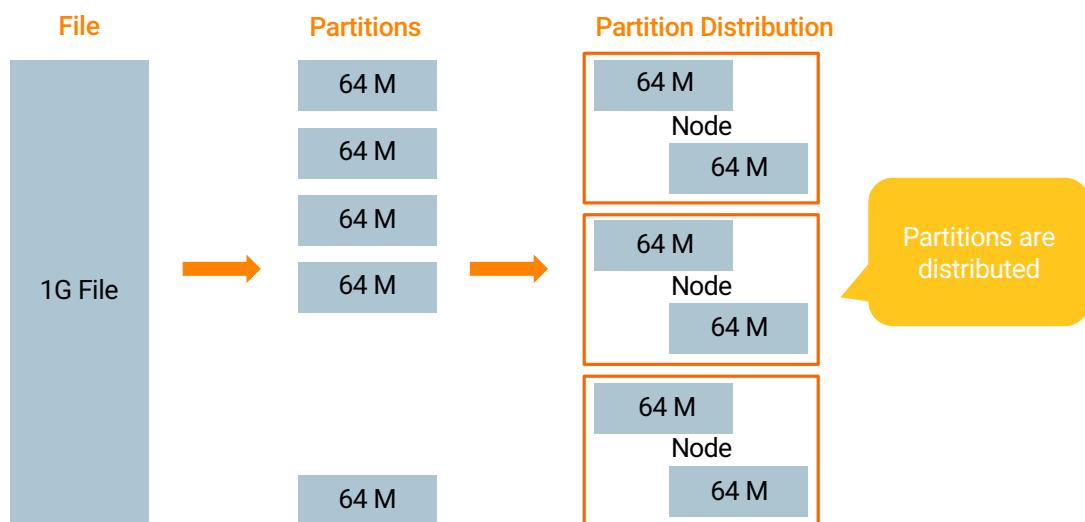
- This will affect how much optimization can be done, as there is no communication between nodes.

Because no work is done until the whole graph is known, the Spark engine can do optimizations at every step.

- In particular, in this example, the log file isn't read to create the first RDD until the engine already knows that it's just looking for the first ERROR line.
- As a result, it can optimize the reading of the log file, and stop when it reaches an ERROR line.

RDD Partitioning

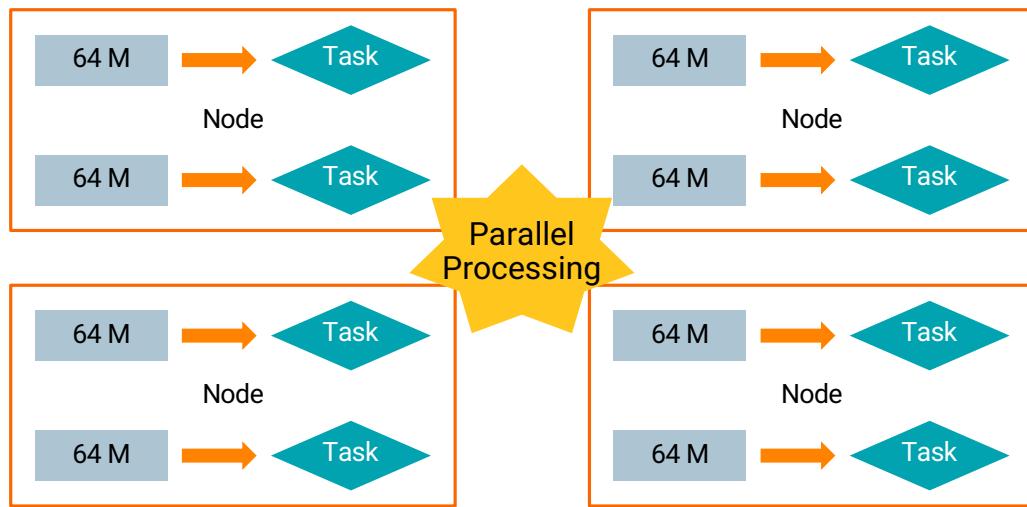
- Data in an RDD is partitioned around the cluster
 - e.g. With HDFS, Spark creates RDD partitions from HDFS blocks



HDFS will store the file in blocks (128M) on data nodes that live in the cluster.
These data nodes can also be Spark workers, a common situation when using HDFS with Spark.
Spark will partition its RDD on the same blocks as HDFS stores it, as this is the most efficient.

RDD Partitions and Parallel Processing

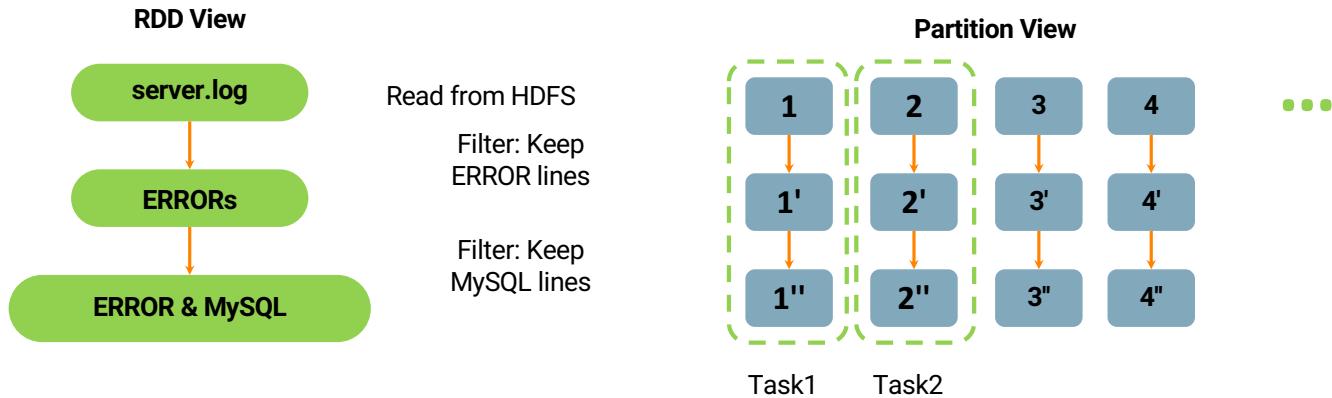
- Nodes execute tasks in parallel on the partitions
 - Spark will co-locate tasks with their data (HDFS block if HDFS)



If you have a series of transformations running on an RDD, then Spark will keep operating on the local data on each node.

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)



As much as possible, Spark will pipeline transformations for a given partition. Transformations will be done (in a stage) by the task that started the processing, and continue to work on the generated RDD partitions. Spark will keep these partitions in memory as much as possible. When it crosses an input boundary (e.g. a shuffle or an input operation), then it will start a new stage. Remember, this all happens once an action is run — until then it's just a DAG of transformations somewhere, with no actual processing having been done.

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 ...)		

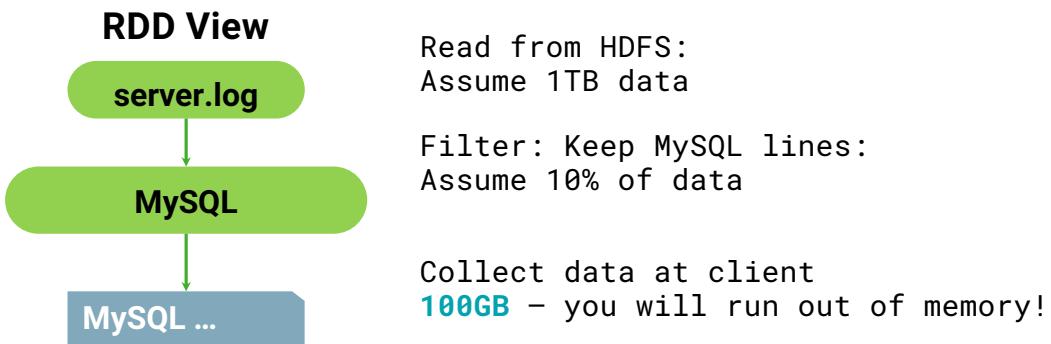


Filter: Keep MySQL lines

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

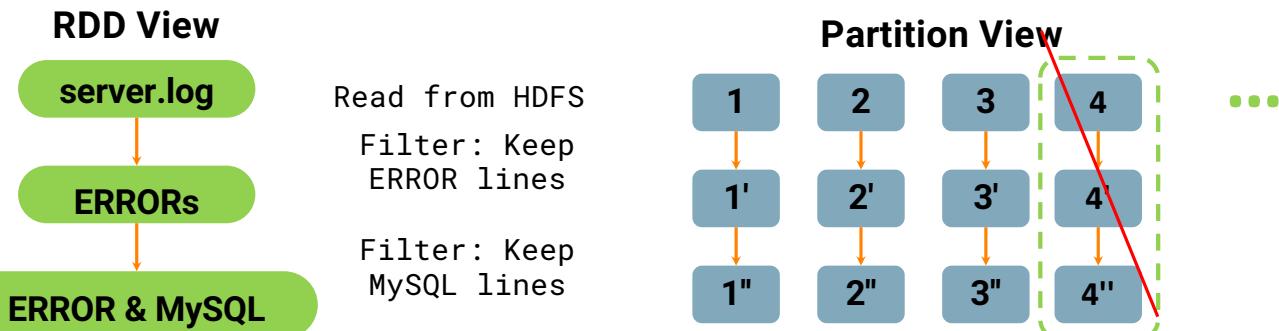
Careful with Data to Client

- 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

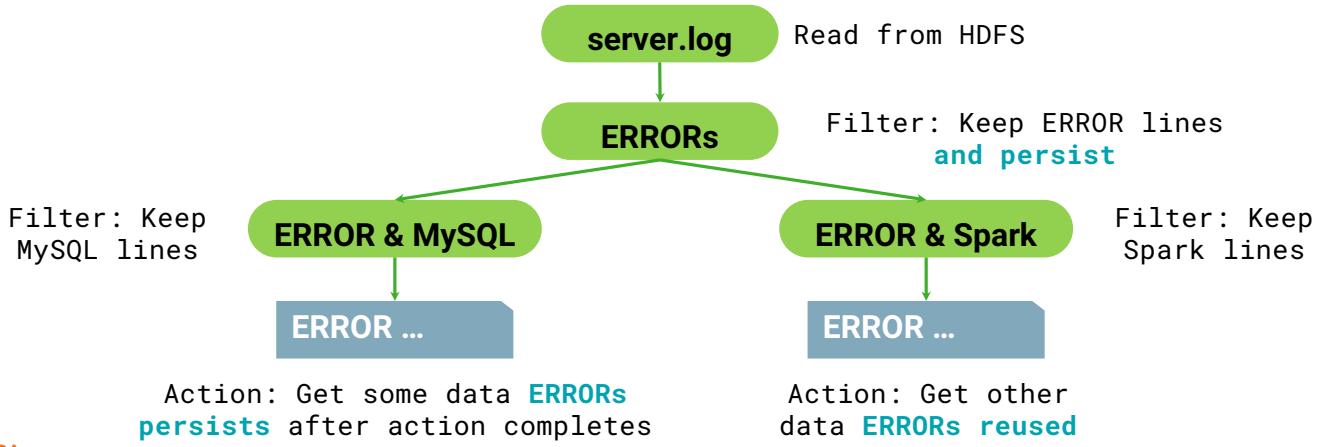


If a partition of an RDD is lost, for example due to a node failure, then it's easy to rebuild it.

- This can often be done locally on a node, without needing to communicate with other nodes.
- For example, if the RDD is based on an HDFS file block, then Spark just reads the block again, applies the series of transformations that it remembers for the RDD (the lineage) and gets the resulting partition back.
- There are many situations where it is not quite as simple — we'll cover them as we encounter them in the course.

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



RDDs are not persistent by design.

- You're likely to have much more data than memory.
- If you tried to persist it all, you'd have to evict a lot of it anyhow, so default is not to keep it.

You want to be careful in what RDDs you persist.

- In general, you'll only persist those that are expensive to create.
- For example, those that involve a shuffle.
- Otherwise, you could be filling your memory with RDDs that are not that useful to persist because they don't take much effort to recreate.
- We'll cover this in some more detail later.

In the example above, since we persist **ERRORs**, when the second action is run, it can start evaluating the transformations from the saved **ERRORs** RDD — it doesn't have to recompute it.

- If we had not save **ERRORs**, it would disappear after the first action, and then need to be recomputed for the second.
- This example is for illustration only — **ERRORs** may be easy to create, so maybe not the best candidate for caching depending on our application — more on that later.

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")

// 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")
```

Language: Scala

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.
I never hope to see one;
But I can tell you, anyhow,
I'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

There is no “overwrite” mode here!

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. All data has to fit in memory of ONE machine => don't use for big datasets	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

1. Resilient Distributed Dataset
2. sc.parallelize() and sc.textfile()
3. False. Transformations result in new RDDs being created. In Spark, data is immutable.
4. flatmap()
5. True
6. Lazy evaluation
7. False. The intersection function performs this task. The distinct function would remove duplicates elements, so that each element is only listed once regardless of how many times it appeared in the original RDD.
8. True

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

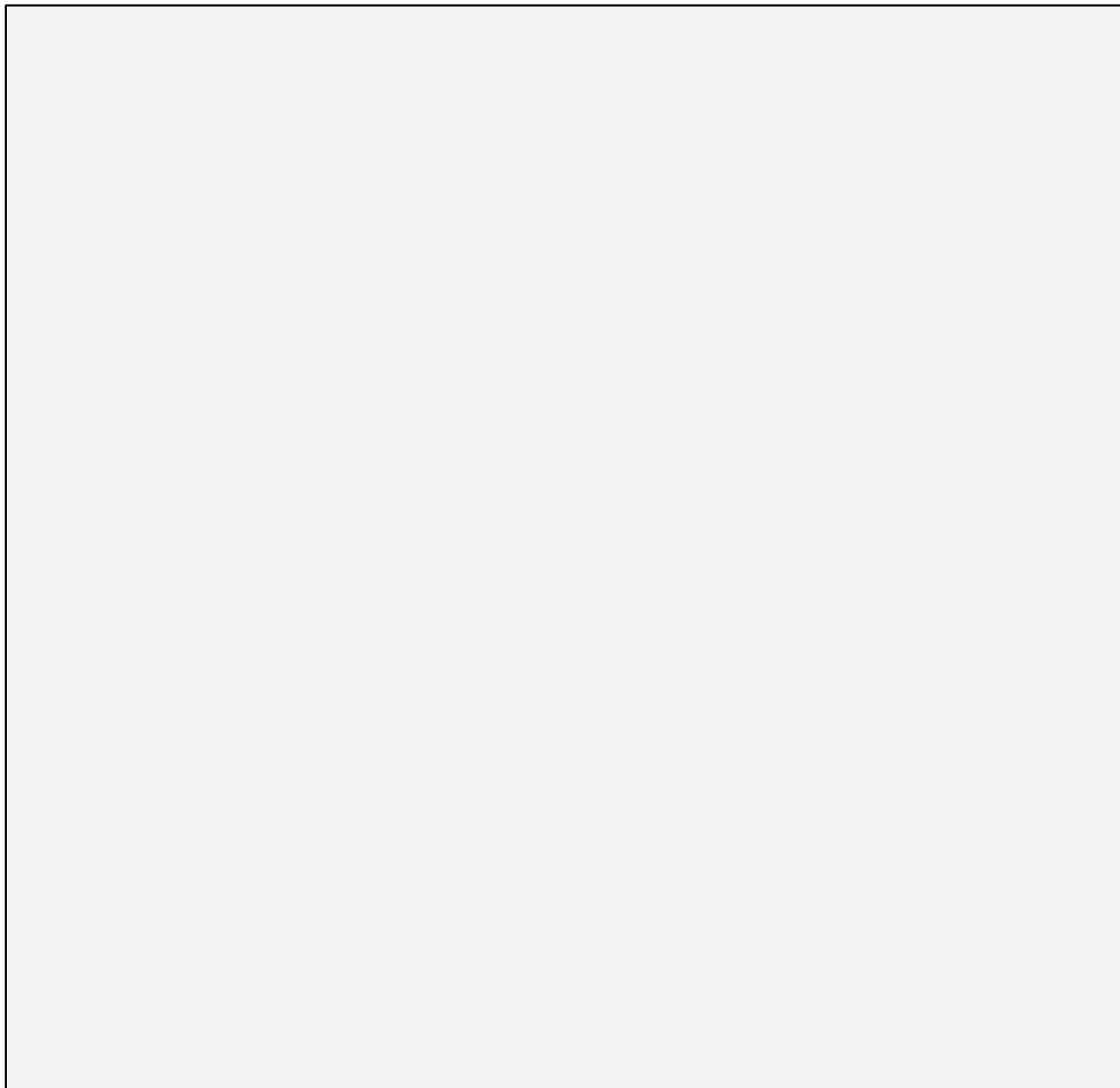
- Resilient Distributed Datasets (RDDs)
- **Exercise: Working with RDDs**

Exercise: Working with RDDs



Working with DataFrames

Chapter 8



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- **Working With Data Frames**
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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**

When Dataset was introduced in Spark 1.6, it was introduced as a separate type from DataFrame.

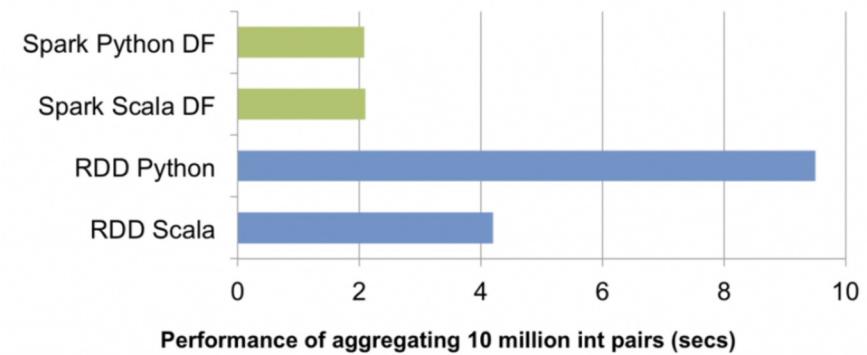
- This was because the functional interface was different, and they did not want to break API compatibility for DataFrame in a 1.x release.
- However, when working on Spark 2, they decided break compatibility with 1.x versions.
- As a result, they were able to unify the DataFrame and Dataset APIs.

In Java, there is no typedef capability, so there is no actual DataFrame type.

- You use Dataset<Row> wherever you need the DataFrame functionality.
- In other languages, such as Python and R, you also don't have a DataFrame type.
 - This is usually not an issue due to the more dynamic nature of these languages.

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.**

1. An API that allows to SQL to generate Spark jobs
 2. Using spark.read, available formats are csv, json,parquet, orc, text
 3. A DataFrame is a Dataset of type row, Datasets are strongly typed objects
 4. False

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

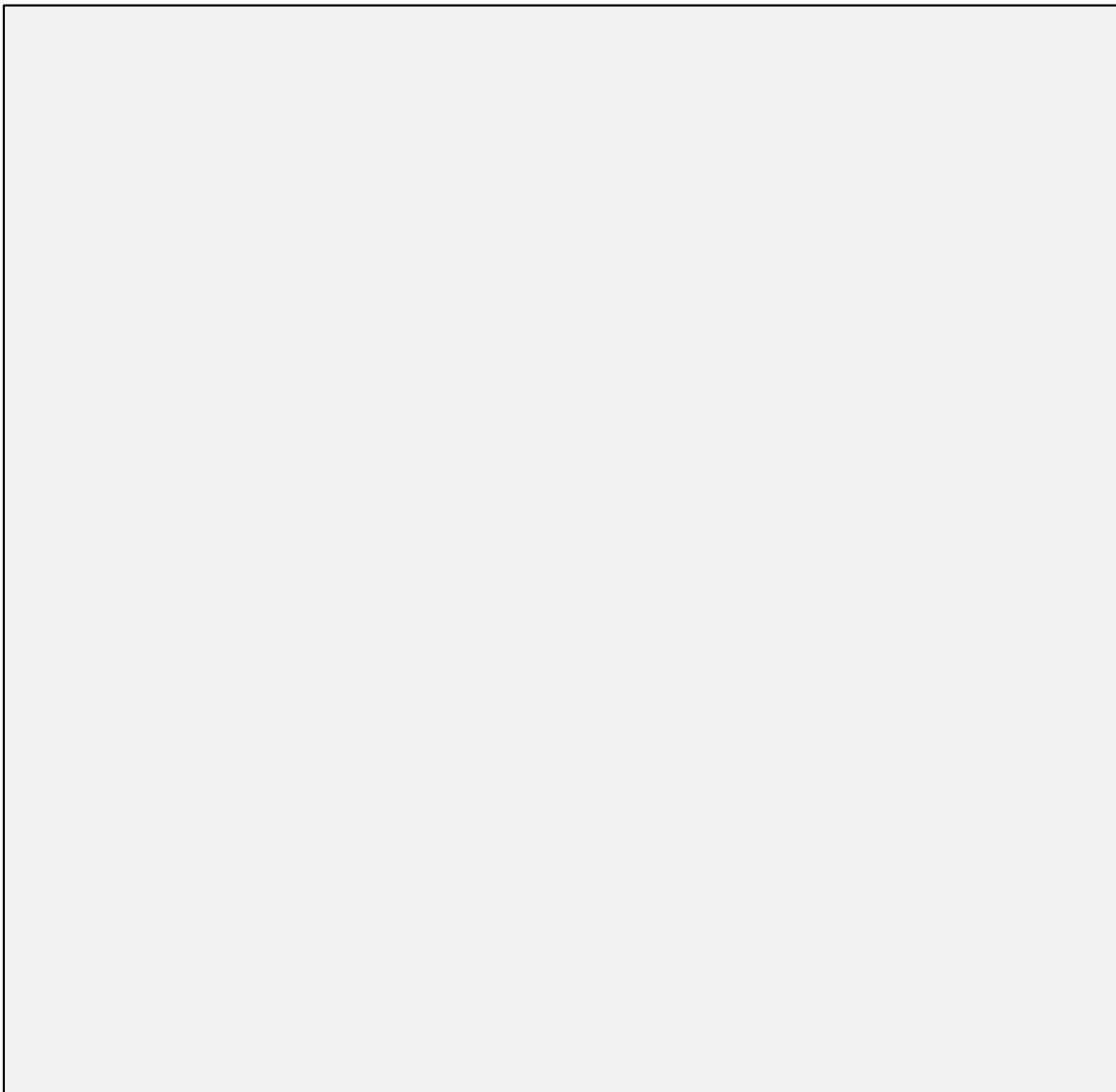
- Introduction to DataFrames

- **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 9



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- **Introduction to Apache Hive**
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

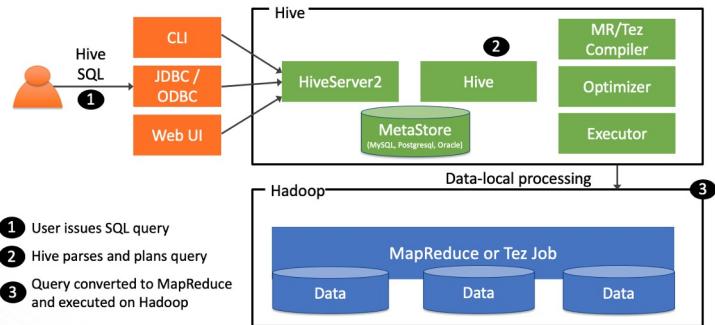
- Hive's Catalog (System Tables)

■ Metastore DB

- RDBMS store for “system tables”

■ LLAP (Low Latency Analytical Processing)

- Low Latency, Always On, Shared Cache Service



HiveServer 2

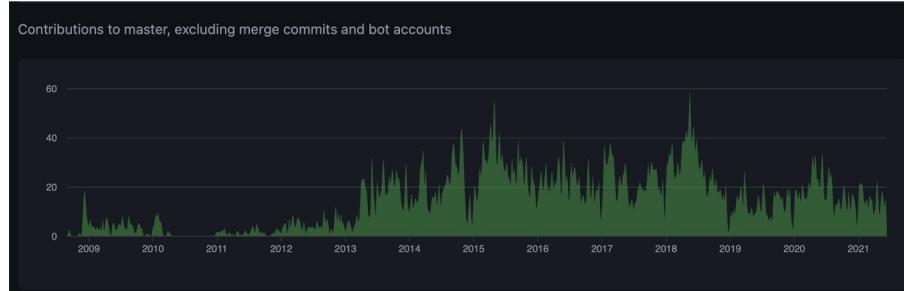
- HiveServer2 (HS2) is a service that enables clients to execute queries against Hive
- HiveServer2 supports multi-client concurrency and authentication. It is designed to provide better support for open API clients like JDBC and ODBC
- HiveServer2 is a single process running as a composite service, 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

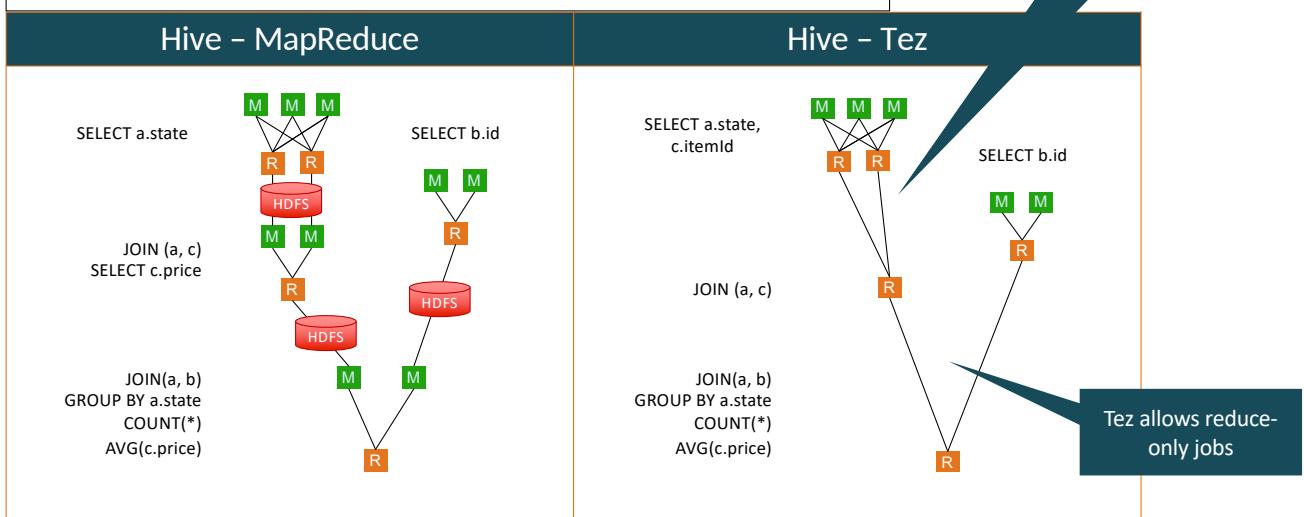
- Throughout its long history, Hive has undergone many improvements
 - The TEZ execution engine
 - Vectorization
 - The ORC format
 - LLAP
 - Transactional tables
 - Materialized Views



Call out that each improvement added its share of complexity to the product

```
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



Spend as much time as you want on this slide. Tez is a huge improvement in Hive execution.

Tez provides a general-purpose, highly customizable framework that simplifies data-processing tasks across both small-scale (low-latency) and large-scale (high-throughput) workloads in Hadoop. It generalizes the MapReduce paradigm to a more powerful framework by providing the ability to execute a complex DAG of tasks for a single job.

<http://tez.apache.org>

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;`

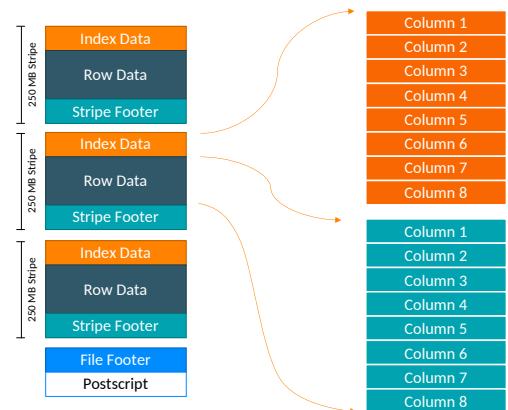
ORC-Backed Hive Tables

- 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

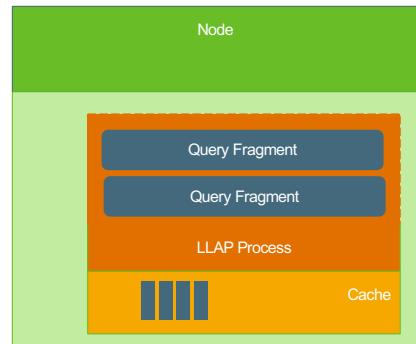
- **High Performance: Columnar storage File.**
- **Efficient Reads: Break into large “stripes” of data efficient read.**
- **Fast Filtering: Built in index, min/max, metadata for fast filtering blocks.**
- **Efficient Compression: Decompose complex row types into primitives: massive compression and efficient 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

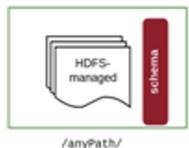
- The data in Hive external tables can reside in any HDFS directory.
- When you drop an External Hive table only the metadata is deleted not the data.

```
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.

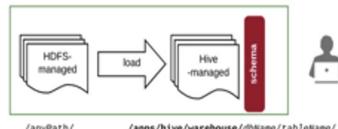


```
CREATE EXTERNAL TABLE myTable  
(name STRING, age INT)  
LOCATION '/anyPath';
```

- 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 3 write and read operations improve the ACID properties and performance of transactional tables. Transactional tables perform as well as other tables. Hive supports all TPC Benchmark DS (TPC-DS) queries.

Hive 3 and later extends atomic operations from simple writes and inserts to support the following operations:

- Writing to multiple partitions
- Using multiple insert clauses in a single SELECT statement

A single statement can write to multiple partitions or multiple tables. If the operation fails, partial writes or inserts are not visible to users. Operations remain performant even if data changes often, such as one percent per hour. Hive 3 and later does not overwrite the entire partition to perform update or delete operations.

Read semantics consist of snapshot isolation. Hive logically locks in the state of the warehouse when a read operation starts. A read operation is not affected by changes that occur during the operation.

https://docs.hortonworks.com/HDPDocuments/HDP3/HDP-3.1.0/using-hiveql/content/hive_3_internals.html

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);
```

```
tm  
--- delta_0000001_0000001_0000  
    |-- 000000_0  
    --- delta_0000002_0000002_0000      //Fails  
    |-- 000000_0  
    --- delta_0000003_0000003_0000  
    |-- 000000_0
```

When an insert-only transaction begins, the transaction manager gets a transaction ID. For every write, the transaction manager allocates a write ID. This ID determines a path to which data is actually written. The following code shows an example of a statement that creates insert-only transactional table:

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

Assume that three insert operations occur, and the second one fails:

```
INSERT INTO tm VALUES(1,1);  
INSERT INTO tm VALUES(2,2); // Fails  
INSERT INTO tm VALUES(3,3);
```

For every write operation, Hive creates a delta directory to which the transaction manager writes data files. Hive writes all data to delta files, designated by write IDs, and mapped to a transaction ID that represents an atomic operation. If a failure occurs, the transaction is marked aborted, but it is atomic:

```
tm  
--- delta_0000001_0000001_0000  
    |-- 000000_0  
    --- delta_0000002_0000002_0000      //Fails  
    |-- 000000_0  
    --- delta_0000003_0000003_0000  
    |-- 000000_0
```

During the read process, the transaction manager maintains the state of every transaction. When the reader starts, it asks for the snapshot information, represented by a high watermark. The watermark identifies the highest transaction ID in the system followed by a list of exceptions that represent transactions that are still running or are aborted.

The reader looks at deltas and filters out, or skips, any IDs of transactions that are aborted or still

running. The reader uses this technique with any number of partitions or tables that participate in the transaction to achieve atomicity and isolation of operations on transactional tables.

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	

Tables that support updates and deletions require a slightly different technique to achieve atomicity and isolation. Hive runs on top of an append-only file system, which means Hive does not perform in-place updates or deletions. Isolation of readers and writers cannot occur in the presence of in-place updates or deletions. In this situation, a lock manager or some other mechanism, is required for isolation. These mechanisms create a problem for long-running queries.

Instead of in-place updates, Hive decorates every row with a row ID. The row ID is a struct that consists of the following information:

- The write ID that maps to the transaction that created the row
- The bucket ID, a bit-backed integer with several bits of information, of the physical writer that created the row
- The row ID, which numbers rows as they were written to a data file

Instead of in-place deletions, Hive appends changes to the table when a deletion occurs. The deleted data becomes unavailable and the compaction process takes care of the garbage collection later.

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;
```

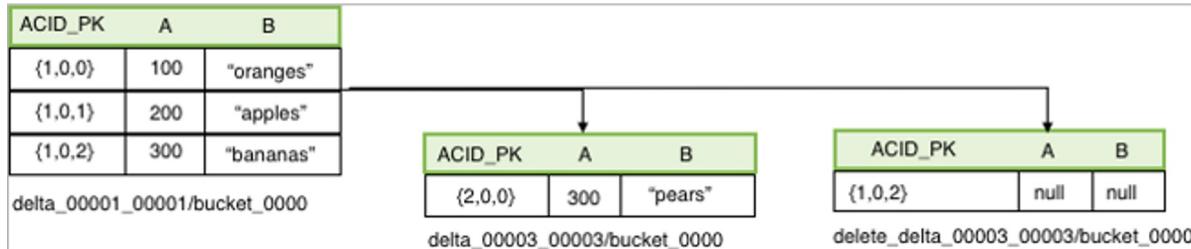
This operation generates a directory and file,
delete_delta_00002_00002/bucket_0000.

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

Hive Transactional Operations (Continued)

Update Operation

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



The reader, which requires the AcidInputFormat, applies all the insert events and encapsulates all the logic to handle delete events. A read operation first gets snapshot information from the transaction manager based on which it selects files that are relevant to that read operation. Next, the process splits each data file into the number of pieces that each process has to work on. Relevant delete events are localized to each processing task. Delete events are stored in a sorted ORC file. The compressed, stored data is minimal, which is a significant advantage of Hive 3. You no longer need to worry about saturating the network with insert events in delta files.

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

Answers

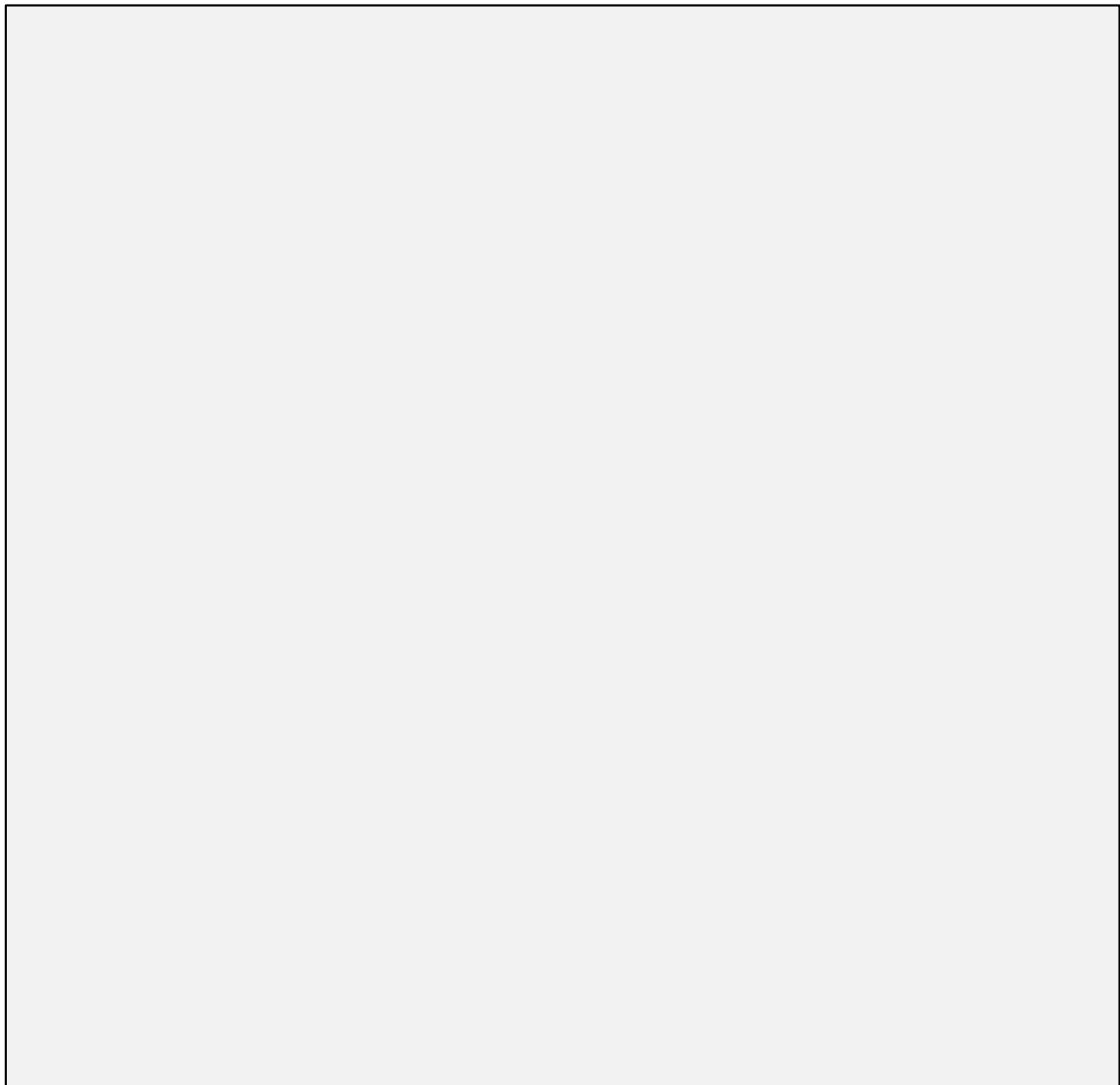
- 1 – HiveServer2 (HS2)
- 2 – MR, Spark & Tez.
- 3 – When a managed table is dropped, all of its underlying files will be deleted.
- 4 – Tez, ORC, Vectorization, ACID transactions

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

Transforming Data with Hive

Chapter 10



Course Chapters

- Introduction
- Why Data Engineering
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- **Transforming Data with Hive**
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- Conclusion
- Appendix: Working with Datasets in Scala

Objectives

- **By the end of this chapter, you will be able to**
 - Execute a query using Beeline
 - Connect to a database and query tables
 - Write Hive SQL statements
 - Create Hive 3.0 ACID transactions
 - Retrieve data using Group By, Sort By, and Distribute By
 - Use materialized views to pre-aggregate data for queries
 - Relate multiple tables in Hive databases using joins and subqueries
 - Create User Defined Functions (UDFs)

Chapter Topics

Transforming Data with Hive

- **Introduction to Hive SQL**
- **Hands-On Exercise: Hive SQL**
- **Hive ACID Transactions**
- **Hands-On Exercise: Hive Transactions**
- **Data Retrieval, Views, and Materialized Views**
- **Hands-On Exercise: Materialized Views**
- **Hive Joins**
- **Hands-On Exercise: Hive Joins**
- **Hive Windowing and Grouping**
- **Hive User Defined Functions**
- **Essential Points**

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-3

Beeline Query Execution Commands

▪ Command Line

- beeline -e <query> to Execute a query
- beeline -f <filename> to Execute query from file

▪ Shell

- !connect command connects to Hive database using JDBC URL
- !run used to execute query from a file.
- !quit command or !q to exit interactive mode

```
[hdfs@ip-172-31-0-110 ~]$ beeline
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/hdp/3.1.0.0-78/hive/lib/log4j-slf4j-impl-2.10.0.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/hdp/3.1.0.0-78/hadoop/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
Connecting to jdbc:hive2://ip-172-31-0-113.eu-central-1.compute.internal:2181,ip-172-31-0-110.eu-central-1.compute.internal:2181,ip-172-31-8-20.eu-central-1.compute.internal:2181/default;serviceDiscoveryMode=zooKeeper;zooKeeperNamespace=hiveserver2
Enter username for jdbc:hive2://ip-172-31-0-113.eu-central-1.compute.internal:2181,ip-172-31-0-110.eu-central-1.compute.internal:2181,ip-172-31-8-20.eu-central-1.compute.internal:2181/default: hive
Enter password for jdbc:hive2://ip-172-31-0-113.eu-central-1.compute.internal:2181,ip-172-31-0-110.eu-central-1.compute.internal:2181,ip-172-31-8-20.eu-central-1.compute.internal:2181/default: ****
19/05/23 08:55:26 [main]: INFO jdbc.HiveConnection: Connected to ip-172-31-6-253.eu-central-1.compute.internal:10000
Connected to: Apache Hive (version 3.1.0.3.1.0.0-78)
Driver: Hive JDBC (version 3.1.0.3.1.0.0-78)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 3.1.0.3.1.0.0-78 by Apache Hive
0: jdbc:hive2://ip-172-31-0-113.eu-central-1.>
```

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-4

Hive QL

SQL Data Types	SQL Semantics
INT	SELECT, LOAD, INSERT from query
TINYINT/SMALLINT/BIGINT	Expressions in WHERE and HAVING
BOOLEAN	GROUP BY, ORDER BY, SORT BY
FLOAT	CLUSTER BY, DISTRIBUTE BY
DOUBLE	Subqueries in FROM clause
STRING	GROUP BY, ORDER BY
BINARY	ROLLUP and CUBE
TIMESTAMP	UNION
ARRAY, MAP, STRUCT, UNION	LEFT, RIGHT and FULL INNER/OUTER JOIN
DECIMAL	CROSS JOIN, LEFT SEMI JOIN
CHAR	Windowing functions (OVER, RANK, etc.)
VARCHAR	Sub-queries for IN/NOT IN, HAVING
DATE	EXISTS / NOT EXISTS

Compare/Contrast RDBMS & Hive

▪ Similarities

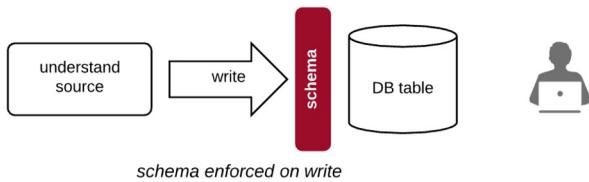
- Query language
- ODBC/JDBC
- Security

▪ Key differences

- Data modeling approach
 - RDBMS: we model for use case ubiquity
 - Hive: we build Query Focused Databases (QFD) based on specific use cases
- Integrity checks
 - Schema on write for RDBMS (data types validated & PK/FK data constraints)
 - Hive features schema on read (fast loading & integrity via the data pipeline)
- Hive focuses on OLAP while RDBMS also addresses the OLTP domain

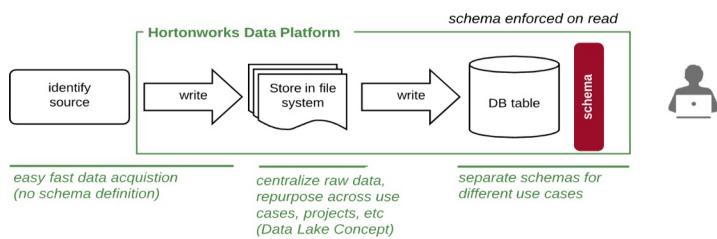
Hive is schema-on-read

Schema-on-write



- Rigid: writes to schema specifications
- Slow to acquire data: analyze, develop, test, acquire (must know query patterns before acquire)
- Broad schemas: single schema serves multiple use cases
- Slow to change schema: analyze, develop, test against all use cases

Schema-on-read



- Flexible: stores raw data to file system (post-processed according to need)
- Rapid to acquire data: data acquisition and schema definition decoupled
- Multiple schemas: schema for each use case
- Allows Data Lake vision: centralize diverse data sources, reuse across projects and initiatives

Create Database

```
CREATE (DATABASE|SCHEMA) [IF  
NOT EXISTS] database_name  
[COMMENT database_comment]  
[LOCATION hdfs_path]  
[WITH DBPROPERTIES  
(property_name=property_value,  
...)];
```

Drop Database

```
DROP (DATABASE|SCHEMA) [IF  
EXISTS] database_name  
[RESTRICT|CASCADE];
```

Alter Database

```
ALTER (DATABASE|SCHEMA)  
database_name SET LOCATION  
hdfs_path;  
-- (Note: Hive 2.2.1, 2.4.0  
and later)
```

Use Database

```
USE database_name;  
USE DEFAULT;
```

Hive Table Design

```
CREATE [TEMPORARY] [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name -- (Note: TEMPORARY  
available in Hive 0.14.0 and later)  
[(col_name data_type [COMMENT col_comment], ... [constraint_specification])]  
[COMMENT table_comment]  
[PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]  
[CLUSTERED BY (col_name, col_name, ...) [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets  
BUCKETS]  
[SKEWED BY (col_name, col_name, ...) -- (Note: Available in Hive 0.10.0 and  
later)]  
ON ((col_value, col_value, ...), (col_value, col_value, ...), ...)  
[STORED AS DIRECTORIES]  
[  
[ROW FORMAT row_format]  
[STORED AS file_format]  
| STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)] -- (Note: Available in  
Hive 0.6.0 and later)  
]  
[LOCATION hdfs_path]  
CLOUDERA TABLE PROPERTIES (property_name=property_value, ...)] -- (Note: Available in Hive 0.6.0 and later)  
Educational Services Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.  
[AS select_statement]; -- (Note: Available in Hive 0.5.0 and later; not supported for external  
tables)
```

Hive Table Design (Continued)

Rename Table

```
ALTER TABLE table_name RENAME TO new_table_name;
```

Alter Table Properties

```
ALTER TABLE table_name SET TBLPROPERTIES table_properties;
```

table_properties:

```
: (property_name = property_value, property_name =  
property_value, ... )
```

Hive Data Types

- **Numeric data types**

- [TINYINT](#) (1-byte signed integer, from -128 to 127)
- [SMALLINT](#) (2-byte signed integer, from -32,768 to 32,767)
- [INT/INTEGER](#) (4-byte signed integer, from -2,147,483,648 to 2,147,483,647)
- [BIGINT](#) (8-byte signed integer, from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807)
- FLOAT (4-byte single precision floating point number)
- DOUBLE (8-byte double precision floating point number)
- DOUBLE PRECISION
- DECIMAL
- NUMERIC (same as DECIMAL, starting with [Hive 3.0.0](#))

- **Misc types**

- BOOLEAN
- BINARY

- **Data/Time types**

- TIMESTAMP
- DATA
- INTERVAL

- **String types**

- STRING
- VARCHAR
- CHAR

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

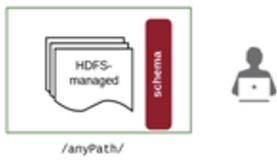
- The data in Hive external tables can reside in any HDFS directory
- When you drop an External Hive table only the metadata is deleted not the data

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

Hive External Versus 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**

Loading Data into Hive

```
LOAD DATA LOCAL INPATH '/tmp/customers.csv' OVERWRITE INTO TABLE customers;
```

```
LOAD DATA INPATH '/user/train/customers.csv' OVERWRITE INTO TABLE customers;
```

```
INSERT INTO TABLE students  
VALUES ('fred flintstone', 35, 1.28), ('barney rubble', 32, 2.32);
```

```
INSERT INTO TABLE birthdays  
SELECT firstName, lastName, birthday  
FROM customers  
WHERE birthday IS NOT NULL;
```

Performing Queries

```
SELECT * FROM customers;
```

```
FROM customers
SELECT firstName, lastName, address, zip
WHERE orderID > 0
ORDER BY zip;
```

```
SELECT customers.*, orders.*
FROM customers
JOIN orders ON
(customers.customerID = orders.customerID);
```

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- **Hands-On Exercise: Hive SQL**
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- Hive Joins
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- Hive User Defined Functions
- Essential Points

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- **Hive ACID Transactions**
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- Hive Joins
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- Hive User Defined Functions
- Essential Points

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-18

Hive 3.0 ACID Transactions

- Hive supports single-table / single-statement transactions and are the default table type
- 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);
```

```
tm  
--- delta_000001_000001_0000  
    --- 00000_0  
    --- delta_000002_000002_0000  
        --- 00000_0  
        --- delta_000003_000003_0000  
            --- 00000_0
```

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;
```

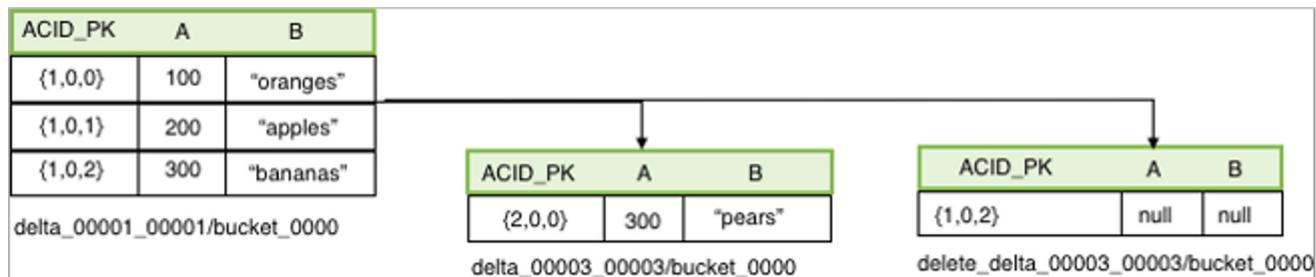
This operation generates a directory and file,
delete_delta_00002_00002/bucket_0000.

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

Hive Transactional Operations (Continued)

Update Operation

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



Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions**
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- Hive Joins
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- Hive User Defined Functions
- Essential Points

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-24

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- **Data Retrieval, Views, and Materialized Views**
- Hands-On Exercise: Materialized Views
- Hive Joins
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- Hive User Defined Functions
- Essential Points

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-25

Hive Data Retrieval - Group By

- **GROUP BY** clause is used to group rows that have the same values
- Set **hive.groupby.position.alias** to true (the default is **false**)

```
SELECT [ALL | DISTINCT] select_expr,  
       select_expr, ...  
  FROM table_reference  
  [WHERE where_condition]  
  [GROUP BY col_list]  
  [HAVING having_condition]  
  [ORDER BY col_list]]  
  [LIMIT number];
```

See <https://stackoverflow.com/questions/29443253/hive-more-clean-way-to-select-as-and-group-by> for more info (and example) on `hive.groupby.position.alias` property.

Hive Data Retrieval - Order By and Sort By

▪ orderBy clause

- Guarantees ordering of the rows
- Must be followed by a “limit” clause
- In Hive 3.0, Order By without limit in subqueries and views will be removed by the optimizer by default (set `hive.remove.orderby.in.subquery` to false)

```
colOrder: ( ASC | DESC )
colNullOrder: (NULLS FIRST | NULLS LAST)
orderBy: ORDER BY colName colOrder? colNullOrder? (',', colName
colOrder? colNullOrder?)*
```

query: SELECT expression (',', expression)* FROM src orderBy

- Sorts the rows before sending the rows to the reducer (depends on column types)
- Guarantees ordering of the rows

```
colOrder: ( ASC | DESC )
sortBy: SORT BY colName colOrder? (',', colName colOrder?)*
query: SELECT expression (',', expression)* FROM src sortBy
```

Hive Data Retrieval - Distribute By

- Hive uses the columns in Distribute By to distribute the rows among reducers
- All rows with the same Distribute By columns will go to the same reducer
- Distribute By does not guarantee clustering or sorting properties on the distributed keys

```
SELECT col1, col2 FROM t1 DISTRIBUTE BY col1;  
SELECT col1, col2 FROM t1 DISTRIBUTE BY
```

Hive Data Retrieval Examples

Order By Statement

```
SELECT Id, Name, Dept FROM employee  
ORDER BY DEPT;
```

Sort By Statement

```
SELECT key, value FROM src SORT BY key  
ASC, value DESC
```

Distribute By Statement

```
insert overwrite table mytable  
select gender,age,salary  
from salaries  
distribute by age  
sort by age;
```

Hive Views

- Hive Views simplify the complexities of a larger table into a Flat structure.
- Using Views versus Tables don't significant alter performance when executing a query
- In some instances, views can be used as 'transient' or 'staging' tables to expedite more

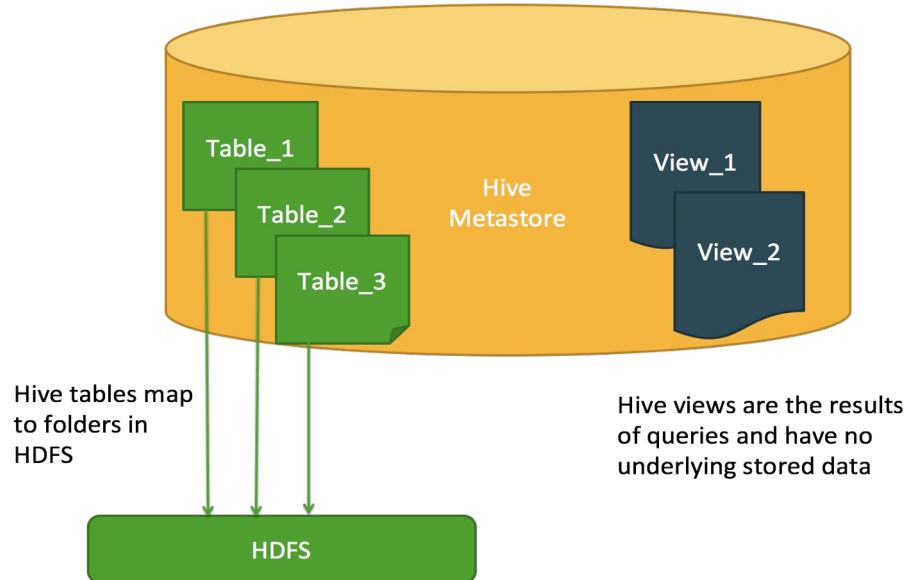
```
CREATE VIEW [IF NOT EXISTS]
[db_name.]view_name [(column_name
[COMMENT column_comment], ... ) ]
[COMMENT view_comment]
[TBLPROPERTIES (property_name =
property_value, ...)]
AS SELECT ...;
```

```
ALTER VIEW [db_name.]view_name SET
TBLPROPERTIES table_properties;

table_properties:
  : (property_name = property_value,
property_name = property_value, ...)
```

```
DROP VIEW [IF EXISTS] [db_name.]view_name;
```

Understanding Views



Hive View Example

```
CREATE VIEW hortonworks_website
  (url COMMENT 'URL of Website page')
  COMMENT 'Hortonworks Website'
AS
SELECT DISTINCT website_url
  FROM page_view
 WHERE page_url='http://www.hortonworks.com';
```

```
DROP VIEW hortonworks_website;
```

Hive 3.0 Materialized Views

- Materialized Views contain data that is pre-aggregated which is used for query execution.
- Materialized views can be stored natively in Hive or in external system such as Apache Druid (Using custom storage handlers)
- The optimizer relies on Apache Calcite to automatically produce rewritings for a large set of query expressions comprising projections, filters join and aggregation operations

Create Statement

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] [db_name.]materialized_view_name  
[DISABLE REWRITE]  
[COMMENT materialized_view_comment]  
[  
    [ROW FORMAT row_format]  
    [STORED AS file_format]  
    | STORED BY 'storage.handler.class.name' [WITH SERDEPROPERTIES (...)]  
]  
[LOCATION hdfs_path]  
[TBLPROPERTIES (property_name=property_value, ...)]  
AS  
<query>;
```

Druid Storage

```
CREATE MATERIALIZED VIEW druid_wiki_mv  
STORED AS 'org.apache.hadoop.hive.druid.DruidStorageHandler'  
AS  
SELECT __time, page, user, c_added, c_removed  
FROM src;
```

Hive 3.0 Materialized View Operations

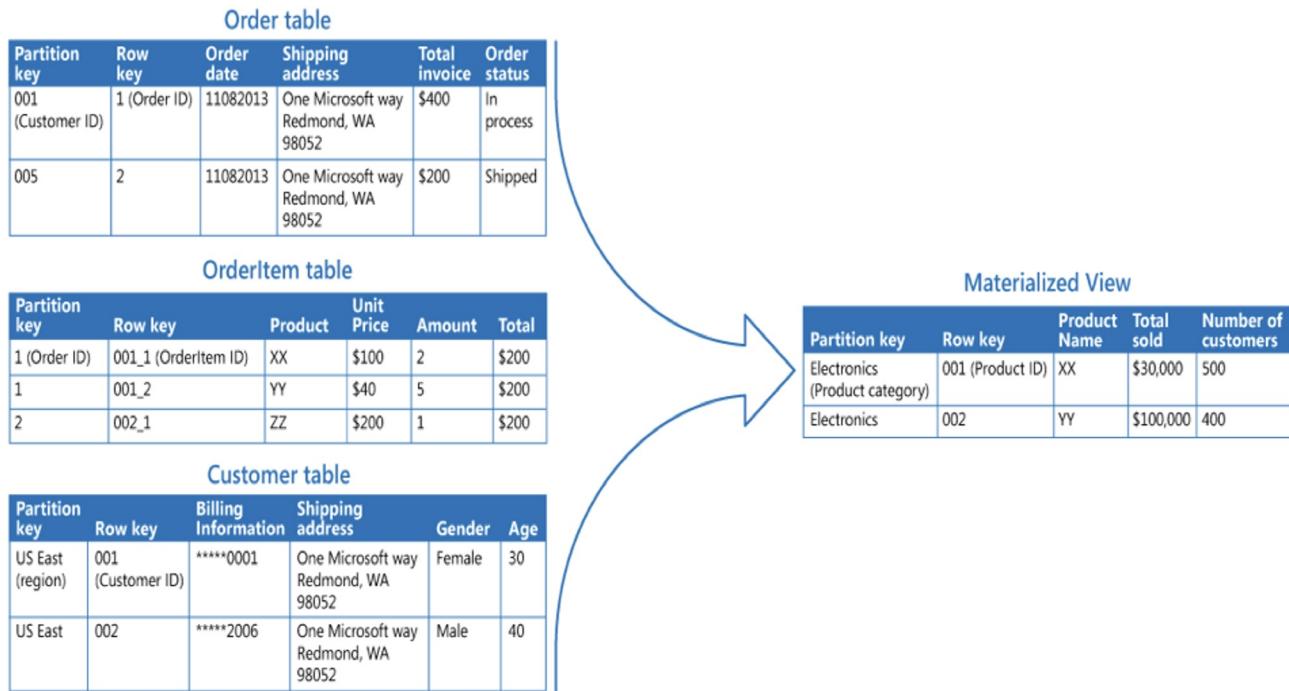
Operation statements

```
-- Drops a materialized view  
DROP MATERIALIZED VIEW [db_name.]materialized_view_name;  
-- Shows materialized views (with optional filters)  
SHOW MATERIALIZED VIEWS [IN database_name]  
['identifier_with_wildcards'];  
-- Shows information about a specific materialized view  
DESCRIBE [EXTENDED | FORMATTED]  
[db_name.]materialized_view_name;
```

Query Rewriting statement

```
ALTER MATERIALIZED VIEW [db_name.]materialized_view_name  
ENABLE|DISABLE REWRITE;
```

Hive Materialized View Example



Hive 3.0 Materialized Views Example (Continued)

Create Materialized View

```
CREATE MATERIALIZED VIEW mv1
AS SELECT empid, deptname, hire_date
FROM emps JOIN depts
ON (emps.deptno = depts.deptno)
WHERE hire_date >= '2017-01-01';
```

Execute Query

```
SELECT empid, deptname
FROM mv1
JOIN depts
ON (emps.deptno = depts.deptno)
WHERE hire_date >= '2017-01-01'
AND hire_date <= '2019-01-01';
```

Explain Statement

```
EXPLAIN EXTENDED SELECT empid, deptname
FROM mv1
JOIN depts
ON (emps.deptno = depts.deptno)
WHERE hire_date >= '2017-01-01'
AND hire_date <= '2019-01-01';
```

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- **Hands-On Exercise: Materialized Views**
- Hive Joins
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- Hive User Defined Functions
- Essential Points

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-37

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- **Hive Joins**
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- Hive User Defined Functions
- Essential Points

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-38

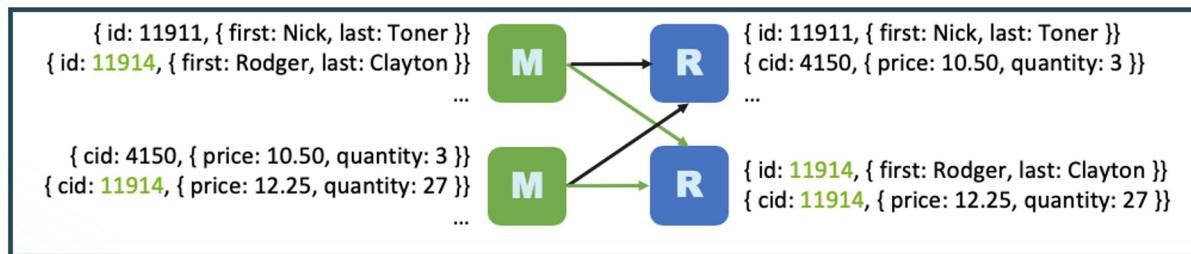
Join Strategies

Type	Approach	Pros	Cons	MapReduce Type
Shuffle Join	Join keys are shuffled using map/reduce and joins performed join side.	Works regardless of data size or layout.	Most resource-intensive and slowest join type.	Reduce-Side
Broadcast Join	Small tables are loaded into memory in all nodes, mapper scans through the large table and joins.	Very fast, single scan through largest table.	All but one table must be small enough to fit in RAM.	Map-Side
Sort-Merge-Bucket Join	Mappers take advantage of co-location of keys to do efficient joins.	Very fast for tables of any size.	Data must be sorted and bucketed ahead of time.	Map-Side

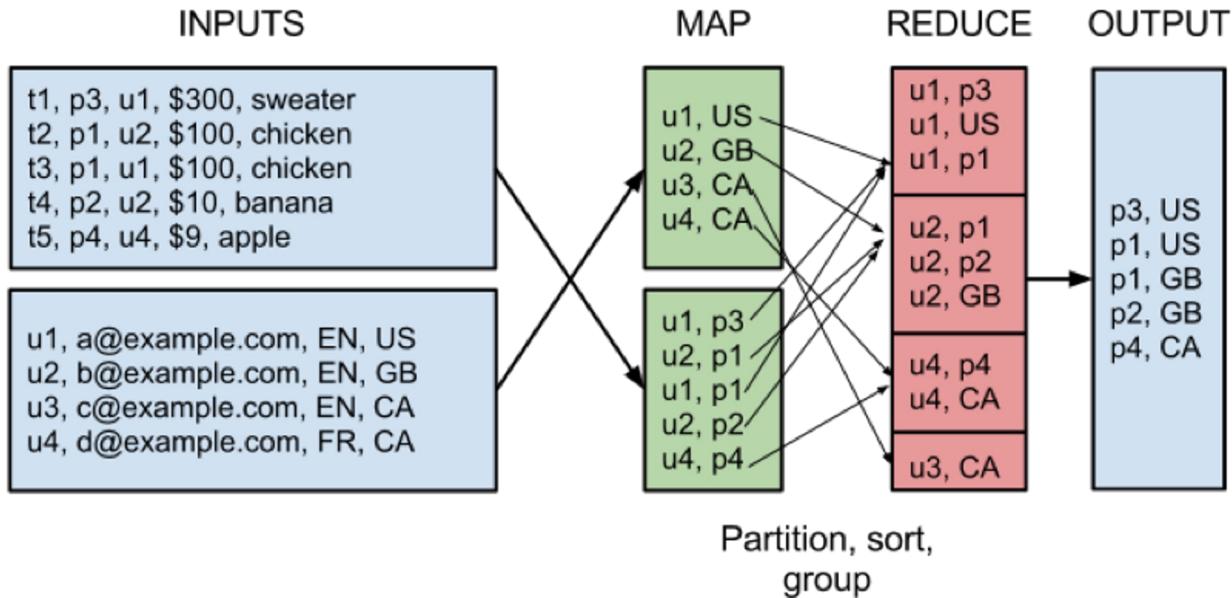
Shuffle Joins – (Default Reducer-Side Join)

customer			orders		
first	last	id	cid	price	quantity
Nick	Toner	11911	4150	10.50	3
Jessie	Simonds	11912	11914	12.25	27
Kasi	Lamers	11913	3491	5.99	5
Rodger	Clayton	11914	2934	39.99	22
Verona	Hollen	11915	11914	40.50	10

SELECT * FROM customer JOIN orders ON customer.id = orders.cid;



Reducer-Side Join Visualized



Broadcast Join – (Map-Side Join)

customer			orders		
first	last	id	cid	price	quantity
Nick	Toner	11911	11914	4150	10.50
Jessie	Simonds	11912		12.25	27
Kasi	Lamers	11913		5.99	5
Rodger	Clayton	11914		39.99	22
Verona	Hollen	11915		40.50	10

SELECT * FROM customer JOIN orders ON customer.id = orders.cid;

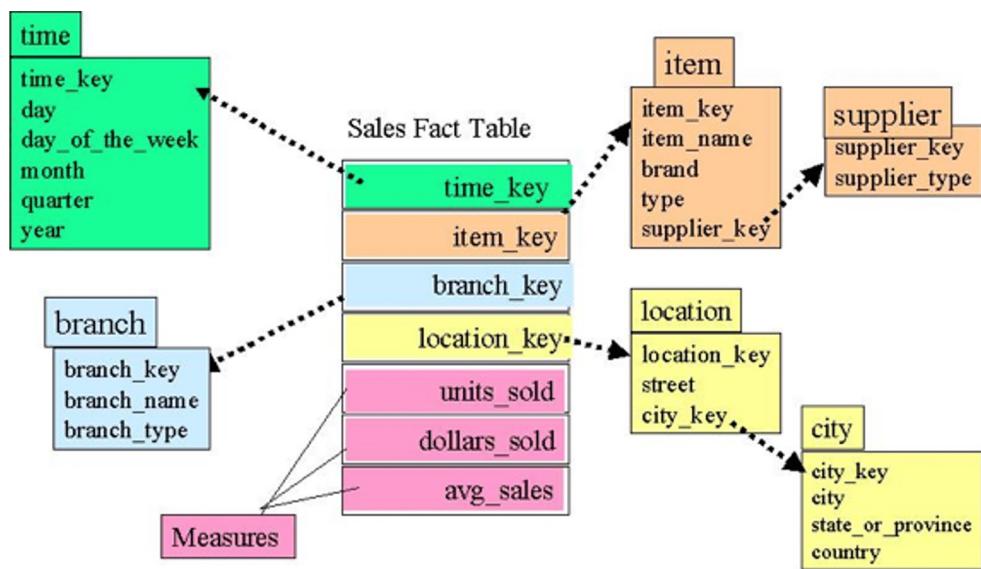
```
{ id: 11914, { first: Rodger, last: Clayton } }  
{ cid: 11914, { price: 12.25, quantity: 27 } }  
cid: 11914, { price: 12.25, quantity: 27 }
```



Records are joined during
the map phase.

Map-Side Join Visualized

- Star schemas (e.g. dimensional tables)
- Good when table is small enough to fit in RAM



Using Hive Broadcast Joins

- **HIVE automatically uses broadcast join when enabled:**
 - `hive.auto.convert.join=true`
- **Control the size of the join table**
 - `hive.auto.convert.join.nonconditionaltask.size`

Soft-Merge-Bucket (SMB) Joins

customer			orders		
first	last	id	cid	price	quantity
Nick	Toner	11911	4150	10.50	3
Jessie	Simonds	11912	11914	12.25	27
Kasi	Lamers	11913	11914	40.50	10
Rodger	Clayton	11914	12337	39.99	22
Verona	Hollen	11915	15912	40.50	10

SELECT * FROM customer join orders ON customer.id = orders.cid;

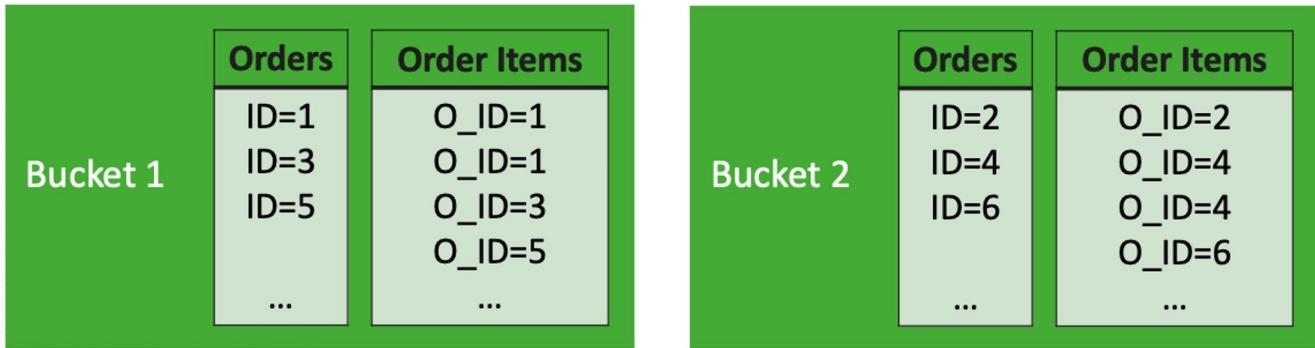
Distribute and sort by the most common join key.

```
CREATE TABLE orders (cid int, price float, quantity int)
CLUSTERED BY(cid) SORTED BY(cid) INTO 32 BUCKETS;
```

```
CREATE TABLE customer (id int, first string, last string)
CLUSTERED BY(id) SORTED BY(id) INTO 32 BUCKETS;
```

SMB-Join Visualized

- Joins individually occur on map-side for each bucket, not the complete dataset, level
 - Hive assumes that join key placement in buckets are aligned for all datasets
 - The collective results represent the comprehensive join request
- Leverages hash partitions and requires equal number of buckets for datasets to be joined



Hive Join - (Cartesian Joins)

- Cross join or Cartesian Joins, are a way to join multiple tables in which all the rows of one table are joined with all the rows of another table
- Support has been added for cartesian product edge joins
(For Outer Joins, this may cause a performance issue and should be set to False)
 - `hive.tez.cartesian-product.enabled=true`

Hive Join Optimization

- **Joins where one side fits in memory**
 - That side is loaded into memory as a hash table
 - Only larger table is scanned
 - Fact tables have smaller footprint in memory
- **Star-schema joins are optimized.**
- **Hints are no longer needed for many use cases.**
- **Map joins are automatically picked up by the optimizer**

Hive Union

- UNION is used to combine the result from multiple SELECT statements into a single result set
- The default behavior for UNION is that duplicate rows are removed from the result

```
select_statement UNION [ALL | DISTINCT] select_statement UNION [ALL | DISTINCT]  
select_statement ...
```

```
SELECT *  
FROM (  
    select_statement  
    UNION ALL  
    select_statement  
) unionResult
```

Hive Subqueries

- Hive supports subqueries only in the FROM clause
- The subquery has to be given a name because every table in a FROM clause must have a name
- Columns in the subquery select list are available in the outer query like columns of a table

```
SELECT ... FROM (subquery) name ...
SELECT ... FROM (subquery) AS name ...
```

```
SELECT col
FROM (
    SELECT a+b AS col
    FROM t1
) t2
```

Hive ngrams

- An **ngram** is a subsequence of text within a large document

```
select ngrams(sentences(val),2,100) from mytable;
```

```
select context_ngrams(sentences(val),  
                      array("error","code",null),100)  
      from mytable;
```

Vectorization

- Vectorized query execution is a Hive feature that greatly reduces CPU usage for typical query operations such as scan, filter and joins
- Vectorized query execution streamlines operations by processing a block of 1024 rows at a time

```
Enabling Vectorized execution
```

```
set hive.vectorized.execution.enabled = true; (Off by default)
```

**Vectorization + ORC files = a huge breakthrough in Hive
query performance**

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- Hive Joins
- **Hands-On Exercise: Hive Joins**
- Hive Windowing and Grouping
- Hive User Defined Functions
- Essential Points

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-53

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- Hive Joins
- Hands-On Exercise: Hive Joins
- **Hive Windowing and Grouping**
- Hive User Defined Functions
- Essential Points

CLOUDERA

Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

10-54

Hive Windowing

- **Windowing in Hive allows the ability to create a window of data to operate aggregation and analytical functions**
- **Windowing in Hive contains the following functions**
 - OVER Clause
 - OVER with a Partition BY statement

The OVER Clause

orders			result set	
cid	price	quantity	cid	max(price)
4150	10.50	3	2934	39.99
11914	12.25	27	4150	10.50
4150	5.99	5	11914	40.50
2934	39.99	22		
11914	40.50	10		

SELECT cid, max(price) FROM orders GROUP BY cid;

orders			result set	
cid	price	quantity	cid	max(price)
4150	10.50	3	2934	39.99
11914	12.25	27	4150	10.50
4150	5.99	5	4150	10.50
2934	39.99	22	11914	40.50
11914	40.50	10	11914	40.50

SELECT cid, max(price) OVER (PARTITION BY cid) FROM orders;

Using Windows

orders			result set	
cid	price	quantity	cid	sum(price)
4150	10.50	3	4150	5.99
11914	12.25	27	4150	16.49
4150	5.99	5	4150	36.49
4150	39.99	22	4150	70.49
11914	40.50	10	11914	12.25
4150	20.00	2	11914	52.75

SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY price ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) FROM orders;

Using Windows (Continued)

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY price  
ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING) FROM orders;
```

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY price  
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) FROM orders;
```

Hive Analytics Function

orders			result set		
cid	price	quantity	cid	quantity	rank
4150	10.50	3	4150	2	1
11914	12.25	27	4150	3	2
4150	5.99	5	4150	5	3
4150	39.99	22	4150	22	4
11914	40.50	10	11914	10	1
4150	20.00	2	11914	27	2

SELECT cid, quantity, rank() OVER (PARTITION BY cid ORDER BY quantity) FROM orders;

Hive Grouping, Cubes, and Rollups

- **ROLLUP** statement enables a Select statement to calculate multiple levels of aggregations across a specified group of dimensions
- **CUBE** creates a subtotal of all possible combinations of the set of column in its argument
- **Grouping sets** create specified levels of aggregations
- The **grouping_ID** function, returns a value of “0” or “1” if that column has been aggregated in that row

```
SELECT key, value,  
GROUPING__ID, count(*)  
FROM T1  
GROUP BY key, value WITH  
ROLLUP;
```

```
GROUP BY a,b,c  
GROUPING SETS  
((a,b,c),(a,b),(a),())
```

```
GROUP BY a,b,c  
WITH CUBE
```

Chapter Topics

Transforming Data with Hive

- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- Hive Joins
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- **Hive User Defined Functions**
- Essential Points

Hive 3.0 User Defined Functions (UDF)

- **User Defined Function (UDF)** are Java-based and used as an extension of Hive's query language (**HiveQL**)
- UDFs are used to create your own logic in terms of code into Hive code
- Three types of UDFs:
 - Regular UDF
 - User Defined Aggregate Function(UDAF)
 - User Defined Tabular Function (UDTF)

Regular User Defined Function (UDF)

- Regular UDFs work on a single row in a table to produce a single row output.
- Customized UDF
 - Overload the evaluate() function to write business logic in UDF
 - Regular UDF logic in Hive JAR is exported to HDFS added to classpath in Hive to be executed

Example Simple UDF

```
SELECT lower(str) from table
```

User Defined Aggregate Function (UDAF)

- User Defined aggregate functions that work on more than one row and give a single row as output
- Customized UDAF
 - Overwrite `init()`, `iterate()`, `terminatePartial()`, `merge()`, and `terminate()` with custom logic
 - UDAF logic in Hive JAR is exported to HDFS added to classpath in Hive to be executed

Example Simple UDAF

```
SELECT Count(col) from table
```

User Defined Tabular Function (UDTF)

- User Defined tabular functions work on one row as input and return multiple rows as output
- Customized UDTF
 - Overwrite initialize(), process() and close() with custom logic
 - UDTF logic in Hive JAR is exported to HDFS added to classpath in Hive to be executed

Example Simple UDTF

```
SELECT explode(split(col,',')) from table
```

Chapter Topics

Transforming Data with Hive

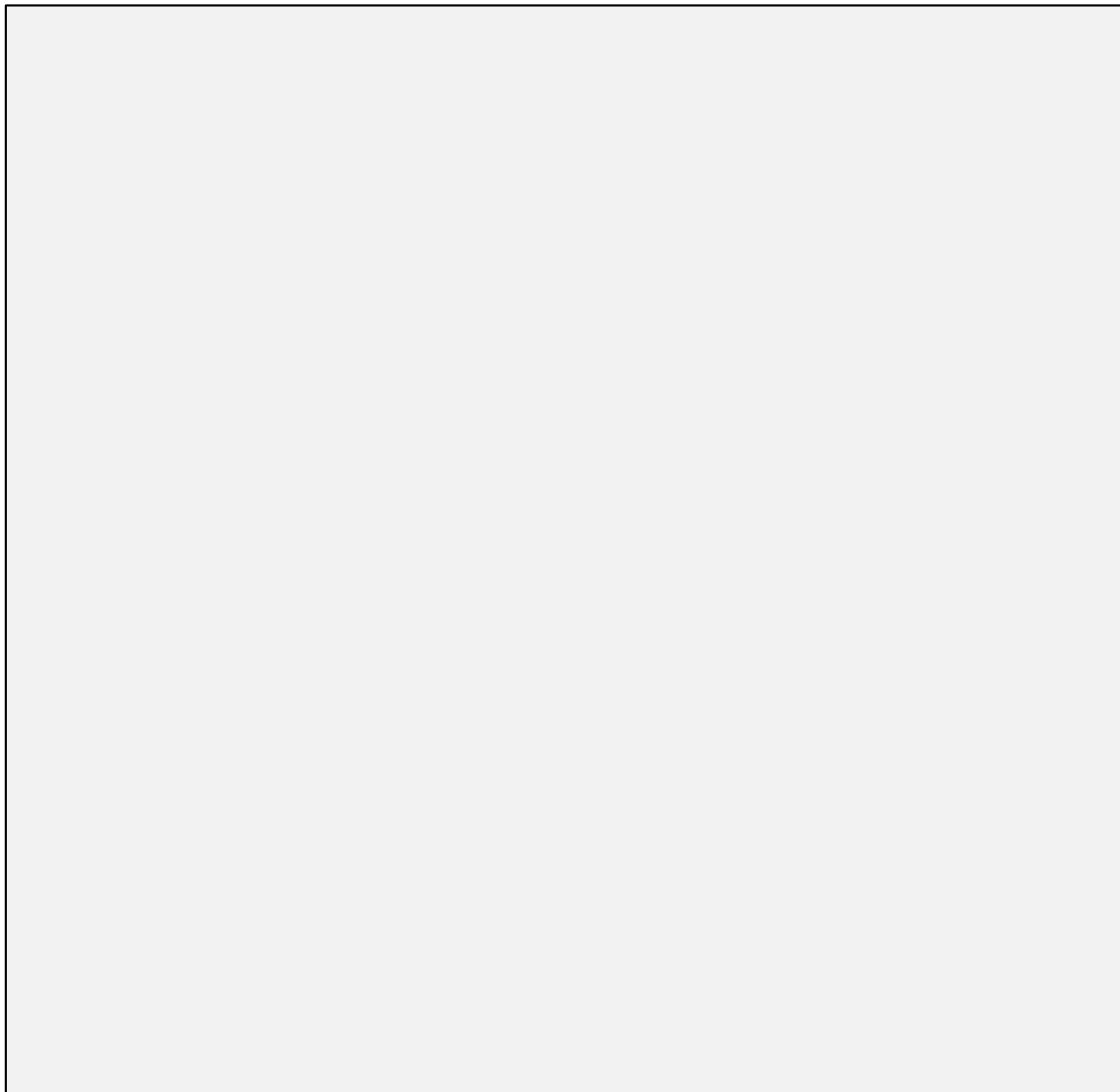
- Introduction to Hive SQL
- Hands-On Exercise: Hive SQL
- Hive ACID Transactions
- Hands-On Exercise: Hive Transactions
- Data Retrieval, Views, and Materialized Views
- Hands-On Exercise: Materialized Views
- Hive Joins
- Hands-On Exercise: Hive Joins
- Hive Windowing and Grouping
- Hive User Defined Functions
- **Essential Points**

Essential Points

- Beeline commands can be used to connect to Hive databases and execute queries
- Hive tables are managed internally or externally
- Hive 3.0 provides ACID support for transactional tables
- Group By, Sort By and Distribute by are functions used by Hive to retrieve data
- Hive 3.0 uses materialized views to pre-aggregate data for queries
- Hive Joins and subqueries are used to relate multiple tables in Hive databases
- Windowing and grouping is used to retrieve data over a specific data clause
- User Defined Functions (UDFs) are Java based extensions of HiveQL

Data Engineering with Hive

Chapter 11



Course Chapters

- Introduction
- Why Data Engineering
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- **Transforming Data with Hive**
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- Conclusion
- Appendix: Working with Datasets in Scala

Objectives

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

- Explain how partitioning subdivides the data
- Create partitioned tables to speed up queries that filter on columns
- Load data into static and dynamic partitions
- Use buckets to break data into ranges
- Access data that is not in a structured tabular format
- Denormalize data using complex data types

Chapter Topics

Data Engineering with Hive

- **Working with Partitions**
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- Working with Skew
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

Hive Data Abstractions

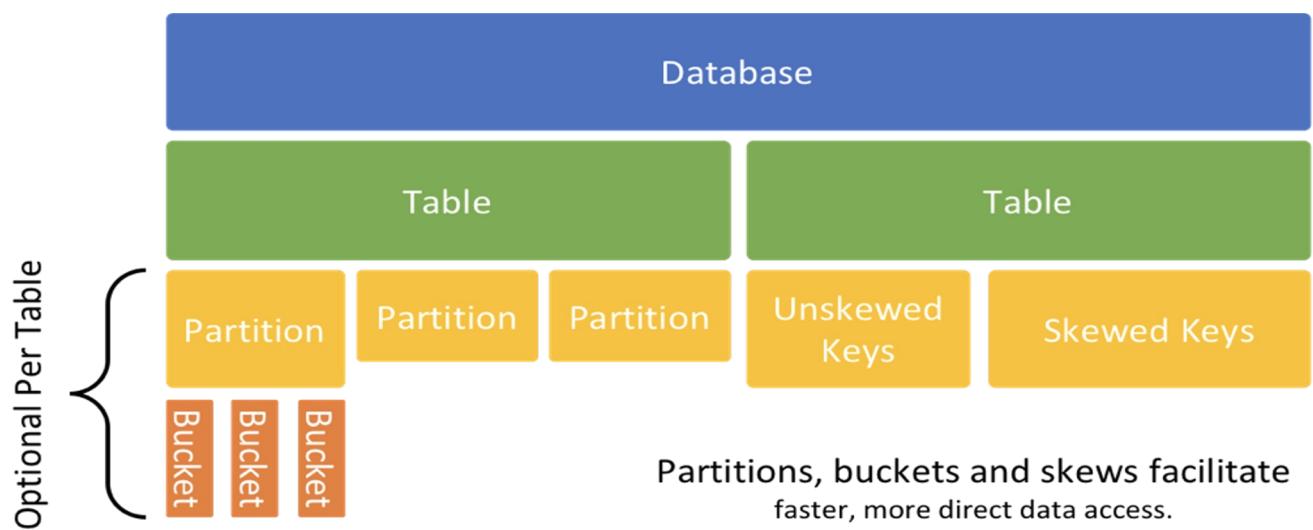


Table Partitioning

- By default, all data files for a table are stored in a single directory
 - All files in the directory are read during a query
- Partitioning subdivides the data
 - Data is physically divided during loading, based on values from one or more columns
- Speeds up queries that filter on partition columns
 - Only the files containing the selected data need to be read
- Does not prevent you from running queries that span multiple partitions

Example: Partitioning Customers by State (1)

- Example: customer is a non-partitioned table

```
CREATE EXTERNAL TABLE customers (
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    state STRING,
    zipcode STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/analyst/dualcore/customers';
```

Example: Partitioning Customers by State (2)

- Data files are stored in a single directory
- All files are scanned for every query

/analyst/dualcore/customers

1000000 Quentin Shepard 32092 West 10th Street Prairie City SD 57649	file1
1000001 Brandon Louis 1311 North 2nd Street Clearfield IA 50840	
1000002 Marilyn Ham 25831 North 25th Street Concord CA 94522	
...	

1050344 Denise Carey 1819 North Willow Parkway Phoenix AZ 85042	file2
1050345 Donna Pettigrew 1725 Patterson Street Garberville CA 95542	
1050346 Hans Swann 1148 North Hornbeam Avenue Sacramento CA 94230	
...	

Example: Partitioning Customers by State (3)

- What if most of Dualcore's analysis on the customer table was done by state?

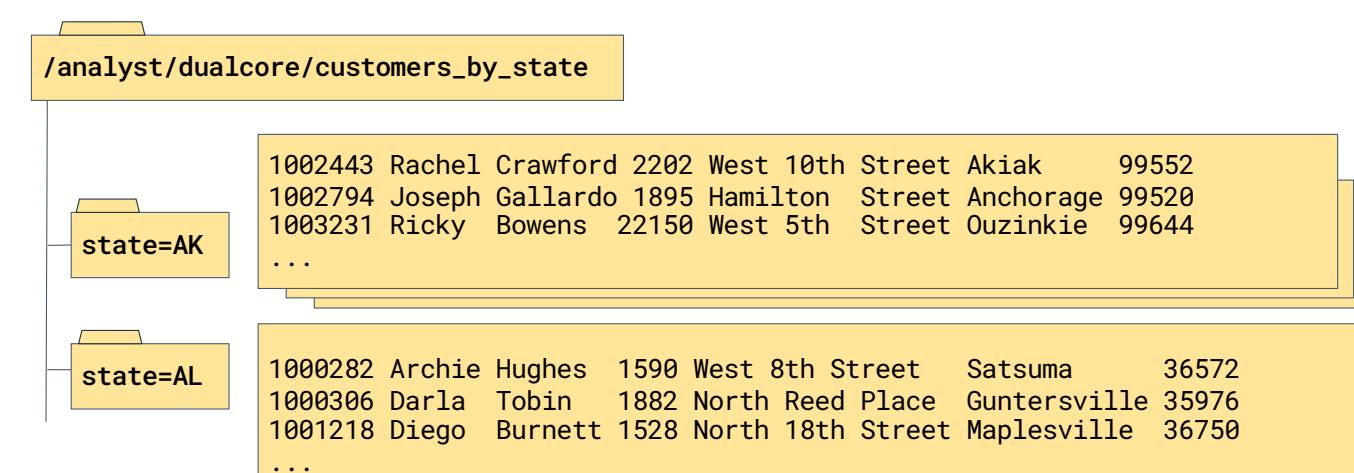
For example:

```
SELECT o.order_date, c.fname, c.lname  
FROM customers c  
JOIN orders o ON (c.cust_id = o.cust_id)  
WHERE c.state='NY';
```

- By default, all queries have to scan all files in the directory
- Use partitioning to store data in separate subdirectories by state
 - Queries that filter by state scan only the relevant subdirectories

Partitioning File Structure

- **Partitioned tables store data in subdirectories**



Creating a Partitioned Table

- Example: customer is a partitioned table using **PARTITIONED BY**

```
CREATE EXTERNAL TABLE customers_by_state (
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    zipcode STRING)
PARTITIONED BY (state STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/analyst/dualcore/customers_by_state';
```

Partitioned Columns

- The partition column is displayed if you **DESCRIBE** the table

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

A partition column is a *virtual column*; column values are not stored in the files

Nested Partitions

- You can also create nested partitions

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



Loading Data into a Partitioned Table

- **Static partitioning**

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

- **Dynamic partitioning**

- Hive/Impala automatically creates partitions
 - Inserted data is stored in the correct partitions based on column values

Static Partitioning

- With static partitioning, you create each partition manually

```
ALTER TABLE customers_by_state ADD PARTITION (state='NY');
```

- Then add data one partition at a time

```
INSERT OVERWRITE TABLE customers_by_state PARTITION(state='NY')
SELECT cust_id, fname, lname, address, city, zipcode
FROM customers
WHERE state='NY';
```

Static Partitioning Example: Partition Calls by Day (1)

- Dualcore's call center generates daily logs detailing calls received

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

call-20161001.log

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

call-20161002.log

Static Partitioning Example: Partition Calls by Day (2)

- The partitioned table is defined the same way

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

Static Partitioning Example: Partition Calls by Day (3)

- **We use static partitioning**
 - The data is already partitioned by day into separate files
 - The data can be loaded one partition at a time
- **With static partitioning, you create new partitions as needed**
 - For each new day of call log data, add a partition:

```
ALTER TABLE call_logs ADD PARTITION (call_date='2016-10-01');
```

- **This command**

1. Adds the partition to the table's metadata
2. Creates subdirectory `call_date=2016-10-01` in
`/warehouse/tablespace/managed/hive/call_logs/`

Static Partitioning Example: Partition Calls by Day (4)

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

```
LOAD DATA INPATH '/analyst/dualcore/call-20161001.log'  
INTO TABLE call_logs  
PARTITION(call_date='2016-10-01');
```

- This command moves the HDFS file call-20161001.log to the partition subdirectory

- To overwrite all data in a partition

```
LOAD DATA INPATH '/analyst/dualcore/call-20161001.log'  
OVERWRITE INTO TABLE call_logs  
PARTITION(call_date='2016-10-01');
```

Hive Shortcut for Loading Data into Static Partitions

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

```
LOAD DATA INPATH '/analyst/dualcore/call-20161002.log'  
  INTO TABLE call_logs  
  PARTITION(call_date='2016-10-02');
```

- This command

1. Adds the partition to the table's metadata if it doesn't exist
2. Creates subdirectory `call_date=2016-10-02` in `call_logs` if it doesn't exist
3. Moves the HDFS file `call-20161002.log` to the partition subdirectory

Dynamic Partitioning

- When loading data with **INSERT**, use the **PARTITION** clause
 - The partition column(s) must be included in the PARTITION clause
 - The partition column(s) must be specified last in the SELECT list
- Hive or Impala creates partitions and inserts data based on values of the partition column
 - The values of the partition column(s) are not included in the files

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

Caution: If the partition column has many unique values, many partitions will be created

Partition Configuration Properties

- Three Hive configuration properties exist to limit creating too many partitions

- **hive.exec.max.dynamic.partitions.pernode**

- Maximum number of dynamic partitions that can be created by any given node involved in a query
 - Default 2000

- **hive.exec.max.dynamic.partitions**

- Total number of dynamic partitions that can be created by one HiveQL statement
 - Default 5000

- **hive.exec.max.created.files**

- Maximum total files (on all nodes) created by a query
 - Default 100000

Viewing, Adding, and Removing Partitions

▪ View the current partitions in a table

```
SHOW PARTITIONS call_logs;
```

▪ Add or drop partitions

```
ALTER TABLE call_logs ADD PARTITION (call_date='2016-06-05');
```

```
ALTER TABLE call_logs DROP PARTITION (call_date='2016-06-05');
```

When to Use Partitioning

- **Use partitioning for tables when**

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

When Not to Use Partitioning

- **Avoid partitioning data into numerous small data files**
 - Partitioning on columns with too many unique values
- **Caution: This can happen easily when using dynamic partitioning!**
 - For example, partitioning customers by first name could produce thousands of partitions

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- **Hands-On Exercise: Hive Partitions**
- Working with Buckets
- Working with Skew
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

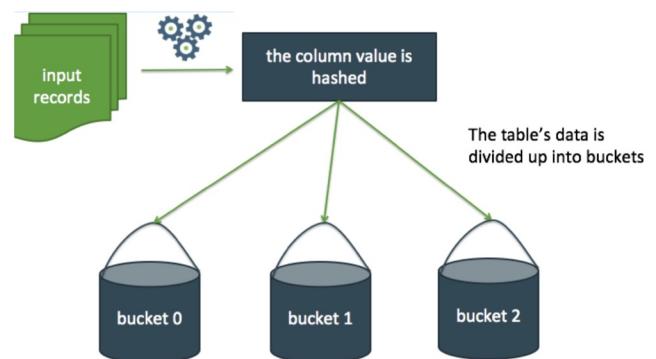
Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- **Working with Buckets**
- Working with Skew
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

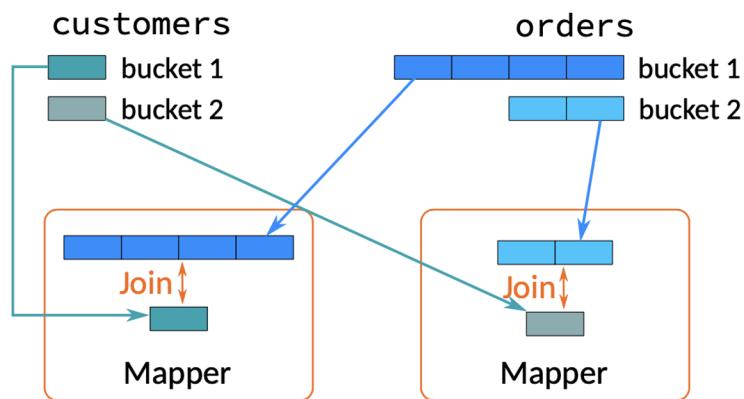
Bucketing Data in Hive

- **Partitioning subdivides data by values in partitioned columns**
 - Stores data in separate subdirectories
 - Divides data based on columns with a limited number of discrete values
 - Generally the preferred way to subdivide data
- **Bucketing data is another way of subdividing data**
 - Stores data in separate files
 - Divides data into buckets in an effectively random way
 - Calculates hash codes based on column values
 - Uses hash code to assign records to buckets
 - Each bucket is one file



Benefits of Bucketing

- Allows faster access based on the key columns
 - Optimizes joins when bucket key column is the join column



- Provides a way to sample data

Creating a Bucketed Table

- **Example of creating a table that supports bucketing**

- Creates a table supporting 20 buckets based on order_id column
 - Each bucket should contain roughly 5% of the table's data

```
CREATE TABLE orders_bucketed
  (order_id INT,
   cust_id INT,
   order_date TIMESTAMP)
CLUSTERED BY (order_id) INTO 20 BUCKETS;
```

- **Column selected for bucketing should have well-distributed values**

- Identifier columns are often a good choice

Inserting and Sampling Data with a Bucketed Table

- To insert data, use **INSERT OVERWRITE TABLE**

```
INSERT OVERWRITE TABLE orders_bucketed SELECT * FROM orders;
```

- To sample data, use **TABLESAMPLE**

- This example selects one of every ten records (10%)

```
SELECT * FROM orders_bucketed TABLESAMPLE (BUCKET 1 OUT OF 10 ON order_id);
```

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- **Working with Skew**
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

Skewed Data Joins in Hive

- Hive supports “compile-time” and “run-time” skewed joins
- Compile-time support requires popular values in metadata

```
create table Sales skewed by (storeID) on (500);  
SET hive.optimize.skewjoin.compiletime=true;
```

- Run-time support computes “popular” values for join key
 - Must be activated
 - SET `hive.optimize.skewjoin=true`;
 - Set threshold for being popular
 - `SET hive.skewjoin.key=100000`;
- In both cases, the plan is broken into different joins:
 - One for the popular values – map join
 - Other for the remaining values – common join

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- Working with Skew
- **Instructor-Led Demo: Skewed Table**
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- Working with Skew
- Instructor-Led Demo: Skewed Table
- **Using Serdes to Ingest Text Data**
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

Text Processing Overview

- **Traditional data processing relies on structured tabular data**
 - Carefully curated information in rows and columns
- **What types of data are we producing today?**
 - Unstructured text data
 - Semi-structured data in formats like JSON
 - Log files
- **Examples of unstructured and semi-structured data include**
 - Free-form notes in electronic medical records
 - Electronic messages
 - Product reviews
- **These types of data also contain great value**
 - Extracting it requires a different approach
 - Apache Hive provides some special functions for text analysis
 - Cloudera Search (based on Apache Solr) is designed specifically for this
 - More sophisticated with higher performance than Hive

Hive Record Formats

- So far we have used only **ROW FORMAT DELIMITED** when creating tables stored as text files
 - Requires data in rows and columns with consistent delimiters
- But Hive supports other record formats
- A SerDe is an interface Hive uses to read and write data
 - SerDe stands for *serializer/deserializer*
- SerDes enable Hive to access data that is not in structured tabular format

Hive SerDes

- You specify the SerDe when creating a table in Hive
 - Sometimes it is specified implicitly
- Hive includes several built-in SerDes for record formats in text files

Name	Reads and Writes Records
LazySimpleSerDe	Using specified field delimiters (default)
RegexSerDe	Based on supplied patterns
OpenCSVSerde	in CSV format
JsonSerDe	in JSON format

Specifying a Hive SerDe

- Previously, we specified the row format using **ROW FORMAT DELIMITED** and **FIELDS TERMINATED BY**
 - LazySimpleSerDe is specified implicitly

```
CREATE TABLE people(fname STRING, lname STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';
```

- You can also specify the SerDe explicitly

- Using **ROW FORMAT SERDE**

```
CREATE TABLE people(fname STRING, lname STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES ('field.delim'='\t');
```

Log Files

- We sometimes need to analyze data that lacks consistent delimiters

- Log files are a common example of this

```
05/23/2016 19:45:19 312-555-7834 CALL_RECEIVED ""
05/23/2016 19:45:23 312-555-7834 OPTION_SELECTED "Shipping"
05/23/2016 19:46:23 312-555-7834 ON_HOLD ""
05/23/2016 19:47:51 312-555-7834 AGENT_ANSWER "Agent ID N7501"
05/23/2016 19:48:37 312-555-7834 COMPLAINT "Item damaged"
05/23/2016 19:48:41 312-555-7834 CALL_END "Duration: 3:22"
```

Creating a Table with Regex SerDe (1)

```
05/23/2016 19:45:19 312-555-7834 CALL_RECEIVED ""
05/23/2016 19:48:37 312-555-7834 COMPLAINT "Item damaged"
```

Log excerpt

```
CREATE TABLE calls (
    event_date STRING,
    event_time STRING,
    phone_num STRING,
    event_type STRING,
    details STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES ("input.regex" =
    "([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) \"([^\"]*)\"");
```

Regex SerDe

- **RegexSerDe** reads records based on a regular expression
- The regular expression is specified using **SERDEPROPERTIES**
 - Each pair of parentheses denotes a field
 - Field value is text matched by pattern within parentheses

Creating a Table with Regex SerDe (2)

```
05/23/2016 19:45:19 312-555-7834 CALL_RECEIVED ""  
05/23/2016 19:48:37 312-555-7834 COMPLAINT "Item damaged"
```

Log excerpt

```
CREATE TABLE calls (  
    event_date STRING,  
    event_time STRING,  
    phone_num STRING,  
    event_type STRING,  
    details STRING)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'  
WITH SERDEPROPERTIES ("input.regex" =  
    "([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) \"([^\"]*)\"");
```

Regex SerDe

event_date	event_time	phone_num	event_type	details
05/23/2016	19:45:19	312-555-7834	CALL_RECEIVED	
05/23/2016	19:48:37	312-555-7834	COMPLAINT	Item damaged

Fixed-Width Formats

- Many older applications produce data in fixed-width formats



- Hive doesn't directly support fixed-width formats
 - But you can overcome this limitation by using RegexSerDe

Fixed-Width Format Example

10309296107596201608290122150akland

CA94618

cust_id order_id order_dt order_tm city state zip

```
CREATE TABLE fixed (
  cust_id INT,
  order_id INT, order_dt STRING, order_tm STRING,
  city STRING, state STRING, zip STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES ("input.regex" =
  "(\\d{7})(\\d{7})(\\d{8})(\\d{6}).{20}(\\w{2})(\\d{5})");
```

cust_id	order_id	order_dt	order_tm	state	zip
1030929	6107596	20160829	012215	CA	94618

CSV Format

- Simple comma-delimited data can be processed using the default SerDe
 - With ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
- But the actual CSV format is more complex, and handles cases including
 - Embedded commas
 - Quoted fields
 - Missing values
- Hive provides **OpenCSVSerde** for processing CSV data
 - Also supports other delimiters such as tab (\t) and pipe (|)

CSV SerDe Example

```
1,Gigabux,gigabux@example.com  
2,"ACME Distribution Co.",acme@example.com  
3,"Bitmonkey, Inc.",bmi@example.com
```

Input Data

```
CREATE TABLE vendors  
(id INT,  
 name STRING,  
 email STRING)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde';
```

CSV SerDe

id	name	email
1	Gigabux	gigabux@example.com
2	ACME Distribution Co.	acme@example.com
3	Bitmonkey, Inc.	bmi@example.com

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- Working with Skew
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- **Hands-On Exercise: Analyzing Text with Hive**
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- Working with Skew
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- **Using Complex Types to Denormalize Data**
- Hands-On Exercise: Complex Data
- Essential Points

Complex Data Types

- **Hive has support for complex data types**
 - Represent multiple values within a single row/columns position

Data Type	Description
ARRAY	Ordered list of values, all of the same type
MAP	Key-value pairs, each of the same type
STRUCT	Named fields, of possibly mixed types

Why Use Complex Values?

- **Can be more efficient**
 - Related data is stored together
 - Avoids computationally expensive join queries
- **Can be more flexible**
 - Store an arbitrary amount of data in a single row
- **Sometimes the underlying data is already structured this way**
 - Other tools and languages represent data in nested structures
 - Avoids the need to transform data to flatten nested structures

Example: Customer Phone Numbers

- Traditional storage of multiple phone numbers for customers

customers table	
cust_id	name
a	Alice
b	Bob
c	Carlos

phones table	
cust_id	phone
a	555-1111
a	555-2222
a	555-3333
b	555-4444
c	555-5555
c	555-6666

```
SELECT  
c.cust_id,  
c.name,  
p.phone  
FROM customers c  
JOIN phones p  
ON (c.cust_id =  
p.cust_id)
```

Query results		
cust_id	name	phone
a	Alice	555-1111
a	Alice	555-2222
a	Alice	555-3333
b	Bob	555-4444
c	Carlos	555-5555
c	Carlos	555-6666

Using ARRAY Columns with Hive (1)

- Using the ARRAY type allows us to store the data in one table

customers_phones table		
cust_id	name	phones
a	Alice	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]
c	Carlos	[555-5555, 555-6666]

```
SELECT name,  
       phones[0],  
       phones[1]  
  FROM customers_phones;
```

Query results		
name	phones[0]	phones[1]
Alice	555-1111	555-2222
Bob	555-4444	NULL
Carlos	555-5555	555-6666

Using ARRAY Columns with Hive (2)

- All elements in an ARRAY column have the same data type
- You can specify a delimiter (default is control+B)

```
CREATE TABLE customers_phones
(cust_id STRING,
 name STRING,
 phones ARRAY<STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '|'
STORED AS TEXTFILE;
```

a, Alice, 555-1111|555-2222|555-3333
b, Bob, 555-4444
c, Carlos, 555-5555|555-6666

Data File

customers_phones table		
cust_id	name	phones
a	Alice	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]
c	Carlos	[555-5555, 555-6666]

Using MAP Columns with Hive (1)

- Using the MAP type allows us to store the data in one table

customers_phones table		
cust_id	name	phones
a	Alice	{home:555-1111, work:555-2222, mobile:555-3333}
b	Bob	{mobile:555-4444}
c	Carlos	{work:555-5555, home:555-6666}

```
SELECT name,  
       phones['home'] AS home  
FROM customers_phones;
```

Query results	
name	home
Alice	555-1111
Bob	NULL
Carlos	555-6666

Using MAP Columns with Hive (2)

- MAP keys must all be one data type, and values must all be one data type
 - MAP<KEY-TYPE , VALUE-TYPE>
- You can specify the key-value separator (default is control+C)

```
CREATE TABLE customers_phones
(cust_id STRING, name STRING,
phones MAP<STRING, STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '|'
MAP KEYS TERMINATED BY ':'
STORED AS TEXTFILE;
```

a, Alice, home:555-1111|work:555-2222|mobile:555-3333
b, Bob, mobile:555-4444
c, Carlos, work:555-5555|home:555-6666

Data File

customers_phones table		
cust_id	name	phones
a	Alice	{home:555-1111, work:555-2222, mobile:555-3333}
b	Bob	{mobile:555-4444}
c	Carlos	{work:555-5555, home:555-6666}

Using STRUCT Columns with Hive

- A STRUCT stores a fixed number of names fields
 - Each field can have a different data type

customers_addr table		
cust_id	name	address
a	Alice	{street:742 Evergreen Terrace, city:Springfield, state:OR, zipcode:97477}
b	Bob	{street:1600 Pennsylvania Ave NW, city:Washington, state:DC, zipcode:20500}
c	Carlos	{street:342 Gravelpit Terrace, city:Bedrock, state:NULL, zipcode:NULL}

CLOU
Educa

```
SELECT name,  
       address.state,  
       address.zipcode  
  FROM customers_addr;
```

Query results

name	state	zipcode
Alice	OR	97477
Bob	DC	20500
Carlos	NULL	NULL

be reproduced or shared without prior written consent from Cloudera.

11-55

Complex Columns in the SELECT List in Hive Queries

- You can include complex columns in the SELECT list in Hive queries
 - Hive returns full ARRAY, MAP or STRUCT columns

customers_phones table			Query results
cust_id	name	phones	phones
a	Alice	[555-1111, 555-2222, 555-3333]	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]	[555-4444]
c	Carlos	[555-5555, 555-6666]	[555-5555, 555-6666]

Returning the Number of Items in a Collection with Hive

- The size function returns the number of items in an ARRAY or MAP
 - An example of a collection function

customers_phones table		
cust_id	name	phones
a	Alice	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]
c	Carlos	[555-5555, 555-6666]

```
SELECT  
    name,  
    size(phones) AS num  
FROM customers_phones;
```

Query results	
name	num
Alice	3
Bob	1
Carlos	2

Converting ARRAY to Records with `explode`

- The `explode` function creates a record for each element in an ARRAY

- An example of a table-generating function

customers_phones table		
cust_id	name	phones
a	Alice	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]
c	Carl os	[555-5555, 555-6666]

```
SELECT  
    explode(phones) AS phone  
FROM customers_phones;
```

Query results
phone
555-1111
555-2222
555-3333
555-4444
555-5555
555-6666

Using explode with a Lateral View

- No other columns can be included in the SELECT list with explode



```
SELECT name, explode(phones) AS phone  
FROM customers_phones;
```

- Use a lateral view to overcome this limitation

- Applies the table-generating function to each ARRAY in the base table
- Joins the resulting output with the rows of the base table

```
SELECT name, phone  
FROM customers_phones  
LATERAL VIEW  
    explode(phones) p AS phone;
```

Query results	
name	phone
Alice	555-1111
Alice	555-2222
Alice	555-3333
Bob	555-4444
Carlos	555-5555
Carlos	555-6666

Loading Data Containing Complex Types

- Impala supports querying complex types only in Parquet or ORC tables
- Impala cannot **INSERT** data containing complex types
- Workaround: Use Hive to **INSERT** data into Parquet or ORC tables
 - **CREATE TABLE** in Hive or Impala, then **INSERT** in Hive

```
INSERT INTO TABLE cust_phones_parquet SELECT * from customers_phones;
```

- Or use a **CREATE TABLE AS SELECT (CTAS)** statement in Hive

```
CREATE TABLE cust_phones_parquet AS STORED AS PARQUET SELECT * FROM customers_phones;
```

Denormalizing Tables with a One-To-Many Relationship

- Denormalizing tables allows you to query data without the need to use joins
- To denormalize tables with a one-to-many relationship, use **ARRAY<STRUCT<>>**
- One row of the new table can include all associated rows from another table

```
CREATE EXTERNAL TABLE rated_products (
    prod_id INT,
    brand STRING,
    name STRING,
    price INT,
    ratings ARRAY<STRUCT <rating:TINYINT,message:STRING>>)
STORED AS PARQUET;
```

Populating a Denormalized Table

- This step must be performed in Hive
- Use `named_struct()` to cast a row of the detail table into the proper structure
- Use `collect_list()` to collect multiple rows into an array

```
INSERT OVERWRITE TABLE rated_products
SELECT p.prod_id, p.brand, p.name, p.price,
       collect_list(named_struct('rating', r.rating, 'message', r.message))
  FROM products p LEFT OUTER JOIN ratings r
    ON (p.prod_id = r.prod_id)
 GROUP BY p.prod_id, p.brand, p.name, p.price;
```

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- Working with Skew
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- **Hands-On Exercise: Complex Data**
- Essential Points

Chapter Topics

Data Engineering with Hive

- Working with Partitions
- Hands-On Exercise: Hive Partitions
- Working with Buckets
- Working with Skew
- Instructor-Led Demo: Skewed Table
- Using Serdes to Ingest Text Data
- Hands-On Exercise: Analyzing Text with Hive
- Using Complex Types to Denormalize Data
- Hands-On Exercise: Complex Data
- Essential Points

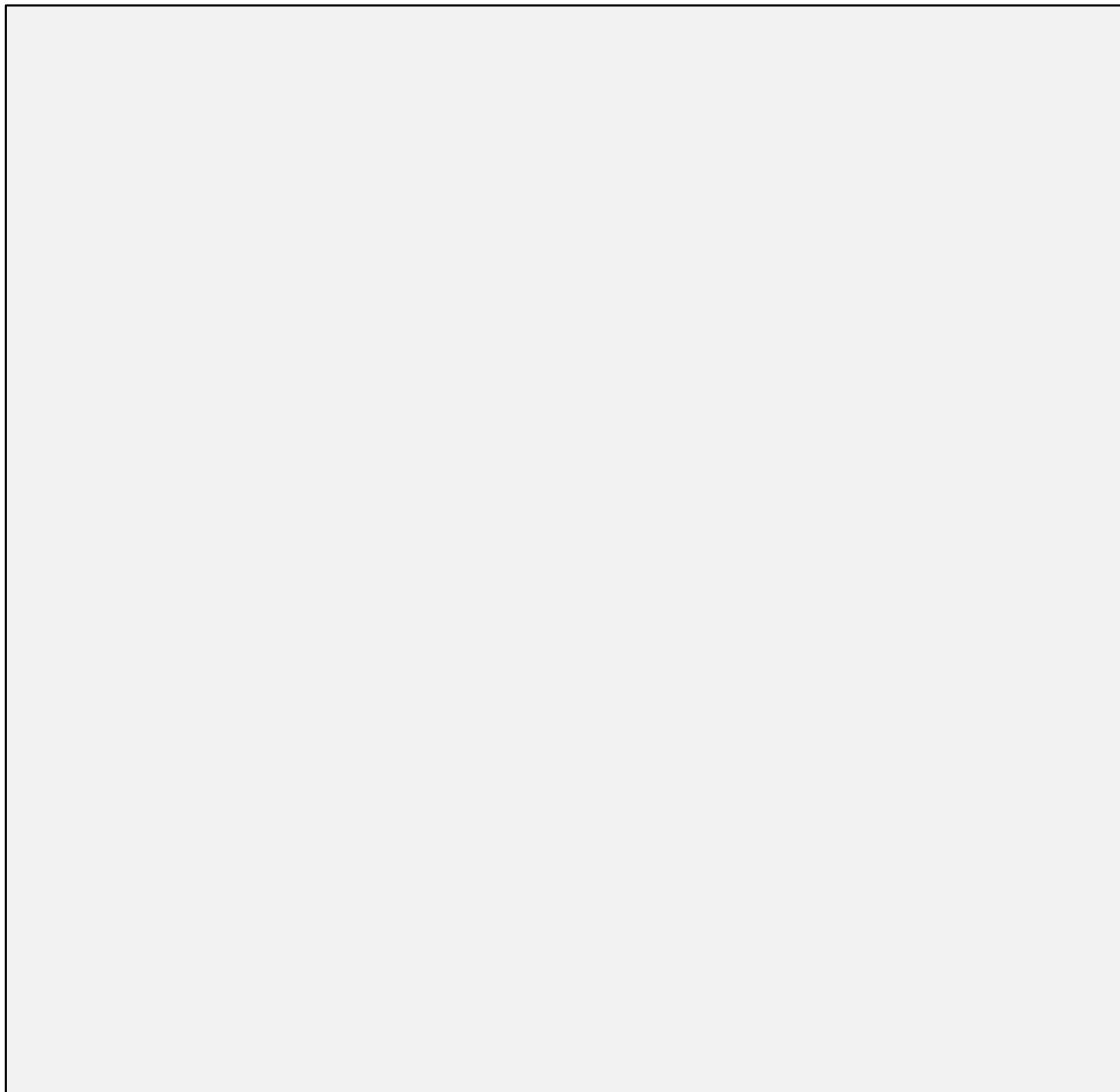
Essential Points

- Dynamic partitioning divides data automatically, based on values from one or more columns
- Use a static partition when you want to manually specify which partition the data is stored in
- Buckets provide extra structure to the data so it may be used for more efficient queries
- Hive supports “compile-time” and “run-time” skewed joins
- SerDes enables Hive to access data that is not in structured tabular format

Bucketing is similar to partitioning – in both cases, data is segregated and stored – but there are a few key differences. Partitioning is based on a column that is repeated in the dataset and involves grouping data by a particular value of the partition column. While bucketing organizes data by a range of values, mainly involving primary key or non-repeated values in a dataset.

Hive and Spark Integration

Chapter 12



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- **Hive Spark Integration**
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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> <ul style="list-style-type: none">1. builtin2. Use Hive 1.2.1, which is bundled with the Spark assembly jar when -Phive is enabled.3. maven4. Use Hive jars of specified version downloaded from Maven repositories. This configuration is not generally recommended for production deployments.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.
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

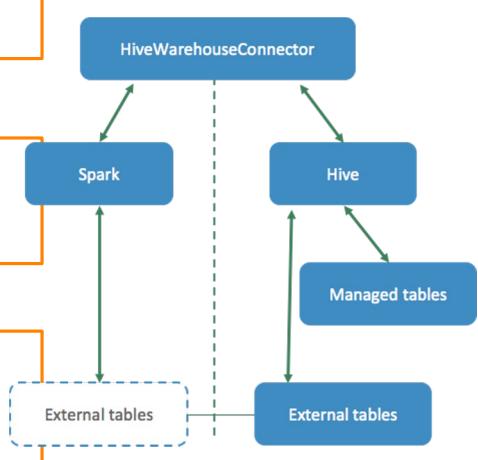
```
import com.hortonworks.hwc.HiveWarehouseSession  
import com.hortonworks.hwc.HiveWarehouseSession._  
val hive = HiveWarehouseSession.session(spark).build()
```

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

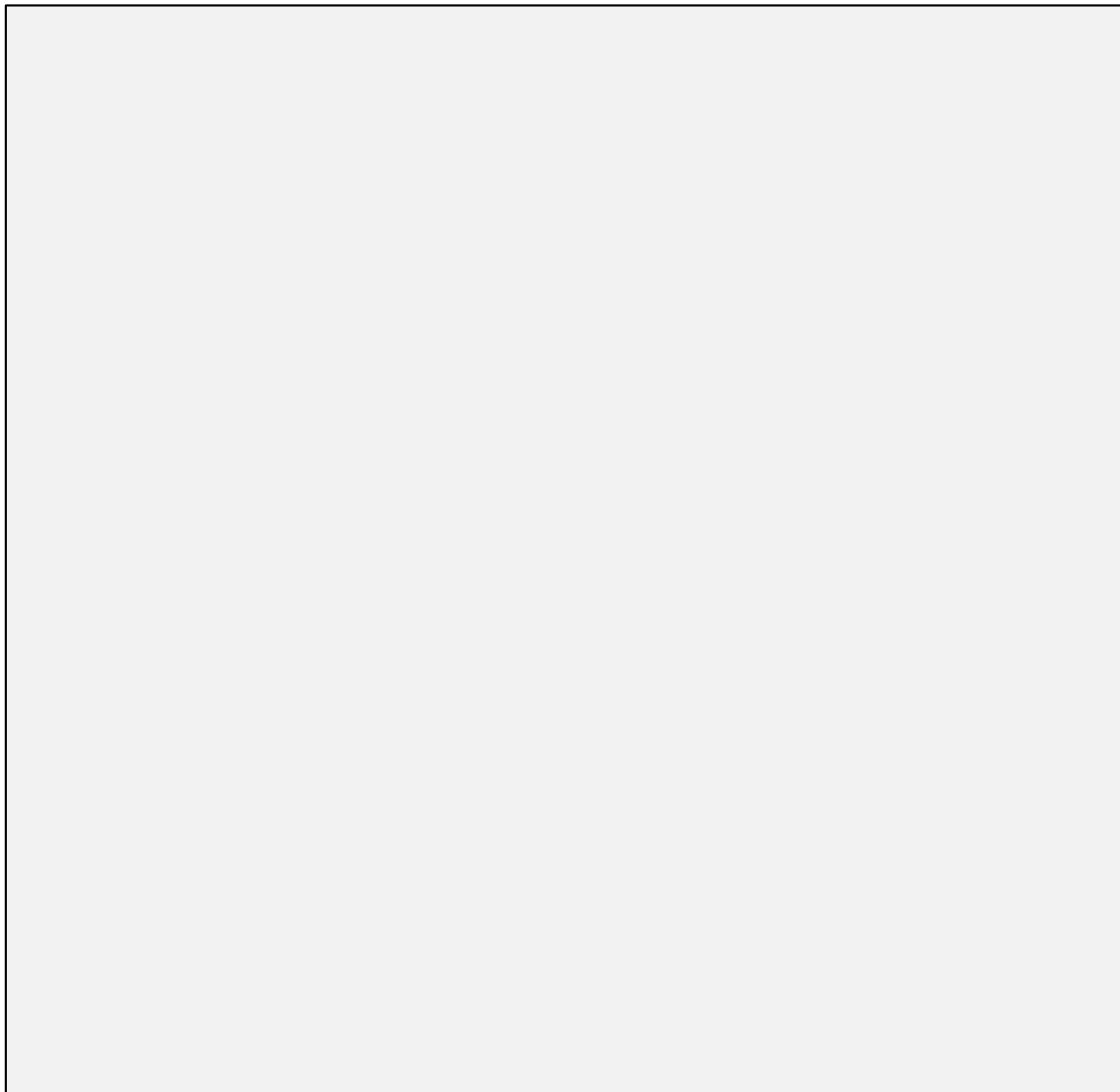
Chapter Topics

Hive and Spark Integration

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

Distributed Processing Challenges

Chapter 13



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- **Distributed Processing Challenges**
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- Conclusion
- Appendix: Working with Datasets in Scala

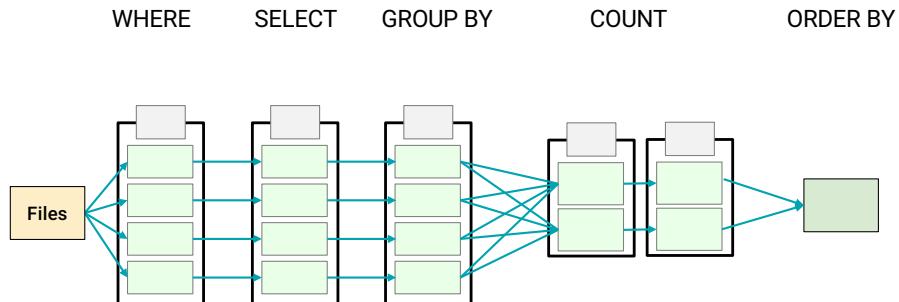
Chapter Topics

Distributed Processing Challenges

- Shuffle
- Skew
- Order

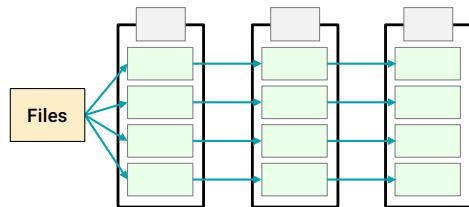
SELECT COUNT(*) FROM WHERE GROUP BY ORDER BY

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



SELECT WHERE

WHERE SELECT

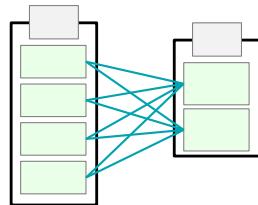


This is the fast part, distributed processing is efficient

- 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**

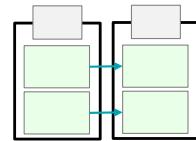
Chapter Topics

Distributed Processing Challenges

- Shuffle
- Skew
- Order

SELECT COUNT(*) FROM WHERE GROUP BY ORDER BY

COUNT

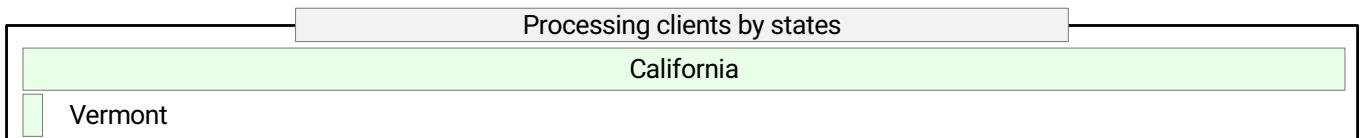


This can also be a problem:

- 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-00001-e0fa63811-....csv
-rw-r--r-- 3 training training 2333 ... devices/part-00002-e0fa63811-....csv
```

```
$ hdfs dfs -ls
devices

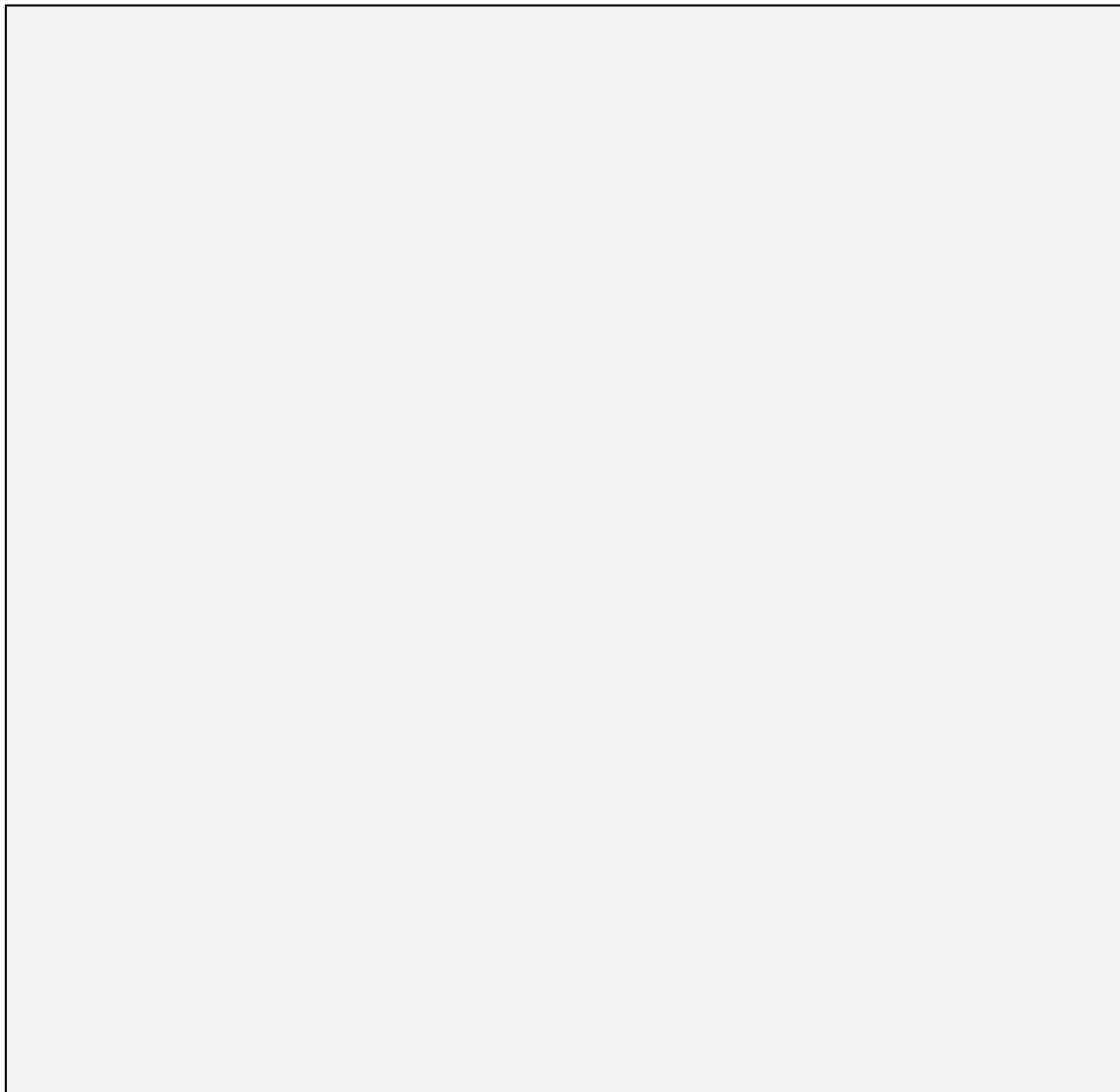
Found 4 items
-rw-r--r-- 3 training training 2149 ... devices/_SUCCESS
-rw-r--r-- 3 training training 2119 ... devices/part-00000-
e0fa6381-....csv
-rw-r--r-- 3 training training 2202 ... devices/part-00001-
e0fa6381-....csv
-rw-r--r-- 3 training training 2333 ... devices/part-00002-
e0fa6381-....csv
```

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 14



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- **Spark Distributed Processing**
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

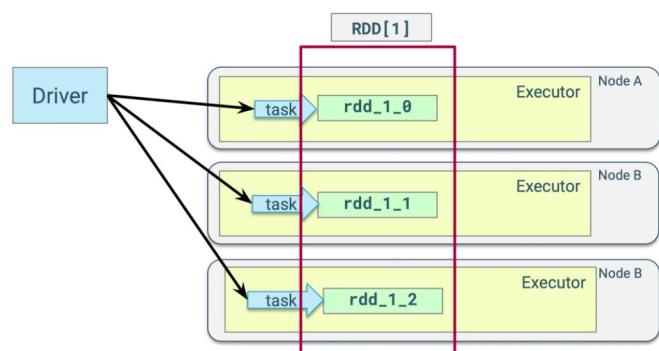
Chapter Topics

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

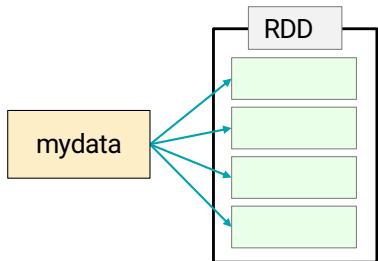
> `myRDD.getNumPartitions()`

Language: Python

Example: Average Word Length by Letter (1)

Language: Python

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



START BUILD

Example code is in examples/spark/AverageWordLengthByLetter.pyspark and .scalaspark.

The example code does something similar to word count, but instead of counting words, it calculates the average length of words by their first letter.

That is, what is the average length of words starting with a, starting with b, and so on.

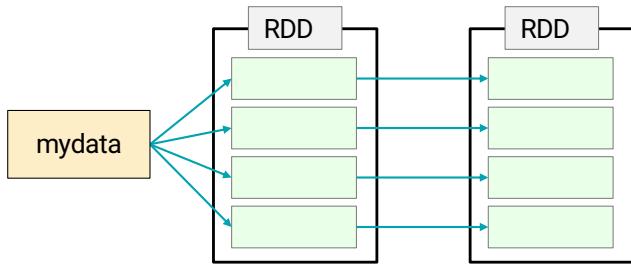
What is interesting here is not the task, though, it is how partitioning works with operations.

We start by reading a file, which as we previously learned creates a partition corresponding to the HDFS blocks (or more, if so specified by the user). So in this case, mydata consists of four HDFS blocks which in turn are loaded into four partitions.

Example: Average Word Length by Letter (2)

Language: Python

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

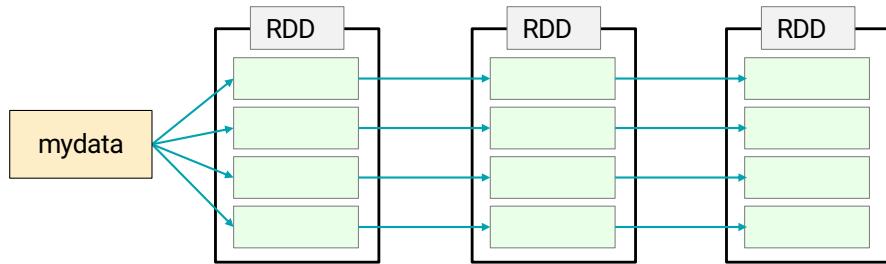


map and flatMap (and related map-like operations such as filter and keyBy) operate locally on each partition. The nature of map operations is that they operate on rows of data independently of the rest of the data set, so they are parallelized by executing the operation on the node where the data is located. For every partition operated on, there is a single output partition. (Usually, if there's more data in the output of the operation than the input, it is possible the result will be too large for a single partition in which case it will map to more partitions. It is still considered a "narrow" dependency, though, in that there's no shuffle required, but data will need to be copied across the network.)

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)))
```

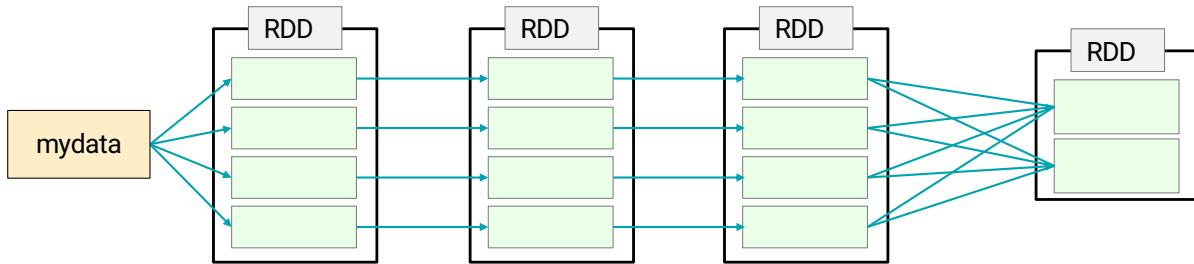


Side note to instructors: Astute students may look at the API docs for map and related functions and note that there is a parameter called `preservesPartitioning` which defaults to `false` in most cases. The purpose of this is not related to partitioning as discussed in this slide (and this chapter). Rather, it indicates whether a particular operation changes keys. If not, then subsequent operations on child RDDs would not have to shuffle data between stages. When possible (such as when data has already been shuffled by a specific partitioner according to a particular set of keys) Spark will avoid shuffling. It doesn't change the relationship between partitions and stages as discussed in this chapter. How partitioning is actually done, and how Spark uses pre-partitioning to avoid unnecessary shuffles, is outside the scope of this class.

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)
```



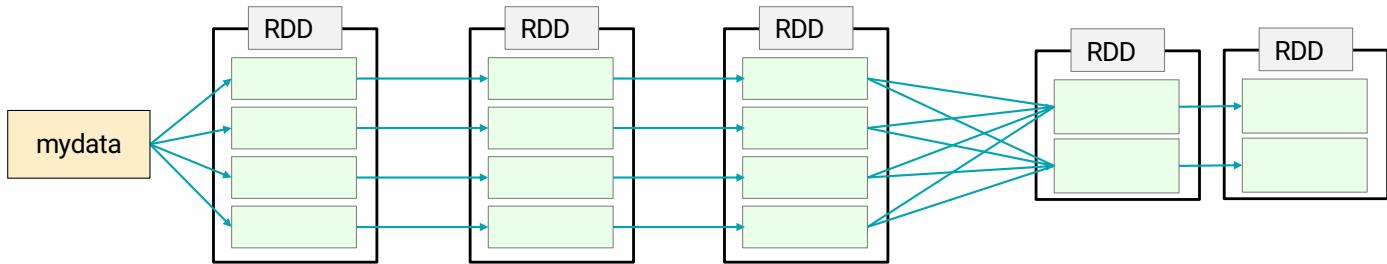
groupByKey is different than the map and flatMap operations. It is a reduce type operation; that is, each task needs access to all the data, not just data in a single partition. The data will need to be copied to each node where a reduce or groupBy operation is occurring; this is called shuffling, and will be discussed more in a few slides.

The resulting number of partitions doesn't correspond to the number of input partitions. In this example the output happens to be two partitions but that's controlExercise, as will be discussed in a few slides. This example supposed that the default level of parallelism had been set to 2. The point here isn't how many partitions there are, just that certain operations require access to all partitions and result in new partitions, and there's a task associated with each partition.

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])))
```



END BUILD

The final step is another map operation. This operation works on the partitions which were output by the groupByKey (or other reduce-related operation). Partitioning is again preserved.

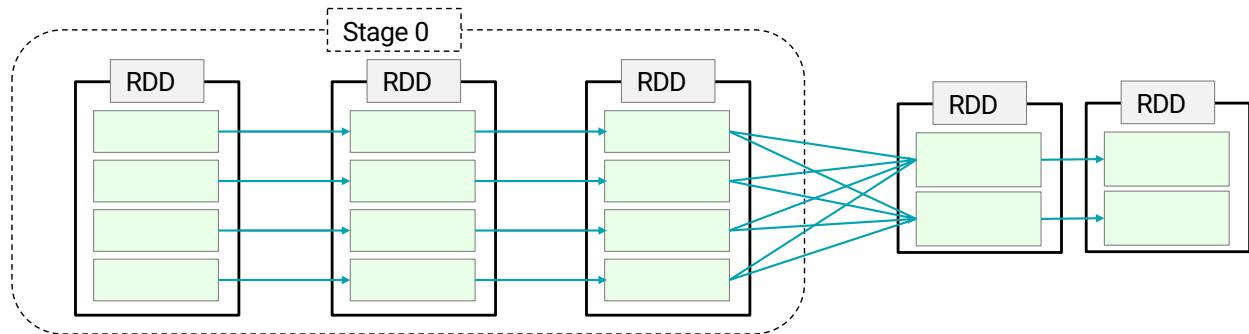
Stages and Tasks

- 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")
```



START BUILD

Stages consist of all the tasks that can be performed on the same node, that is, that can be done without shuffling data between nodes.

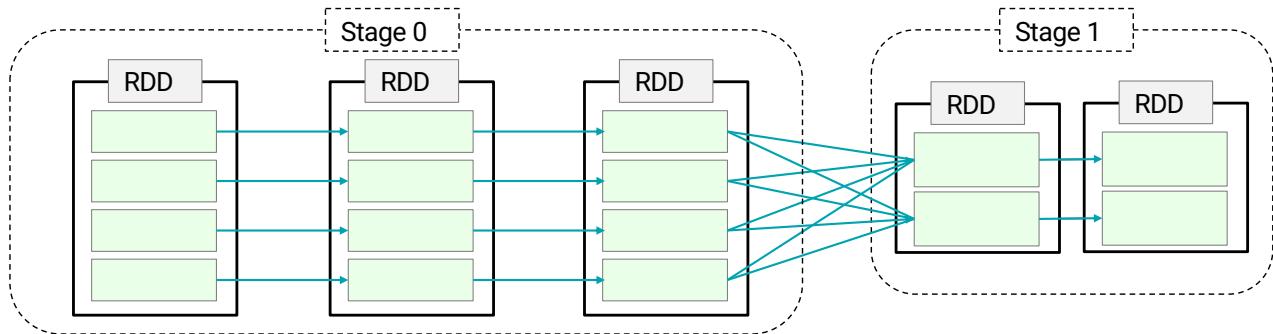
This diagram shows how Spark would calculate the stages for the Average Word Length by Letter program we just saw.

Note that stages are numbered starting with 0 rather than 1.
Stage 1 is dependent on (is a “child” of) Stage 0.

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")
```



START BUILD

Stages consist of all the tasks that can be performed on the same node, that is, that can be done without shuffling data between nodes.

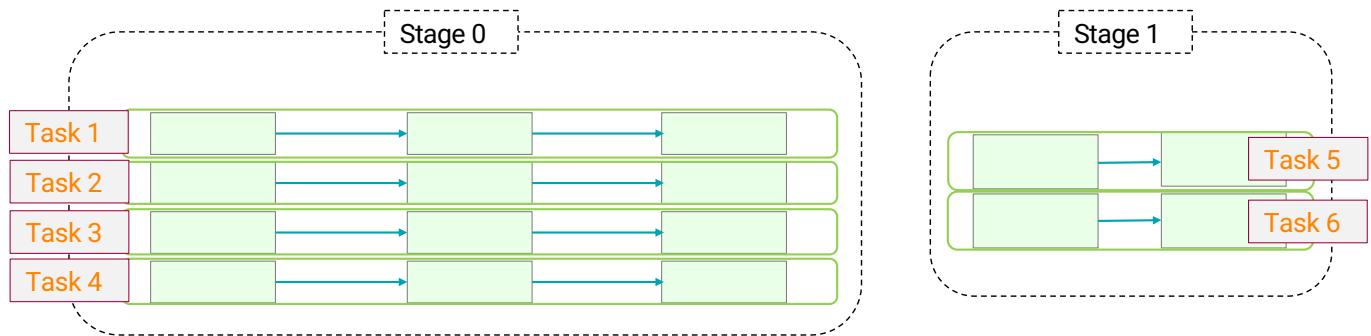
This diagram shows how Spark would calculate the stages for the Average Word Length by Letter program we just saw.

Note that stages are numbered starting with 0 rather than 1.
Stage 1 is dependent on (is a “child” of) Stage 0.

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")
```

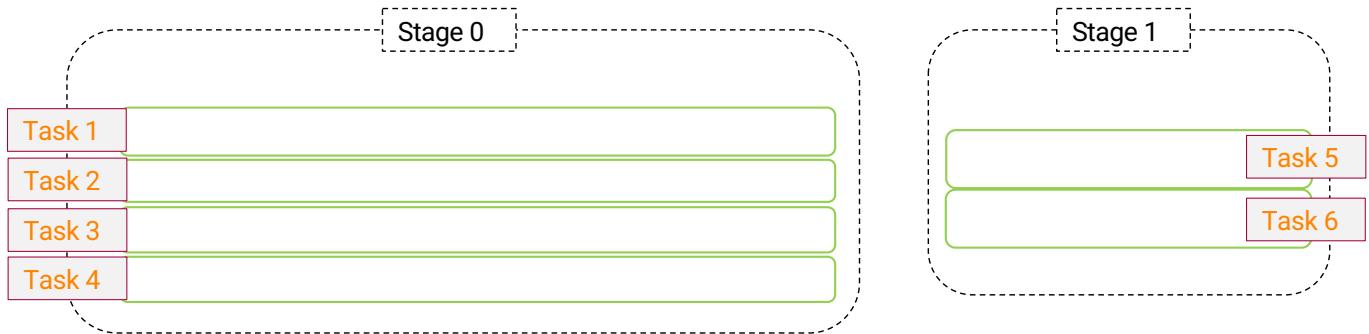


Tasks which preserve partitioning (like map) can be pipelined together, essentially collapsing multiple tasks into one.

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")
```

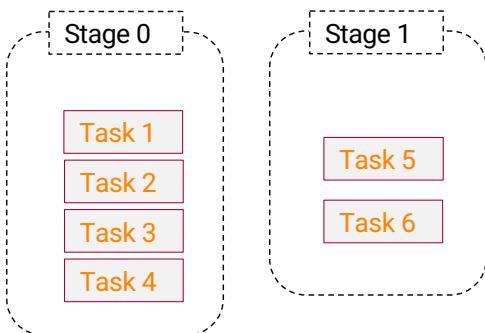


When operations are pipelined together, you can think of them as just a single task.

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")
```

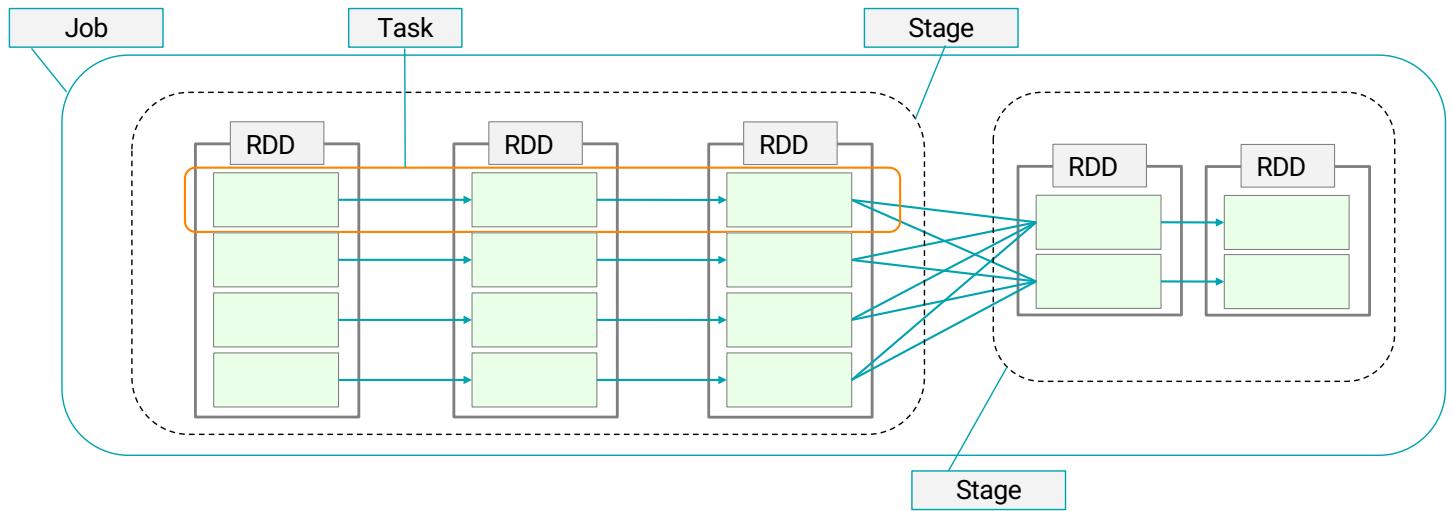


END BUILD

The result of this calculation of stages will result in six tasks.

Note that this is conceptual; some operations will actually result in multiple tasks which you can see if you look at the Stages UI. But this explanation should be sufficient to get the idea across.

Summary of Spark Terminology



In Spark, a *job* is defined as the set of tasks triggered by an action (such as `saveAsTextFile` or `collect`).

In our example, that action was `count()`.

A Spark application can consist of any number of jobs. (Especially considering that the Spark shell is an application, and obviously can run numerous jobs, as you have been doing all along in the exercises.)

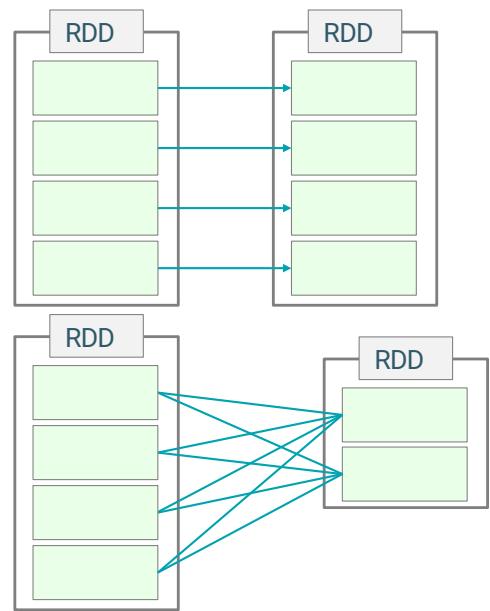
For Hadoop MR users: The Spark definition of *job* is different than in MR. In MR, a job is the same as an instance of an application, and always consists of a single map phase and a single (or no) reduce phase. Spark jobs can have any number of reduce and map phases.

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

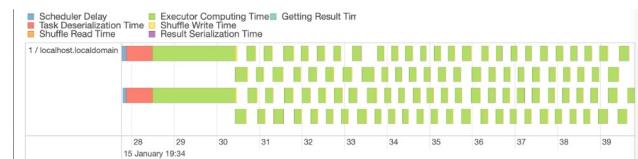
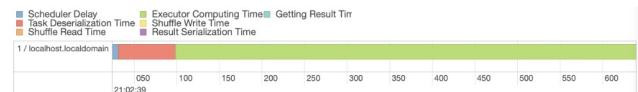
How Execution Plans are Created

- 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)

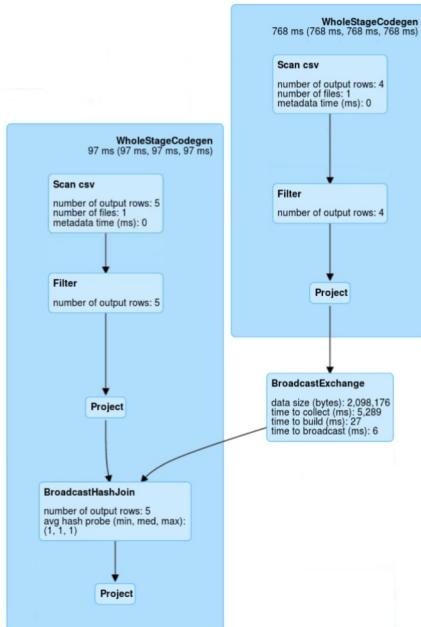
SQL

Completed Queries: 3

Completed Queries (3)

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

Example: Catalyst Execution Plan (5)



Tungsten Overview

- Improve Spark execution by optimizing CPU / memory usage
 - Understands and optimizes for hardware architectures
 - Tunes optimizations for Spark's characteristics

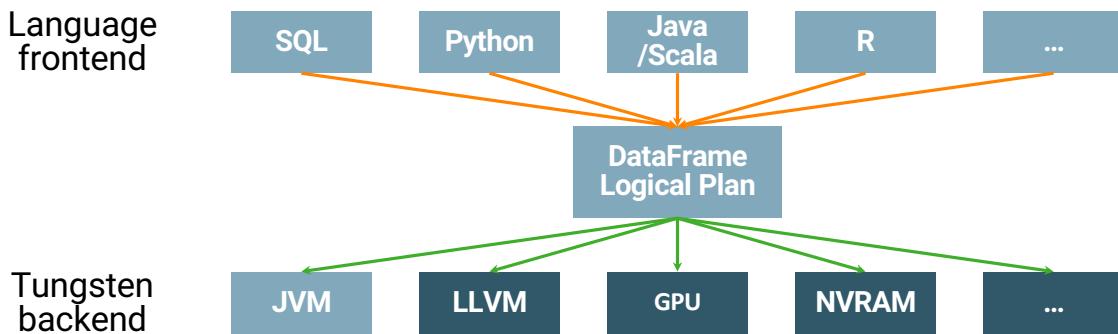


Image from:

<https://www.slideshare.net/databricks/2015-0616-spark-summit>
<https://community.hortonworks.com/articles/72502/what-is-tungsten-for-apache-spark.html>

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

Java serialization performance is a well-known issue in using Java.
Kryo is a "fast and efficient object graph serialization framework for Java."
It improves on native Java serialization performance.
It was one of the choices used with Spark before Tungsten was developed.
See: <https://github.com/EssotericSoftware/kryo>

Java objects have a very large memory overhead.
e.g. a string with 4 bytes of data can take 48 bytes and more to store in Java.
See: <https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>

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.

1 - False
2 - False: an Application is a sequence of jobs
3 - Catalyst and Tungsten
4 - False, Lambdas impede Catalyst
5 - False
6 - False
7 - False, they are 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

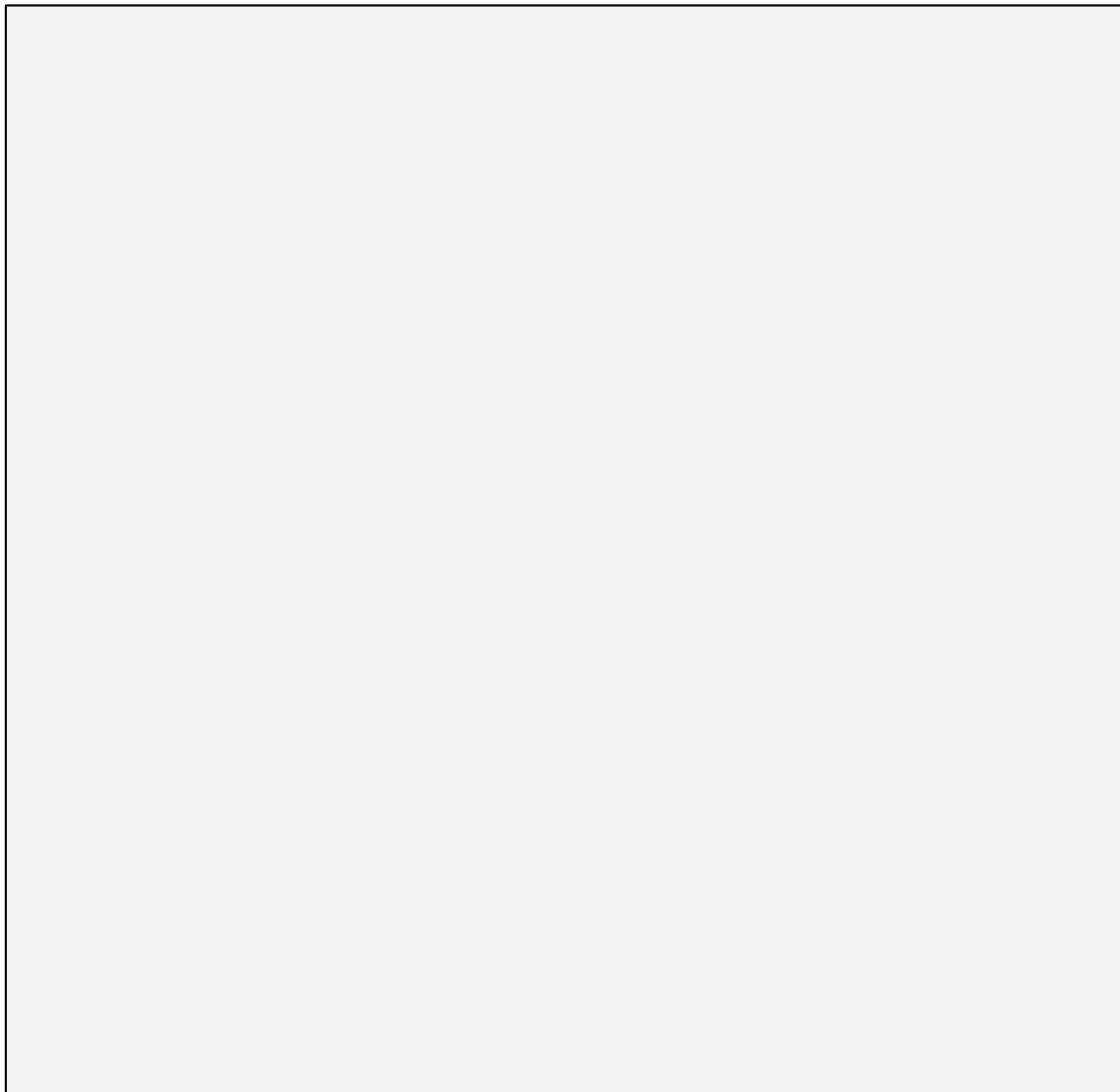
Chapter Topics

Spark Distributed Processing

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

Spark Distributed Persistence

Chapter 15



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence**
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- Conclusion
- Appendix: Working with Datasets in Scala

Distributed Data Persistence

After completing this chapter, you will be able to

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

Chapter Topics

Spark Distributed Persistence

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

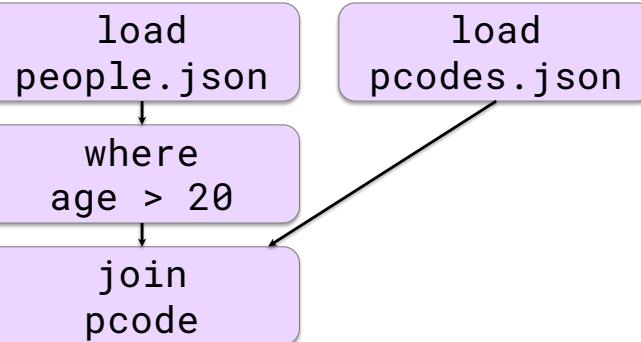
Persistence

- 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()
```

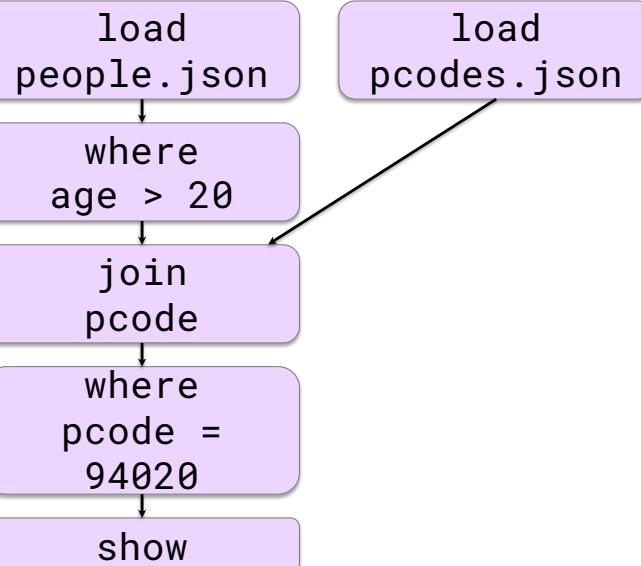
Language: Python



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()
```

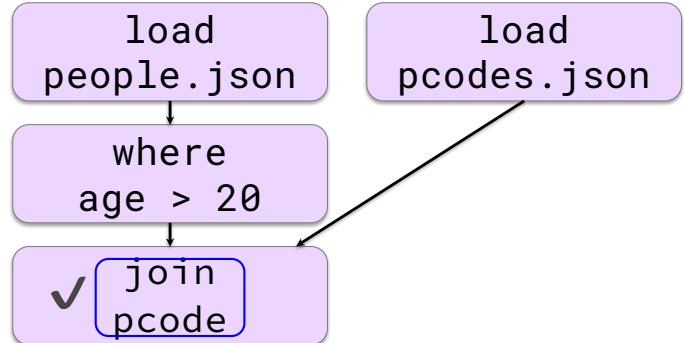
Language: Python



Example: DataFrame Persistence (3)

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

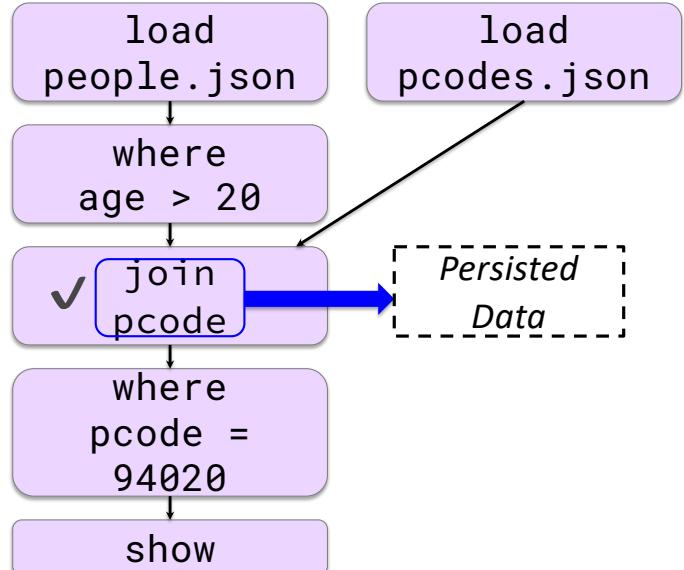
Language: *Python*



Example: DataFrame Persistence (4)

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

Language: Python



Example: DataFrame Persistence (5)

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

Language: Python

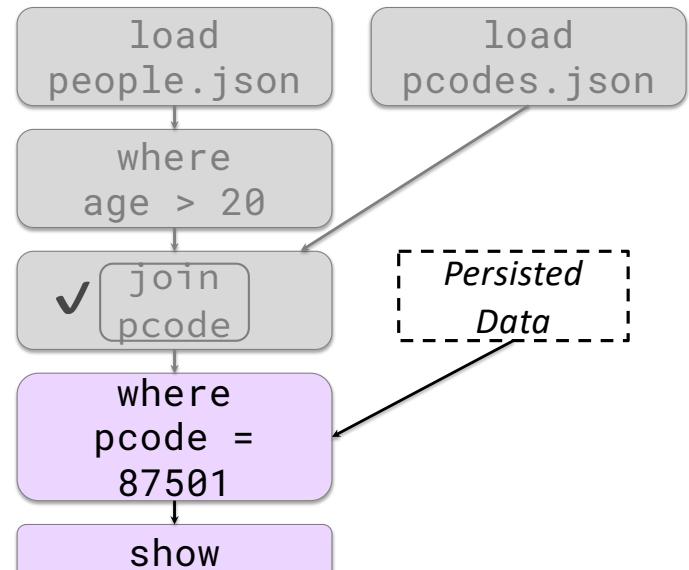


Table and View Persistence

- 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

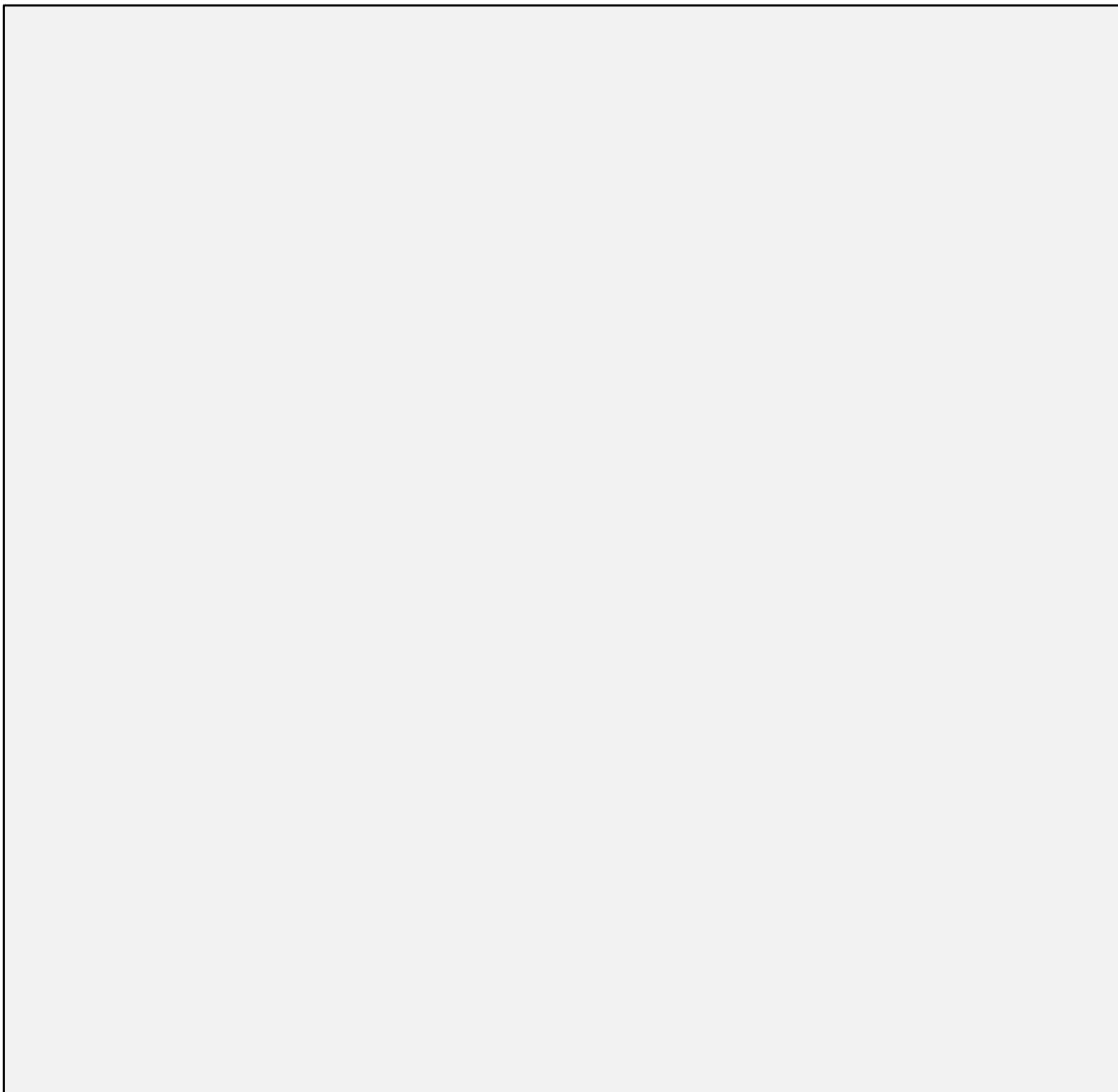
Chapter Topics

Spark Distributed Persistence

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

Working with the Data Engineering Service

Chapter 16



Course Chapters

- Introduction
- Why Data Engineering
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- **Working with the Data Engineering Service**
- Working with Airflow
- Workload Manager Introduction
- Conclusion
- Appendix: Working with Datasets in Scala

Chapter Topics

Title

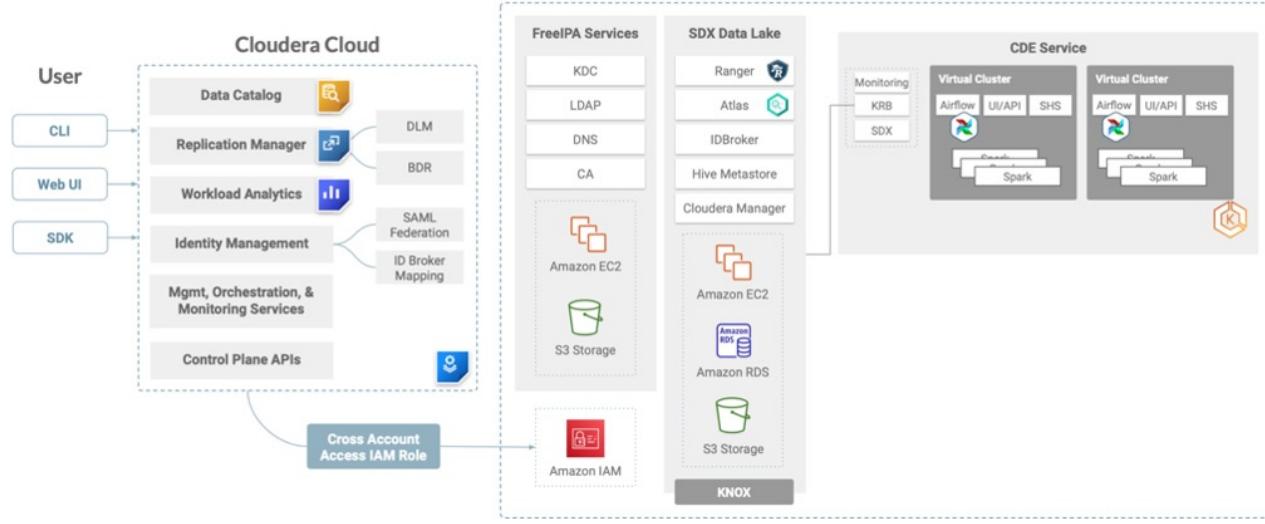
- **Working with the Data Engineering Service**

- Exercises:

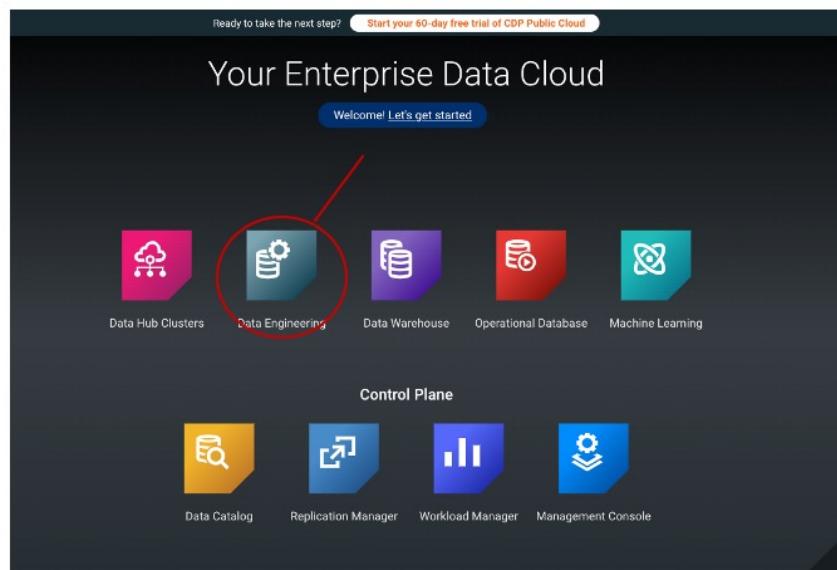
- Data Engineering Data Service Walkthrough
- Create and Trigger Ad Hoc Spark Jobs
- Add Schedule to Ad Hoc Spark Jobs
- Spark Job Data Lineage Using Atlas

CDP-AWS High Level Architecture

CDP Environment



New Data Engineering Experience

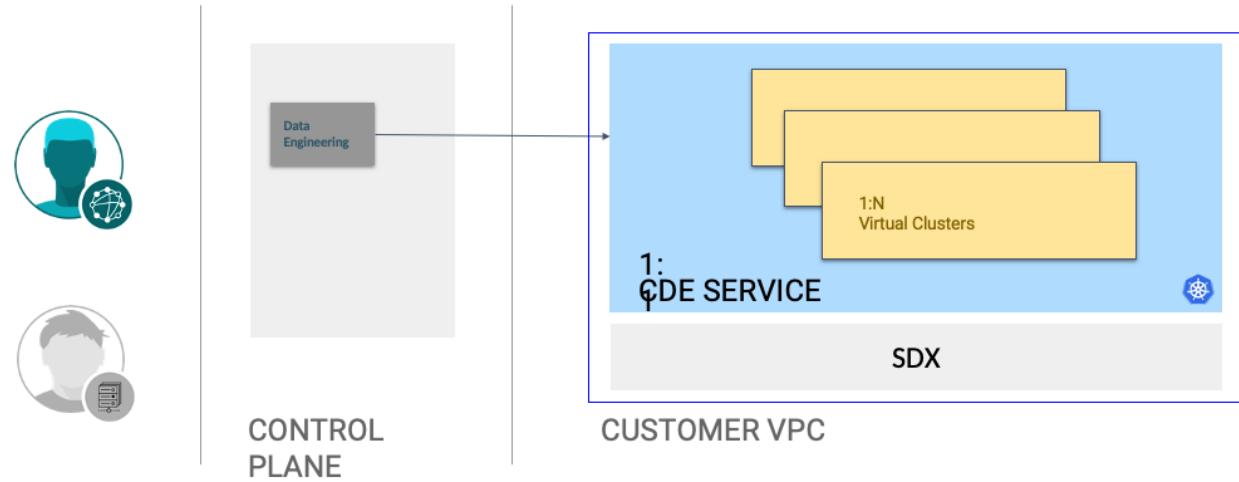


CLOUDERA
Educational Services

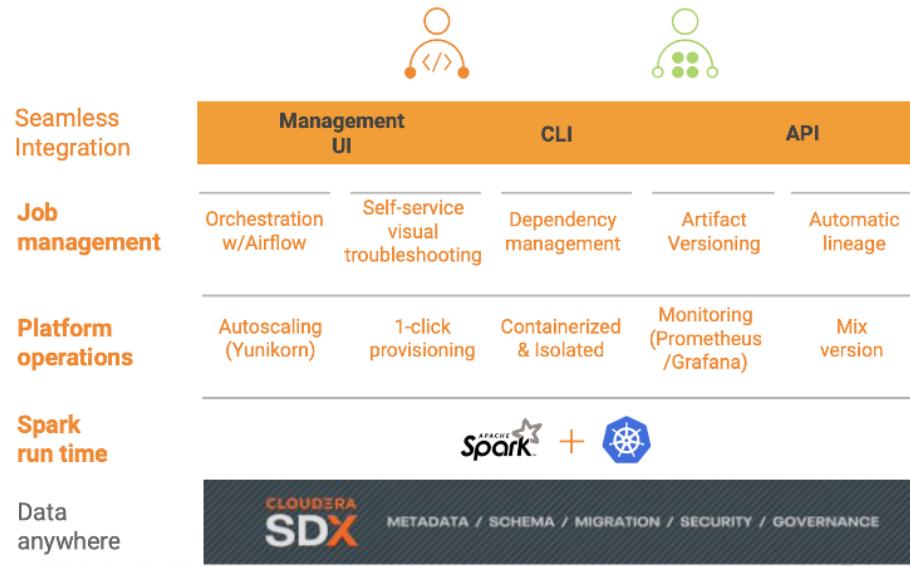
Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

16-4

CDE Service Components and Relationships



Cloudera Data Engineering



Autoscaling

The screenshot shows the Cloudera Data Engineering interface. On the left, there's a sidebar with a dark background and the Cloudera logo. The main area has a light gray background.

Overview

Environments [1]

- dex-mow-int-env** (aws)
Enabled
NODES: 3 CPU: 24 MEMORY: 92 GB

Virtual Clusters / dex-mow-int-env [2]

- HeavyETL-workload** (dex-mow-int-env)
Running
PODS: 9 CPU: 4.5 MEMORY: 9 GB JOBS: 0
- SalesOps-workload** (dex-mow-int-env)
Running
PODS: 9 CPU: 4.5 MEMORY: 9 GB JOBS: 0

Isolated virtual clusters (Annotation: A red circle with a '1' is above the notification bell icon.)

Autoscaling (Annotation: An arrow points from the word 'Autoscaling' to a step-function line graph showing resource usage over time.)

CLOUDERA
Educational Services

Copyright © 2010-2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

16-7

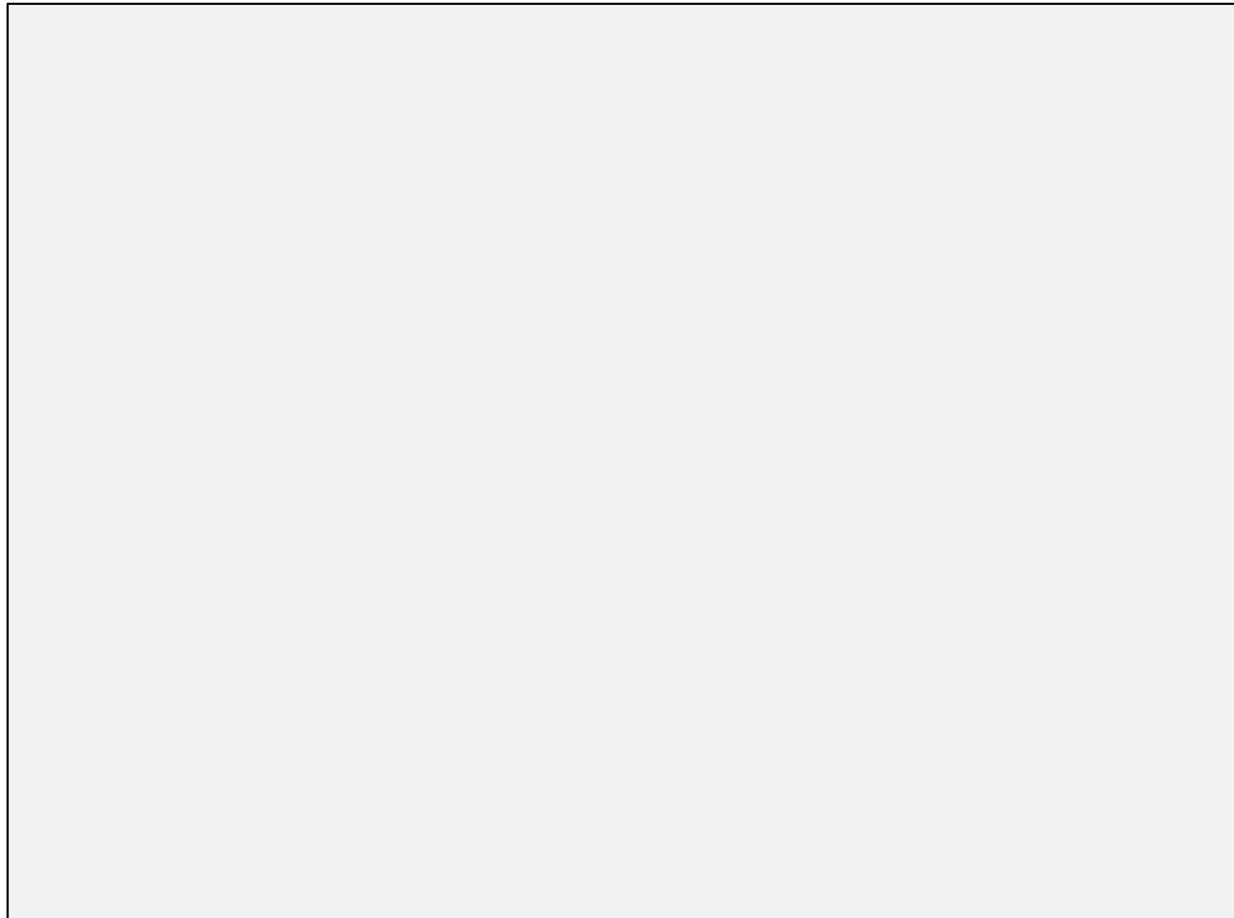
Provision Isolated Workloads with Quotas

The screenshot shows the Cloudera Data Engineering interface for creating a new cluster. On the left, there's a sidebar with the Cloudera logo and 'Data Engineering' text. The main area has a title 'Overview / Create a Cluster'. It includes fields for 'Cluster Name' (set to 'SalesOps-VirtualCluster'), 'CDE Service' (set to 'dex-dev-us-west-2'), and two slider controls for 'Auto-Scale Range': 'CPU' (set to 35) and 'Memory (GB)' (set to 100). To the right is a 'Summary' panel with details: Cluster Name 'SalesOps-VirtualCluster', CDE Service 'cluster-rk6pb89', Auto-Scale Range (CPU 35 Max, Memory 100 GB Max), and 'Airflow enabled'. A red notification bell icon in the top right corner has a '3' badge.

CLOUDERA
Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

16-8



Monitor Capacity & Resource Usage Thru Grafana

The screenshot shows the Cloudera Data Engineering interface with a Grafana dashboard titled "Overview / slevente-test-2". The dashboard includes the following components:

- Metrics Summary:** Shows "Running" status, version 1, created by Szemion Levente, node count 18, CPU 15, memory 60Gi, and applications 0.
- Environment:** DATA LAKE, dev-dev-us-west-2-dl.
- Charts:**
 - CPU usage seconds: A line chart showing CPU usage over time, with a significant spike around 08:00.
 - Memory usage: A line chart showing memory usage over time, with a sharp increase starting around 08:00.
 - Network usage: A line chart showing network usage over time, with a peak around 08:00.
 - Filesystem usage: A line chart showing filesystem usage over time, with a peak around 08:00.
- Kubernetes Metrics:** A detailed table showing Kubernetes metrics for various pods and services, including CPU usage, memory usage, and network bandwidth.
- Summary Metrics:** Two large circular summary metrics: one green (value 13) and one red (value 43).
- Status Indicators:** A donut chart showing system status (red, yellow, green), a gauge for "Container memory" (value 10), and a gauge for "Pods running" (value 3).

At the bottom left, there are links for "Get Started", "Help", and "Shaun Almadian". At the bottom right, it says "Copyright © 2010-2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera." and "16-9".

Easy Job Deployment

CLOUDERA Data Engineering

Jobs / Create Job

Job Details

Name * ETLjob-2

User * eisae_sahmadian

Upload File * Choose file

Main Class * org.cloudera.cde.app.Application

Arguments (Optional) Argument Add Argument

Configurations (Optional) spark.yarn.access.hadoopFileSystems s3a://dex Add Configuration Advanced Configurations >

Advanced Configurations < Upload additional files, customize no. of executors, driver and executor cores and memory

Upload Jars (dependencies) Select jar file(s)

Upload dependencies Select file(s) to upload

Executors: 1 to 20 (20)

Driver Cores: 1 to 4 (4)

Executor Cores: 1 to 8 (8)

Driver Memory (GB): 1 to 32 (32)

Executor Memory (GB): 1 to 32 (32)

Schedule * Run Now Schedule

Every year on every day of every month at 6,18 : 15,30

At 15 and 30 minutes past the hour, at 06:00 AM, 08:00 AM, and 08:00 PM

Use a Cron Expression * * * * *

Start Date Tuesday, May 5, 2020 at 4:32:21 PM

UIC Time: Tuesday, May 5, 2020 at 11:32 PM UTC

End Date Wednesday, May 6, 2020 at 4:32:21 PM

Scheduling Configurations

Enable Catchup Kick off Job Runs for intervals that has not been run along with the current interval

Depends on Previous If the job runs are dependant, then the catchup will occur serially

Flexible Scheduler

CLOUDERA Educational Services

Copyright © 2010-2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

16-10

Flexible Orchestration Backed By Airflow

Embedded Apache Airflow

The screenshot shows the Cloudera Data Engineering interface. On the left, there's a sidebar with 'Job Runs', 'Jobs', 'Resources', and 'Schedules'. The main area is titled 'Schedule' and has tabs for 'DAGs', 'Data Profiling', 'Browse', and 'Admin'. A search bar at the top right says 'HeavyETL-Workload'. Below the tabs is a table titled 'DAGs' with columns: #, DAG, Schedule, Owner, and Recent. It lists four DAGs: 'analytics-pyspark-job1', 'demo-etl-job', 'demo-job-1', and 'ingestion-ETL-job-1'. The 'ingestion-ETL-job-1' row is highlighted with a green circle. An arrow points from the text 'Auto-generated Pipelines' to this row. To the right of the table is a code editor window titled 'DAG: ingestion-ETL-job-1'. The title bar says 'On | DAG: ingestion-ETL-job-1'. The code editor shows Python code for defining the DAG:

```
from dateutil import parser
from datetime import timezone
from airflow import DAG
from cde_job_run_operator.operator import CDEJobRunOperator

default_args = {
    'owner': 'Airflow',
    'depends_on_past': False,
    'wait_for_downstream': False,
    'start_date': parser.isoparse('2020-08-04T15:12:47.262Z').replace(tzinfo=timezone.utc),
    'end_date': parser.isoparse('2020-08-05T15:12:47.262Z').replace(tzinfo=timezone.utc),
    'connection_id': 'cde_runtime_api',
    'job_name': 'ingestion-ETL-job-1',
    'user': 'sahmadian',
}

dag = DAG(
    'ingestion-ETL-job-1',
    schedule_interval='0,15,30,45 * * * *',
    default_args=default_args,
    catchup=False
)

task = CDEJobRunOperator(
    task_id='ingestion-ETL-job-1-task',
    connection_id='cde_runtime_api',
    job_name='ingestion-ETL-job-1',
    user='sahmadian',
    dag=dag
)
```

An arrow points from the text 'Pipelines as Python Code' to the code editor.

CLOUDERA
Educational Services

Copyright © 2010-2022 Cloudera. All rights reserved. No part of this document may be reproduced in whole or in part without the express written permission of Cloudera.

Deploy Airflow Jobs

The screenshot shows the Cloudera Data Engineering interface. On the left, there's a sidebar with 'Job Runs', 'Jobs' (selected), and 'Resources'. The main area is titled 'Jobs / Create Job'.

Job Details:

- Job Type: Spark Airflow (highlighted with a red arrow)
- Name: Job Name (empty)
- DAG File: File Resource
- Upload File: Choose file (Python file selected)

Buttons: Cancel, Create and Run

Jobs List:

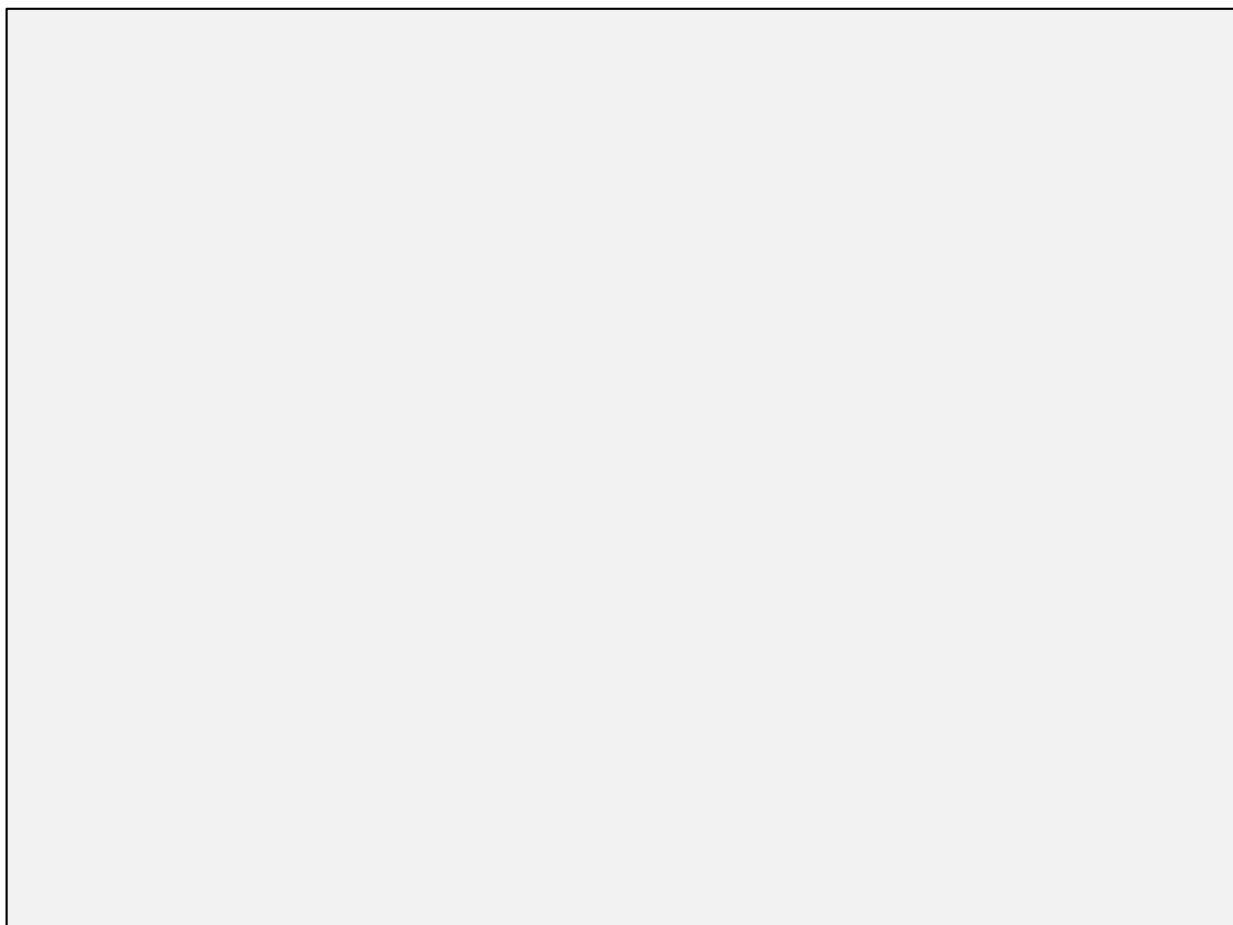
Status	Job	Type	Schedule
Idle	vivek-airflow-job-2	Airflow	*/20 ****
Idle	ps-pi	Spark	Ad-Hoc
Idle	tpcds	Spark	Ad-Hoc
Idle	ps-skew	Spark	Ad-Hoc

Red arrows point from the 'Airflow' radio button in the creation form to the 'vivek-airflow-job-2' row in the list, indicating the selection of the job type and its corresponding entry in the list.

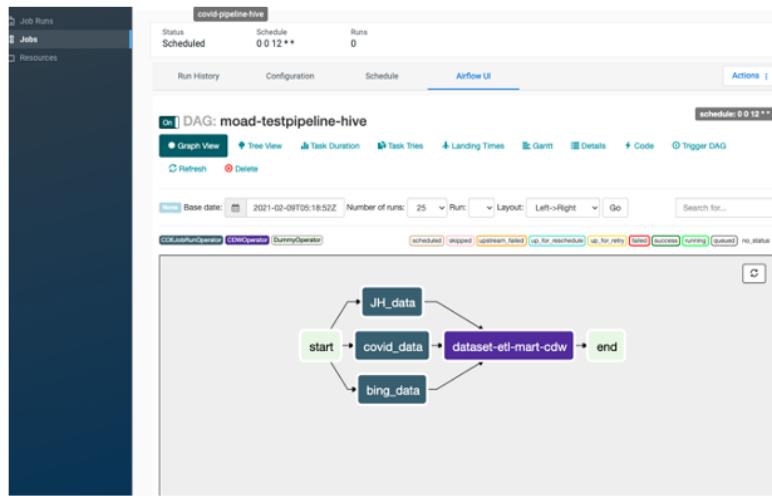
CLOUDERA
Educational Services

Copyright © 2010-2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

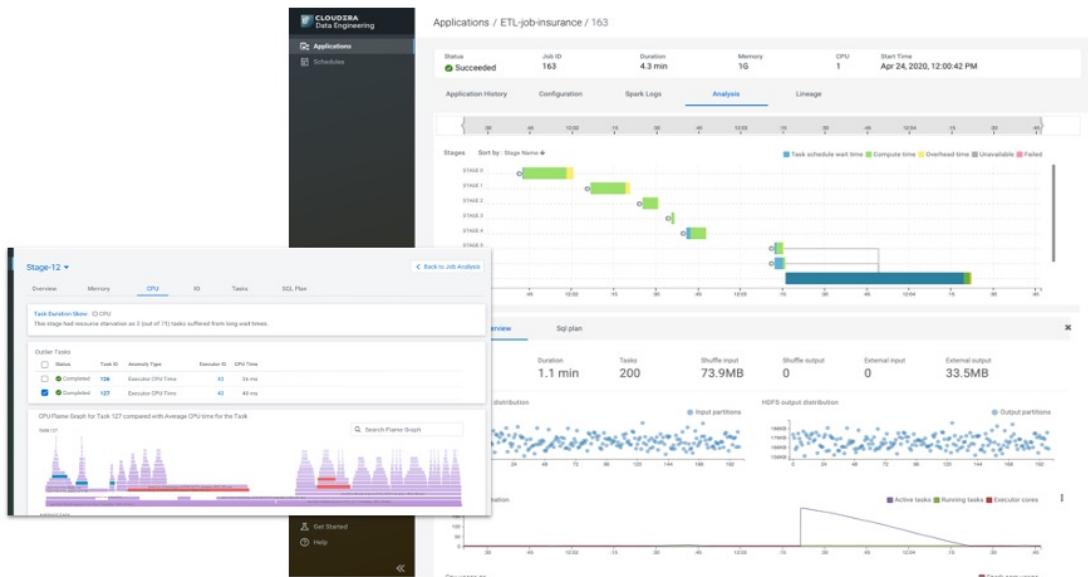
16-12



Build Spark And Hive Pipelines With Dependencies



Visual Troubleshooting & Performance Tuning

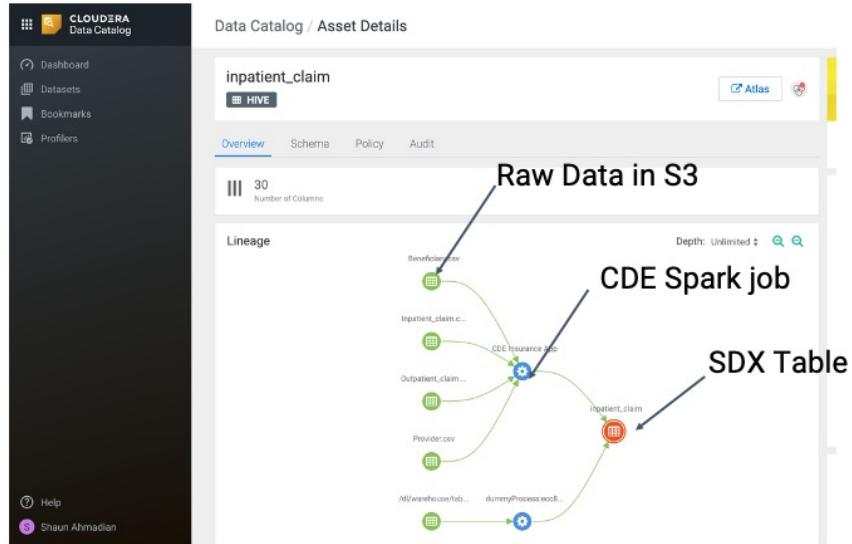


CLOUDERA
Educational Services

Copyright © 2010-2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

16-14

Automatic Capture Of Data Lineage in SDX



Rich API

The screenshot shows a Swagger UI interface for a RESTful API. At the top, there's a navigation bar with the Swagger logo, a dropdown for 'doc.json', and a 'Explore' button. Below the header, the API documentation is organized into sections:

- applications**:
 - `GET /applications` List all applications
 - `POST /applications` Create an application
 - `GET /applications/{name}` Describe application
 - `DELETE /applications/{name}` Delete application
 - `PATCH /applications/{name}` Update application
- applications schedule**:
 - `POST /applications/{name}/schedule/clear` Clear application schedule
 - `POST /applications/{name}/schedule/mark-success` Mark application schedule as successful
 - `POST /applications/{name}/schedule/pause` Pause application schedule
 - `POST /applications/{name}/schedule/unpause` Unpause application schedule
- info**:
 - `GET /info` Information about the instance
- jobs**:
 - `GET /jobs` List jobs
 - `POST /jobs` Run a job
 - `GET /jobs/{id}` Describe job

Knowledge Check

- 1. CDE can run Spark on YARN, Spark on Kubernetes and Spark on MESOS.**
- 2. CDE is only available in CDP Public Cloud.**
- 3. CDE allows you to chain multiple Spark jobs using Oozie or Airflow.**
- 4. CDE allows you to run Flink jobs.**
- 5. CDE updates the data lineage in Atlas.**
- 6. CDE comes with a basic API.**

1. Kubernetes
2. CDP Public Cloud and CDP Private Cloud
3. Airflow
4. False
5. True
6. No, it comes with a rich API

Chapter Topics

Title

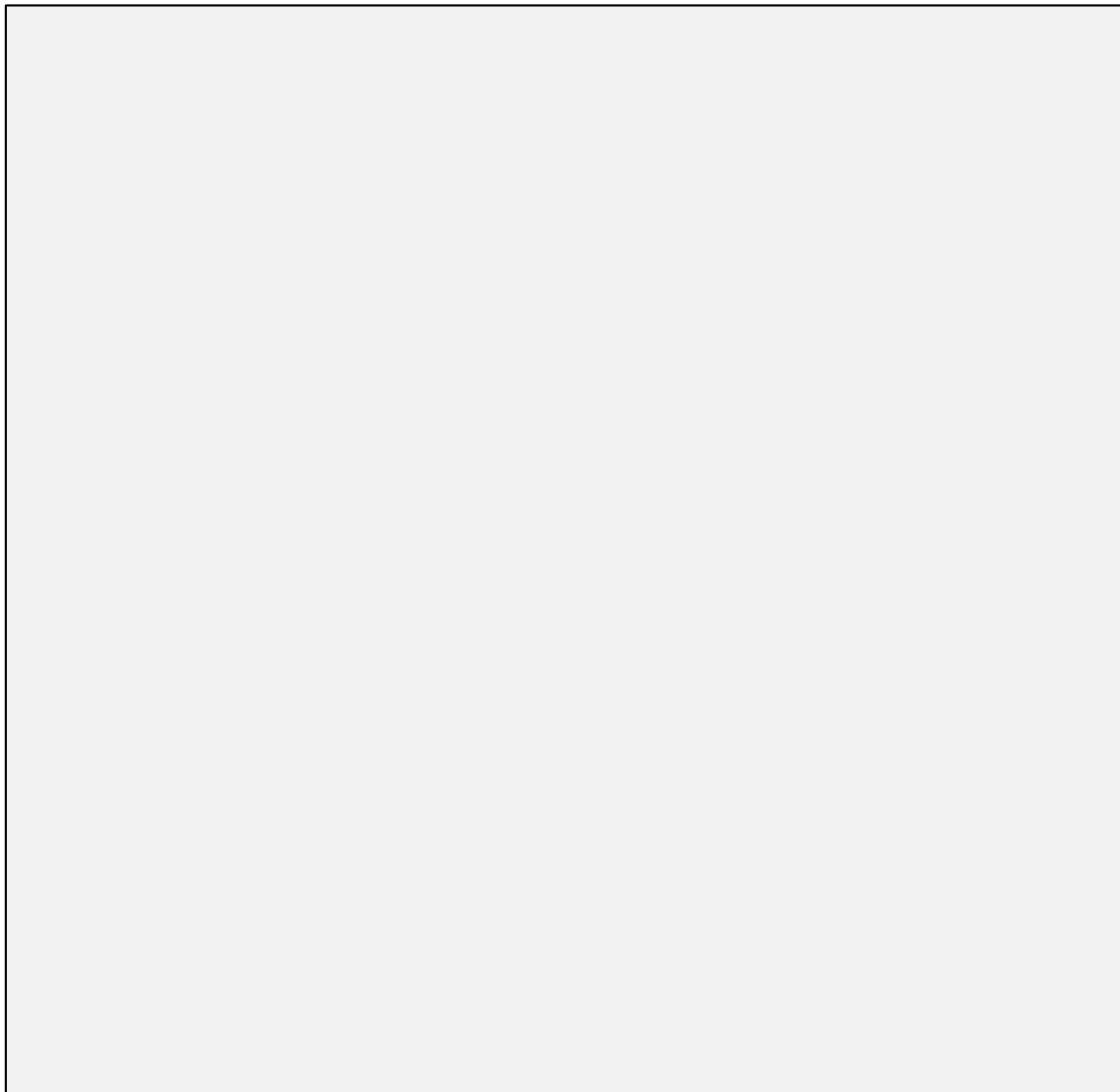
- Working with the Data Engineering Service

- Exercises:

- Data Engineering Data Service Walkthrough
- Create and Trigger Ad Hoc Spark Jobs
- Add Schedule to Ad Hoc Spark Jobs
- Spark Job Data Lineage Using Atlas

Working with Airflow

Chapter 17



Course Chapters

- Introduction
- Why Data Engineering
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- **Working with Airflow**
- Workload Manager Introduction
- Conclusion
- Appendix: Working with Datasets in Scala

Goal of Orchestration in CDE

- **DE and ML practitioners who want to easily define data/ETL pipelines**
- **Multi-step with various types of transformations**
 - Mix of Spark and Hive
- **Managed service with out of the box integration with CDP**

AIRFLOW INTEGRATION



▪ An Integrated, Purpose Built for Data Engineers



SIMPLIFY PIPELINES

- Construct complex pipelines from simpler steps
- Easier to diagnose and optimize
- Modular & increase reusability



OPEN

- Preferred tooling for DE and ML practitioners
- 100s of existing operators
- Strong community support

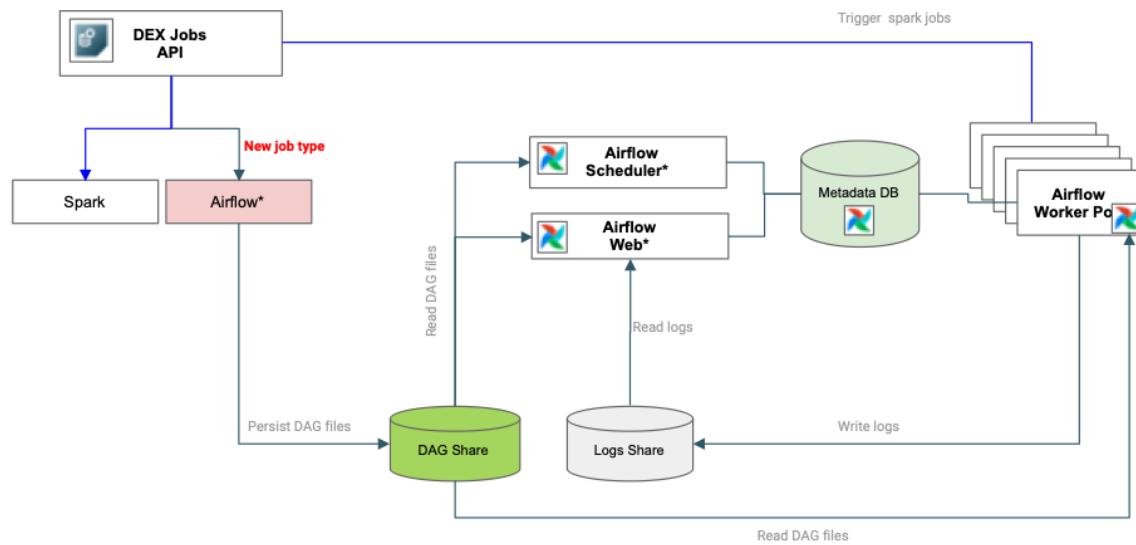


INTEGRATED LIFECYCLE

- Out of the box integration with CDP
- Flexibility to run both Spark and Hive jobs
- Extendible to other CDP end points

Not only about adoption, but driving consumption -reduce barriers to moving workloads to CML

Virtual Cluster Airflow Components



CDE (Spark)

- Define your Spark job(s) in CDE
- Use CDE operator in DAG to reference the Spark job(s)

```
from cloudera.cdp.airflow.operators.cde_operator import CDEJobRunOperator
...
ingest_step1 = CDEJobRunOperator(
    task_id='ingest',
    retries=3,
    dag=dag,
    job_name='automotive-ingest-job'
)
```

Example Airflow DAG

Import operators

main DAG definition
Name & schedule

Operator/step definitions

Define step ordering

```
spark_airflow_dag_hive.py ×
1  from dateutil import parser
2  from datetime import datetime, timedelta
3  from datetime import timezone
4  from airflow import DAG
5  from cloudera.cdp.airflow.operators.cde_operator import CDEJobRunOperator
6  from cloudera.cdp.airflow.operators.cdw_operator import CDWOperator
7
8  default_args = {
9      'owner': 'csso_sahmadian',
10     'retry_delay': timedelta(seconds=5),
11     'depends_on_past': False,
12     'start_date': parser.isoparse('2020-12-08T07:33:37.393Z').replace(tzinfo=timezone.utc)
13 }
14
15 dag = DAG(
16     'airflow-pipeline-demo',
17     default_args=default_args,
18     schedule_interval='@daily',
19     catchup=False,
20     is_paused_upon_creation=False
21 )
22
23 ingest_step1 = CDEJobRunOperator(
24     task_id='ingest',
25     retries=3,
26     dag=dag,
27     job_name='automotive-ingest-job'
28 )
29
30 prep_step2 = CDEJobRunOperator(
31     task_id='data_prep',
32     dag=dag,
33     job_name='insurance-claims-job'
34 )
35
36 vw_query = """
37 show databases;
38 """
39
40 mart_step3 = CDWOperator(
41     task_id='dataset-etl-mart-cdw',
42     dag=dag,
43     cli_conn_id='cdw-hive-aws-demo-2',
44     hql=vw_query,
45     schema='default',
46     ## CDW related args ##
47     use_proxy_user=False,
48     query_isolation=True
49 )
50
51 ingest_step1 >> prep_step2 >> mart_step3
52
53
```

CLOUDERA
Educational Services

Copyright © 2010-2022 Cloudera. A

CDE jobs page showing Airflow & Spark types

Jobs

heavyETL-workload

Search Jobs

Filter By: Status Type

Create Job

Airflow job

Spark jobs

Feedback

Status	Job	Type	Schedule	Modified On	Actions
Running	spark-workflow-dag	Airflow	Scheduled	Dec 21, 2020, 10:17:56 AM	⋮
Running	insurance-claims-job	Spark	Ad-Hoc	Dec 21, 2020, 10:01:48 AM	⋮
Running	automotive-ingest-job	Spark	Ad-Hoc	Dec 21, 2020, 10:00:15 AM	⋮
Running	iot_e2e_test	Spark	Ad-Hoc	Dec 21, 2020, 2:42:25 AM	⋮
Running	DeepAnalysis-1608257144206	Spark	Ad-Hoc	Dec 18, 2020, 5:45:49 PM	⋮
Running	inh_dabur_dabur	Spark	Ad-Hoc	Dec 18, 2020, 5:45:49 PM	⋮

CLOUDERA
Educational Services

Copyright © 2010–2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

17-7

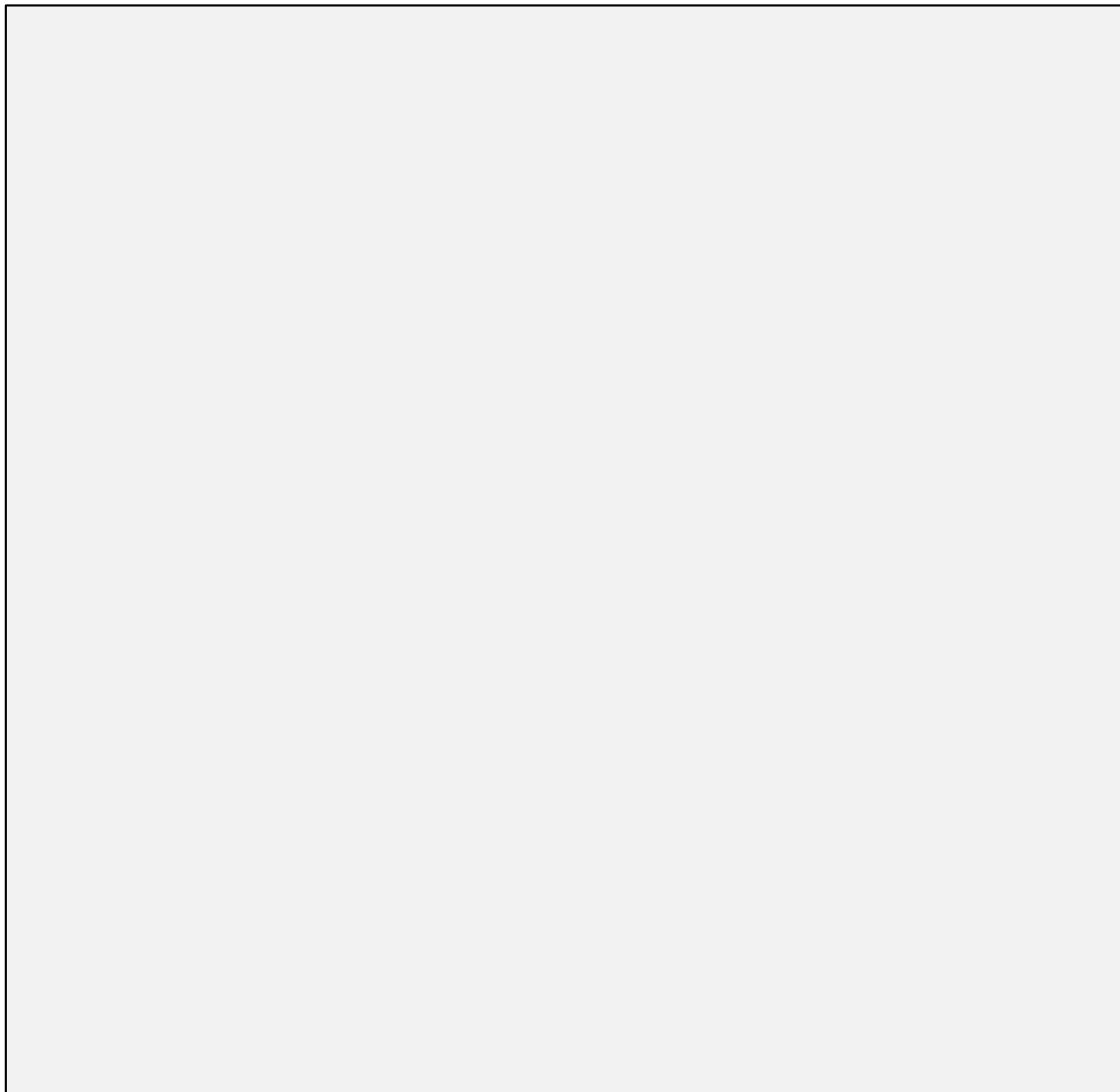
Chapter Topics

Title

- Working with Airflow
- **Exercise: Orchestrate a Set of Jobs Using Airflow**

Workload Manager (WXM) Introduction

Chapter 18



Course Chapters

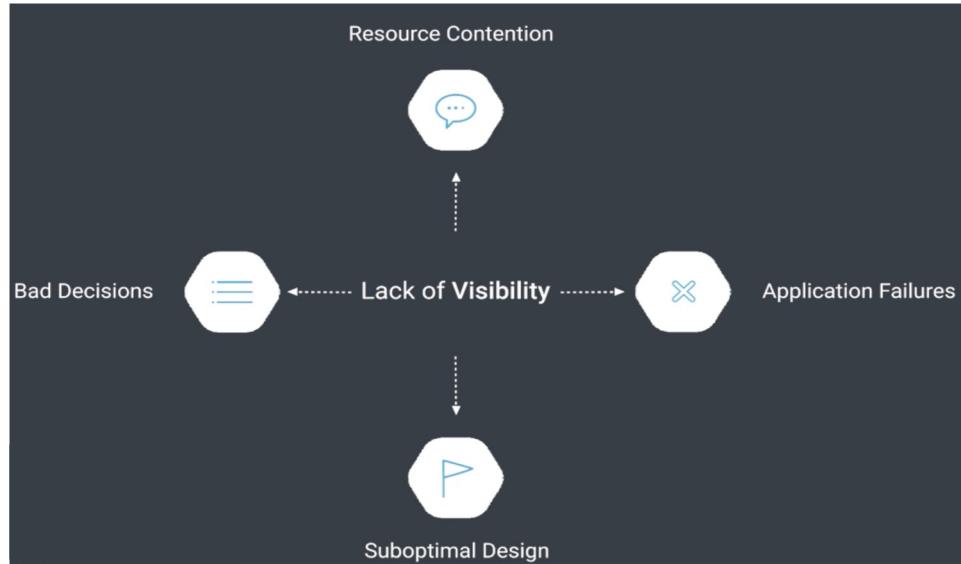
- Introduction
- Why Data Engineering
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- **Workload Manager Introduction**
- Conclusion
- Appendix: Working with Datasets in Scala

Chapter Topics

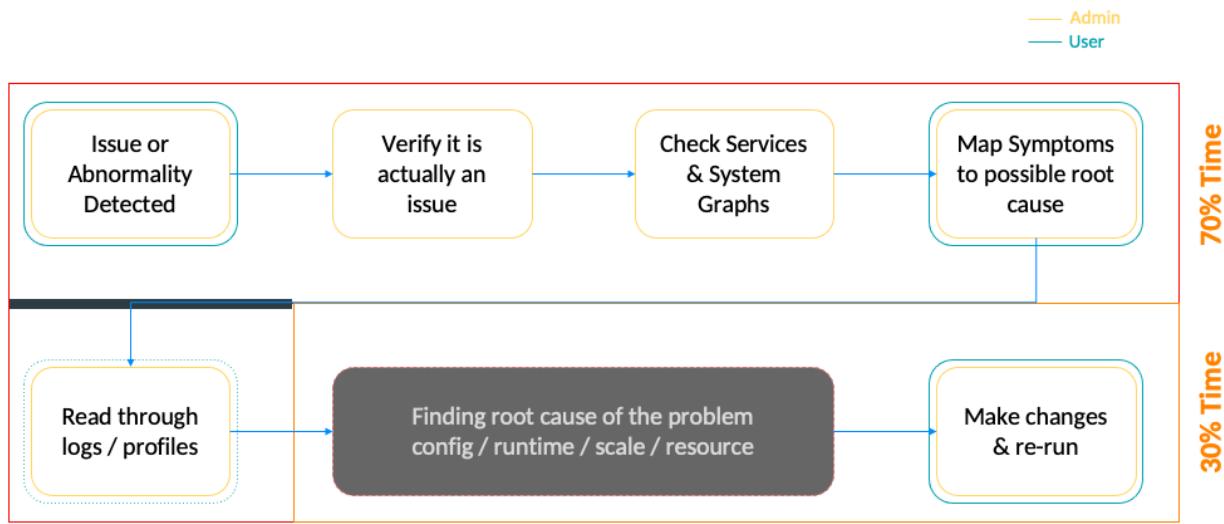
Title

- Why Should You Care?
- What Is It?
- How Does It Work?
- Who Should Use It?
- WXM For Developers: Diagnostic Data Collection Details
- Use Case Example: Troubleshooting Abnormal Job Durations
- Demo: WXM for Spark Deep Dive

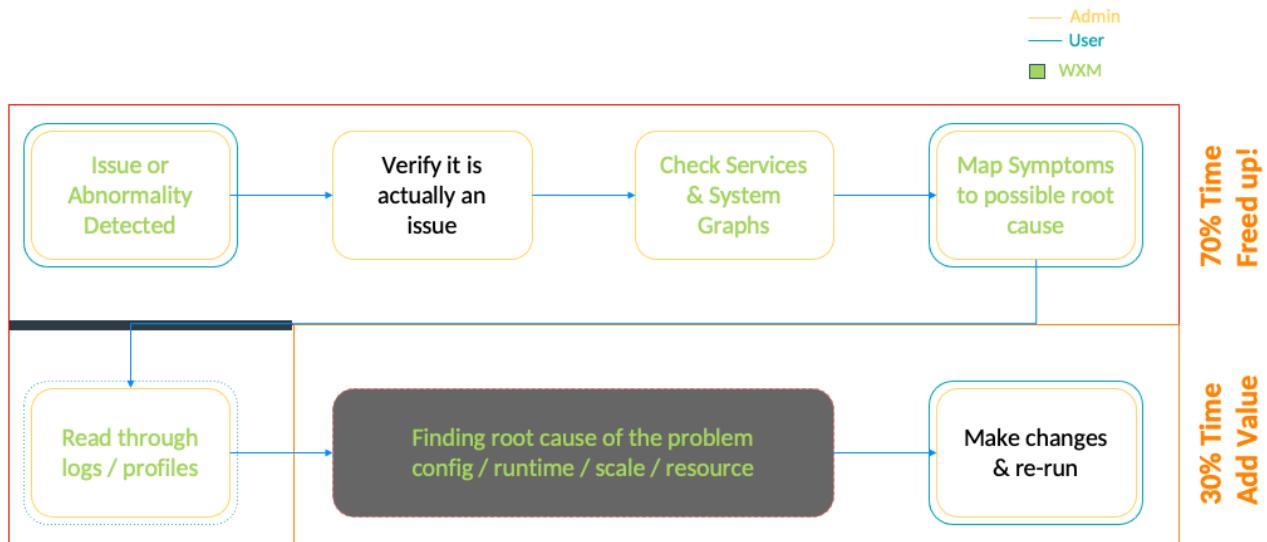
Workload Manager Increase Visibility



Typical Issue Lifecycle



Typical Issue Lifecycle with Workload Manager



Chapter Topics

Title
<ul style="list-style-type: none">▪ Why Should You Care?▪ What Is It?▪ How Does It Work?▪ Who Should Use It?▪ WXM For Developers: Diagnostic Data Collection Details▪ Use Case Example: Troubleshooting Abnormal Job Durations▪ Demo: WXM for Spark Deep Dive

What is Workload Manager?

- **Workload Manager is a tool that provides**
 - insights to help you gain in-depth understanding of the workloads you send to clusters managed by Cloudera Manager
 - information that can be used for troubleshooting failed jobs and optimizing slow jobs that run on those clusters
- **It uses information about job executions to display metrics about the performance of a job and compares the current run of a job to previous runs of the same job by creating baselines.**
- **You can use the knowledge gained from this information to identify and address abnormal or degraded performance or potential performance improvements**

Workload Manager is a tool that provides insights to help you gain in-depth understanding of the workloads you send to clusters managed by Cloudera Manager. In addition, it provides information that can be used for troubleshooting failed jobs and optimizing slow jobs that run on those clusters. After a job ends, information about job execution is sent to Workload Manager with the Telemetry Publisher, a role in the Cloudera Manager Management Service.

Workload Manager uses the information to display metrics about the performance of a job. Additionally, Workload Manager compares the current run of a job to previous runs of the same job by creating baselines. You can use the knowledge gained from this information to identify and address abnormal or degraded performance or potential performance improvements.

Chapter Topics

Title
<ul style="list-style-type: none">▪ Why Should You Care?▪ What Is It?▪ How Does It Work?▪ Who Should Use It?▪ WXM For Developers: Diagnostic Data Collection Details▪ Use Case Example: Troubleshooting Abnormal Job Durations▪ Demo: WXM for Spark Deep Dive

How Does it Work?

- When you enable Workload Manager, the Cloudera Management Service starts the Telemetry Publisher role
- Telemetry Publisher collects and transmits metrics as well as configuration and log files from Impala, Oozie, Hive, YARN, and Spark services for jobs running on CDH clusters to Workload Manager
- This data is collected in the following ways:
 - **Pull** — Telemetry Publisher pulls diagnostic data from these services periodically (once per minute, by default). That's for
 - **Push** — A Cloudera Manager Agent pushes diagnostic data from these services to Telemetry Publisher within 5 seconds after a job finishes. That's for
- After the diagnostic data reaches Telemetry Publisher, it is stored temporarily in its data directory and periodically (once per minute) exported to Workload Manager

When you enable Workload Manager, the Cloudera Management Service starts the Telemetry Publisher role. Telemetry Publisher collects and transmits metrics as well as configuration and log files from Impala, Oozie, Hive, YARN, and Spark services for jobs running on CDH clusters to Workload Manager. Telemetry Publisher collects metrics for all clusters that use the environments where Workload Manager is enabled.

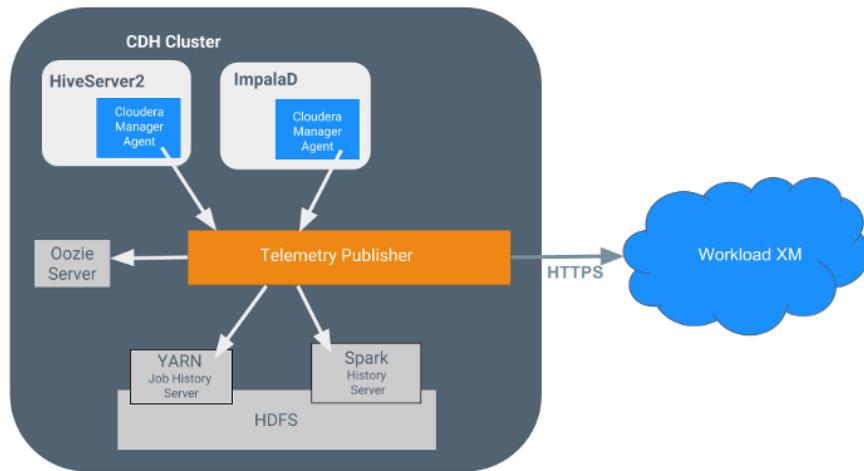
The above diagram shows the sources from which you can configure Telemetry Publisher to collect diagnostic data. This data is collected in the following ways:

- **Pull** — Telemetry Publisher pulls diagnostic data from these services periodically (once per minute, by default). These sources are indicated with the outbound arrows leading from Telemetry Publisher in the above diagram. They are Oozie, YARN, and Spark.
- **Push** — A Cloudera Manager Agent pushes diagnostic data from these services to Telemetry Publisher within 5 seconds after a job finishes. These sources are indicated with the inbound arrows to Telemetry

Publisher. They are Hive and Impala.

After the diagnostic data reaches Telemetry Publisher, it is stored temporarily in its data directory and periodically (once per minute) exported to Workload Manager.

The Telemetry Publisher Connections



When you enable Workload Manager, the Cloudera Management Service starts the Telemetry Publisher role. Telemetry Publisher collects and transmits metrics as well as configuration and log files from Impala, Oozie, Hive, YARN, and Spark services for jobs running on CDH clusters to Workload Manager. Telemetry Publisher collects metrics for all clusters that use the environments where Workload Manager is enabled.

The above diagram shows the sources from which you can configure Telemetry Publisher to collect diagnostic data. This data is collected in the following ways:

- **Pull** – Telemetry Publisher pulls diagnostic data from these services periodically (once per minute, by default). These sources are indicated with the outbound arrows leading from Telemetry Publisher in the above diagram. They are Oozie, YARN, and Spark.
- **Push** – A Cloudera Manager Agent pushes diagnostic data from these services to Telemetry Publisher within 5 seconds after a job finishes. These sources are indicated with the inbound arrows to Telemetry

Publisher. They are Hive and Impala.

After the diagnostic data reaches Telemetry Publisher, it is stored temporarily in its data directory and periodically (once per minute) exported to Workload Manager.

Chapter Topics

Title

- Why Should You Care?
- What Is It?
- How Does It Work?
- Who Is It For?
- WXM For Developers: Diagnostic Data Collection Details
- Use Case Example: Troubleshooting Abnormal Job Durations
- Demo: WXM for Spark Deep Dive

Who Is It For?



Hadoop / System
/ Database Admin



Data Architects



Data Engineering
/ BI Developer



System
Integrator



Central Support
Team



Sales Engg / PS

Workload XM is a multi-persona tool. It is not only aimed at Admins but is extremely useful to data architects and application developers like DE/BI developers. Another very important persona is the Central Support Team at the customer & Cloudera

Chapter Topics

Title

- Why Should You Care?
- What Is It?
- How Does It Work?
- Who Is It For?
- **WXM For Developers: Diagnostic Data Collection Details**
- Use Case Example: Troubleshooting Abnormal Job Durations
- Demo: WXM for Spark Deep Dive

MapReduce Jobs

- **Telemetry Publisher polls the YARN Job History Server for recently completed MapReduce jobs**
- **For each of these jobs, Telemetry Publisher collects the configuration and jhist file, which is the job history file that contains job and task counters, from HDFS**
- **Telemetry Publisher can be configured to collect MapReduce task logs from HDFS and send them to Workload Manager**
 - By default, this log collection is turned off

- **MapReduce Jobs** — Telemetry Publisher polls the YARN Job History Server for recently completed MapReduce jobs. For each of these jobs, Telemetry Publisher collects the configuration and jhist file, which is the job history file that contains job and task counters, from HDFS. Telemetry Publisher can be configured to collect MapReduce task logs from HDFS and send them to Workload Manager. By default, this log collection is turned off.

Spark Applications

- Telemetry Publisher polls the Spark History Server for recently completed Spark applications.
- For each of these applications, Telemetry Publisher collects their event log from HDFS.
- Telemetry Publisher only collects Spark application data from Spark version 2.2 and later.
- Telemetry Publisher can be configured to collect the executor logs of Spark applications from HDFS and send them to Workload Manager,
 - but this data collection is turned off by default.
- **Important:** CDH version 5.x is packaged with Spark 1.6 so you cannot configure Telemetry Publisher data collection for CDH 5.x clusters unless you are using CDS 2.2 Powered by Apache Spark or later versions with those clusters

- **Spark Applications** – Telemetry Publisher polls the Spark History Server for recently completed Spark applications. For each of these applications, Telemetry Publisher collects their event log from HDFS. Telemetry Publisher only collects Spark application data from Spark version 2.2 and later. Telemetry Publisher can be configured to collect the executor logs of Spark applications from HDFS and send them to Workload Manager, but this data collection is turned off by default.

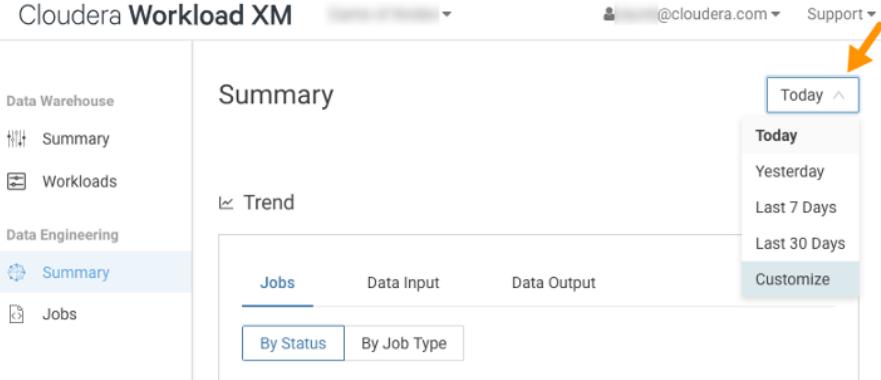
Important: CDH version 5.x is packaged with Spark 1.6 so you cannot configure Telemetry Publisher data collection for CDH 5.x clusters unless you are using CDS 2.2 Powered by Apache Spark or later versions with those clusters.

Chapter Topics

Title

- Why Should You Care?
- What Is It?
- How Does It Work?
- Who Is It For?
- WXM For Developers: Diagnostic Data Collection Details
- **Use Case Example: Troubleshooting Abnormal Job Durations**
- Demo: WXM for Spark Deep Dive

1 - Specify a Time Range



The screenshot shows the Cloudera Workload XM interface. On the left, there's a sidebar with sections for Data Warehouse (Summary, Workloads), Data Engineering (Summary, Jobs), and a CLOUDERA Educational Services logo. The main area is titled 'Summary' and contains a 'Trend' section with tabs for 'Jobs', 'Data Input', and 'Data Output'. Below these tabs are buttons for 'By Status' and 'By Job Type'. In the top right corner, there's a dropdown menu for specifying a time range. The menu includes options like 'Today', 'Yesterday', 'Last 7 Days', 'Last 30 Days', and 'Customize'. An orange arrow points to the 'Today' button, which is highlighted with a blue border. The background of the dropdown menu has a light gray gradient.

Cloudera Workload XM

Summary

Trend

Jobs Data Input Data Output

By Status By Job Type

Today Yesterday Last 7 Days Last 30 Days Customize

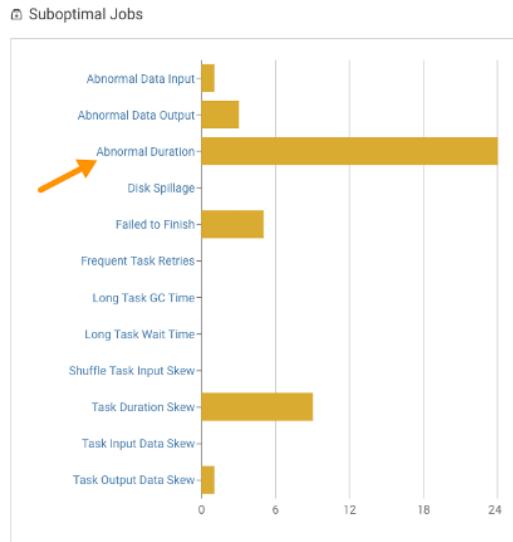
CLOUDERA Educational Services

Copyright © 2010-2022 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

18-17

On the Data Engineering Summary page, click the time range in the upper right corner of the page and specify a time range you are interested in.

2 - Select Abnormal Duration



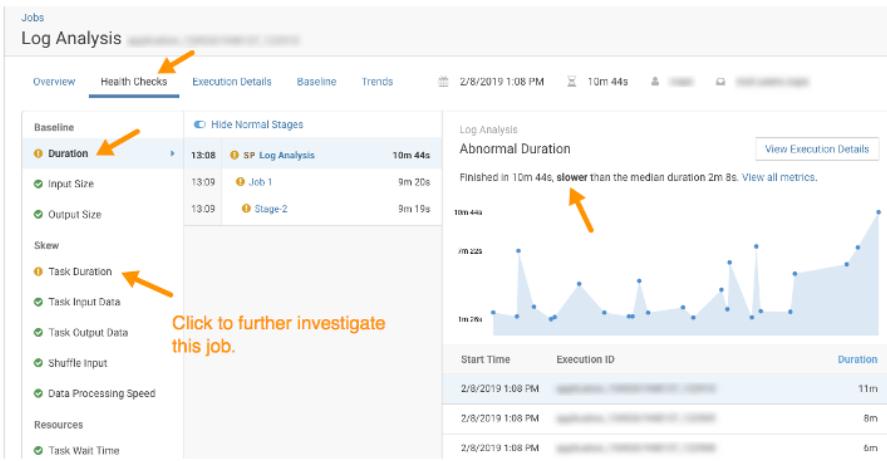
In the Suboptimal Jobs graph, click the Abnormal Duration health check to view the number of jobs with an abnormal duration that executed within the selected time frame. Any jobs that fall outside of the baseline duration will be marked as slow. Hover over the graph to see how many jobs triggered each health check.

3 - Select a Duration Range

The screenshot shows the 'Jobs' section of the Cloudera Data Engineering interface. At the top, there are filters for Type (All), Status (All), Health Check (Health Check), Duration (Duration), and Range (Range). A yellow arrow points to the 'Duration' dropdown menu, which is open and displays 'All', '0s - 1h 0m 0s', '1h 0m 0s - 5h 0m 0s', '5h 0m 0s +', and 'Customize...'. Another yellow arrow points to the 'Customize...' option. Below the dropdown, a message says 'Choose a duration range from the Duration list, or choose Customize to enter a custom maximum-minimum duration range.' To the right of the dropdown, there is a table titled 'Health Issue' with several items listed under it, such as 'Abnormal Data Output', 'Task Output Data Skew', etc.

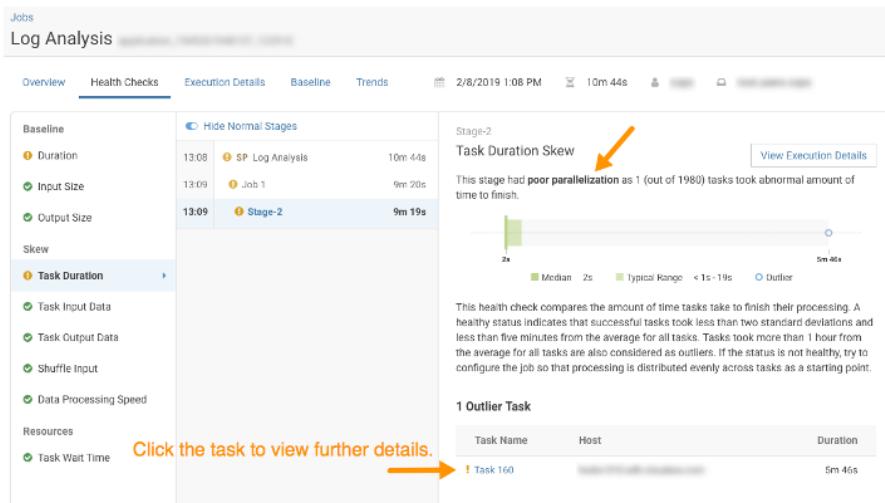
After clicking Abnormal Duration in the Suboptimal Jobs graph, a list of all slow jobs displays on the Data Engineering Jobs page. These jobs have all triggered the Duration health check.
From the Duration drop-down list, select a duration range or select Customize to enter a custom minimum or maximum duration to view any jobs that meet that duration criteria.

4 - Inspect the Job Details



Click on the Job name to view more detailed information, and then on the Jobs detail page, click on the Health Checks tab. Under the Duration health check, you can see that this job finished much slower than the normal duration.

5 - Identify the Outlier Task



After clicking Task Duration, you can see that this job contains a task that took an abnormal amount of time to finish.

Click the task to view further details about it.

6 - Identify the Cause of the Abnormal Duration

Jobs
insert_overwrite_table_all_prod_user

Execution Details Click Execution Details to view the query code and configuration information.

Overview Health Checks Execution Details Baseline Trends 4/18/2018 8:51 PM 54m 37s ID

Baseline

Duration	20:51	HV	54m 37s
Input Size	20:51	Init Stage-1	49m 9s
Output Size	20:51	Map Stage	43m 22s

Skew

Task Duration	20:51	Map Stage	43m 22s
Task Input Data	20:51	Map Stage	43m 22s
Task Output Data	20:51	Map Stage	43m 22s
Shuffle Input	20:51	Map Stage	43m 22s
Data Processing Speed	20:51	Map Stage	43m 22s

Resources

Task Wait Time	20:51	Map Stage	43m 22s
Task GC Time	20:51	Map Stage	43m 22s
Disk Spillage	20:51	Map Stage	43m 22s
Task Retries	20:51	Map Stage	43m 22s

Task Details

Map Stage / Task 000645

Metric	Task	Average
GC Time elapsed	21m 53s	22s 200ms
CPU time spent	7h 34m	10m 16s
Successful Attempt Duration	26m 11s	6m 24s
Wait Duration	3s 980ms	10m
Data writer (Local)	167.2 MB	102 MB
Map output materialized bytes	79.1 MB	49.2 MB
Duration	26m 19s	16m 20s
Data read (Local)	342.9 MB	261.7 MB
Map output bytes	781.9 MB	612.2 MB
Spilled Records	12.2M	9.7M

After clicking the task, the Task Details pane displays details about its run. In the following example, garbage collection is taking significantly more time than the average task.
Click Task GC Time to view more information about garbage collection for this job.

7 - Drill Down on the Execution Details

Time	Status	Stage	Duration
20:51	OK	Stage-1	54m 37s
21:40	OK	Stage-2	4m 47s
21:45	OK	Stage-2	26s

On the Task GC Time page, click the Execution Details tab, and then click one of the MapReduce stages.

8 - View the Related Configuration Settings

The screenshot shows the Cloudera Manager interface for managing Hadoop jobs. On the left, there's a list of jobs with their start times, durations, and stages. In the center, under the 'Configurations' section for 'Stage-1', there is a search bar with the placeholder 'Search [map memory]'. Below the search bar, it says 'Type part of the configuration property name to search for it.' A tooltip above the search bar also says 'Search [map memory]'. At the bottom of the configurations section, it says 'Displaying 1 - 1' and has a '10 per page' dropdown.

In the MapReduce stage Summary page, click View Configurations, and then enter part of the MapReduce memory configuration property name to search for and view the configuration for garbage collection:
In the above case, setting this property to 1024 might be causing the mapper JVM to have insufficient memory, which triggers too frequent garbage collection. Increasing this number might improve performance on your cluster.

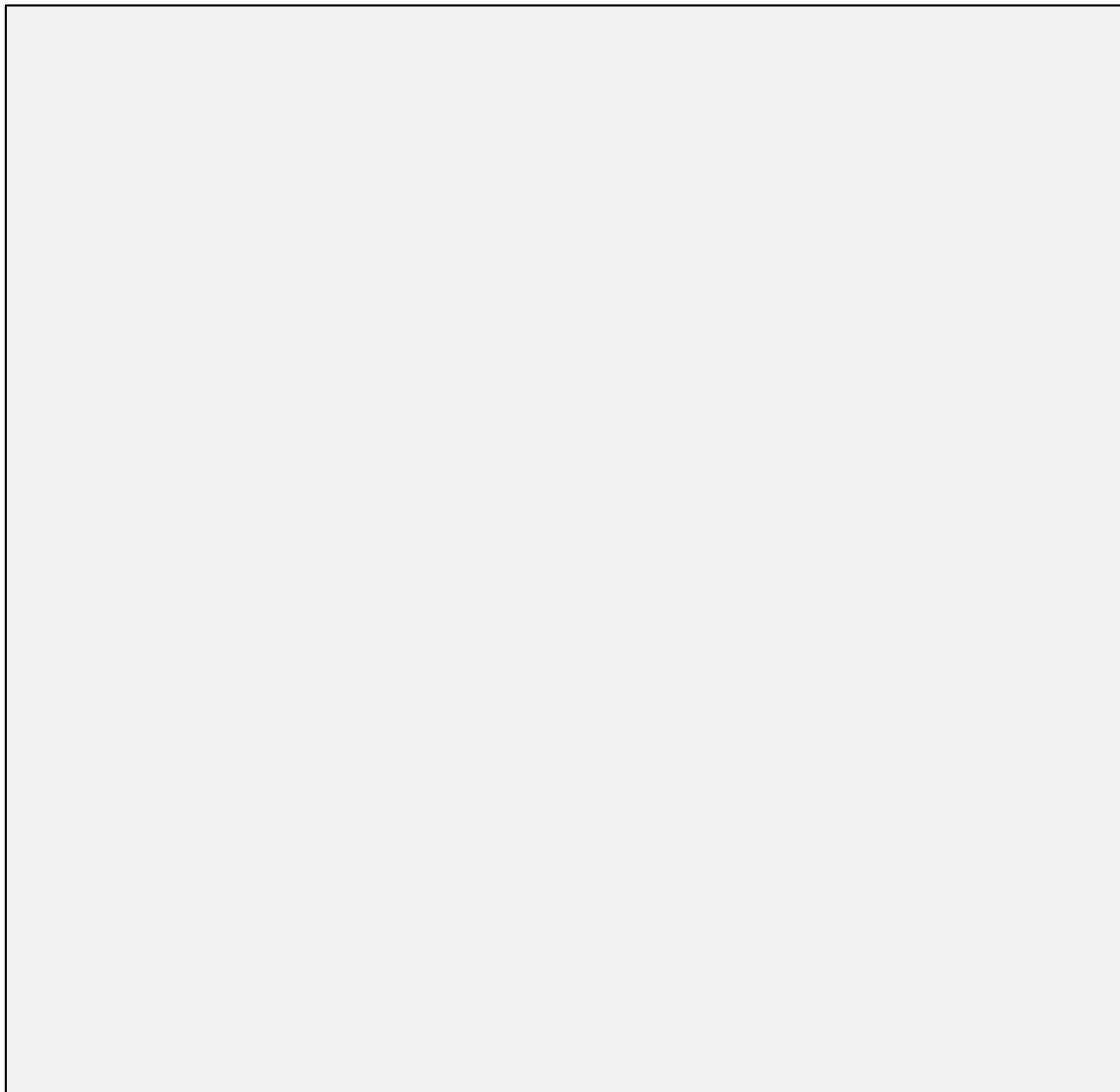
Chapter Topics

Title

- Why Should You Care?
- What Is It?
- How Does It Work?
- Who Is It For?
- WXM For Developers: Diagnostic Data Collection Details
- Use Case Example: Troubleshooting Abnormal Job Durations
- Demo: WXM for Spark Deep Dive

Conclusion

Chapter 19



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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

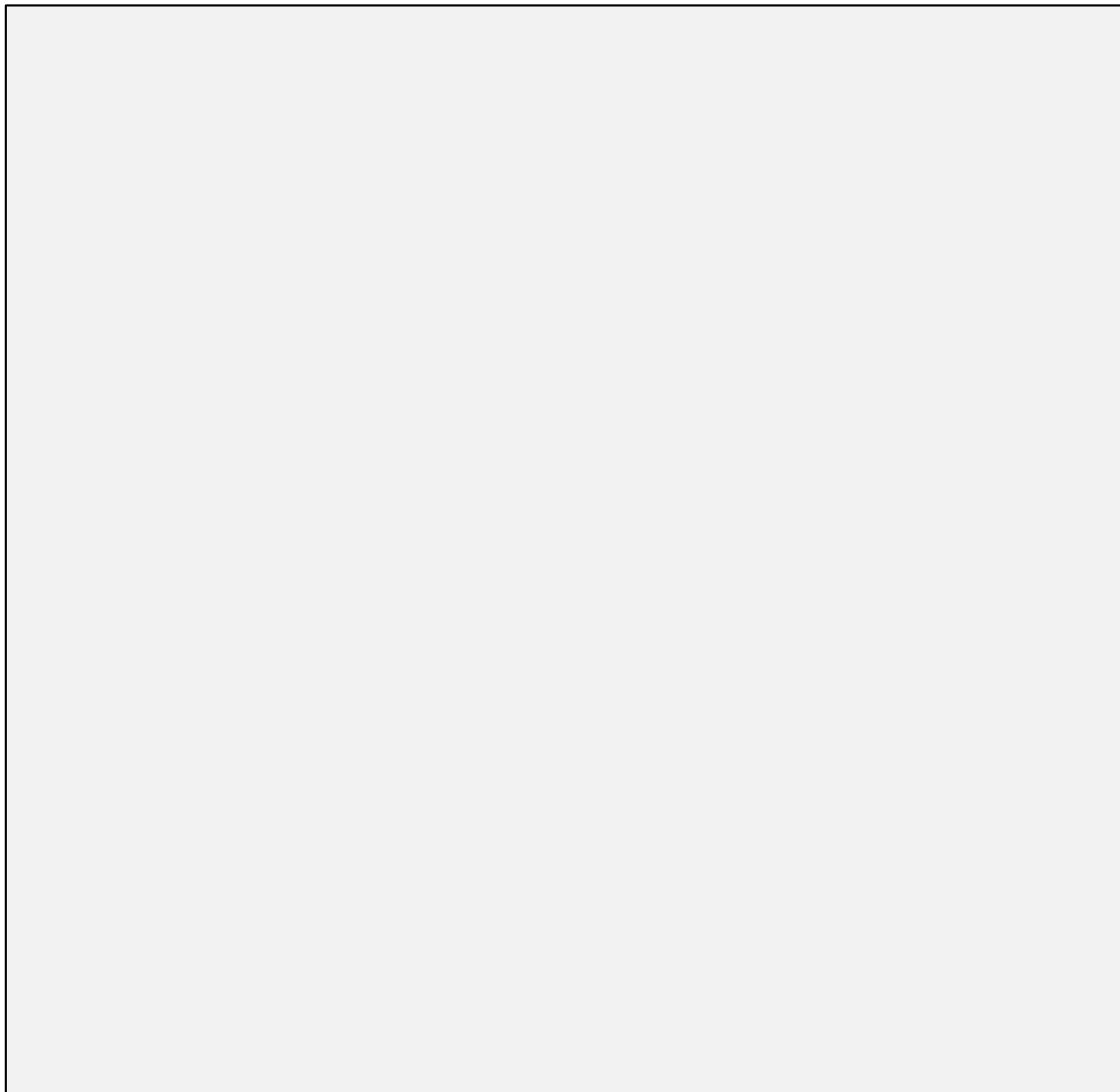
- Administrator Training: CDP Private Cloud Base

■ For data analysts and data scientists

- Cloudera Data Analyst Training
- Cloudera Data Scientist Training

Working with Datasets in Scala

Appendix A



Course Chapters

- Introduction
- Why Data Engineering?
- Introduction to Zeppelin
- HDFS Introduction
- YARN Introduction
- Distributed Processing History
- Working With RDDs
- Working With Data Frames
- Introduction to Apache Hive
- Transforming Data with Hive
- Data Engineering with Hive
- Hive Spark Integration
- Distributed Processing Challenges
- Spark Distributed Processing
- Spark Distributed Persistence
- Working with the Data Engineering Service
- Working with Airflow
- Workload Manager Introduction
- 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