

CLOUDERA

Educational Services

Cloudera Data Analyst Training



Introduction

Chapter 1

Course Chapters

- **Introduction**
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Trademark Information

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

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

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

Chapter Topics

Introduction

- **About This Course**
- Introductions
- About Cloudera
- About Cloudera Educational Services
- Course Logistics
- Hands-On Exercise: Launching the Exercise Environment

Course Objectives (1)

During this course, you will learn

- How the open source ecosystem of big data tools addresses challenges not met by traditional RDBMSs
- How Apache Hive and Apache Impala are used to provide SQL access to data
- How Hive and Impala syntax and data formats, including functions and subqueries, help answer questions about data
- How to create, modify, and delete tables, views, and databases; load data; and store results of queries
- How to create and use partitions and different file formats

Course Objectives (2)

- How to combine two or more datasets using JOIN or UNION, as appropriate
- What analytic and windowing functions are, and how to use them
- How to store and query complex or nested data structures
- How to process and analyze semi-structured and unstructured data
- Different techniques for optimizing Hive and Impala queries
- How to extend the capabilities of Hive and Impala using parameters, custom file formats and SerDes, and external scripts
- How to determine whether Hive, Impala, an RDBMS, or a mix of these is best for a given task

Chapter Topics

Introduction

- About This Course
- **Introductions**
- About Cloudera
- About Cloudera Educational Services
- Course Logistics
- Hands-On Exercise: Launching the Exercise Environment

Introductions

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

Chapter Topics

Introduction

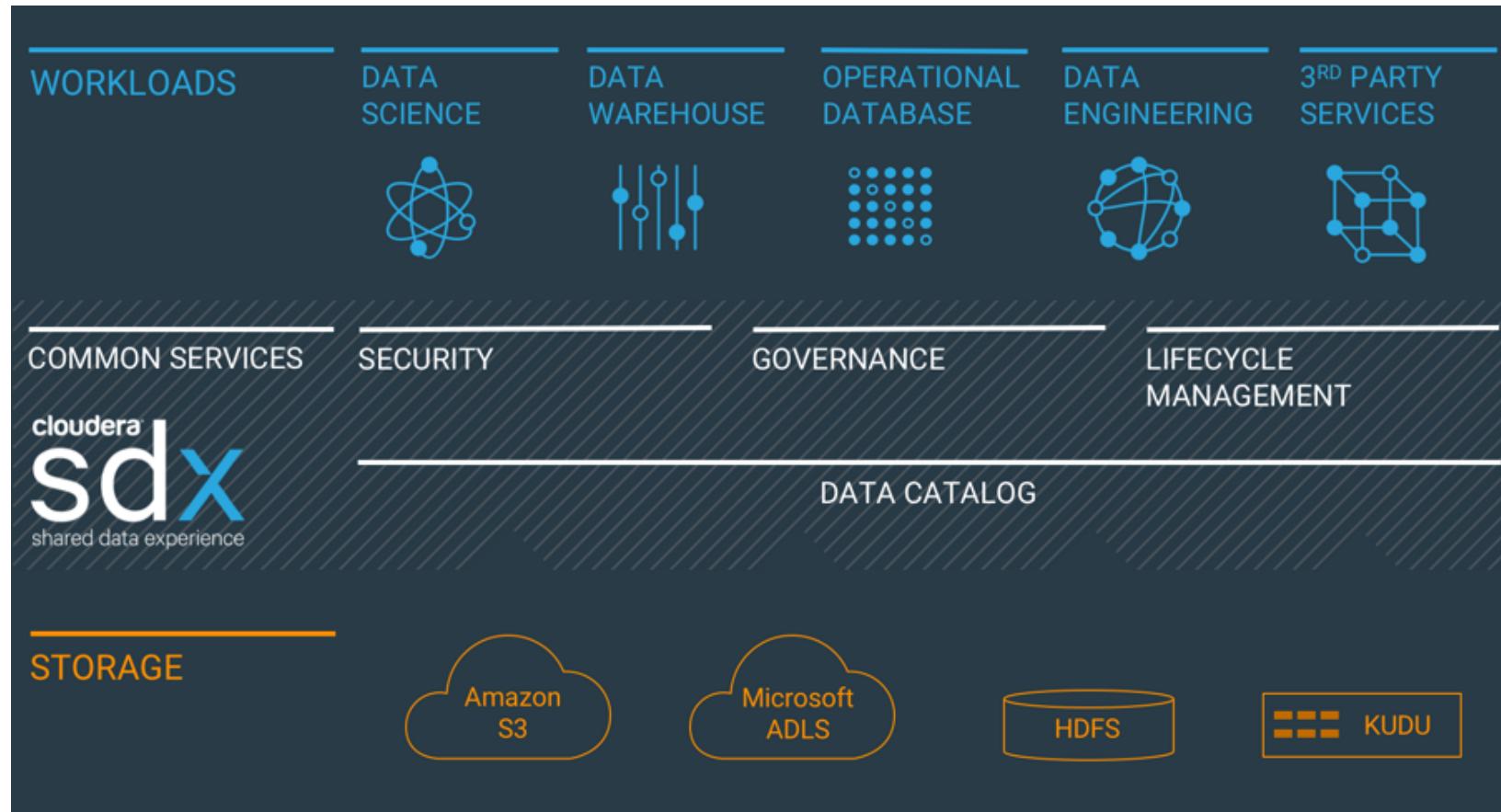
- About This Course
- Introductions
- **About Cloudera**
- About Cloudera Educational Services
- Course Logistics
- Hands-On Exercise: Launching the Exercise Environment

About Cloudera



- Cloudera (founded 2008) and Hortonworks (founded 2011) merged in 2019
- The “new” Cloudera creates world’s leading big data platform
- Delivering an Enterprise Data Cloud for any data from the Edge to AI
- Leader in training, certification, support, and consulting for big data professionals

Cloudera Shared Data Experience



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

Cloudera Enterprise Data Hub



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

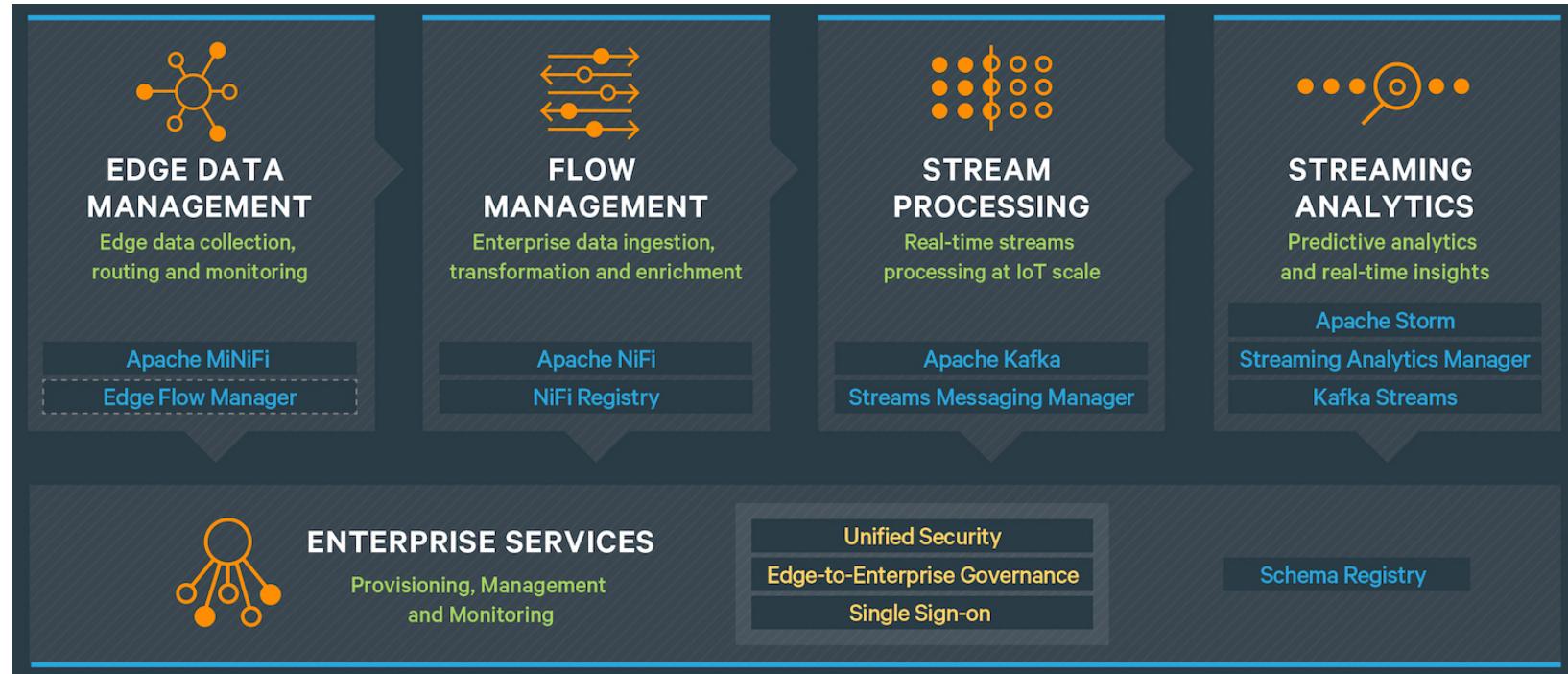
Cloudera Data Science Workbench (CDSW)

- Fast, easy, and secure self-service data science in enterprise environments
 - Browser-based interface
 - Direct access to a secure Cloudera (CDH) or Hortonworks (HDP) cluster running Spark and other tools
 - Isolated environments for running Python, R, and Scala code
 - Teams, version control, collaboration, and sharing

The screenshot shows the CDSW interface with the following components:

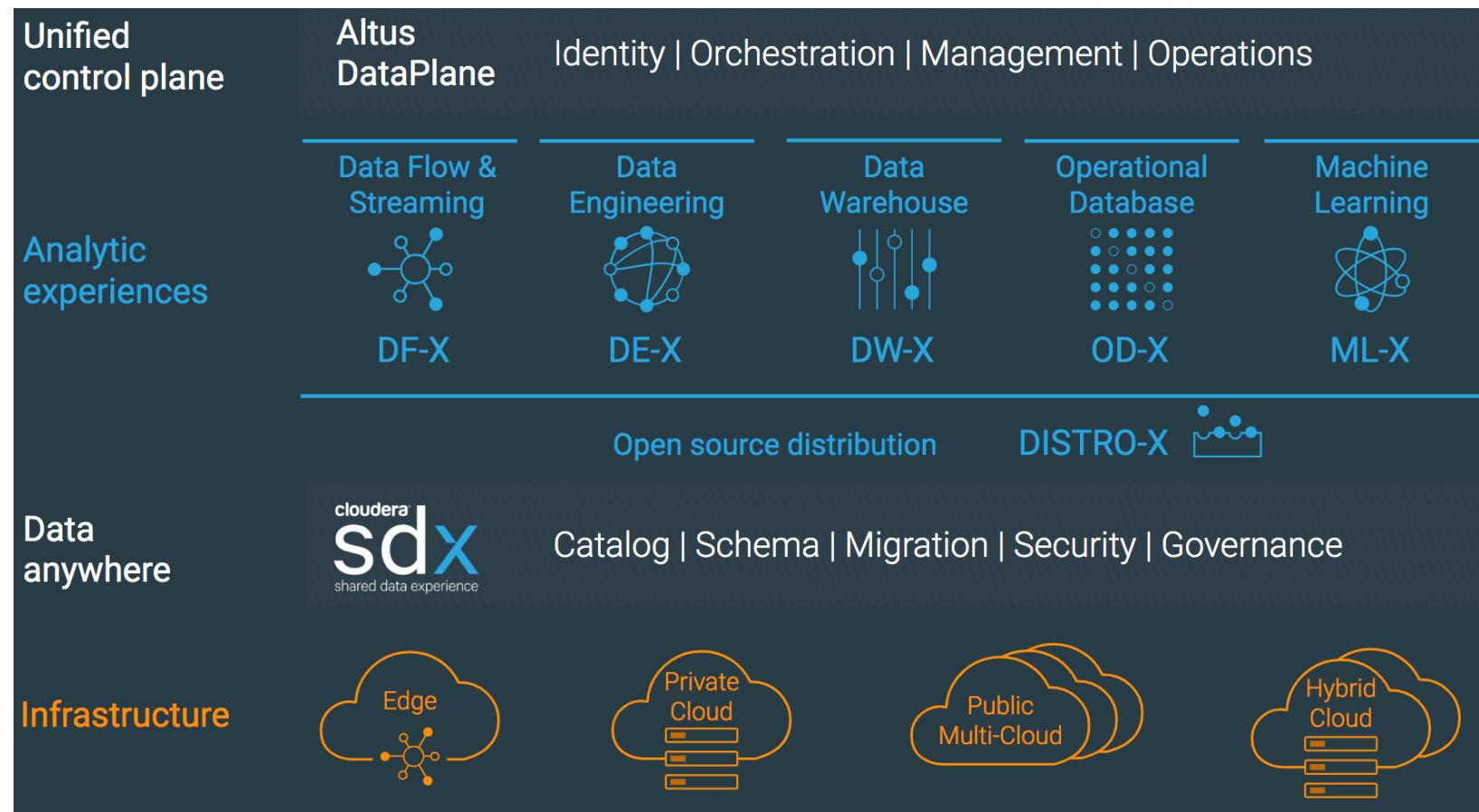
- Top Bar:** Shows project statistics: 4 sessions running, 3 jobs running, 2 models running, 2.00 vCPU, and 8.00 GB memory.
- Left Sidebar:** Includes links for Projects, Sessions, Experiments, Models, and Jobs.
- Central Area:**
 - Terminal:** A code editor window titled "analysis.py" containing Python code for data manipulation and visualization.
 - Terminal Access:** A terminal window showing a scatter plot comparing total_bill and tip for smokers and non-smokers.
 - Data Grid:** A table showing the first five rows of the dataset with columns: index, total_bill, tip, sex, smoker, day, and time.

Cloudera DataFlow



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

Cloudera Data Platform



- **Public, private, and hybrid clouds**
- **Shared data experience**
- **Powered by open source**
- **Analytics from the Edge to AI**
- **Unified data control plane**

Chapter Topics

Introduction

- About This Course
- Introductions
- About Cloudera
- **About Cloudera Educational Services**
- Course Logistics
- Hands-On Exercise: Launching the Exercise Environment

Cloudera Educational Services

- **The leader in Apache Hadoop-based training**
- **We offer a variety of courses, both instructor-led (in physical or virtual classrooms) and self-paced OnDemand, 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
- **We also offer private courses**
 - Can be delivered on-site or virtually
 - Can be tailored to suit customer needs

Cloudera OnDemand

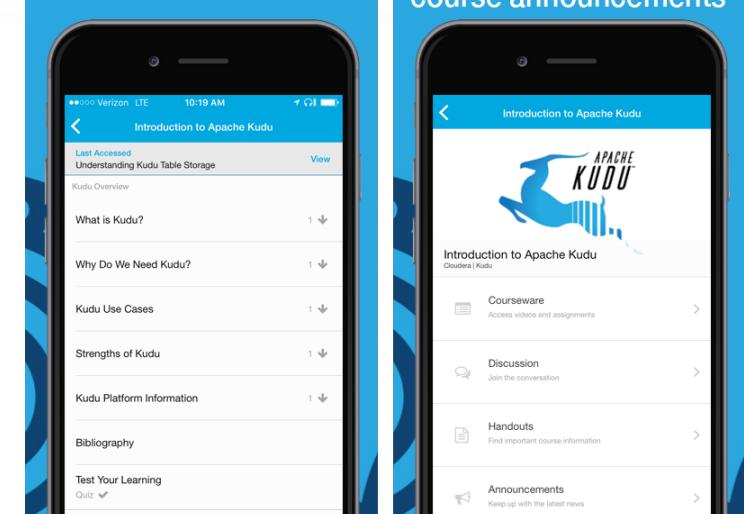
- Our OnDemand catalog includes
 - Courses for developers, data analysts, administrators, and data scientists, updated regularly
 - Exclusive OnDemand-only courses on security, Cloudera Data Science Workbench, and installing on Microsoft Azure
 - Free courses such as *Essentials* and *Cloudera Director*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

Accessing Cloudera OnDemand

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

The screenshot shows the Cloudera OnDemand web interface. At the top, there are tabs for Home, Course (which is selected), Discussion, and Progress. Below the tabs is a search bar labeled "Search course content...". A sidebar on the left lists course categories: "Introduction to Apache Hadoop and the Hadoop Ecosystem" (selected), "Apache Hadoop Overview", "Data Storage and Ingest", "Data Processing", "Data Analysis and Exploration", "Other Ecosystem Tools", "Intro to the Hands-On Exercises", "Hands-On Exercise: Accessing the Exercise Environment", "Hands-On Exercise: General Notes", "Hands-On Exercise: Query Hadoop Data with Apache Impala", "Test Your Learning" (with a "Quiz" link), "Apache Hadoop File Storage", and "Data Processing on an Apache Hadoop Cluster". The main content area displays the "Apache Hadoop Overview" page, which includes a title, a navigation bar, a sidebar with "Common Hadoop Use Cases" (listing ETL, Data analysis, Text mining, Index building, Graph creation and analysis, Pattern recognition, Data storage, Collaborative filtering, Prediction models, Sentiment analysis, and Risk assessment), and a video player showing "0:56 / 3:27". To the right of the video player is a transcript section with a "Start of transcript. Skip to the end." link. The bottom right corner features a mobile device icon with the text "Ask questions, read forums, and receive course announcements".

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



Cloudera Certification

- The leader in Apache Hadoop-based certification
- All Cloudera professional certifications are hands-on, performance-based exams requiring you to complete a set of real-world tasks on a working multi-node CDH cluster
- We offer two levels of certifications
 - **Cloudera Certified Professional (CCP)**
 - The industry's most demanding performance-based certification, CCP Data Engineer evaluates and recognizes your mastery of the technical skills most sought after by employers
 - CCP Data Engineer
 - **Cloudera Certified Associate (CCA)**
 - To achieve CCA certification, you complete a set of core tasks on a working CDH cluster instead of guessing at multiple-choice questions
 - CCA Spark and Hadoop Developer
 - CCA Data Analyst
 - CCA Administrator

Chapter Topics

Introduction

- About This Course
- Introductions
- About Cloudera
- About Cloudera Educational Services
- **Course Logistics**
- Hands-On Exercise: Launching the Exercise Environment

Logistics

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

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

Chapter Topics

Introduction

- About This Course
- Introductions
- About Cloudera
- About Cloudera Educational Services
- Course Logistics
- **Hands-On Exercise: Launching the Exercise Environment**

Hands-On Exercise: Starting the Exercise Environment

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



Apache Hadoop Fundamentals

Chapter 2

Course Chapters

- Introduction
- **Apache Hadoop Fundamentals**
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Apache Hadoop Fundamentals

In this chapter, you will learn

- How the needs of big data pose challenges for traditional RDBMSs
- How systems designed for big data enable better storage and processing for big data
- What are several tools in the big data open source ecosystem

Chapter Topics

Apache Hadoop Fundamentals

- **The Motivation for Hadoop**
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

Volume

- **Every day...**
 - Over 2.6 billion shares are traded on the New York Stock Exchange
 - Facebook accepts 300 million photo uploads
 - Google processes over 24 petabytes of data
- **Every minute...**
 - Facebook users share over 3.2 million pieces of content
 - Email users send about 200 million messages
- **Every second...**
 - Financial institutions process more than 10,000 credit card transactions
 - Amazon Web Services fields more than 650,000 requests
- **And it's not stopping...**
 - They can't collect it all yet, but the Large Hadron Collider produces 572 terabytes of data per second

Velocity

- **We are generating data faster than ever**
 - Automated processes are faster than manual processes
 - Numerous interconnected systems export and transfer data frequently
- **Online interactions and streaming video require fast data transfer**

Variety

- **We are producing a wide variety of data**
 - Social network connections
 - Server and application log files
 - Electronic medical records
 - Images, audio, and video
 - RFID and wireless sensor network events
 - Product ratings on shopping and review websites
 - And much more...
- **Not all of this maps cleanly to the relational model**

Value

- **This data has many valuable applications**
 - Product recommendations
 - Demand prediction
 - Marketing analysis
 - Fraud detection
 - And many, many more...
- **We must process it to extract that value**
 - And processing *all the data* can yield more accurate results

A System that Scales

- **Traditional data stores:**
 - Can't handle a big data workload on any one machine
 - Weren't designed to scale out to multiple machines
- **Key problems:**
 - How to store large amounts of data reliably
 - How to analyze that data

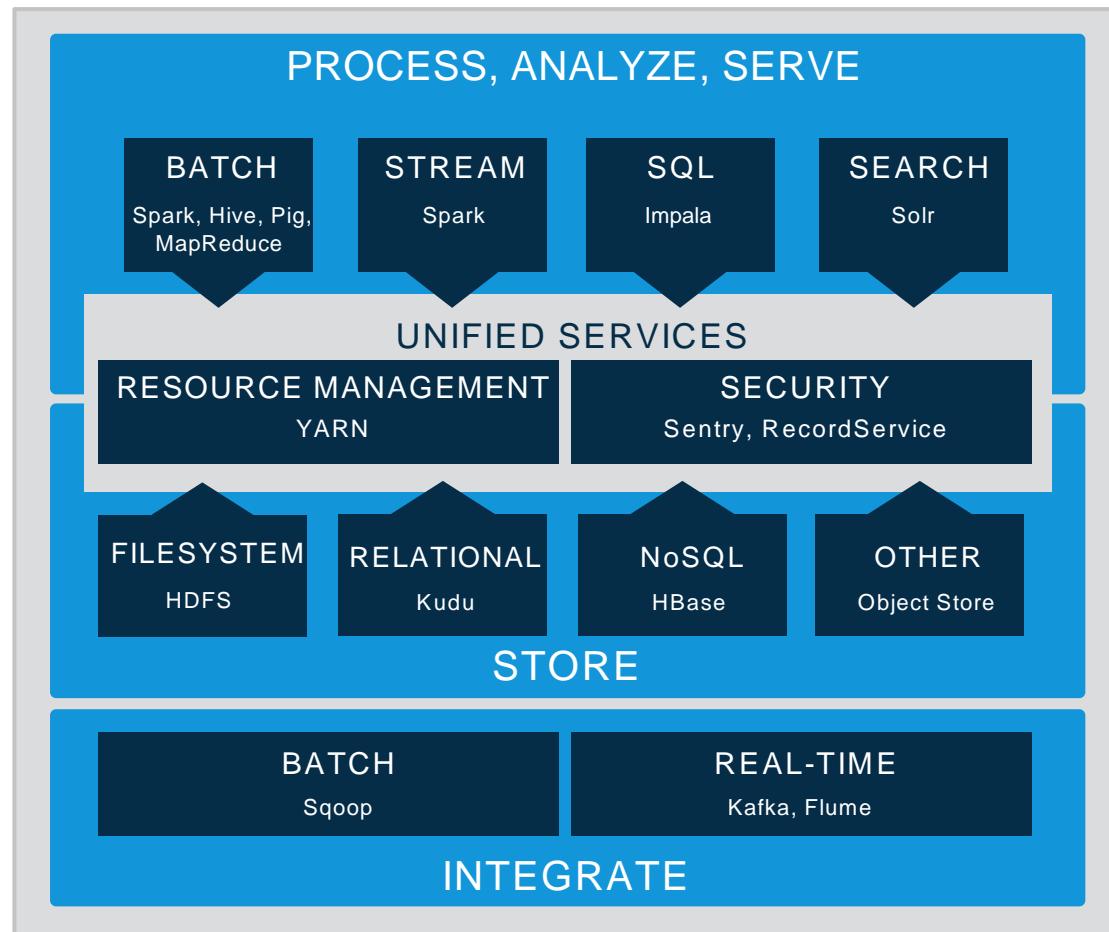
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- **Hadoop Overview**
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

What Is Apache Hadoop?

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



Scalability

- **Hadoop is a distributed system**
 - A collection of servers running Hadoop software is called a *cluster*
- **Individual servers within a cluster are called *nodes***
 - Typically industry-standard servers running Linux
 - Each node both stores and processes data
- **Add more nodes to the cluster to increase scalability**
 - A cluster may contain up to several thousand nodes
 - You can scale out incrementally as required

Fault Tolerance

- **Adding nodes increases capacity but also the chance that any one of them will fail**
 - Solution: Build redundancy into the system and handle it automatically
- **Files loaded into Hadoop are replicated across nodes in the cluster**
 - If a node fails, its data is replicated using one of the other copies
- **Data processing jobs are broken into individual tasks**
 - Each task takes a small amount of data as input
 - Thousands of tasks (or more) often run in parallel
 - If a node fails during processing, its tasks are rescheduled elsewhere
- **Routine failures are handled automatically without any loss of data**

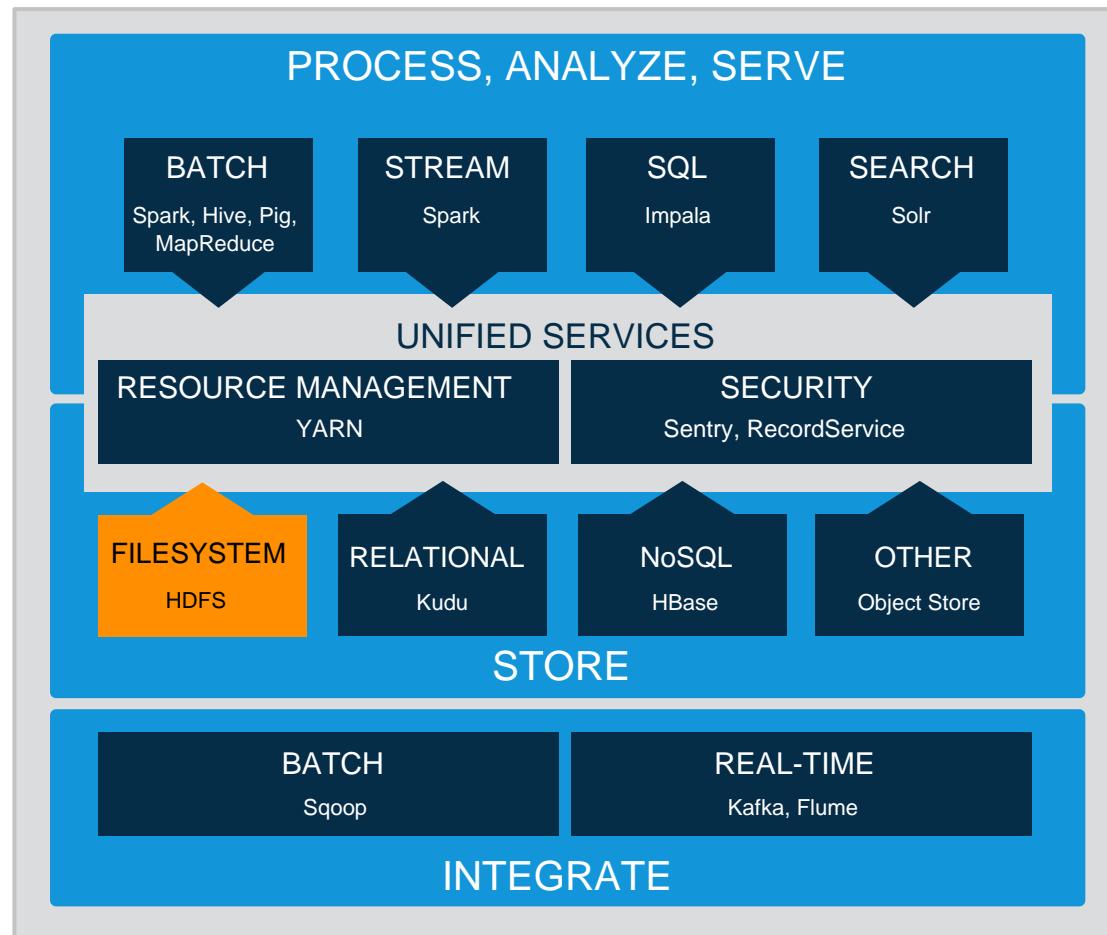
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- **Data Storage: HDFS**
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

HDFS: Hadoop Distributed File System

- HDFS is one of the storage layer options for Hadoop
- Provides inexpensive and reliable storage for massive amounts of data
- Other Hadoop components work with data in HDFS
 - Such as MapReduce and Apache Hive, Impala, and Spark

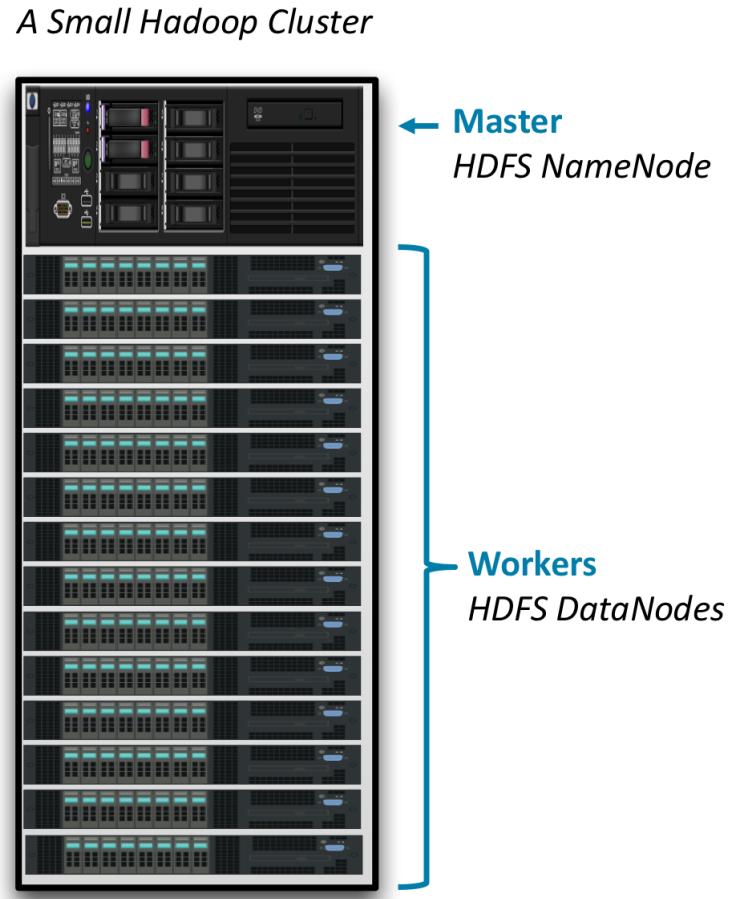


HDFS Characteristics

- **Optimized for sequential access to a relatively small number of large files**
 - Each file is likely to be 100MB or larger
 - Multi-gigabyte files are typical
- **In some ways, HDFS is similar to a UNIX filesystem**
 - Hierarchical: Everything descends from a root directory
 - Uses UNIX-style paths (such as `/sales/rpt/asia.txt`)
 - UNIX-style file ownership and permissions
- **There are also some major deviations from UNIX**
 - No concept of a current directory
 - Cannot modify files once written
 - Must use Hadoop-specific utilities or custom code to access HDFS

HDFS Architecture

- Hadoop has a master/worker architecture
- HDFS master daemon: NameNode
 - Manages namespace and metadata
 - Monitors worker nodes
- HDFS worker daemon: DataNode
 - Reads and writes the actual data



Accessing HDFS Using the Command Line

- **HDFS is not a general-purpose filesystem**
 - Not built into the OS, so only specialized tools can access it
 - End users typically access HDFS using the `hdfs dfs` command
- **Example: Display the contents of the `/user/fred/sales.txt` file**

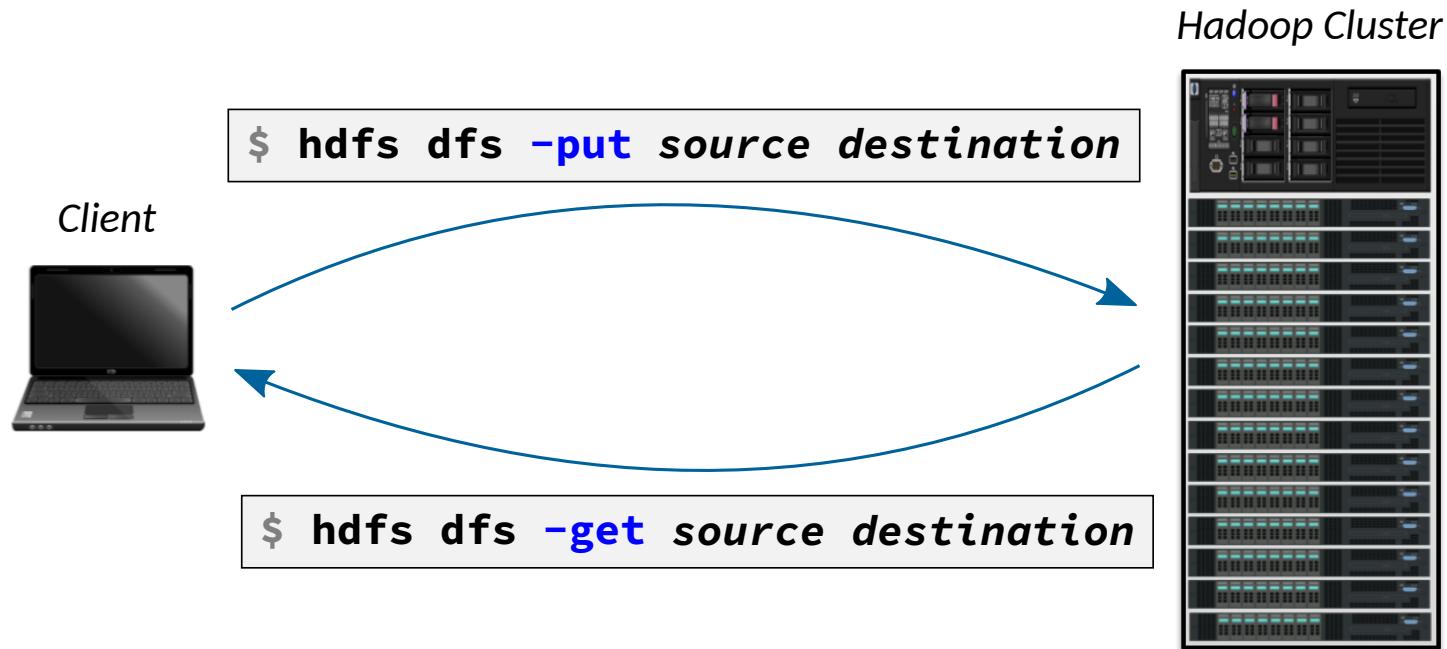
```
$ hdfs dfs -cat /user/fred/sales.txt
```

- **Example: Create a directory (below the root) called reports**

```
$ hdfs dfs -mkdir /reports
```

Copying Local Data to and from HDFS

- Remember that HDFS is distinct from your local filesystem
 - Use `hdfs dfs -put` to copy local files to HDFS
 - Use `hdfs dfs -get` to fetch a local copy of a file from HDFS



More hdfs dfs Command Examples

- Copy */localdir/input.txt* from local disk to user's directory in HDFS

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

- This copies the file to */user/username/input.txt*
- Supply a directory or filename to change the destination default

- Get a directory listing of the HDFS root directory

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

- Delete the file */reports/sales.txt*

```
$ hdfs dfs -rm /reports/sales.txt
```

Using the Hue HDFS File Manager

- Hue is a web interface for Hadoop
 - Hadoop User Experience
- Hue includes an application for browsing and managing files in HDFS
 - To use Hue, browse to `http://hue_server:8888/hue/home`

The screenshot shows the Hue HDFS File Manager interface. On the left, there's a sidebar with a 'Toggle Menu' button (circled in red), a 'File Browser' section (with a 'Browse Files' button circled in blue), and a 'Upload Files' button. The main area is titled 'File Browser' with a 'Home' link and a path '/analyst' (circled in red). It features a search bar, an 'Actions' dropdown, and buttons for 'Move to trash' and 'Upload'. A 'Manage Files' button is also present. The central part of the screen displays a table of file and directory listings:

	Name	Size	User	Group	Permissions	Date
<input type="checkbox"/>	tf		hdfs	supergroup	drwxr-xr-x	November 02, 2018 08:54 AM
<input type="checkbox"/>	.		training	supergroup	drwxr-xr-x	November 06, 2018 09:53 AM
<input type="checkbox"/>	ad_data1		training	supergroup	drwxr-xr-x	November 06, 2018 09:53 AM
<input type="checkbox"/>	ad_data1.txt	30.1 MB	training	supergroup	-rw-r--r--	November 06, 2018 09:53 AM
<input type="checkbox"/>	dualcore		training	supergroup	drwxrwxrwx	November 02, 2018 11:16 AM

At the bottom, there are buttons for 'Show 45 of 3 items', 'Page 1 of 1', and navigation arrows. A note at the bottom left says 'Click or Drop files here'.

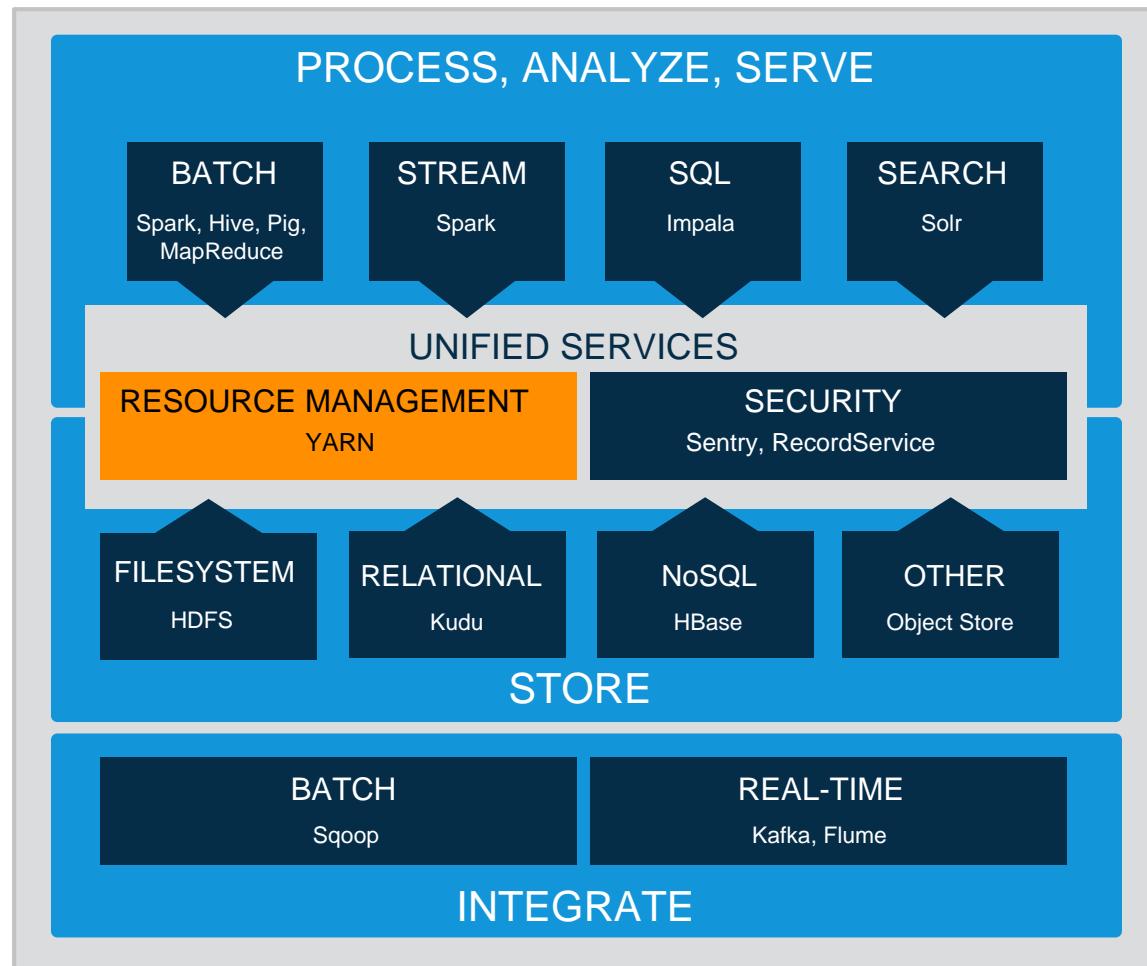
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- **Distributed Data Processing: YARN, MapReduce, and Spark**
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

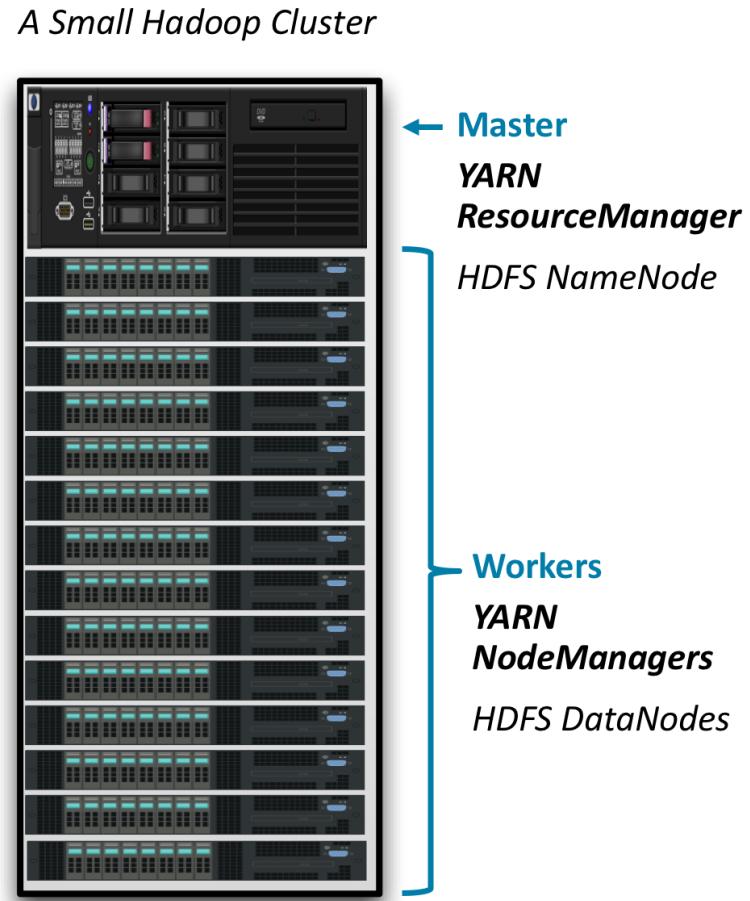
Workload Management: YARN

- Many different Hadoop tools may be running jobs on the cluster at the same time
- Distributing and monitoring work across the cluster requires workload management
- YARN (Yet Another Resource Negotiator) allocates and monitors cluster resources



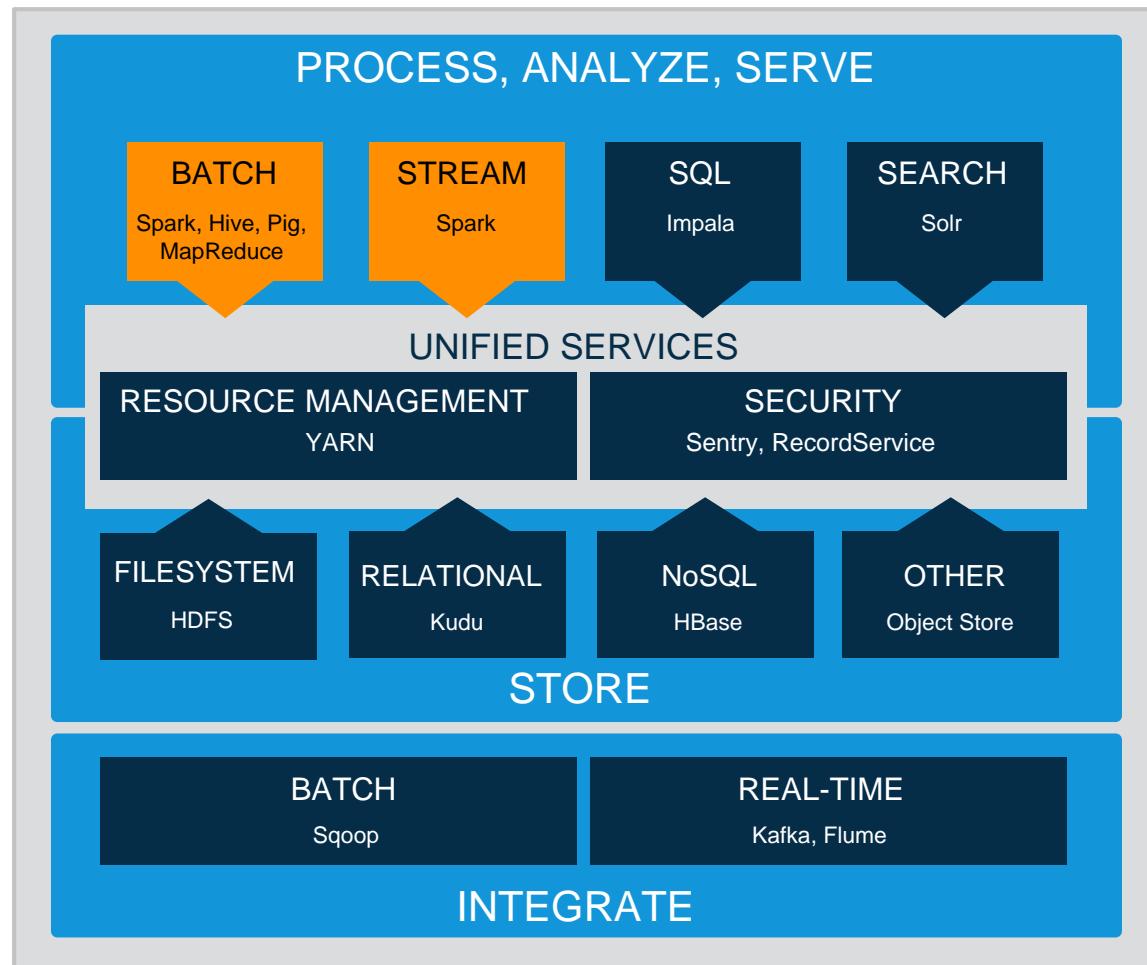
Hadoop Cluster Architecture

- **Master/Worker Architecture**
 - Like HDFS, YARN runs on a master node and worker nodes
- **YARN master daemon: ResourceManager**
 - Accepts jobs, tracks resources, monitors and distributes work
- **YARN worker daemon: NodeManager**
 - Launches tasks to run on worker nodes, reports status back to master
- **HDFS and YARN are colocated**
 - Worker nodes run both HDFS and YARN on the same machines



General Data Processing

- Hadoop includes two general data processing engines
 - MapReduce
 - Spark
- Higher-level tools like Hive use these engines to process data
- Developers can use the MapReduce or Spark APIs to write custom data processing code



Hadoop MapReduce

- **MapReduce is the original processing engine for Hadoop**
 - Still a commonly used general data processing engine
- **Based on the “map-reduce” programming model**
 - A style of processing data popularized by Google
- **Provides a set of programming libraries**
 - Primarily supports Java
 - Streaming MapReduce provides (limited) support for scripting languages such as Python
- **Benefits of MapReduce**
 - Simplicity
 - Flexibility
 - Scalability

Apache Spark

- **Spark is the next-generation general data processing engine**
- **Builds on the same “map-reduce” programming model as MapReduce**
- **Originally developed at AMPLab at the University of California, Berkeley**
- **Supports Scala, Java, Python, and R**
- **Has the same benefits as MapReduce, plus...**
 - Improved performance using in-memory processing
 - Higher-level programming model to speed development

Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- **Data Processing and Analysis: Hive and Impala**
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

Data Processing and Analysis with Hadoop (1)

- **Hadoop MapReduce and Spark are powerful data processing engines but...**
 - Hard to master
 - Require programming skills
 - Slow to develop, hard to maintain
- **Hadoop includes several other tools for data processing and analysis**
 - Tools for data analysts, not programmers

Apache Hive

- **Hive is an abstraction on top of Hadoop**
 - Reduces development time
 - Uses a SQL-like language called HiveQL

```
SELECT customers.cust_id, SUM(cost) AS total
  FROM customers
  JOIN orders
    ON (customers.cust_id =
orders.cust_id)
 GROUP BY customers.cust_id
 ORDER BY total DESC;
```



- **A Hive Server runs on a master node**
 - Turns HiveQL queries into MapReduce or Spark jobs
 - Submits those jobs to the cluster

Apache Impala

- **Impala is a massively parallel SQL engine that runs on a Hadoop cluster**
 - Inspired by Google's Dremel project
 - Can query data stored in HDFS or HBase tables
- **Uses Impala SQL**
 - Very similar to HiveQL
- **High performance**
 - Typically at least 10 times faster than Hive on MapReduce
 - High-level query language (subset of SQL-92)
- **Developed by Cloudera**
 - Donated to the Apache Software Foundation
 - 100% Apache-licensed open source



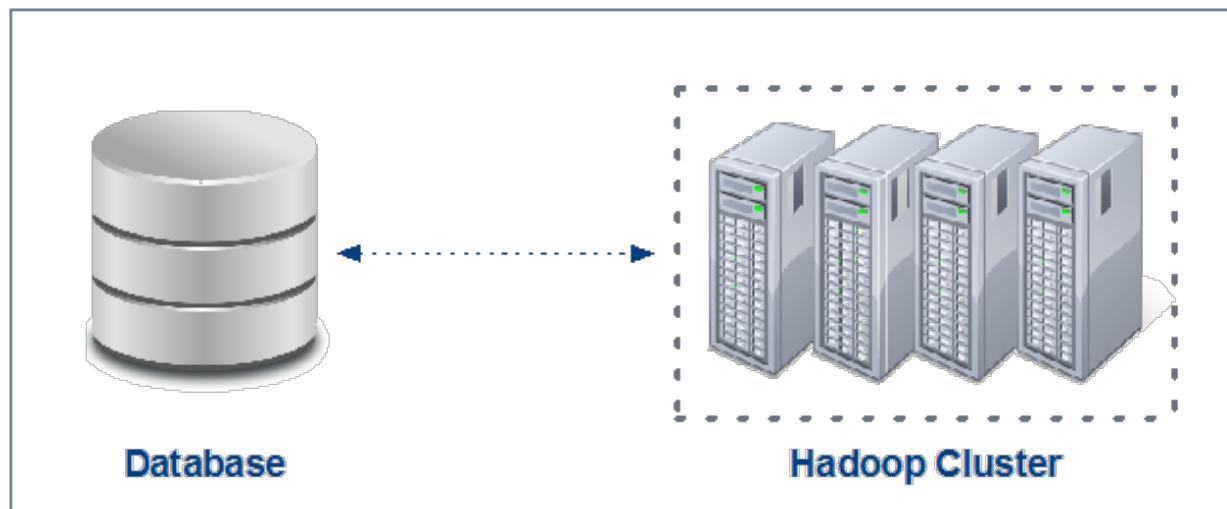
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- **Database Integration: Sqoop**
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

Apache Sqoop

- **Sqoop exchanges data between an RDBMS and Hadoop**
 - The name is a contraction of “SQL to Hadoop”
- **It can import all tables, a single table, or a portion of a table into HDFS**
 - Does this very efficiently using a map-only MapReduce job
 - Result is a directory in HDFS containing comma-delimited text files
- **Sqoop can also export data from HDFS back to the database**



Importing Tables with Sqoop

- This example imports the `customers` table from a MySQL database
 - Will create directory `customers` in the user's home directory in HDFS
 - Directory will contain comma-delimited text files

```
$ sqoop import \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table customers
```

- Adding the `--direct` parameter may offer better performance
 - Uses a database-specific direct connector instead of generic JDBC
 - This option is not compatible with all databases
- Some string columns may require a special setting:
-`Dorg.apache.sqoop.splitter.allow_text_splitter=true`
 - If primary key column is string type
 - As of Sqoop 1.4.7/C6 (check if using an older version)
 - If splitting a text column

Specifying Import Directory with Sqoop

- Use `--target-dir` to specify the import directory

```
$ sqoop import \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table customers \
  --target-dir /mydata/customers
```

- Or use `--warehouse-dir` to specify the parent directory

```
$ sqoop import \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table customers \
  --warehouse-dir /mydata
```

- Both commands create `/mydata/customers` directory in HDFS

Importing an Entire Database with Sqoop

- Import all tables from the database (fields will be tab-delimited)

```
$ sqoop import-all-tables \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --warehouse-dir /mydata \
  --fields-terminated-by '\t'
```

Importing Partial Tables with Sqoop

- Import only specified *columns* from products table

```
$ sqoop import \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table products \
  --warehouse-dir /mydata \
  --columns "prod_id,name,price"
```

- Import only matching *rows* from products table

```
$ sqoop import \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table products \
  --warehouse-dir /mydata \
  --where "price >= 1000"
```

Incremental Imports with Sqoop

- What if new records are added to the database?
 - Could re-import all records, but this is inefficient
- Sqoop's incremental append mode imports only new records
 - Based on value of last record in specified column

```
$ sqoop import \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table orders \
  --warehouse-dir /mydata \
  --incremental append \
  --check-column order_id \
  --last-value 6713821
```

Handling Modifications with Incremental Imports

- What if existing records are also modified in the database?
 - Incremental append mode doesn't handle this
- Sqoop's `lastmodified` append mode adds *and* updates records
 - Caveat: You must maintain a timestamp column in your table

```
$ sqoop import \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table shipments \
  --warehouse-dir /mydata \
  --incremental lastmodified \
  --check-column last_update_date \
  --last-value "2016-12-21 05:36:59"
```

Exporting Data from Hadoop to RDBMS with Sqoop

- We have seen several ways to pull records from an RDBMS into Hadoop
 - It is sometimes also helpful to *push* data in Hadoop back to an RDBMS
- Sqoop supports this with **export**
 - The target table must already exist in the RDBMS

```
$ sqoop export \
  --connect jdbc:mysql://localhost/company \
  --username jdoe --password bigsecret \
  --table product_recommendations \
  --export-dir /mydata/recommender_output
```

Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- **Other Hadoop Data Tools**
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

Apache HBase

- **HBase is “the Hadoop database”**
- **Can store massive amounts of data**
 - Gigabytes, terabytes, and even petabytes of data in a table
 - Tables can have many thousands of columns
- **Scales to provide very high write throughput**
 - Hundreds of thousands of inserts per second
- **Fairly primitive when compared to an RDBMS**
 - NoSQL : There is no high-level query language
 - Use API to scan/get/put values based on keys



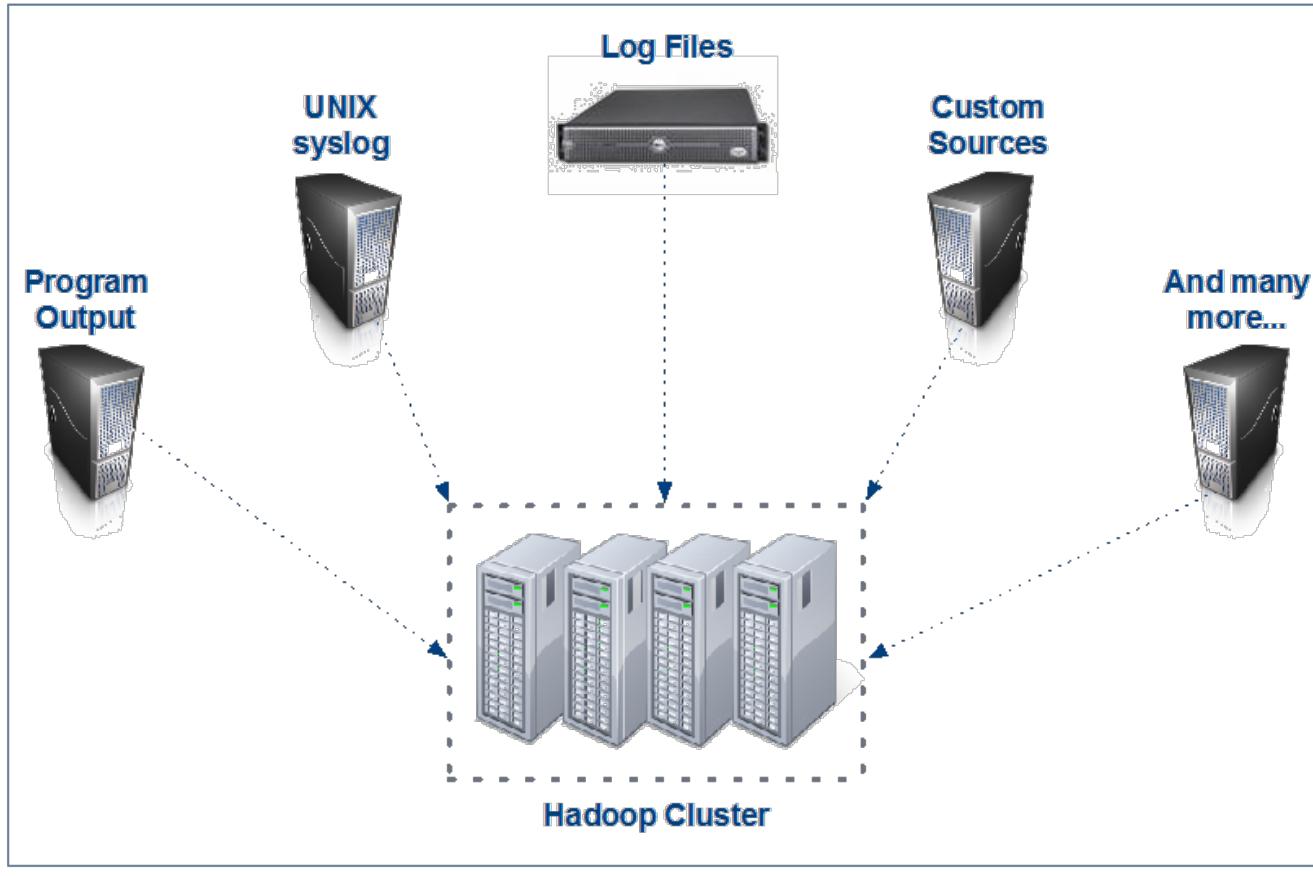
Apache Kudu

- Distributed columnar (key-value) storage for structured data
- Allows random access and updating data (unlike HDFS)
- Supports SQL-based analytics
- Works directly on native file system; is not built on HDFS
- Integrates with Spark, MapReduce, and Apache Impala
- Created at Cloudera, donated to Apache Software Foundation

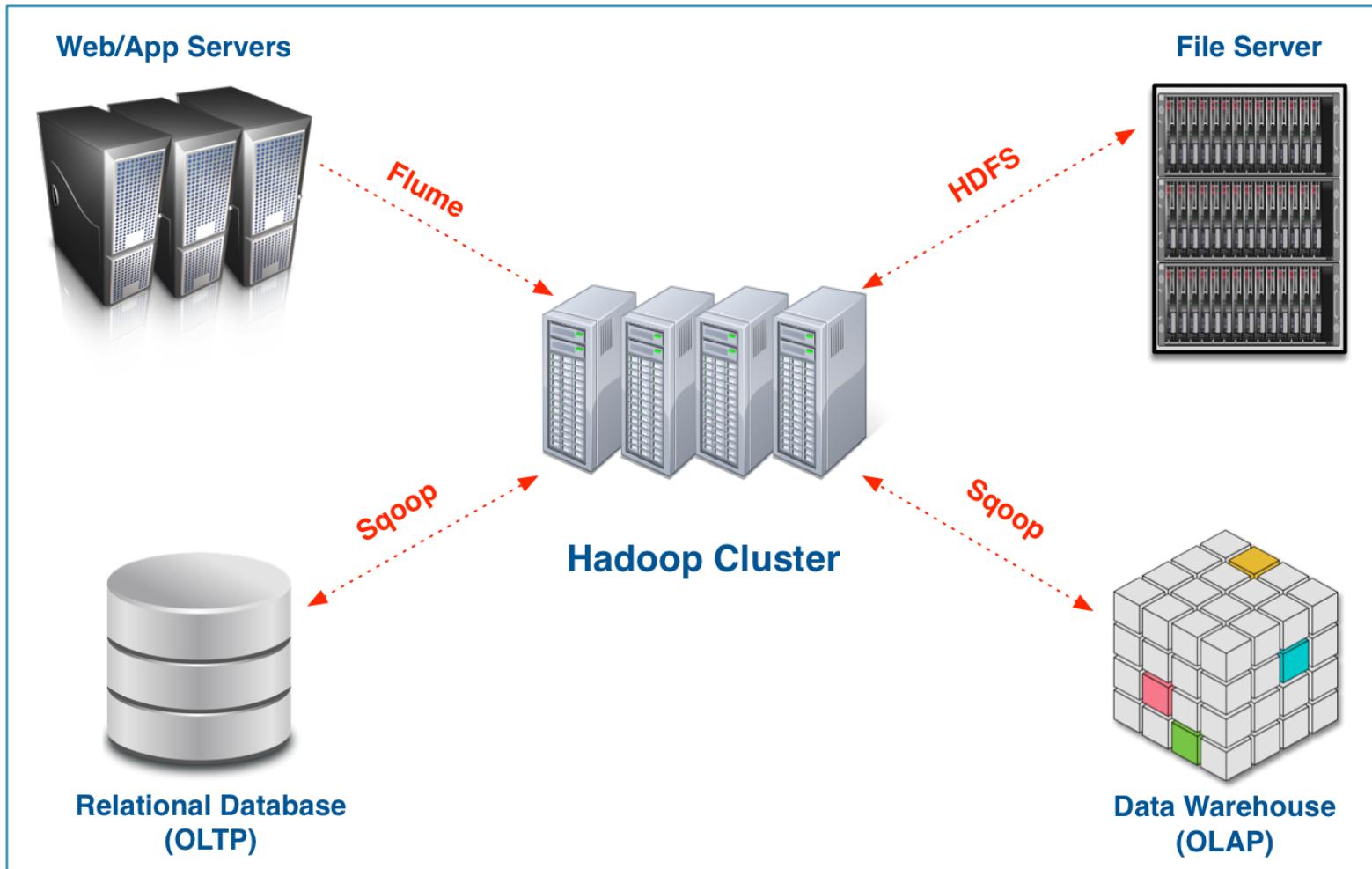


Apache Flume

- Flume imports data into HDFS *as it is being generated* by various sources



Recap: Data Center Integration



Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- Other Hadoop Data Tools
- **Exercise Scenario Explanation**
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

Hands-On Exercises: Scenario Explanation

- **Exercises throughout the course will reinforce the topics being discussed**
 - Exercises simulate common tasks that use the tools you will learn about in class
 - Most exercises depend on data generated in earlier exercises
- **Scenario: Dualcore Inc. is a leading electronics retailer**
 - More than 1,000 brick-and-mortar stores
 - Dualcore also has a thriving e-commerce website
- **Dualcore has hired you to help find value in its data; you will**
 - Process and analyze data from internal and external sources
 - Identify opportunities to increase revenue
 - Find new ways to reduce costs
 - Help other departments achieve their goals

Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- **Essential Points**
- Hands-On Exercise: Data Ingest with Hadoop Tools

Essential Points

- We are generating more data—and faster—than ever before
- Most of this data maps poorly to structured relational tables
- The ability to store and process this data can yield valuable insight
- Hadoop offers scalable data storage and processing
- There are lots of tools in the Hadoop ecosystem that help you to integrate Hadoop with other systems, manage complex jobs, and ease analysis

Bibliography

The following offer more information on topics discussed in this chapter

- **10 Common Hadoop-able Problems (recorded presentation)**
 - <http://tiny.cloudera.com/dac02a>
- **Apache Sqoop Cookbook (O'Reilly book)**
 - <http://tiny.cloudera.com/sqoopbook>
- **HDFS Commands Guide**
 - <http://tiny.cloudera.com/hdfscommands>
- **Hadoop: The Definitive Guide, 4th Edition (O'Reilly book)**
 - <http://tiny.cloudera.com/hadooptdg4>
- **Sqoop User Guide**
 - <http://tiny.cloudera.com/sqoopuser>

Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Hive and Impala
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- **Hands-On Exercise: Data Ingest with Hadoop Tools**

About the Training Virtual Machine

- During this course, you will perform numerous hands-on exercises using the provided hands-on environment
 - A virtual machine (VM) running Linux
- This environment has Hadoop installed in *pseudo-distributed mode*
 - A cluster comprised of a single node
 - Typically used for testing code before deploying to a large cluster

Hands-On Exercise: Data Ingest with Hadoop Tools

- In this exercise, you will gain practice adding data from the local filesystem and from a relational database server to HDFS
 - You will analyze this data in subsequent exercises
 - Please refer to the Hands-On Exercise Manual for instructions



Introduction to Apache Hive and Impala

Chapter 3

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala**
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Introduction to Apache Hive and Impala

In this chapter, you will learn

- **What Apache Hive and Impala's main purposes in a big data system are**
- **How data and metadata are stored**
- **How the features of RDBMSs are different from those of Hive and Impala**

Chapter Topics

Introduction to Apache Hive and Impala

- **What Is Hive?**
- What Is Impala?
- Why Use Hive and Impala?
- Schema and Data Storage
- Comparing Hive and Impala to Traditional Databases
- Use Cases
- Essential Points

What Is Apache Hive?

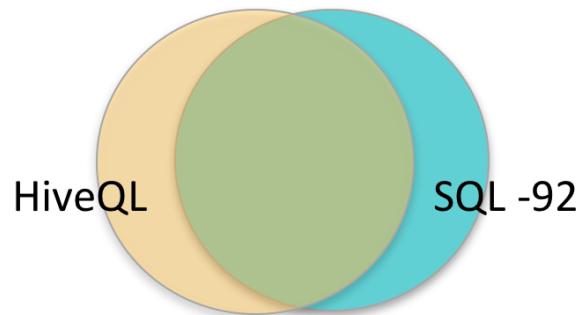
- **Hive is data warehouse infrastructure for distributed systems like Apache Hadoop**
 - Alternative to writing low-level MapReduce code
 - Uses a SQL-like language called HiveQL
 - Generates jobs that run on the distributed system
 - Originally developed by Facebook
 - Now an open source Apache project



HiveQL

- HiveQL implements a subset of SQL-92
 - Plus a few extensions found in MySQL and Oracle SQL dialects

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



Hive Overview

- Hive turns HiveQL queries into data processing jobs
- Then it submits those jobs to the data processing engine (MapReduce or Spark) to execute on the cluster

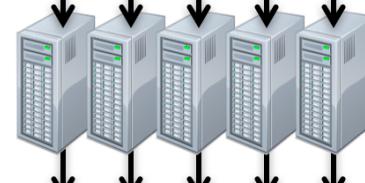
```
SELECT zipcode, SUM(cost) AS total  
FROM customers  
JOIN orders  
ON (customers.cust_id = orders.cust_id)  
WHERE zipcode LIKE '63%'  
GROUP BY zipcode  
ORDER BY total DESC;
```

- Parse HiveQL
- Make optimizations
- Plan execution
- Submit job(s) to cluster
- Monitor progress



Data Processing Engine

Hadoop Cluster



HDFS

Chapter Topics

Introduction to Apache Hive and Impala

- What Is Hive?
- **What Is Impala?**
- Why Use Hive and Impala?
- Schema and Data Storage
- Comparing Hive and Impala to Traditional Databases
- Use Cases
- Essential Points

What Is Apache Impala?

- **Impala is a high-performance SQL engine for vast amounts of data**
 - Massively parallel processing (MPP)
 - Inspired by Google's Dremel project
 - Query latency measured in milliseconds
- **Impala runs on distributed systems (like Hadoop)**
 - Can query data stored in HDFS, Apache HBase tables, Apache Kudu, S3
 - Reads and writes data in common Hadoop file formats
- **Developed by Cloudera**
 - Donated to the Apache Software Foundation
 - 100% open source, released under the Apache software license

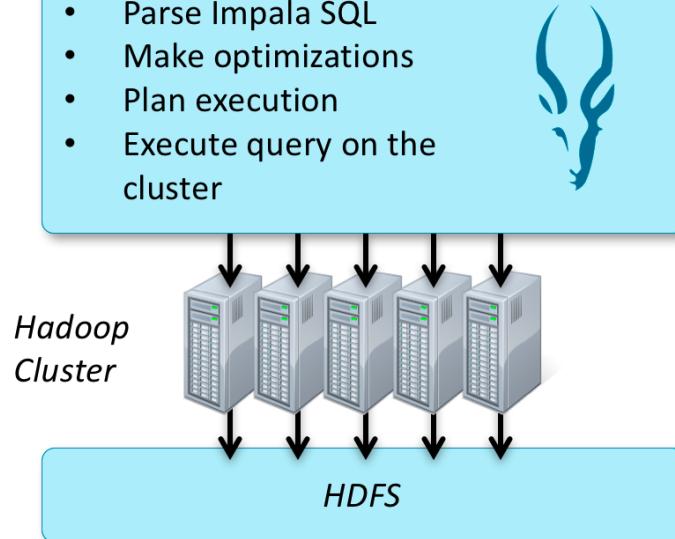


Impala Overview

- Executes Impala SQL queries directly on the cluster
- Does not rely on a general-purpose data processing engine like MapReduce or Spark
- Highly optimized for queries
- Almost always at least five times faster than either Hive or Pig
 - Often 20 times faster or more

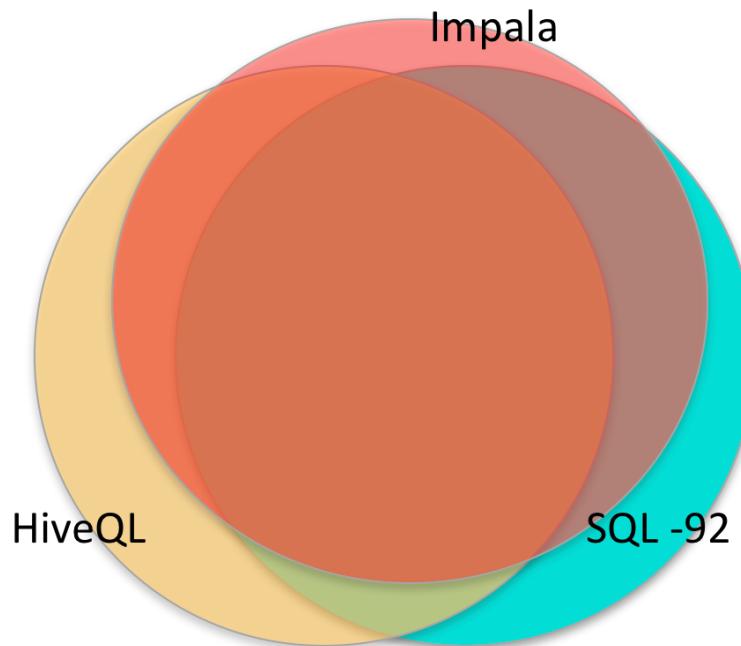
```
SELECT zipcode, SUM(cost) AS total  
  FROM customers  
  JOIN orders  
    ON (customers.cust_id = orders.cust_id)  
 WHERE zipcode LIKE '63%'  
 GROUP BY zipcode  
 ORDER BY total DESC;
```

- Parse Impala SQL
- Make optimizations
- Plan execution
- Execute query on the cluster



Impala Query Language

- Impala supports a subset of HiveQL
 - Plus a few additional commands



Chapter Topics

Introduction to Apache Hive and Impala

- What Is Hive?
- What Is Impala?
- **Why Use Hive and Impala?**
- Schema and Data Storage
- Comparing Hive and Impala to Traditional Databases
- Use Cases
- Essential Points

Why Use Hive or Impala?

- **More productive than writing MapReduce directly**
 - Five lines of HiveQL or Impala SQL might be equivalent to 200 lines or more of Java
- **Brings large-scale data analysis to a broader audience**
 - No software development experience required
 - Leverage existing knowledge of SQL
- **Extensible through custom user-defined components**
- **Many business intelligence (BI) tools support Hive and Impala**

Comparing Hive and Impala

- **Hive and Impala are different tools, but are closely related**
 - They use very similar variants of SQL
 - They share the same data warehouse and metadata storage
 - They are often used together
- **This course first focuses on areas in common between Hive and Impala**
 - HiveQL and Impala SQL are very similar
 - Differences will be noted
- **Later, the course emphasizes some areas in which they differ**

Chapter Topics

Introduction to Apache Hive and Impala

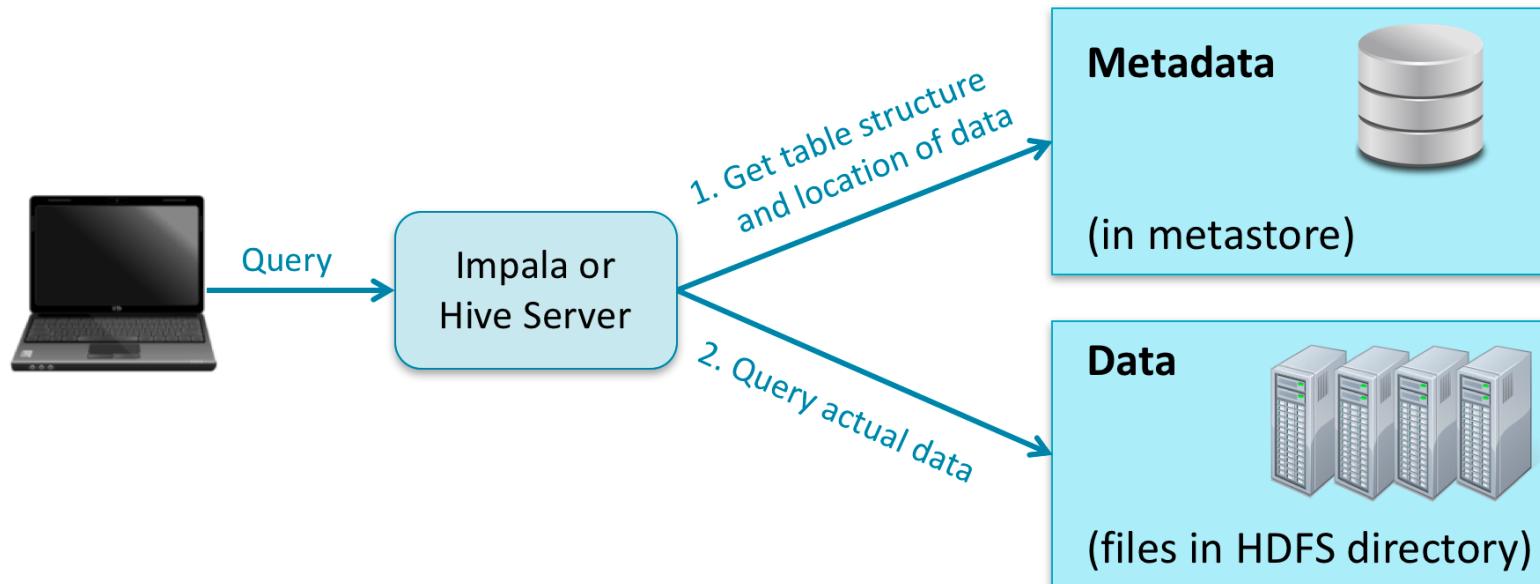
- What Is Hive?
- What Is Impala?
- Why Use Hive and Impala?
- **Schema and Data Storage**
- Comparing Hive and Impala to Traditional Databases
- Use Cases
- Essential Points

How Hive and Impala Query Data (1)

- **Queries operate on tables, just as queries do in an RDBMS**
- **A table has two components:**
 - Metadata
 - Specifies structure and location of the data
 - Defined when table is created
 - Stored in the *metastore*, which is contained in an RDBMS
 - Data
 - Typically in an HDFS directory with one or more files
 - Default path: `/user/hive/warehouse/tablename`
 - Can be in any of several formats for storage and retrieval
- **Hive and Impala work with the same tables**
 - Same metastore for metadata
 - Same directory in HDFS for data

How Hive and Impala Query Data (2)

- Hive and Impala use the metastore to determine data format and location
- The query itself operates on data stored in a filesystem (typically HDFS)



Chapter Topics

Introduction to Apache Hive and Impala

- What Is Hive?
- What Is Impala?
- Why Use Hive and Impala?
- Schema and Data Storage
- **Comparing Hive and Impala to Traditional Databases**
- Use Cases
- Essential Points

Your Cluster Is Not a Database Server

- **Client-server database management systems have many strengths**
 - Have very fast response time
 - Include support for transactions
 - Allow modification of existing records
 - Can serve thousands of simultaneous clients
- **Hive and Impala do not turn your cluster into an RDBMS**
 - No support for updating and deleting records
 - No transaction support
 - No referential integrity

Comparing Hive and Impala to a Relational Database

Feature	RDBMS	Hive	Impala
Query language	SQL (full)	SQL (subset)	SQL (subset)
Update individual records	Yes	No*	Not†
Delete individual records	Yes	No*	Not†
Transactions	Yes	No*	No
Index support	Extensive	Limited	No
Latency	Very low	High	Low
Data size	Terabytes	Petabytes	Petabytes
Storage cost	Very high	Very low	Very low

* Hive now has limited, experimental support for UPDATE, DELETE, and transactions.

† Using Impala, you can update or delete individual rows in Kudu tables.

Query Fault Tolerance

- **Queries in both Hive and Impala are distributed across nodes**
- **Hive answers queries by running MapReduce or Spark jobs**
 - Takes advantage of the underlying engine's fault tolerance
 - If a node fails during a query, MapReduce or Spark runs the task on another node
- **Impala has its own execution engine**
 - Currently lacks fault tolerance
 - If a node fails during a query, the query will fail
 - Re-run the query (typically still faster than using Hive)

Chapter Topics

Introduction to Apache Hive and Impala

- What Is Hive?
- What Is Impala?
- Why Use Hive and Impala?
- Schema and Data Storage
- Comparing Hive and Impala to Traditional Databases
- **Use Cases**
- Essential Points

Use Case: Log File Analytics

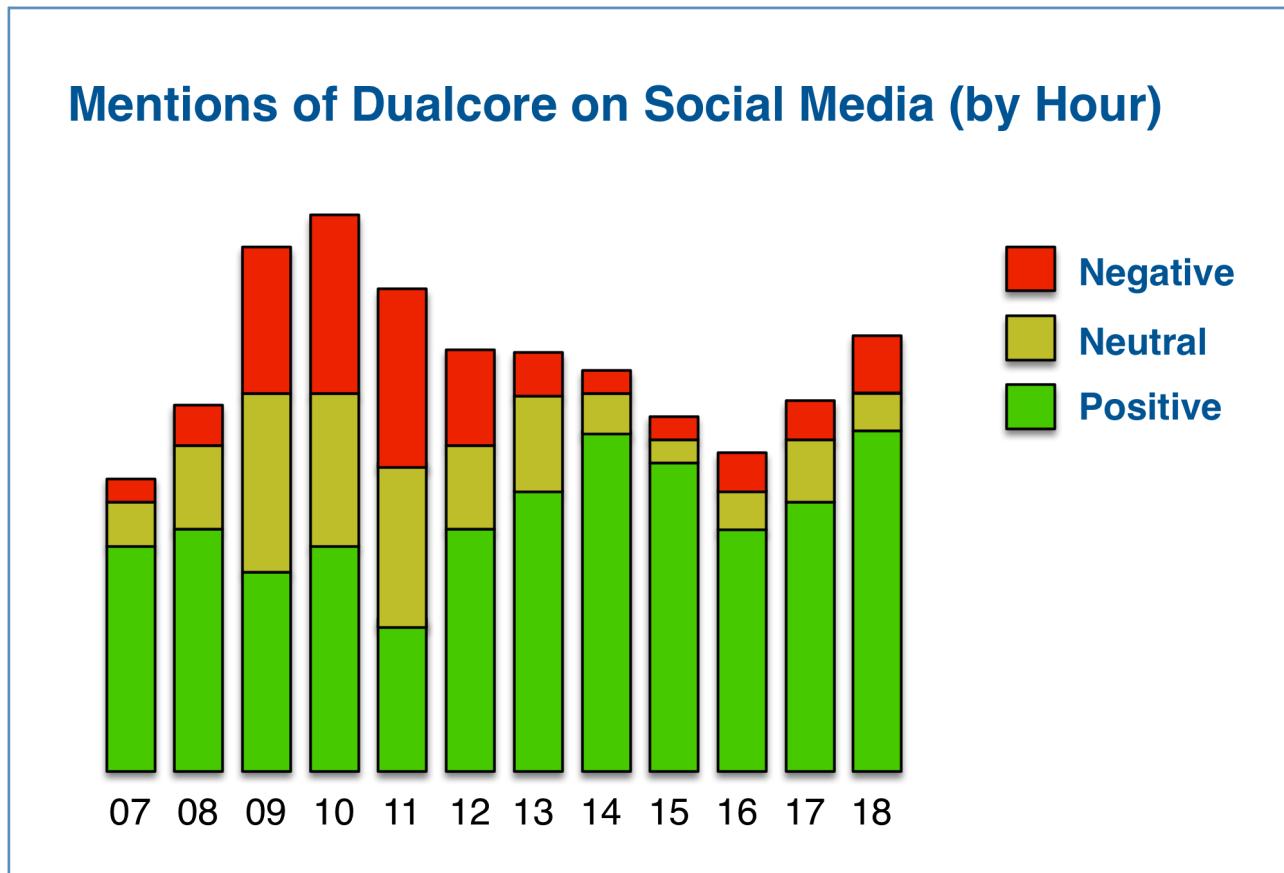
- Server log files are an important source of data
- Hive and Impala allow you to treat a directory of log files like a table
 - Allows SQL-like queries against raw data

Dualcore Inc. Public Website (June 1 - 8)

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

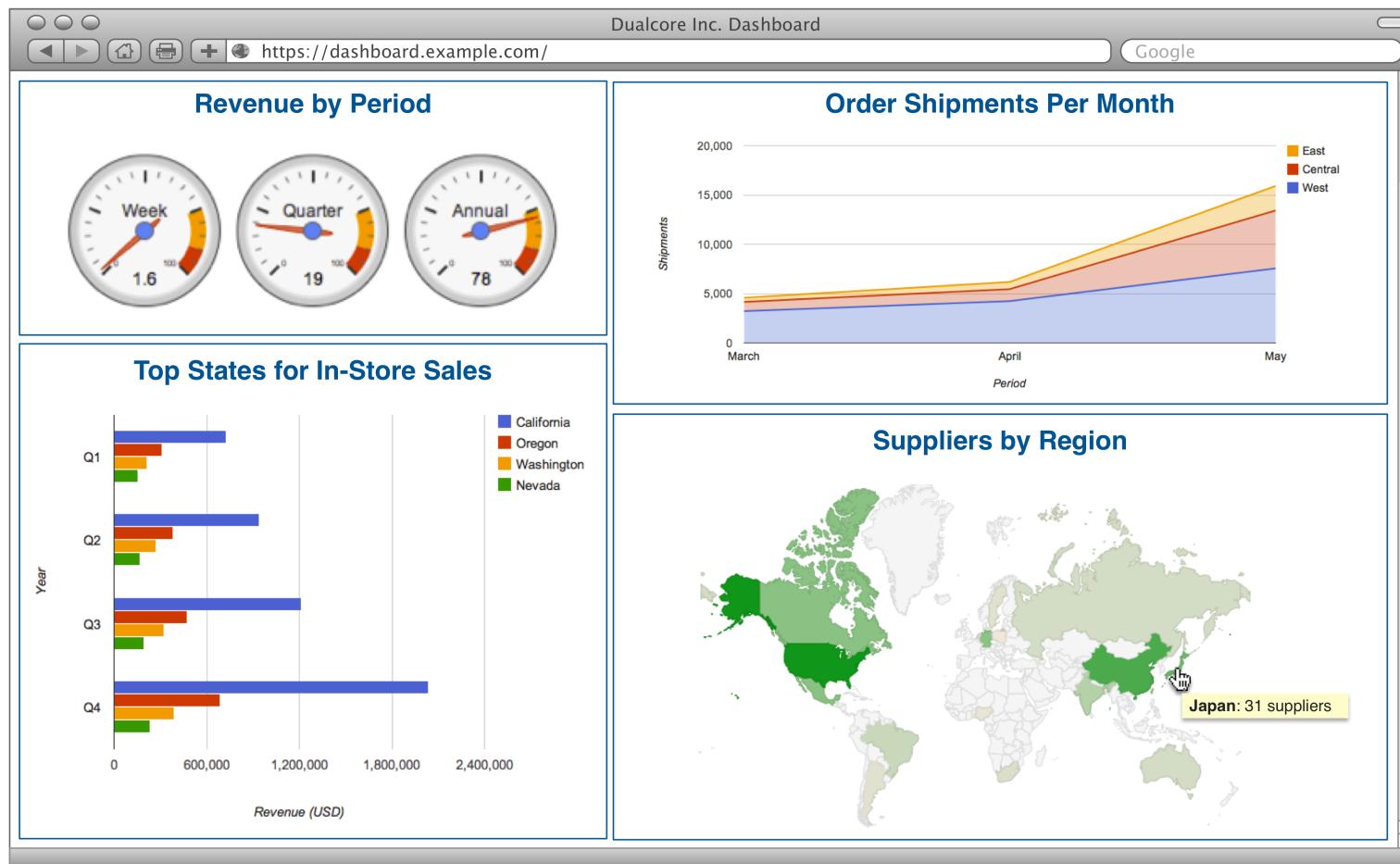
Use Case: Sentiment Analysis

- Many organizations use Hive or Impala to analyze social media



Use Case: Business Intelligence

- Many leading business intelligence tools support Hive and Impala



Chapter Topics

Introduction to Apache Hive and Impala

- What Is Hive?
- What Is Impala?
- Why Use Hive and Impala?
- Schema and Data Storage
- Comparing Hive and Impala to Traditional Databases
- Use Cases
- Essential Points

Essential Points

- **Apache Hive and Apache Impala are tools for performing SQL queries on big data in distributed systems**
 - Hive generates and runs MapReduce or Spark jobs
 - Impala executes queries directly on the distributed system
 - Uses a very fast specialized SQL engine, not MapReduce or Spark
- **Hive and Impala store data and metadata separately**
 - Metadata (information about the table) is kept in the metastore
 - Data is kept in files in a storage system (like HDFS, Kudu, or S3)
- **Hive and Impala differ from RDBMSs in**
 - Transactions and updating individual records
 - Hive has limited, experimental support for these; Impala does not
 - Index support: limited with Hive and none with Impala
 - Latency: very low for RDBMSs, low for Impala, high for Hive
 - Storage: Hive and Impala can handle much more at a low cost

Bibliography

The following offer more information on topics discussed in this chapter

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



Querying with Apache Hive and Impala

Chapter 4

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- **Querying with Apache Hive and Impala**
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Querying with Apache Hive and Impala

In this chapter, you will learn

- What key similarities and differences between SELECT statements for RDBMS and for Hive and Impala are
- What data types can be used for Hive and Impala
- Why you might want to use a Hive or Impala shell
- How to start shells for Hive and Impala
- How to execute queries in shells, from the command line, and in Hue

Chapter Topics

Querying with Apache Hive and Impala

- **Databases and Tables**
- Basic Hive and Impala Query Language Syntax
- Data Types
- Using Hue to Execute Queries
- Using Beeline (Hive's Shell)
- Using the Impala Shell
- Essential Points
- Hands-On Exercise: Running Queries from Shells, Scripts, and Hue

Tables

- **Data for Hive and Impala tables is stored on the filesystem (such as HDFS)**
 - Each table maps to a single directory
- **A table's directory may contain multiple files**
 - Typically delimited text files, but many formats are supported
 - Subdirectories are not allowed
 - Exception: partitioned tables
- **The metastore gives context to this data**
 - Helps map raw data in filesystem to named columns of specific types

Databases

- **Each table belongs to a specific database**
- **Early versions of Hive supported only a single database**
 - It placed all tables in the same database (named `default`)
 - This is still the default behavior
- **Hive and Impala support multiple databases**
 - Helpful for organization and authorization

Exploring Databases and Tables (1)

- Use **SHOW DATABASES** to list the databases in the metastore

```
> SHOW DATABASES;
```

- Use **DESCRIBE DATABASE** to display metadata about a database

```
> DESCRIBE DATABASE default;
```

Exploring Databases and Tables (2)

- Hive and Impala both connect to the default database by default
- Switch between databases with the USE command

(new session of Hive or Impala)

```
> SELECT * FROM customers; ①
> USE sales;
> SELECT * FROM customers; ②
```

- ① Queries table in the **default** database
- ② Queries table in the **sales** database

Exploring Databases and Tables (3)

- Use SHOW TABLES to list the tables in a database

```
> USE accounting;  
> SHOW TABLES;  
+-----+  
| tab_name |  
+-----+  
| invoices |  
| taxes   |  
+-----+
```

Shows tables in the current database (accounting)

```
> SHOW TABLES IN sales;  
+-----+  
| tab_name |  
+-----+  
| customers |  
| prospects |  
+-----+
```

Shows tables in the sales database

Exploring Databases and Tables (4)

- The **DESCRIBE** command displays basic structure for a table

```
> DESCRIBE orders;
+-----+-----+-----+
| name      | type       | comment |
+-----+-----+-----+
| order_id   | int        |          |
| cust_id    | int        |          |
| order_date  | timestamp  |          |
+-----+-----+-----+
```

- Use the **DESCRIBE FORMATTED** command for detailed information
 - Input and output formats, file locations, and other information

Chapter Topics

Querying with Apache Hive and Impala

- Databases and Tables
- **Basic Hive and Impala Query Language Syntax**
- Data Types
- Using Hue to Execute Queries
- Using Beeline (Hive's Shell)
- Using the Impala Shell
- Essential Points
- Hands-On Exercise: Running Queries from Shells, Scripts, and Hue

An Introduction to HiveQL and Impala SQL

- **HiveQL is Hive's query language**
 - Based on a subset of SQL-92, plus Hive-specific extensions
- **Impala SQL is very similar to HiveQL**
 - Differences are noted in this course
- **Some differences compared to standard SQL**
 - Some features are not supported
 - Others are only partially implemented

Syntax Basics

- **Keywords** are words that have a particular meaning
 - Examples: SELECT, FROM, WHERE, AS
 - Not case-sensitive, but often capitalized by convention
 - Are *reserved* for their specific use
- **Statements** are terminated by a semicolon
 - May span multiple lines
- **Comments begin with --(double hyphen)**

```
SELECT cust_id, fname, lname
      FROM customers
     WHERE zipcode='60601';      -- downtown Chicago
```

- No multiline comments in Hive
- Impala supports /* */ multiline comment syntax

Identifiers

- ***Identifiers*** are words used to identify a column, table, or database
- **Quoted identifiers** are identifiers enclosed in backquotes (such as `column`)
 - Allows reserved words such as keywords to be used as identifiers
 - Sometimes considered best practice for saved queries (such as in scripts)
 - Protects query from future changes in the list of reserved words
 - Not used in this course for convenience and readability
 - Example:

```
SELECT `select` FROM `from`;
```

Selecting Data from Tables

- The **SELECT** statement retrieves data from tables
 - Can specify an ordered list of individual columns

```
SELECT cust_id, fname, lname FROM customers;
```

- An asterisk matches all columns in the table

```
SELECT * FROM customers;
```

- Use **DISTINCT** to remove duplicates

```
SELECT DISTINCT zipcode FROM customers;
```

Sorting Query Results

- The ORDER BY clause sorts the result set
 - Default order is ascending (same as using ASC keyword)
 - Specify DESC keyword to sort in descending order



```
SELECT brand, name FROM products  
    ORDER BY price DESC;
```

- Hive requires the field(s) you ORDER BY to be included in the SELECT



```
SELECT brand, name, price FROM products  
    ORDER BY price DESC;
```

To sort by an expression, put it in the **SELECT** list and use an alias

```
SELECT brand, name, price - cost AS profit FROM products  
    ORDER BY profit;
```

Limiting Query Results

- The **LIMIT** clause sets the maximum number of rows returned

```
SELECT brand, name, price FROM products LIMIT 10;
```

- Caution: no guarantee regarding which 10 results are returned
 - Use **ORDER BY** with **LIMIT** for top-N queries

```
SELECT brand, name, price FROM products  
ORDER BY price DESC LIMIT 10;
```

Using a WHERE Clause to Filter Results

- WHERE clauses restrict rows to those matching specified criteria
 - String comparisons are case-sensitive

```
SELECT * FROM orders WHERE order_id=1287;
```

```
SELECT * FROM customers WHERE state  
IN ('CA', 'OR', 'WA', 'NV', 'AZ');
```

- You can combine expressions using AND or OR

```
SELECT * FROM customers  
WHERE fname LIKE 'Ann%'  
AND (city='Seattle' OR city='Portland');
```

Table Aliases

- Table aliases can help simplify complex queries
 - Using AS to specify table aliases is also supported

```
SELECT o.order_date, c.fname, c.lname
FROM customers c JOIN orders AS o
ON (c.cust_id = o.cust_id)
WHERE c.zipcode='94306';
```

Subqueries in the FROM Clause

- Hive and Impala support subqueries in the FROM clause
 - The subquery must be named (high_profits in this example)

```
SELECT prod_id, brand, name
  FROM (SELECT *
        FROM products
        WHERE (price - cost) / price > 0.65
        ORDER BY price DESC
        LIMIT 10) high_profits
 WHERE price > 1000
 ORDER BY brand, name;
```

Subqueries in the WHERE Clause

- Hive and Impala allow subqueries in the WHERE clause
 - Use to filter one table based on criteria in another table

```
SELECT cust_id, fname, lname
  FROM customers c
 WHERE state = 'NY'
   AND c.cust_id IN (SELECT cust_id
                      FROM orders
                     WHERE order_id > 6650000);
```

- Support for subqueries differs between Hive and Impala
 - Both support *uncorrelated* subqueries
 - Both support *correlated* subqueries, but Hive's support is limited
 - Impala supports *scalar* subqueries, but Hive does not (as of C6.0)

Chapter Topics

Querying with Apache Hive and Impala

- Databases and Tables
- Basic Hive and Impala Query Language Syntax
- **Data Types**
- Using Hue to Execute Queries
- Using Beeline (Hive's Shell)
- Using the Impala Shell
- Essential Points
- Hands-On Exercise: Running Queries from Shells, Scripts, and Hue

Data Types

- Each column has an associated data type
- Hive and Impala both support more than a dozen types
 - Most are similar to ones found in relational databases
 - Hive and Impala also support certain *complex types*
- Use the DESCRIBE command to see data types for each column in a table

```
> DESCRIBE products;
```

name	type	comment
prod_id	int	
brand	string	
name	string	
price	int	
cost	int	
shipping_wt	int	

Integer Types

- Integer types are appropriate for whole numbers
 - Both positive and negative values are allowed

Name	Description	Example Value
TINYINT	Range: -128 to 127	17
SMALLINT	Range: -32,768 to 32,767	5842
INT	Range: -2,147,483,648 to 2,147,483,647	84127213
BIGINT	Range: ≈ -9.2 quintillion to ≈ 9.2 quintillion	632197432180964

Decimal Types

- **Decimal types are appropriate for numbers that can include fractional parts**
 - Both positive and negative values allowed
 - Use DECIMAL when exact values are required!
 - Blue digits in the examples below are not accurate

Name	Description	Example Value
FLOAT	Decimals	3.141592 <code>7410125732</code>
DOUBLE	More precise decimals	3.141592653589793 <code>1</code>
DECIMAL(p,s)*	Exact precision	3.1415926535897932 using DECIMAL(17,16)

* Parameter p (*precision*) specifies the total number of digits, and s (*scale*) specifies the number of digits after the decimal point.

Character Types

- Character types are used to represent alphanumeric text values

Name	Description	Example Value
STRING	Character sequence	Impala rules!
CHAR(n)	Fixed-length character sequence	Impala rules!_____ using CHAR(16)
VARCHAR(n)	Variable length character sequence (maximum length n)	Impala rul using VARCHAR(10)

Other Simple Types

- There are a few other data types

Name	Description	Example Value
BOOLEAN	True or false	true
TIMESTAMP	Instant in time	2016-06-14 16:51:05
BINARY (Hive-only)	Raw bytes	N/A

Data Type Conversion

- Hive auto-converts a STRING column used in numeric context



```
> SELECT zipcode FROM customers LIMIT 1;  
60601  
> SELECT zipcode + 1.5 FROM customers LIMIT 1;  
60602.5
```

- Impala requires an explicit cast operation for this



```
> SELECT zipcode + 1.5 FROM customers LIMIT 1;  
ERROR: AnalysisException: Arithmetic operation...
```



```
> SELECT cast(zipcode AS FLOAT) + 1.5  
FROM customers LIMIT 1;  
60602.5
```

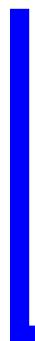
Handling of Out-of-Range Values

- Hive and Impala handle out-of-range numeric values differently
 - Hive returns NULL
 - Impala returns the minimum or maximum value for that type

```
Rashida    29  
Hugo       17  
Abigail   129  
Esma      -999  
Kenji     -999
```

```
CREATE TABLE names_and_ages  
(name STRING,  
 age TINYINT)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY '\t';
```

```
SELECT age FROM names_and_ages;
```



age	age
29	29
17	17
NULL	NULL
NULL	127
NULL	-128

Chapter Topics

Querying with Apache Hive and Impala

- Databases and Tables
- Basic Hive and Impala Query Language Syntax
- Data Types
- **Using Hue to Execute Queries**
- Using Beeline (Hive's Shell)
- Using the Impala Shell
- Essential Points
- Hands-On Exercise: Running Queries from Shells, Scripts, and Hue

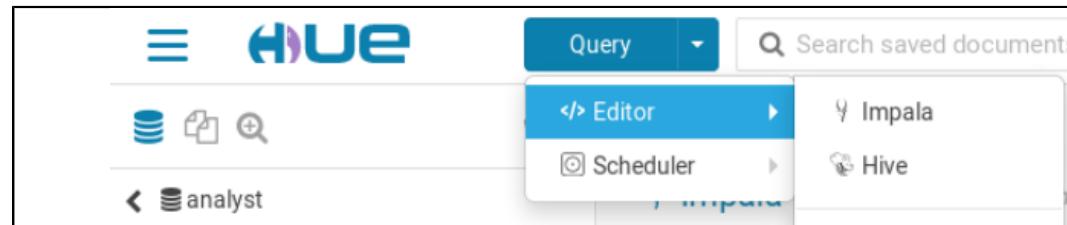
Interacting with Hive and Impala

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

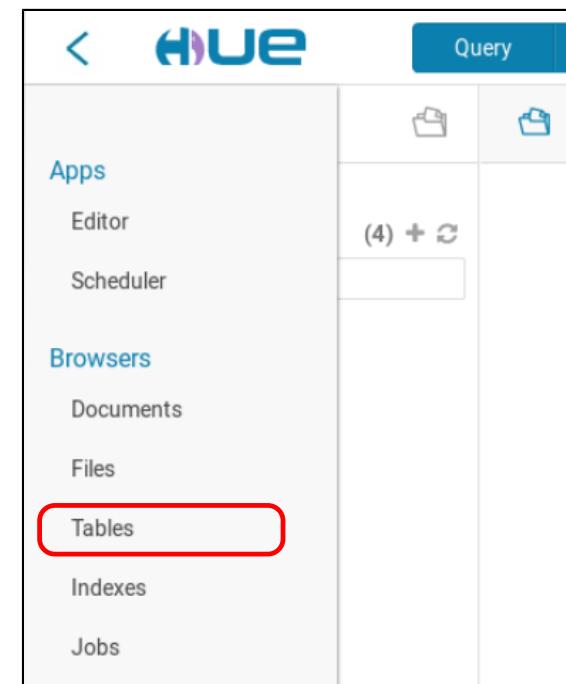
Using Hue with Hive and Impala

You can use Hue to...

- Query data with Hive or Impala



- View and manage tables



The Hue Query Editor: Databases

- The Hive and Impala query editors are nearly identical

Choose a database

Explore schema and sample data

Impala

Tables

Filter...

customer
cust_id (int)
fname (string)
lname (string)
address (string)
city (string)
state (string)
zipcode (string)

order_details

orders

products

0.69s analyst text ?

1| SELECT * FROM customers WHERE state='CA';

Query History Saved Queries

Results (1,024+)

	cust_id	fname	lname	address
1	1000002	Marilyn	Ham	25831 North 25th Stre
2	1000006	Gerard	Franks	356 Turner Street
3	1000010	Mason	Preston	2656 West 13th Street

Tables

Filter...

assistant Functions

analyst.customers

cust_id		int
fname		string
lname		string
address		string
city		string
state		string
zipcode		string

The Hue Query Editor: Queries

The screenshot shows the Hue Query Editor interface. On the left, there's a sidebar with icons for HUE, Tables, and a search bar. The main area is titled "Impala" and shows a query editor with a query history section containing a single query:

```
1| SELECT * FROM customers WHERE state='CA';
```

Below the query editor is a "Results (1,024+)" section displaying a table with columns: cust_id, fname, lname, and address. The first three rows of the table are:

	cust_id	fname	lname	address
1	1000002	Marilyn	Ham	25831 North 25th Stre
2	1000006	Gerard	Franks	356 Turner Street
3	1000010	Mason	Preston	2656 West 13th Street

On the right side, there's an "Assistant" panel with tabs for "Tables" and "Functions". The "Tables" tab is active, showing the schema for the "analyst.customers" table:

analyst.customers		
cust_id		int
fname		string
lname		string
address		string
city		string
state		string
zipcode		string

Annotations are present in the interface:

- A callout box labeled "Execute queries" points to the play button icon in the query editor.
- A callout box labeled "Enter, edit, and save queries" points to the text input field in the query editor.
- A callout box labeled "Inspect tables used in the query" points to the "Tables" tab in the Assistant panel.

The Hue Query Editor: Results

The screenshot shows the Hue Query Editor interface for an Impala query. The query is:

```
1| SELECT * FROM customers WHERE state='CA';
```

The results table has columns: cust_id, fname, lname, address. The data is:

	cust_id	fname	lname	address
1	1000002	Marilyn	Ham	25831 North 25th Street
2	1000006	Gerard	Franks	356 Turner Street
3	1000010	Mason	Preston	2656 West 13th Street

Annotations with arrows point to:

- A button labeled "View history" pointing to the history icon in the top right.
- A button labeled "View results" pointing to the results table.
- A button labeled "Visualize or save query results" pointing to the download icon in the bottom left.

Chapter Topics

Querying with Apache Hive and Impala

- Databases and Tables
- Basic Hive and Impala Query Language Syntax
- Data Types
- Using Hue to Execute Queries
- **Using Beeline (Hive's Shell)**
- Using the Impala Shell
- Essential Points
- Hands-On Exercise: Running Queries from Shells, Scripts, and Hue



Starting Beeline (Hive's Shell)

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

```
$ beeline -u jdbc:hive2://host:10000 \
-n username -p password
Connecting to jdbc:hive2://host:10000
Connected to: Apache Hive (version 2.1.1-cdh6.0.0)
Beeline version 2.1.1-cdh6.0.0 by Apache Hive
0: jdbc:hive2://host:10000>
```

Log messages have been truncated



Executing Queries in Beeline

- SQL commands are terminated with semicolon (;

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

lname	fname
Ham	Marilyn
Franks	Gerard
...	
Falgoust	Jennifer

```
50 rows selected (15.829 seconds)
```

```
0: url>
```



Using Beeline

- **Beeline commands start with “!”**
 - No terminator character
- **Some commands**
 - `!connect url` connects to a different Hive server
 - `!exit` exits the shell
 - `!help` shows the full list of commands
 - `!verbose` shows additional details of queries

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

- **Press Enter to execute a query or command**



Executing Hive Queries from the Command Line

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

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

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

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

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

```
$ beeline --silent=true -u ...
```

Chapter Topics

Querying with Apache Hive and Impala

- Databases and Tables
- Basic Hive and Impala Query Language Syntax
- Data Types
- Using Hue to Execute Queries
- Using Beeline (Hive's Shell)
- **Using the Impala Shell**
- Essential Points
- Hands-On Exercise: Running Queries from Shells, Scripts, and Hue



Starting the Impala Shell

- You can execute statements in the Impala shell
 - This interactive tool is similar to Beeline
- Execute the `impala-shell` command to start the shell

```
$ impala-shell
Connected to localhost.localdomain:21000
Server version: impalad version 3.0.0-cdh6.0.0
Welcome to the Impala shell.
[localhost.localdomain:21000] >
```

Log messages have been truncated

- Use `-i hostname:port` option to connect to another server

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

Log messages have been truncated



Using the Impala Shell

- **Enter semicolon-terminated statements at the prompt**
 - Commands must be terminated with a semicolon
 - Press **Enter** to execute a query or command
 - Use **quit;** to exit the shell
- **Press Tab twice at the prompt to see all available commands**
- **Run `impala-shell --help` for a full list of command line options**



Executing Queries in the Impala Shell

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

```
Query: select lname, fname FROM customers WHERE state = 'CA'  
LIMIT 50
```

lname	fname
Ham	Marilyn
Franks	Gerard
Preston	Mason
Cortez	Pamela
...	
Falguost	Jennifer

```
Returned 50 row(s) in 0.17s
```



Interacting with the Operating System

- Use shell to execute system commands from within Impala shell

```
> shell date;  
Tue Nov 6 09:30:27 PST 2018
```

- Also use shell to run hdfs dfs commands

```
> shell hdfs dfs -mkdir /reports/sales/2018;
```



Running Impala Queries from the Command Line

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

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

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

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

- Use the **--quiet** option to suppress informational messages

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

- Use **-o** to capture output to file, optionally specifying a delimiter

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

Chapter Topics

Querying with Apache Hive and Impala

- Databases and Tables
- Basic Hive and Impala Query Language Syntax
- Data Types
- Using Hue to Execute Queries
- Using Beeline (Hive's Shell)
- Using the Impala Shell
- **Essential Points**
- Hands-On Exercise: Running Queries from Shells, Scripts, and Hue

Essential Points

- **HiveQL and Impala SQL syntax are familiar to those who know SQL**
 - A subset of SQL-92, plus extensions
 - For example:
 - In Hive, ORDER BY column must be in the SELECT list
 - Support for subqueries differs
- **Hive and Impala support several data types**
 - Most simple column types are similar to SQL data types
 - They also support Boolean, binary, and complex data types
 - Hive and Impala handle type inconsistencies and out-of-range values differently
- **Hive and Impala allow interactive, ad-hoc querying**
 - Use shells with `beeline` or `impala-shell` commands
 - Execute from the command line using the same commands with options
 - Use the query editors in Hue

Bibliography

The following offer more information on topics discussed in this chapter

- **HiveQL language manual on the Hive wiki**
 - <http://tiny.cloudera.com/hqlmanual>
- **Impala documentation on the Cloudera website**
 - <http://tiny.cloudera.com/cdh5impala>
- **Beeline documentation on the Hive wiki**
 - <http://tiny.cloudera.com/beeline>

Chapter Topics

Querying with Apache Hive and Impala

- Databases and Tables
- Basic Hive and Impala Query Language Syntax
- Data Types
- Using Hue to Execute Queries
- Using Beeline (Hive's Shell)
- Using the Impala Shell
- Essential Points
- **Hands-On Exercise: Running Queries from Shells, Scripts, and Hue**

Hands-On Exercise: Running Queries from Shells, Scripts, and Hue

- In this exercise, you will run queries from the Impala and Beeline shells, scripts, and Hue**
- Please refer to the Hands-On Exercise Manual for instructions**



Common Operators and Built-In Functions

Chapter 5

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- **Common Operators and Built-In Functions**
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Common Operators and Built-In Functions

In this chapter, you will learn

- How to write queries for Apache Hive and Impala that use common operators and built-in functions
- What is the difference between scalar functions and aggregate functions
- What are the effects of using scalar and aggregate functions
- How to write queries for Apache Hive and Impala that use aggregate functions
- How to write queries that filter results on calculated aggregations

Chapter Topics

Common Operators and Built-In Functions

- **Operators**
- Scalar Functions
- Aggregate Functions
- Essential Points
- Hands-On Exercise: Using Functions

Using Operators in HiveQL and Impala SQL

- As with other SQL dialects, use operators as part of an expression
- For example:
 - In SELECT list

```
SELECT name, price, cost, price - cost FROM products;
```

- In WHERE clause

```
SELECT name, price, cost FROM products
WHERE cost >= price;
```

Arithmetic Operators in HiveQL and Impala SQL

- **Basic four:** +, -, *, /
 - $84 / 4 = 21$
 - $84 - 4 = 80$
- **Modulo:** %
 - This is the remainder from integer division
 - Example: $25 \% 7 = 4$
- **Unary negative:** —
 - Unary because it takes only one operand
 - $-(3 - 5) = -(-2) = 2$
- **No operator for exponentiation**
 - Use `pow()` function instead
 - $\text{pow}(5, 3) = 5 * 5 * 5 = 125$

Comparison Operators in HiveQL and Impala SQL

■ Equality and Inequality

- =
- != or <>
- <, >, <=, >=

■ IN

- 5 IN (2, 3, 5, 7): true
- 6 IN (2, 3, 5, 7): false

■ BETWEEN

- 5 BETWEEN 2 AND 7: true
- 9 BETWEEN 2 AND 7: false
- 7 BETWEEN 2 and 7: true

■ LIKE

- 'name@example.com' LIKE '%.com': true
- 'Mickey' LIKE 'M*ey': false

Comparisons and NULL Values

- Typical operators return NULL when at least one operand is NULL
 - $5 = \text{NULL} \rightarrow \text{NULL}$
 - $5 \neq \text{NULL} \rightarrow \text{NULL}$
 - $5 < \text{NULL} \rightarrow \text{NULL}$
 - $\text{NULL} = \text{NULL} \rightarrow \text{NULL}$
 - $\text{NULL} \neq \text{NULL} \rightarrow \text{NULL}$
- Alternative operators are IS (NOT) DISTINCT FROM and \Leftrightarrow
 - IS DISTINCT FROM
 - Same as \neq for non-NULL values
 - $5 \text{ IS DISTINCT FROM } \text{NULL} \rightarrow \text{true}$
 - $\text{NULL IS DISTINCT FROM } \text{NULL} \rightarrow \text{false}$
 - IS NOT DISTINCT FROM or \Leftrightarrow (NULL-safe equality)
 - Same as $=$ for non-NULL values
 - $5 \Leftrightarrow \text{NULL} \rightarrow \text{false}$
 - $\text{NULL} \Leftrightarrow \text{NULL} \rightarrow \text{true}$

Logical and Null Operators in HiveQL and Impala SQL

- **Logical operators**
 - Binary: AND, OR
 - Unary: NOT
- **Null operators**
 - IS NULL
 - IS NOT NULL

Column Aliases

- Column aliases can replace expressions in result headers
 - Otherwise Hive or Impala will generate column names
 - The AS keyword is optional but recommended

```
SELECT name, price, cost, price - cost AS profit  
FROM products;
```

name	price	cost	profit
USB Card Reader	1839	1275	564
Composite AV Cable (12 in.)	2549	1762	787
...

Chapter Topics

Common Operators and Built-In Functions

- Operators
- **Scalar Functions**
- Aggregate Functions
- Essential Points
- Hands-On Exercise: Using Functions

Built-In Functions

- **Hive and Impala offer dozens of built-in functions**
 - Many are identical to those found in other SQL dialects
 - Others are Hive- or Impala-specific
- **Example function invocation**
 - Function names are not case-sensitive

```
SELECT concat(fname, ' ', lname) AS fullname  
FROM customers;
```

Getting Information about Functions

- To see a list of all functions and operators in Hive, or user-defined functions in Impala:

```
> SHOW FUNCTIONS;
```



- To see a list of built-in functions in Impala:

```
> SHOW FUNCTIONS IN _impala_builtins;
```

- To see information about a function (Hive only)



```
> DESCRIBE FUNCTION upper;  
upper(str) - Returns str with all characters  
changed to uppercase
```

Trying Built-In Functions

- To test a built-in function, use a **SELECT** statement with no **FROM** clause

```
> SELECT abs(-459.67);  
459.67
```

```
> SELECT upper('Fahrenheit');  
FAHRENHEIT
```

Scalar Functions

- ***Scalar functions operate independently on the values in each row***
- **The number of arguments depends on the function**
 - `rand()`
 - `abs(-283)`
 - `pow(2, 5)`
- **Arguments can be column references, literal values, or expressions**
 - Column references: `year(order_dt)`
 - Literal values: `lower('Function')`
 - Expressions:
 - `year(order_dt + 5)`
 - `round(price * tax, 2)`
- **By convention, scalar functions are usually written in lowercase**

Built-In Mathematical Functions

- These functions perform numeric calculations

Function Description	Example Invocation	Input	Output
Round to specified # of decimals	<code>round(total_price, 2)</code>	23.492	23.49
Return nearest integer above	<code>ceil(total_price)</code>	23.492	24
Return nearest integer below	<code>floor(total_price)</code>	23.492	23
Return absolute value	<code>abs(temperature)</code>	-49	49
Return square root	<code>sqrt(area)</code>	64	8
Return a random number	<code>rand()</code>		0.584977

Built-In Date and Time Functions

- These functions work with **TIMESTAMP** values

Function Description	Example Invocation	Input	Output
Return current date and time	<code>current_timestamp()</code>		<code>2016-06-14 16:51:05.0</code>
Convert to UNIX format	<code>unix_timestamp(order_dt)</code>	<code>2016-06-14 16:51:05</code>	<code>1465923065</code>
Convert to string format	<code>from_unixtime(mod_time)</code>	<code>1465923065</code>	<code>2016-06-14 16:51:05</code>
Extract date portion	<code>to_date(order_dt)</code>	<code>2016-06-14 16:51:05</code>	<code>2016-06-14</code>
Extract year portion	<code>year(order_dt)</code>	<code>2016-06-14 16:51:05</code>	<code>2016</code>
Return # of days between dates	<code>datediff(ship_dt, order_dt)</code>	<code>2016-06-17, 2016-06-14</code>	<code>3</code>

Converting Date and Time Formats

- The default date and time string format is `yyyy-MM-dd HH:mm:ss`
 - Hive and Impala expect dates and times in strings to use this pattern
 - Dates and times are rendered as strings in this format
- You can convert to and from different formats
 - In Hive, use `date_format`; in Impala, use `from_timestamp`



```
SELECT date_format(current_timestamp(), 'yyyy/MM/dd');
```



```
SELECT date_format('1984-09-25', 'yyyy/MM/dd');
```



```
SELECT from_timestamp(current_timestamp(), 'yyyy/MM/dd');
```



```
SELECT from_timestamp('1984-09-25', 'yyyy/MM/dd');
```

Built-In String Functions

- These functions operate on strings

Function Description	Example Invocation	Input	Output
Convert to uppercase	<code>upper(name)</code>	Bob	BOB
Convert to lowercase	<code>lower(name)</code>	Bob	bob
Remove whitespace at start/end	<code>trim(name)</code>	Bob	Bob
Remove only whitespace at start	<code>ltrim(name)</code>	Bob	Bob
Remove only whitespace at end	<code>rtrim(name)</code>	Bob	Bob
Extract portion of string	<code>substring(name, 2, 4)</code>	Samuel	amue
Replace characters in string	<code>translate(name, 'uel', 'my')</code>	Samuel	Sammy

String Concatenation

- **concat combines strings**
 - The `concat_ws` variation combines them with a separator

Example Invocation	Output
<code>concat('alice', '@example.com')</code>	<code>alice@example.com</code>
<code>concat_ws(' ', 'Bob', 'Smith')</code>	<code>Bob Smith</code>
<code>concat_ws('/', 'Amy', 'Sam', 'Ted')</code>	<code>Amy/Sam/Ted</code>

Parsing URLs

- The `parse_url` function parses web addresses (URLs)
- The following examples assume the following URL as input
 - <http://www.example.com/click.php?A=42&Z=105#r1>

Example Invocation	Output
<code>parse_url(url, 'PROTOCOL')</code>	<code>http</code>
<code>parse_url(url, 'HOST')</code>	<code>www.example.com</code>
<code>parse_url(url, 'PATH')</code>	<code>/click.php</code>
<code>parse_url(url, 'QUERY')</code>	<code>A=42&Z=105</code>
<code>parse_url(url, 'QUERY', 'A')</code>	<code>42</code>
<code>parse_url(url, 'QUERY', 'Z')</code>	<code>105</code>
<code>parse_url(url, 'REF')</code>	<code>r1</code>

Other Built-In Functions

- Here are some other interesting functions

Function Description	Example Invocation	Input	Output
Convert to another type	<code>cast(weight AS INT)</code>	3.581	3
Selectively return value	<code>if(price > 1000, 'A', 'B')</code>	1500	A
Selectively return value (multiple cases)	<code>case when price > 1000 then 'A' when price < 100 then 'C' else 'B' end</code>	5	C

Chapter Topics

Common Operators and Built-In Functions

- Operators
- Scalar Functions
- **Aggregate Functions**
- Essential Points
- Hands-On Exercise: Using Functions

Aggregate Functions

- **Aggregate functions combine values from multiple rows**
- **They can work over all rows in a table, returning only one row**

```
SELECT AVG(price) FROM products;
```

- **Or they can work over groups of rows**
 - Group rows based on a column expression
 - GROUP BY *column*
- SELECT brand, AVG(price) FROM products GROUP BY brand;
- Combine rows in the group
 - Results consist of one row for each group
 - Individual row values are not available
- **By convention, aggregate functions are typically uppercase**

Example: Record Grouping and Aggregate Functions

- GROUP BY groups selected data by one or more columns
 - Columns in SELECT list must be in GROUP BY clause or aggregated
- Question: How many products of each brand are in the products table?

```
SELECT brand, COUNT(prod_id) AS num FROM products  
GROUP BY brand;
```

products table			
prod_id	brand	name	price
1	Dualcore	USB Card Reader	18.39
2	Dualcore	HDMI Cable	11.99
3	Dualcore	VGA Cable	1.99
4	Gigabux	6-cell Battery	40.50
5	Gigabux	8-cell Battery	50.50
6	Gigabux	Wall Charger	20.00
7	Gigabux	Auto Charger	20.00



Query results	
brand	num
Dualcore	3
Gigabux	4

Built-In Aggregate Functions

- Hive and Impala offer many aggregate functions, including these

Function Description	Example Invocation
Count all rows	COUNT(*)
Count all rows where field is not NULL	COUNT(fname)
Count all rows where field is unique and not NULL	COUNT(DISTINCT fname)
Return the largest value	MAX(price)
Return the smallest value	MIN(price)
Add all supplied values and return result	SUM(price)
Return the average of all supplied values	AVG(price)

The HAVING Clause

- You cannot filter on aggregate functions using WHERE
- Use HAVING instead
 - Put the HAVING clause after the GROUP BY clause
 - The aggregate function does *not* have to be in the SELECT list or the GROUP BY clause
- Question: What is the average profit of products in the products table, by brand, for brands with at least 50 products?

```
SELECT brand, AVG(price-cost) AS avg_profit
      FROM products
      GROUP BY brand
      HAVING COUNT(prod_id) >= 50;
```

Example: GROUP BY with WHERE and HAVING

- Question: How many employees do we have in each state with a salary of less than \$20,000?

```
SELECT state, COUNT(*) AS num FROM employees  
WHERE salary < 20000  
GROUP BY state;
```

- Question: Which states have more than 400 employees whose salary is less than \$20,000?

```
SELECT state, COUNT(*) AS num FROM employees  
WHERE salary < 20000  
GROUP BY state  
HAVING COUNT(*) > 400;
```

Chapter Topics

Common Operators and Built-In Functions

- Operators
- Scalar Functions
- Aggregate Functions
- **Essential Points**
- Hands-On Exercise: Using Functions

Essential Points

- Use operators and functions in queries wherever you can use an expression
- Scalar functions operate independently on the values in each row
- Aggregate functions work with values from multiple rows
 - Aggregate all the rows, or use GROUP BY to aggregate groups of rows
 - Results have one row for each aggregated group
- The syntax for aggregate functions is

```
SELECT col_1, function(col_2)
  FROM table GROUP BY col_1
```

- Columns in SELECT list must be in GROUP BY clause or aggregated
- To filter results on aggregate calculations, use the HAVING clause
 - Put it after the GROUP BY clause

Bibliography

The following offer more information on topics discussed in this chapter

- **Hive Built-In Functions**
 - <http://tiny.cloudera.com/hivefunctions>
- **Impala Built-In Functions**
 - <http://tiny.cloudera.com/impalafunctions>

Chapter Topics

Common Operators and Built-In Functions

- Operators
- Scalar Functions
- Aggregate Functions
- Essential Points
- **Hands-On Exercise: Using Functions**

Hands-On Exercise: Using Functions

- In this exercise, you will write queries using functions to analyze data in tables
- Please refer to the Hands-On Exercise Manual for instructions



Data Management

Chapter 6

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- **Data Management**
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Data Management

In this chapter, you will learn

- What is the proper syntax to create and modify databases and tables
- What is the difference between tables and views
- When a view is helpful
- How to store results of queries into an existing or a new table
- How to store query results into a directory
- How to load data into a big data table

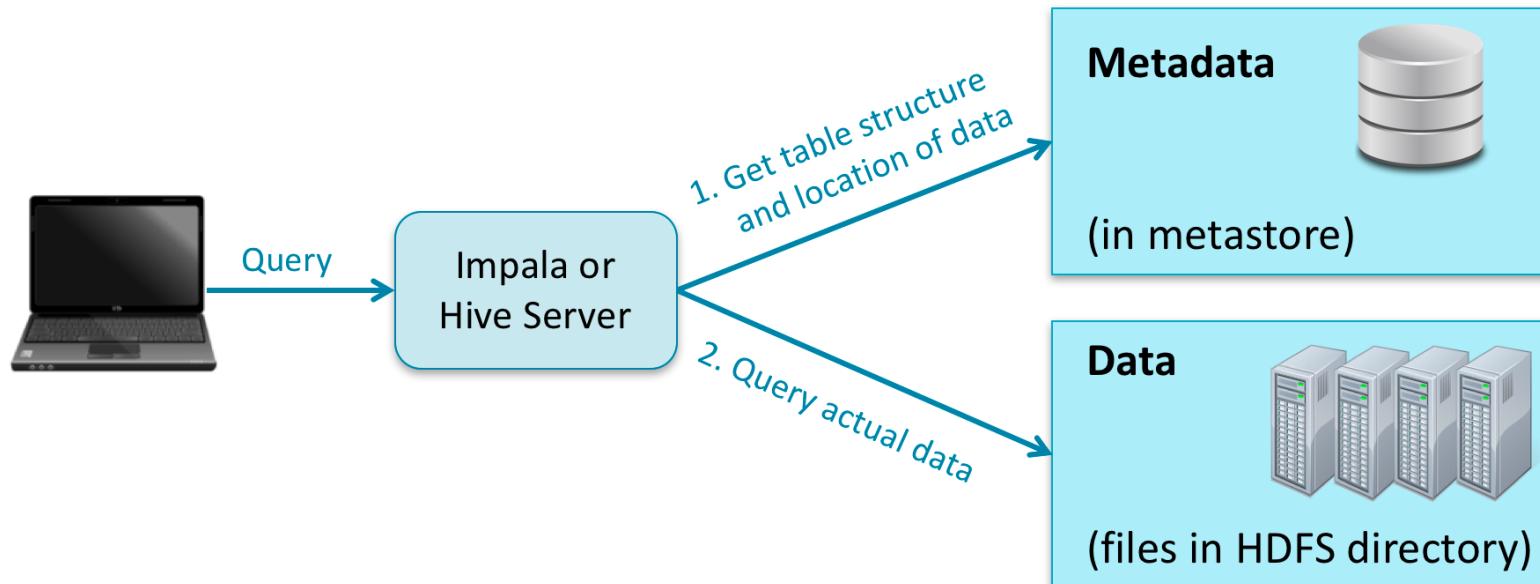
Chapter Topics

Data Management

- **Data Storage**
- Creating Databases and Tables
- Loading Data
- Altering Databases and Tables
- Simplifying Queries with Views
- Storing Query Results
- Essential Points
- Hands-On Exercise: Data Management

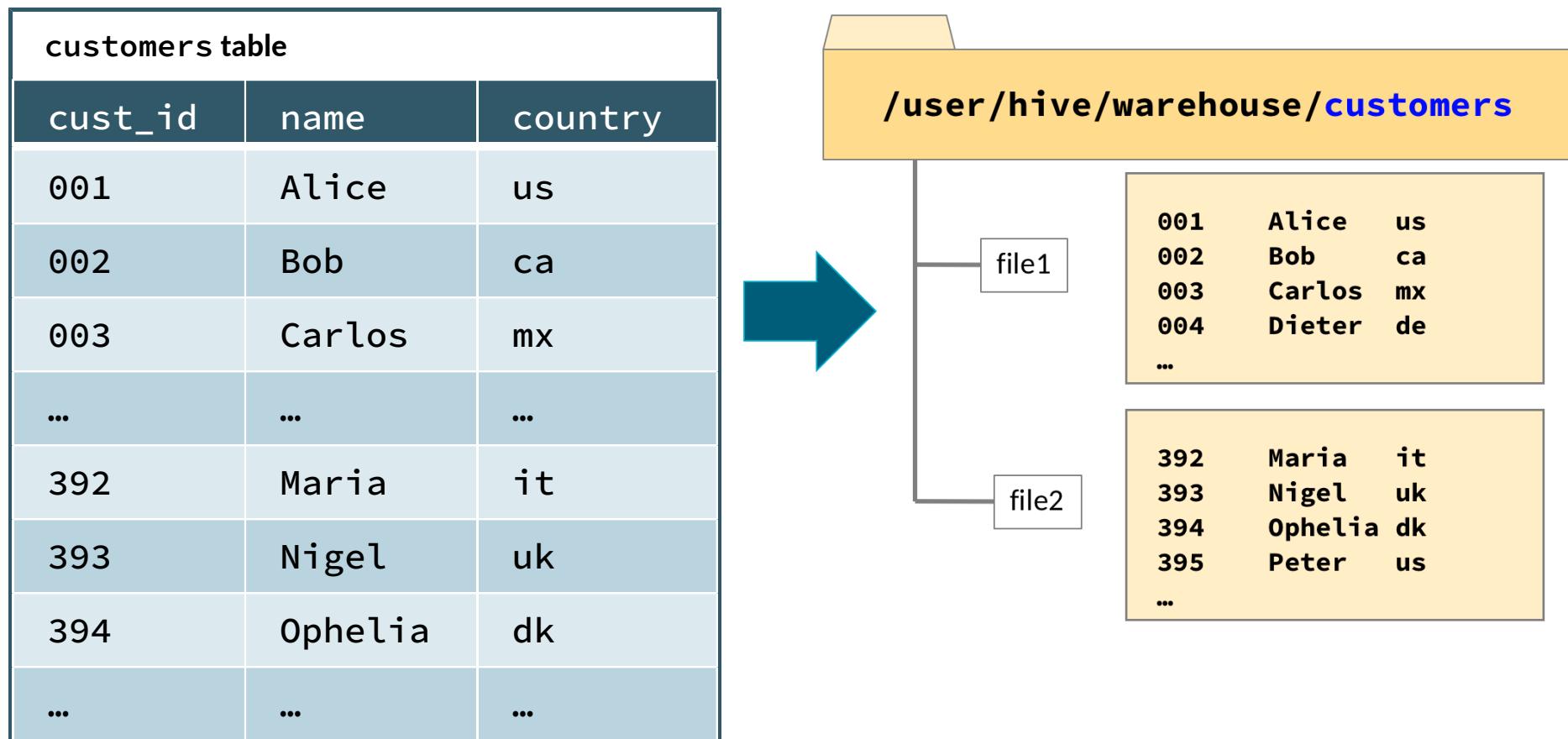
Recap: How Hive and Impala Query Data

- Hive and Impala use the metastore to determine data format and location
- The query itself operates on data stored in a filesystem (typically HDFS)



The Warehouse Directory

- By default, Hive and Impala store data in the HDFS directory `/user/hive/warehouse`
 - Other storage options include HBase, Apache Kudu, and S3
- In HDFS, each table's data is stored in a subdirectory named after the table



Impala Metadata Cache Invalidation

- Hive retrieves metadata from the metastore with every query
- Impala caches metadata to reduce query latency
 - Table structure and data location from the metastore
 - File information from storage system (for example, HDFS)
- Changes made to tables outside Impala cause the cache to be out of sync
 - Creating tables with Hive
 - Importing table data with Sqoop
 - Adding table data in the storage system
- After such changes, you should prompt Impala to update the cache



Updating the Impala Metadata Cache

External Metadata Change	Required Action	Effect on Local Caches
Table schema modified or new data added to a table	<code>REFRESH <i>tablename</i>;</code>	Reloads the metadata for one table immediately; reloads storage block locations for new data files only
New table added, or data in a table extensively altered, such as by HDFS balancing	<code>INVALIDATE METADATA <i>tablename</i>;</code>	Marks the metadata for a single table as stale; when the metadata is needed, all storage block locations are retrieved

- In Hue, you can also use the Refresh button
- **Caution: `INVALIDATE METADATA` with no table name affects *all* users**
 - Marks the entire cache as stale, to be rebuilt completely when needed
 - Can be time-consuming with large tables or lots of tables
 - Use only when needed

Accessing the Metastore Using Hue

■ Hue Table Browser

- An alternative to using SQL commands to manage metadata
- Allows you to create, load, preview, and delete databases and tables

The screenshot shows the Hue Table Browser interface. On the left, a sidebar menu lists 'Apps' (Editor, Scheduler), 'Browsers' (Documents, Files), and 'Tables'. A red arrow points from the 'Tables' menu item to the main browser area. The main area shows the 'Table Browser' for the 'analyst' database. The database properties are listed as follows:

PROPERTIES	
Owner	anonymous (USER)
Location	Location

The 'TABLES' section displays four tables:

Table	Description
customers	Add a description...
order_details	Add a description...
orders	Add a description...
products	Add a description...

Accessing the Metastore with HCatalog

- Hive and Impala have built-in capability to access the metastore directly
- Other tools like Spark lack built-in direct metastore access
- HCatalog is a Hive sub-project that provides access to the metastore
 - Accessible through command line and REST API
 - Allows you to define and describe tables using Hive syntax
 - Enables integration with applications that lack direct metastore access

Chapter Topics

Data Management

- Data Storage
- **Creating Databases and Tables**
- Loading Data
- Altering Databases and Tables
- Simplifying Queries with Views
- Storing Query Results
- Essential Points
- Hands-On Exercise: Data Management

Creating a Database

- Hive and Impala databases are namespaces for organizing tables
- To create a new database

```
CREATE DATABASE dualcore;
```

1. Adds the database definition to the metastore

2. Creates a storage directory in HDFS

For example, /user/hive/warehouse/dualcore.db

- To create a new database conditionally

— Avoids error if database already exists (useful for scripting)

```
CREATE DATABASE IF NOT EXISTS dualcore;
```

Creating a Table (1)

- Basic syntax for creating a table:

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

- Creates an empty subdirectory in the database's warehouse directory in HDFS
 - Default database:
/user/hive/warehouse/tablename
 - Named database:
/user/hive/warehouse/dbname.db/tablename
- Creates the metadata for the table in the metastore

Creating a Table (2)

```
CREATE TABLE dbname.tablename (colname DATATYPE, ...) ①  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY char  
STORED AS {TEXTFILE|SEQUENCEFILE|...};
```

- ① Specify the database name (optional), a name for the table, and list the column names and data types.

Creating a Table (3)

```
CREATE TABLE dbname.tablename (colname DATATYPE, ...)  
ROW FORMAT DELIMITED(1)  
  FIELDS TERMINATED BY char  
  STORED AS {TEXTFILE|SEQUENCEFILE|...};
```

- ① This line states that fields in each file in the table's directory are delimited by some character.

Hive's default field delimiter is control+A, but you may specify an alternate field delimiter...

Creating a Table (4)

```
CREATE TABLE dbname.tablename (colname DATATYPE, ...)  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY char(1)  
STORED AS {TEXTFILE|SEQUENCEFILE|...};
```

- ① ...for example, tab-delimited data would require that you specify **FIELDS TERMINATED BY '\t'**.

Creating a Table (5)

```
CREATE TABLE dbname.tablename (colname DATATYPE, ...)  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY char  
STORED AS {TEXTFILE|SEQUENCEFILE|...};(1)
```

- ① Finally, you may declare the format to use for the data files in the warehouse subdirectory. **STORED AS TEXTFILE** is the default and does not need to be specified.

Example Table Creation

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

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

- This record shows what the data to be loaded into **jobs** will look like

```
1,Data Analyst,135000,2016-12-21 15:52:03
```

Creating Tables Based on Existing Schema

- Use LIKE to create a new table using the definition of an existing table

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

- Column definitions are derived from the existing table's definition
 - New table will contain no data

Controlling Table Data Location

- By default, table data is stored in the warehouse directory (within HDFS)
- This is not always ideal
 - Data might be part of a bigger workflow
- Use LOCATION to specify the directory where table data resides

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

- For locations outside HDFS, a fully qualified path will be needed
 - LOCATION 's3a://path.to.bucket/jobs'
 - LOCATION 'adl://path.to.bucket/jobs'

Externally Managed Tables

- **CAUTION:** Dropping a table removes its data in HDFS
- **Using EXTERNAL when creating the table avoids this behavior**
 - Dropping an *external (unmanaged)* table removes only its *metadata*

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

Creating Tables Using the Table Browser

- Hue's Table Browser provides a table creation wizard
 - Supports most, but not all, table options

The screenshot shows the Hue Table Browser interface. On the left, there is a sidebar with icons for databases, tables, and files. Below these are lists for 'Tables' (customers, order_details, orders, products) and 'Jobs'. At the top, there is a navigation bar with 'Query', a search bar ('Search saved documents...'), and user information ('Jobs', 'training'). The main area is titled 'Table Browser' and shows the 'analyst' database. It displays 'Impala' as the connection type and the path 'Databases > analyst'. Below this, there are sections for 'PROPERTIES' (Owner: anonymous (USER), Location: Location) and 'TABLES'. The 'TABLES' section contains a table with four rows: 'customers', 'order_details', 'orders', and 'products'. Each row has a checkbox, a table icon, and a description column with placeholder text ('Add a description...'). At the bottom of the table list, there are buttons for 'View', 'Query', 'Drop', and '+ New'. The '+ New' button is highlighted with a red box.

Table	Description
<input type="checkbox"/> <i>customers</i>	Add a description...
<input type="checkbox"/> <i>order_details</i>	Add a description...
<input type="checkbox"/> <i>orders</i>	Add a description...
<input type="checkbox"/> <i>products</i>	Add a description...

Chapter Topics

Data Management

- Data Storage
- Creating Databases and Tables
- **Loading Data**
- Altering Databases and Tables
- Simplifying Queries with Views
- Storing Query Results
- Essential Points
- Hands-On Exercise: Data Management

Data Validation

- **Hive and Impala are generally *schema-on-read***
 - Most common method to load data is simply to move files into place
 - Loading data into tables is therefore very fast
 - Errors in file format will be discovered when queries are performed
 - Exception: You can use an `INSERT ... VALUES` statement which validates on execution
 - Not suitable for loading large amounts of data
- **Missing or invalid data typically will be represented as `NULL`**

Loading Data in HDFS

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

```
$ hdfs dfs -put sales.txt /user/hive/warehouse/sales/
```

- This example loads data from the user's directory in HDFS to `sales`

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

- Alternatively, use the `LOAD DATA INPATH` command

- Done from within Hive or Impala
 - This changes the file location within HDFS, just like the second command above
 - Source can be either a file or directory

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

Overwriting Data from Files

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

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

Loading Data Using the Table Browser

- You can load data when creating a table

The screenshot shows the Hue interface with the 'Table Browser' tab selected. On the left, there's a sidebar for the 'analyst' database containing tables like 'customers', 'order_details', 'orders', and 'products'. The main area shows the 'Tables' section with a 'Refresh' button. A red arrow points from the '+ New' button in the toolbar at the bottom right of the main area towards a separate window titled 'Import to table'. This window has two tabs: 'SOURCE' (selected) and 'TARGET'. Tab 1 under SOURCE says 'Pick data from file' with a 'File' type dropdown and a 'Path' input field. Tab 2 under TARGET says 'Move it to table analyst'.

- You can also load the data into an existing table

The screenshot shows the Hue interface with the 'Table Browser' tab selected. The path 'Databases > analyst > customers' is shown. The toolbar at the bottom right includes a 'Query' button, an 'Import' button (highlighted with a red box), a 'Drop' button, and a 'Refresh' button. Below the toolbar, there are tabs for 'Overview', 'Sample (0)', 'Details', and 'Privileges'. Under 'PROPERTIES', it says 'External and stored in location'. Under 'STATS', it says 'Data last updated on 05/24/2019 3:33 PM'.

Loading Data from a Relational Database

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

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

Chapter Topics

Data Management

- Data Storage
- Creating Databases and Tables
- Loading Data
- **Altering Databases and Tables**
- Simplifying Queries with Views
- Storing Query Results
- Essential Points
- Hands-On Exercise: Data Management

Removing a Database

- Removing a database has similar syntax to creating it

```
DROP DATABASE dualcore;
```

```
DROP DATABASE IF EXISTS dualcore;
```

- These commands will fail if the database contains tables
 - Add the CASCADE keyword to force removal
 - Supported in all production versions of Hive
 - Supported in Impala 2.3 (CDH 5.5.0) and higher

```
DROP DATABASE dualcore CASCADE;
```

**CAUTION: This
command might
remove data in HDFS!**

Removing a Table

- Table removal syntax is similar to database removal

```
DROP TABLE IF EXISTS customers;
```

```
DROP TABLE customers;
```

- Managed (internal) tables

- Metadata is removed
- Data in HDFS is removed
- *Caution: No rollback or undo feature!*

- Unmanaged (external) tables

- Metadata is removed
- Data in HDFS is *not* removed

Renaming and Moving Tables

- Use `ALTER TABLE` to modify a table or its columns
- Rename an existing table
 - Changes metadata
 - If table is managed, renames directory in HDFS

```
ALTER TABLE customers RENAME TO clients;
```

- Move an existing table to a different database
 - Changes metadata
 - If table is managed, moves directory in HDFS

```
ALTER TABLE default.clients
    RENAME TO dualcore.clients;
```

Renaming and Modifying Columns

- Rename a column by specifying its old and new names
 - Type must be specified even if it is unchanged

```
ALTER TABLE clients CHANGE fname first_name STRING;
```



- You can also modify a column's type
 - The old and new column names will be the same
 - Does not change the data in HDFS
 - You must ensure the data in HDFS conforms to the new type

```
ALTER TABLE clients CHANGE salary salary BIGINT;
```



- You can also change name and type using a single command

```
ALTER TABLE suppliers CHANGE supp_id supplier_id BIGINT;
```



Reordering Columns in Hive

- Use AFTER or FIRST to reorder columns in Hive
 - Does not change the data in HDFS
 - You must ensure the data in HDFS matches the new order

```
ALTER TABLE jobs  
    CHANGE salary salary INT AFTER id;
```

```
ALTER TABLE jobs  
    CHANGE salary salary INT FIRST;
```

Adding and Removing Columns

- Add new columns to a table

- Appended after any existing columns
 - Does not change the data in HDFS
 - If column does not exist in data in HDFS, then values will be NULL

```
ALTER TABLE jobs  
ADD COLUMNS (city STRING, bonus INT);
```

- Remove columns from a table (Impala only)

- Does not change the data in HDFS

```
ALTER TABLE jobs DROP COLUMN salary;
```



Replacing Columns and Reproducing Tables

- Use **REPLACE COLUMNS** to change the entire table's column definitions
 - Any column not listed will be dropped from the metadata
 - Does not change the data in HDFS

```
ALTER TABLE jobs REPLACE COLUMNS (
    id INT,
    title STRING,
    salary INT
);
```

- Use **SHOW CREATE TABLE** to display a statement to reproduce the table
 - Displays **CREATE TABLE** statement to create table in its current state
 - Use instead of recreating sequence of **CREATE** and **ALTER** statements

Changing Other Properties

- You can change other properties using ALTER TABLE
- For example, change a managed table to external

```
ALTER TABLE tablename SET TBLPROPERTIES('EXTERNAL'='TRUE')
```

- Note that EXTERNAL and TRUE must be uppercase
- See the ALTER TABLE statement documentation for more
 - Impala: <http://tiny.cloudera.com/impala-altertable>
 - Hive: <http://tiny.cloudera.com/hive-altertable>

Chapter Topics

Data Management

- Data Storage
- Creating Databases and Tables
- Loading Data
- Altering Databases and Tables
- **Simplifying Queries with Views**
- Storing Query Results
- Essential Points
- Hands-On Exercise: Data Management

Simplifying Complex Queries

- Complex queries can become cumbersome
 - Imagine typing this several times for different orders

```
SELECT o.order_id, order_date, p.prod_id, brand, name
  FROM orders o
  JOIN order_details d
    ON (o.order_id = d.order_id)
  JOIN products p
    ON (d.prod_id = p.prod_id)
 WHERE o.order_id=6584288;
```

Creating Views

- A **view** is a saved query on a table or set of tables
 - You can query a view as if it were a table

```
CREATE VIEW order_info AS
    SELECT o.order_id, order_date, p.prod_id, brand, name
    FROM orders o
    JOIN order_details d ON (o.order_id = d.order_id)
    JOIN products p ON (d.prod_id = p.prod_id);
```

- The query is now greatly simplified

```
SELECT * FROM order_info WHERE order_id=6584288;
```

- Views are also helpful to provide limited access to table data
 - Prohibit access to sensitive information in some columns or rows
- You cannot directly add data to a view

Exploring Views

- **SHOW TABLES** lists the tables *and views* in a database
 - There is no separate command to list only views

```
SHOW TABLES;
```

- Use **DESCRIBE FORMATTED** to see a view's underlying query

```
DESCRIBE FORMATTED order_info;
```

- Use **SHOW CREATE TABLE** to display a statement to create the view

```
SHOW CREATE TABLE order_info;
```

Modifying and Removing Views

- Use ALTER VIEW to change the underlying query

```
ALTER VIEW order_info AS  
SELECT order_id, order_date FROM orders;
```

- Or to rename a view

```
ALTER VIEW order_info  
RENAME TO order_information;
```

- Use DROP VIEW to remove a view

```
DROP VIEW order_info;
```

Chapter Topics

Data Management

- Data Storage
- Creating Databases and Tables
- Loading Data
- Altering Databases and Tables
- Simplifying Queries with Views
- **Storing Query Results**
- Essential Points
- Hands-On Exercise: Data Management

Saving Query Output to a Table

- **SELECT statements display their results on screen**
- **To save results to a table, use `INSERT OVERWRITE TABLE`**
 - Destination table must already exist
 - Existing contents will be deleted

```
INSERT OVERWRITE TABLE nyc_customers
  SELECT * FROM customers
  WHERE state = 'NY' AND city = 'New York';
```

- **`INSERT INTO TABLE` adds records without first deleting existing data**

```
INSERT INTO TABLE nyc_customers
  SELECT * FROM customers
  WHERE state = 'NY' AND city = 'Brooklyn';
```

Creating Tables Based on Existing Data

- Create a table based on a SELECT statement
 - Known as CREATE TABLE AS SELECT (CTAS)

```
CREATE TABLE ny_customers
  ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
  STORED AS TEXTFILE
AS
  SELECT cust_id, fname, lname
  FROM customers
  WHERE state = 'NY';
```

- Column definitions are derived from the existing table
- Column names are inherited from the existing names
 - Use aliases in the SELECT statement to specify new names



Writing Output in Hive

- Hive also lets you save output to a directory
 - This example writes to a directory in HDFS*

```
INSERT OVERWRITE DIRECTORY '/analyst/dualcore/ny/'  
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'  
SELECT * FROM customers  
WHERE state = 'NY';
```

- For other storage locations, qualify the path
 - For example, 's3a://dualcore/ny/'
 - Or, 'adl://dualcore/ny/'

*Note: Some older versions of Hive did *not* delete existing contents as expected



Saving to Multiple Directories in Hive (1)

- This query will save output to an HDFS file

```
INSERT OVERWRITE DIRECTORY 'ny_customers'  
    SELECT cust_id, fname, lname  
        FROM customers WHERE state = 'NY';
```

- It could also be written as follows

```
FROM customers c  
INSERT OVERWRITE DIRECTORY 'ny_customers'  
    SELECT cust_id, fname, lname WHERE state='NY';
```



Saving to Multiple Directories in Hive (2)

- You sometimes might want to save query output to multiple directories
 - Hive SELECT queries can take a long time to complete
- Hive allows this with a single query
 - Much more efficient than using multiple queries
 - Result is two directories in HDFS

```
FROM customers c
  INSERT OVERWRITE DIRECTORY 'ny_names'
    SELECT fname, lname
      WHERE state = 'NY'
  INSERT OVERWRITE DIRECTORY 'ny_count'
    SELECT COUNT(DISTINCT cust_id)
      WHERE state = 'NY';
```



Saving to Multiple Directories in Hive (3)

- The following query produces the same result

```
FROM (SELECT * FROM customers WHERE state='NY') nycust
  INSERT OVERWRITE DIRECTORY 'ny_names'
    SELECT fname, lname
  INSERT OVERWRITE DIRECTORY 'ny_count'
    SELECT COUNT(DISTINCT cust_id);
```

Chapter Topics

Data Management

- Data Storage
- Creating Databases and Tables
- Loading Data
- Altering Databases and Tables
- Simplifying Queries with Views
- Storing Query Results
- **Essential Points**
- Hands-On Exercise: Data Management

Essential Points

- **Use CREATE TABLE to create tables and DROP TABLE to delete them**
 - Creating from existing tables is often helpful
- **Use ALTER TABLE to modify tables, including adding or removing columns**
- **Each table's data is stored in a directory in the storage system (such as HDFS)**
 - Load data by moving files into the directory or using LOAD DATA INPATH
 - Metadata is stored elsewhere
 - In Impala, metadata is cached and sometimes needs refreshing
- **Views are saved *queries* on a table or set of tables**
 - Help to simplify complex and repetitive queries
 - Can provide limited access to table data
- **You can store query *results* into a table or other directory**
 - Use INSERT OVERWRITE TABLE or INSERT INTO TABLE to save to an existing table
 - Use CTAS (CREATE TABLE AS SELECT) to save as a new table
 - Use INSERT OVERWRITE DIRECTORY for non-table directories

Chapter Topics

Data Management

- Data Storage
- Creating Databases and Tables
- Loading Data
- Altering Databases and Tables
- Simplifying Queries with Views
- Storing Query Results
- Essential Points
- **Hands-On Exercise: Data Management**

Hands-On Exercise: Data Management

- In this exercise, you will create and load tables using the Hue Table Browser, Impala SQL, and Sqoop
- Please refer to the Hands-On Exercise Manual for instructions



Data Storage and Performance

Chapter 7

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- **Data Storage and Performance**
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Data Storage and Performance

In this chapter, you will learn

- How partitioning can improve efficiency
- How to create a partitioned table
- How to load data into a partitioned table
- What are some different file formats for Apache Hive and Impala
- Why Apache Avro and Apache Parquet might be good choices for file formats
- How use different file formats, including Avro and Parquet, to load or store data

Chapter Topics

Data Storage and Performance

- **Partitioning Tables**
- Loading Data into Partitioned Tables
- When to Use Partitioning
- Choosing a File Format
- Using Avro and Parquet File Formats
- Essential Points
- Hands-On Exercise: Data Storage and Performance

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

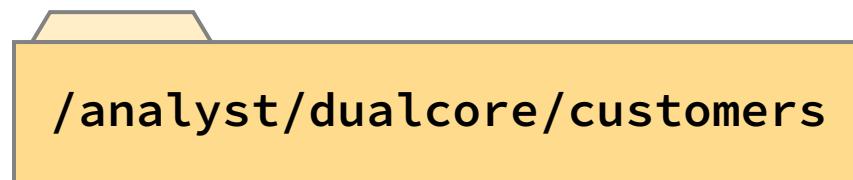
Example: Partitioning Customers by State (1)

- Example: `customers` 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'
LOCATION '/dualcore/customers';
```

Example: Partitioning Customers by State (2)

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



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

...

file1

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

...

file2

Example: Partitioning Customers by State (3)

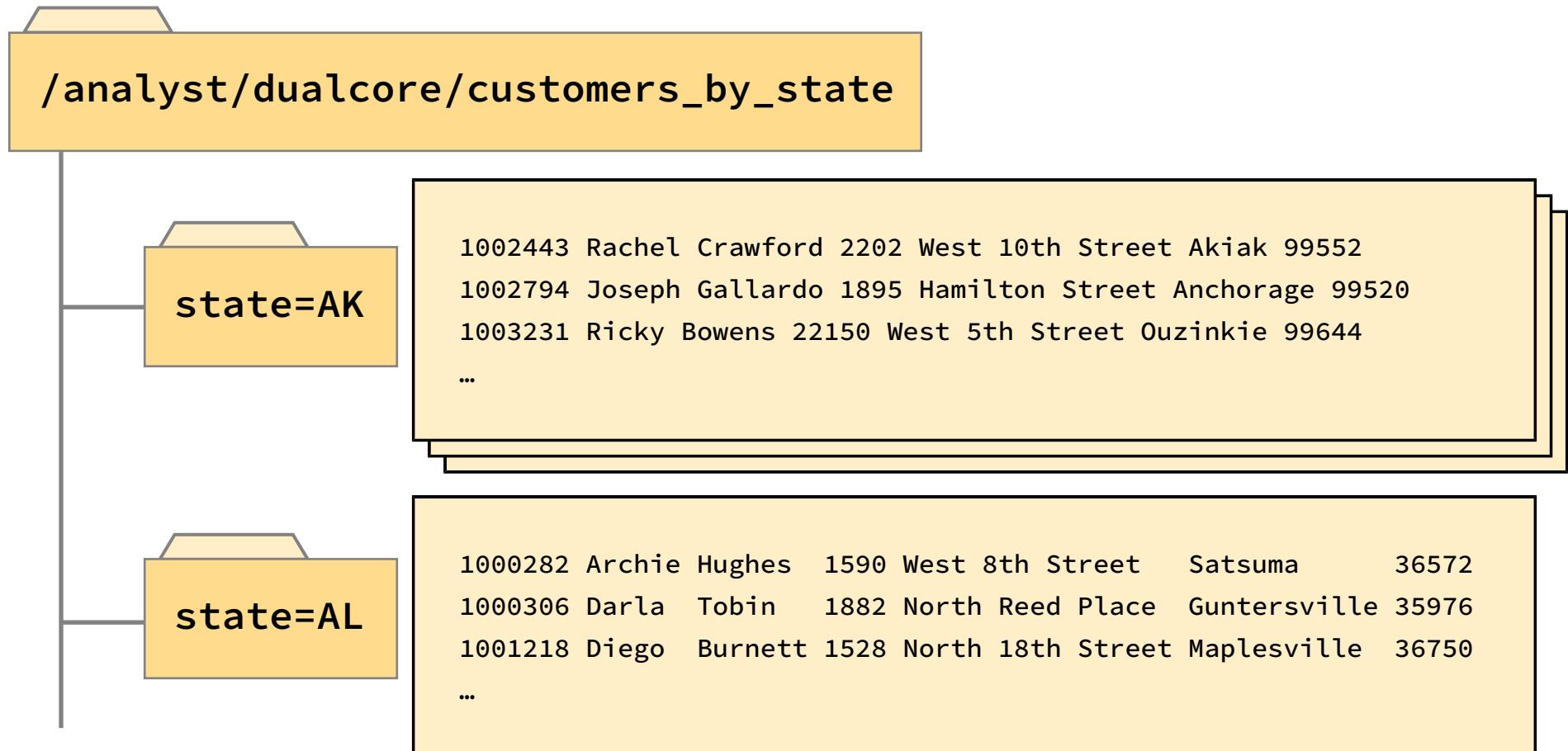
- What if most of Dualcore's analysis on the customer table was done by state? For example:

```
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
 - Queries that filter on partitioned fields limit amount of data read



Creating a Partitioned Table

- Create 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'
LOCATION '/dualcore/customers_by_state';
```

Partition 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    |          | ①
+-----+-----+-----+
```

- ① 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)
```

/analyst/dualcore/customers_by_state_and_zipcode

state=AK

zipcode=96520

zipcode=96552

1002794 Joseph Gallardo 1895 Hamilton Street Anchorage

1009777 Tree van Nilson 1331 Village Lane Anchorage

...

1002443 Rachel Crawford 2202 West 10th Street Akiak

1003232 Penny Lane 233 West 5th Street Akiak

...

Chapter Topics

Data Storage and Performance

- Partitioning Tables
- **Loading Data into Partitioned Tables**
- When to Use Partitioning
- Choosing a File Format
- Using Avro and Parquet File Formats
- Essential Points
- Hands-On Exercise: Data Storage and Performance

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

`call-20161001.log`

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

`call-20161002.log`

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

Static Partitioning Example: Partition Calls by Day (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 ',';
```

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
`/user/hive/warehouse/call_logs/`

Static Partitioning Example: Partition Calls by Day (4)

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

```
LOAD DATA INPATH '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 '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 '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
`/user/hive/warehouse/call_logs/call_date=2016-10-02` 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;
```



Partitioning in Hive (1)

- **Hive's default disallows using dynamic partitioning**
 - Intended to prevent users from accidentally creating a huge number of partitions
 - Remove this limitation by changing a configuration property

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

- **Note: Hive properties set in Beeline are for the current session only.**
 - Your system administrator can configure properties permanently



Partitioning in Hive (2)

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

Viewing, Adding, and Removing Partitions

- To view the current partitions in a table

```
SHOW PARTITIONS call_logs;
```

- Use ALTER TABLE to add or drop partitions

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

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

Chapter Topics

Data Storage and Performance

- Partitioning Tables
- Loading Data into Partitioned Tables
- **When to Use Partitioning**
- Choosing a File Format
- Using Avro and Parquet File Formats
- Essential Points
- Hands-On Exercise: Data Storage and Performance

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 Storage and Performance

- Partitioning Tables
- Loading Data into Partitioned Tables
- When to Use Partitioning
- **Choosing a File Format**
- Using Avro and Parquet File Formats
- Essential Points
- Hands-On Exercise: Data Storage and Performance

Choosing a File Format

- Hive and Impala support many different file formats for data storage

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

- Format options

- TEXTFILE
- SEQUENCEFILE
- AVRO
- PARQUET
- RCFILE
- ORCFILE (Hive only)*

* Supported, but not recommended, in CDH

Considerations for Choosing a File Format

- **Hadoop and its ecosystem support many file formats**
 - You can ingest data in one format and convert to another as needed
- **Selecting the format for your dataset involves several considerations**
 - Ingest pattern
 - Tool compatibility
 - Expected lifetime
 - Storage and performance requirements
- **Which format is best? It depends on your data and use cases**
 - The following slides offer some general guidance on common formats

Text File Format

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

SequenceFile Format

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

Apache Avro File Format

- **Apache Avro is an efficient data serialization framework**
- **Avro also defines a data file format for storing Avro records**
 - Similar to SequenceFile format
- **Efficient storage due to optimized binary encoding**
- **Widely supported throughout the Hadoop ecosystem**
 - Can also be used outside of Hadoop
- **Ideal for long-term storage of important data**
 - Many languages can read and write Avro files
 - Embeds schema in the file, so will always be readable
 - Schema evolution can accommodate changes
- **Verdict: Excellent interoperability and performance**
 - Best choice for general-purpose storage in Hadoop

Columnar Formats

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

Organization of data in traditional row-based formats			
id	name	city	occupation
1	Alice	Palo Alto	Accountant
2	Bob	Sunnyvale	Accountant
3	Bob	Palo Alto	Dentist
4	Bob	Palo Alto	Manager
5	Carol	Palo Alto	Manager

Row-based storage to disk



Organization of data in columnar formats			
id	name	city	occupation
1	Alice	Palo Alto	Accountant
2	Bob	Sunnyvale	Accountant
3	Bob	Palo Alto	Dentist
4	Bob	Palo Alto	Manager
5	Carol	Palo Alto	Manager

Column-based storage to disk



Columnar File Formats: RCFile and ORCFile

■ RCFile

- A column-oriented format originally created for Hive tables
- All data stored as strings (inefficient)
- **Verdict:** Poor performance and limited interoperability

■ ORCFile



- An improved version of RCFile
- Currently supported only in Hive, not Impala
- More efficient than RCFile, but not well supported outside of Hive
- **Verdict:** Improved performance but limited interoperability

Columnar File Formats: Apache Parquet

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

Chapter Topics

Data Storage and Performance

- Partitioning Tables
- Loading Data into Partitioned Tables
- When to Use Partitioning
- Choosing a File Format
- **Using Avro and Parquet File Formats**
- Essential Points
- Hands-On Exercise: Data Storage and Performance

Using Avro File Format (1)

- Avro embeds a schema definition in the file itself
- But an Avro table still must have a schema definition
 - This is stored in the metastore or in a separate Avro schema file
- To store the table schema in the metastore, create the table as usual

```
CREATE TABLE order_details_avro (order_id INT, prod_id INT)
    STORED AS AVRO;
```

- To use a separate schema file, specify the `avro.schema.url` property

```
CREATE TABLE order_details_avro
    STORED AS AVRO
    TBLPROPERTIES ('avro.schema.url'=
        'hdfs://localhost/dualcore/order_details.avsc');
```

Using Avro File Format (2)

- Avro schemas are represented in JSON
 - Use `avro.schema.literal` to create a table using a JSON schema

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

Using Parquet File Format (1)

- Create a new table stored in Parquet format

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

- Load data from another table into a Parquet table

```
INSERT OVERWRITE TABLE order_details_parquet
    SELECT * FROM order_details;
```

Using Parquet File Format (2)

- Parquet embeds a schema definition in the file itself, just as Avro does
- In Impala, use `LIKE PARQUET` to create a table using the column definitions of an existing Parquet data file
- Example: Create a new table to access an existing Parquet file in HDFS



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

Chapter Topics

Data Storage and Performance

- Partitioning Tables
- Loading Data into Partitioned Tables
- When to Use Partitioning
- Choosing a File Format
- Using Avro and Parquet File Formats
- **Essential Points**
- Hands-On Exercise: Data Storage and Performance

Essential Points

- **Partitioning improves performance of queries that filter on partition columns**
 - Create using PARTITIONED BY (*col_name col_type*)
- **Load the data into partitioned tables with static or dynamic partitions**
 - For static, create the partition with ADD PARTITION then load data using PARTITION(*col_name=value*)
 - For dynamic, load the data using PARTITION(*col_name*) and the engine creates the partitions automatically
- **Apache Hive and Impala accept different file formats, for example**
 - Text files: basic, convenient, human-readable
 - Avro: best choice for general-purpose storage
 - Parquet: best choice for columnar storage
- **Specify the file format when creating the table using STORED AS**
- **Copy Avro or Parquet column definitions on creation**
 - Avro: specify the `avro.schema.url` property in TBLPROPERTIES
 - Parquet: in Impala, use `LIKE PARQUET` to specify a Parquet schema

Bibliography

The following offer more information on topics discussed in this chapter

- **Using Impala at Scale at Allstate**
 - <http://tiny.cloudera.com/allstateimpala>
- **Introducing Parquet: Efficient Columnar Storage for Apache Hadoop**
 - <http://tiny.cloudera.com/dac16a>

Chapter Topics

Data Storage and Performance

- Partitioning Tables
- Loading Data into Partitioned Tables
- When to Use Partitioning
- Choosing a File Format
- Using Avro and Parquet File Formats
- Essential Points
- **Hands-On Exercise: Data Storage and Performance**

Hands-On Exercise: Data Storage and Performance

- In this exercise, you will create a table for ad click data, partitioned by the network on which the ad was displayed
- Please refer to the Hands-On Exercise Manual for instructions



Working with Multiple Datasets

Chapter 8

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- **Working with Multiple Datasets**
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Working with Multiple Datasets

In this chapter, you will learn

- How using UNION will combine two or more datasets
- How using basic joins will combine two or more datasets
- How to write queries using appropriate JOIN types to provide desired results
- How NULL values affect results in a JOIN key column

Chapter Topics

Working with Multiple Datasets

- **UNION and Joins**
- Handling NULL Values in Joins
- Advanced Joins
- Essential Points
- Hands-On Exercise: Working with Multiple Datasets

Combining Query Results with a Union

- **UNION ALL unifies output from multiple SELECTs into a single result set**
 - The order and types of columns in each query must match
 - In Hive, the names of columns also must match

```
SELECT cast(cust_id AS string) AS id, fname, lname
      FROM customers
      WHERE state = 'NY'
UNION ALL
SELECT emp_id AS id, fname, lname
      FROM employees
      WHERE state = 'NY';
```

- **Without the ALL keyword, UNION also removes duplicate values**
 - Impala supports this
 - Hive supports this as of version 1.2.0

Joins

- **Joining disparate datasets is a common operation**
 - Uses a shared column between the datasets
 - Combines rows by matching values in the shared columns
- **Hive and Impala support several types of joins**
 - Inner joins
 - Outer joins (left, right, and full)
 - Cross joins
 - Left semi-joins
- **Join conditions must use equality comparisons when using Hive**
 - Valid: `customers.cust_id = orders.cust_id`
 - Invalid: `customers.cust_id <> orders.cust_id`
 - Impala supports some non-equijoin queries
- **For best performance in Hive, list the largest table last in your query**

Join Syntax

- Use the following syntax* for joins

```
SELECT c.cust_id, name, total  
  FROM customers c  
 JOIN orders o ON (c.cust_id = o.cust_id);
```

- The above example is an inner join
 - Can replace JOIN with another type (such as RIGHT OUTER JOIN)
- You can sometimes use *implicit join syntax*

```
SELECT c.cust_id, name, total  
  FROM customers c, orders o  
 WHERE (c.cust_id = o.cust_id);
```

* Unless noted, all syntax in this chapter is for both Hive and Impala

Inner Join

- **Inner joins exclude records with non-matching key values**
- **For example, imagine an inner join of these tables on `cust_id`**
 - What do you think the results would be?

customers table		
cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table		
order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

Inner Join Example

customers table		
cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table		
order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total  
FROM customers c  
JOIN orders o  
ON (c.cust_id = o.cust_id);
```

Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871

Outer Joins (1)

- **Outer joins include records with non-matching key values**
- **Left outer joins**
 - Contain all records from the left (first) table
 - Contain records from the right only if they match
- **Right outer joins**
 - Contain all records from the right (second) table
 - Contain records from the left only if they match
- **Full outer joins include all records from both**

Outer Joins (2)

- Joining on `cust_id` again, which rows are excluded:
 - Using a left outer join with `customers` listed first?
 - Using a right outer join with `customers` listed first?
 - Using a full outer join with `customers` listed first?

customers table		
cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table		
order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

Left Outer Join Example

customers table		
cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table		
order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total  
FROM customers c  
LEFT OUTER JOIN orders o  
ON (c.cust_id = o.cust_id);
```

Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871
d	Dieter	NULL

Right Outer Join Example

customers table		
cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table		
order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total  
FROM customers c  
RIGHT OUTER JOIN orders o  
ON (c.cust_id = o.cust_id);
```

Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871
NULL	NULL	2137

Full Outer Join Example

customers table		
cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table		
order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total  
FROM customers c  
FULL OUTER JOIN orders o  
ON (c.cust_id = o.cust_id);
```

Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871
d	Dieter	NULL
NULL	NULL	2137

Using an Outer Join to Find Unmatched Entries

customers table		
cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table		
order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total  
FROM customers c  
FULL OUTER JOIN orders o  
ON (c.cust_id = o.cust_id)  
WHERE c.cust_id IS NULL  
OR o.total IS NULL;
```

Result of query

cust_id	name	total
d	Dieter	NULL
NULL	NULL	2137

Chapter Topics

Working with Multiple Datasets

- UNION and Joins
- Handling NULL Values in Joins
- Advanced Joins
- Essential Points
- Hands-On Exercise: Working with Multiple Datasets

NULL Values in Join Key Columns

- NULL values typically are not matched

customers_with_null table		
cust_id	name	country
a	Alice	us
...
d	Dieter	de
NULL	Unknown	NULL

orders_with_null table		
order_id	cust_id	total
1	a	1539
...
5	z	2137
6	NULL	1789

```
SELECT c.cust_id, name, total  
FROM customers_with_null c  
JOIN orders_with_null o  
ON (c.cust_id = o.cust_id);
```

Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871

NULL-Safe Join Example

- Use NULL-safe equality to match NULLs

customers_with_null table		
cust_id	name	country
a	Alice	us
...
d	Dieter	de
NULL	Unknown	NULL

orders_with_null table		
order_id	cust_id	total
1	a	1539
...
5	z	2137
6	NULL	1789

```
SELECT c.cust_id, name, total  
FROM customers_with_null c  
JOIN orders_with_null o  
ON (c.cust_id <=> o.cust_id);
```

Result of query

cust_id	name	total
a	Alice	1539
a	Alice	6352
b	Bob	1456
c	Carlos	1871
NULL	Unknown	1789

Chapter Topics

Working with Multiple Datasets

- UNION and Joins
- Handling NULL Values in Joins
- **Advanced Joins**
- Essential Points
- Hands-On Exercise: Working with Multiple Datasets



Non-Equijoin (Impala only)

- Previous joins were *equijoins*
 - Join conditions expressed as one or more equality conditions
- Impala allows *non-equijoins*
 - Join conditions expressed by inequalities
 - Unequal: != or <>
 - Less than: <, <=
 - Greater than: >, >=
 - BETWEEN
 - Join conditions expressed by other comparisons
 - IN
 - LIKE
 - REGEXP



Non-Equijoin Example Using Inequalities (1)

employees table		
fname	lname	salary
Sean	Baca	19554
Ana	Bobo	26596
Mary	Beal	53485
Gary	Burt	21191

salary_grades table		
grade	min_amt	max_amt
1	10000	19999
2	20000	29999
3	30000	39999
4	40000	49999
5	50000	59999

```
SELECT fname, lname, grade FROM
employees e
JOIN salary_grades g
ON (salary >= min_amt
    AND salary <= max_amt);
```

Or equivalently

```
SELECT fname, lname, grade FROM
employees e
JOIN salary_grades g
ON (salary BETWEEN
    min_amt AND max_amt);
```

What do you think the results will be?



Non-Equijoin Example Using Inequalities (2)

employees table		
fname	lname	salary
Sean	Baca	19554
Ana	Bobo	26596
Mary	Beal	53485
Gary	Burt	21191

salary_grades table		
grade	min_amt	max_amt
1	10000	19999
2	20000	29999
3	30000	39999
4	40000	49999
5	50000	59999

```
SELECT fname, lname, grade FROM
employees e
JOIN salary_grades g
ON (salary >= min_amt
    AND salary <= max_amt);
```

Result of query		
fname	lname	grade
Sean	Baca	1
Ana	Bobo	2
Mary	Beal	5
Gary	Burt	2



Non-Equijoin Example Using IN

employees_citizenship table		
name	citizenship	residence
Alice	us	fr
Bob	gb	us
Carlos	mx	us
Dieter	de	de

eu_countries table	
country	code
France	fr
Germany	de
United Kingdom	gb
...	...

```
SELECT * FROM employees_citizenship e JOIN eu_countries c  
ON (e.citizenship IN (c.code));
```

name	citizenship	residence	country	code
Bob	gb	us	United Kingdom	gb
Dieter	de	de	Germany	de

Cross Joins

- ***Cross joins form Cartesian products***
 - Involves no matching at all
 - Creates every possible combination
 - Potentially generates a huge amount of data
- **Pairing two disk types with four sizes creates how many rows?**

disks table
name
Internal hard disk
External hard disk

sizes table
size
1.0 terabytes
2.0 terabytes
3.0 terabytes
4.0 terabytes

Cross Joins

- ***Cross joins form Cartesian products***
 - Involves no matching at all
 - Creates every possible combination
 - Potentially generates a huge amount of data
- **Pairing two disk types with four sizes creates how many rows?**

disks table
name
Internal hard disk
External hard disk

sizes table
size
1.0 terabytes
2.0 terabytes
3.0 terabytes
4.0 terabytes

Cross Join Example

disks table

name

Internal hard disk

External hard disk

sizes table

size

1.0 terabytes

2.0 terabytes

3.0 terabytes

4.0 terabytes

```
SELECT * FROM disks  
CROSS JOIN sizes;
```

Result of query

name	size
Internal hard disk	1.0 terabytes
Internal hard disk	2.0 terabytes
Internal hard disk	3.0 terabytes
Internal hard disk	4.0 terabytes
External hard disk	1.0 terabytes
External hard disk	2.0 terabytes
External hard disk	3.0 terabytes
External hard disk	4.0 terabytes

Using Cross Join for Non-Equijoin

employees table

fname	lname	salary
Sean	Baca	19554
Ana	Bobo	26596
Mary	Beal	53485
Gary	Burt	21191

salary_grades table

grade	min_amt	max_amt
1	10000	19999
2	20000	29999
3	30000	39999
4	40000	49999
5	50000	59999

```
SELECT fname, lname, grade  
FROM employees  
CROSS JOIN salary_grades  
WHERE salary  
BETWEEN min_amt AND max_amt;
```

Result of query

fname	lname	grade
Sean	Baca	1
Ana	Bobo	2
Mary	Beal	5
Gary	Burt	2

Cross Joins without CROSS Keyword

- Cross joins also occur when you leave off join conditions
- The following all produce the same result

- Using CROSS JOIN

```
SELECT * FROM disks CROSS JOIN sizes;
```

- Using explicit join syntax and no join condition (ON)

```
SELECT * FROM disks JOIN sizes;
```

- Using implicit join syntax and no join condition (WHERE)

```
SELECT * FROM disks, sizes;
```

Left Semi-Joins (1)

- A less common type of join is the LEFT SEMI JOIN
 - It is a special (and efficient) type of inner join
 - It behaves more like a filter than a join
- Left semi-joins only return records from the table on the left
 - There must be a match in the table on the right
 - Join conditions and other criteria are specified in the ON clause

```
SELECT c.cust_id
  FROM customers c
    LEFT SEMI JOIN orders o
      ON (c.cust_id = o.cust_id AND o.total > 1500);
```

Left Semi-Joins (2)

- Use LEFT SEMI JOIN as an alternative to using a subquery
 - These two queries return the same result

```
SELECT cust_id FROM customers c
  WHERE c.cust_id IN
    (SELECT cust_id FROM orders WHERE total > 1500);
```

```
SELECT c.cust_id
  FROM customers c
  LEFT SEMI JOIN orders o
    ON (c.cust_id = o.cust_id AND o.total > 1500);
```

Chapter Topics

Working with Multiple Datasets

- UNION and Joins
- Handling NULL Values in Joins
- Advanced Joins
- **Essential Points**
- Hands-On Exercise: Working with Multiple Datasets

Essential Points

- **UNION will combine two or more datasets by appending rows**
- **Joins will combine rows by matching on a common column**
 - Inner joins only include rows from any table that have a match
 - Outer joins will include rows that do not have a match
 - Left outer joins include unmatched rows in the left (first) table
 - Right outer joins include unmatched rows in the right (second) table
 - Full outer joins include all rows
 - Non-equijoins allow inequalities for the match condition (Impala only)
 - Cross joins match each row from one table with each row from the other
- **NULL values typically are not matched**
 - Use the NULL-safe equality operator ($\text{<}=\text{>}$) to match NULLs

Chapter Topics

Working with Multiple Datasets

- UNION and Joins
- Handling NULL Values in Joins
- Advanced Joins
- Essential Points
- **Hands-On Exercise: Working with Multiple Datasets**

Hands-On Exercise: Working with Multiple Datasets

- In this exercise, you will use joins to analyze sales and product data
- Please refer to the Hands-On Exercise Manual for instructions



Analytic Functions and Windowing

Chapter 9

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing**
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Analytic Functions and Windowing

In this chapter, you will learn

- What are the fundamental differences between analytic functions and aggregate functions
- How to translate a complex query using aggregate functions and subqueries into a query using analytic functions
- How to use analytic functions that are not also aggregate functions
- What windowing is
- How to write queries that use windowing

Chapter Topics

Analytic Functions and Windowing

- **Using Analytic Functions**
- Instructor-Led Demonstration: Answering Questions with Analytic Functions
- Other Analytic Functions
- Sliding Windows
- Essential Points
- Hands-On Exercise: Analytic Functions

Aggregate and Analytic Functions

- Aggregate and *analytic functions* both use values over multiple rows
- Aggregate functions
 - Combine rows by group
 - Return one result row for each group
- Analytic functions
 - Calculate without combining
 - Return one result row for each table row
 - Supported in Hive since 0.11 (CDH 5.0)
 - Partial support in Impala 2.0 (CDH 5.2.0)
 - Full support in Impala 2.3 (CDH 5.5.0)

Analytic Function Windows

- Define sets of rows using `OVER (PARTITION BY column)`
 - Optionally, include `ORDER BY` as well
- Each set of rows is called a *window*
 - Working with them is called *windowing*
- Example: `OVER (PARTITION BY brand)`

prod_id	brand	name	price
1	Dualcore	USB Card Reader	18.39
2	Dualcore	HDMI Cable	11.99
3	Dualcore	VGA Cable	1.99
4	Gigabux	6-cell Battery	40.50
5	Gigabux	8-cell Battery	50.50
6	Gigabux	Wall Charger	20.00
7	Gigabux	Auto Charger	20.00

The diagram illustrates two windows defined by the 'brand' column. A vertical bracket on the right side of the table spans rows 1-3 and 4-7, labeled 'Window'. Another vertical bracket spans rows 4-7, also labeled 'Window'.

Example: Aggregation over a Window

- Question: What is the average price for the products in each brand?

```
SELECT prod_id, brand, price,  
       AVG(price) OVER (PARTITION BY brand) AS avg  
FROM products;
```

products table			
prod_id	brand	name	price
1	Dualcore	USB Card Reader	18.39
2	Dualcore	HDMI Cable	11.99
3	Dualcore	VGA Cable	1.99
4	Gigabux	6-cell Battery	40.50
5	Gigabux	8-cell Battery	50.50
6	Gigabux	Wall Charger	20.00
7	Gigabux	Auto Charger	20.00



Query results			
prod_id	brand	price	avg
1	Dualcore	18.39	10.79
2	Dualcore	11.99	10.79
3	Dualcore	1.99	10.79
4	Gigabux	40.50	32.75
5	Gigabux	50.50	32.75
6	Gigabux	20.00	32.75
7	Gigabux	20.00	32.75

Example: Aggregation Using GROUP BY

- Question: What is the average price of the products in each brand?

```
SELECT brand,  
       AVG(price) AS avg  
  FROM products GROUP BY brand;
```

products table			
prod_id	brand	name	price
1	Dualcore	USB Card Reader	18.39
2	Dualcore	HDMI Cable	11.99
3	Dualcore	VGA Cable	1.99
4	Gigabux	6-cell Battery	40.50
5	Gigabux	8-cell Battery	50.50
6	Gigabux	Wall Charger	20.00
7	Gigabux	Auto Charger	20.00



Query results	
brand	avg
Dualcore	10.79
Gigabux	32.75

Compare: Using Windows or Using GROUP BY

```
SELECT prod_id, brand, price,  
       AVG(price) OVER (PARTITION BY brand) AS avg  
  FROM products;
```

```
SELECT brand,  
       AVG(price) AS avg  
  FROM products  
 GROUP BY brand;
```

prod_id	brand	price	avg
1	Dualcore	18.39	10.79
2	Dualcore	11.99	10.79
3	Dualcore	1.99	10.79
4	Gigabux	40.50	32.75
5	Gigabux	50.50	32.75
6	Gigabux	20.00	32.75
7	Gigabux	20.00	32.75

brand	avg
Dualcore	10.79
Gigabux	32.75

Aggregate Functions Usable as Analytic Functions

- These common aggregate functions can be used as analytic functions
 - AVG
 - COUNT
 - MAX
 - MIN
 - SUM
- Other aggregate functions might not
 - Statistics functions such as STDDEV and VARIANCE are supported by Hive but not Impala
 - Test or check the documentation before relying on other aggregate functions

Chapter Topics

Analytic Functions and Windowing

- Using Analytic Functions
- **Instructor-Led Demonstration: Answering Questions with Analytic Functions**
- Other Analytic Functions
- Sliding Windows
- Essential Points
- Hands-On Exercise: Analytic Functions

Instructor-Led Demonstration: Answering Questions with Analytic Functions

- Your instructor will now demonstrate how to use analytic functions to answer questions about data
 - Each has many solutions, some using grouping instead of windows
 - Try to suggest an approach, even if you don't have the exact query in mind

Questions

1. What is the price of the least expensive product?
2. Which product is least expensive?
3. What is the price of the least expensive product *in each brand*?
4. What is *difference* between a product's price and the minimum price for that brand?
5. Which product is the least expensive for each brand?
6. What's the second lowest price? What's the fifth lowest?

Chapter Topics

Analytic Functions and Windowing

- Using Analytic Functions
- Instructor-Led Demonstration: Answering Questions with Analytic Functions
- **Other Analytic Functions**
- Sliding Windows
- Essential Points
- Hands-On Exercise: Analytic Functions

Ranking Analytic Functions

- Additional analytic functions only for windowing
- Use ORDER BY clause inside OVER (...) for these functions
 - OVER (PARTITION BY *col_1* ORDER BY *col_2*)

Function	Returns
ROW_NUMBER	Row number within the window (consecutive, ties are arbitrarily incremented)
RANK	Rank of current value within the window (ranks skip numbers when there are ties)
DENSE_RANK	Rank of current value within the window (with consecutive rankings)
PERCENT_RANK	Rank of current value within the window, expressed as a decimal
NTILE(n)	The n -tile (of n) within the window that the current value is in—this places the values as evenly as possible into n divisions
CUME_DIST	Cumulative distribution of current value within the window

Example: RANK and ROW_NUMBER

Question: What is the ranking of each product by price, within each brand?

```
SELECT prod_id, brand, price,  
       RANK() OVER (PARTITION BY brand ORDER BY price) AS rank,  
       ROW_NUMBER() OVER (PARTITION BY brand ORDER BY price) AS n  
  FROM products;
```

products table			
prod_id	brand	name	price
1	Dualcore	USB Card Reader	18.39
2	Dualcore	HDMI Cable	11.99
3	Dualcore	VGA Cable	1.99
4	Gigabux	6-cell Battery	40.50
5	Gigabux	8-cell Battery	50.50
6	Gigabux	Wall Charger	20.00
7	Gigabux	Auto Charger	20.00

Query results



prod_id	brand	price	rank	n
3	Dualcore	1.99	1	1
2	Dualcore	11.99	2	2
1	Dualcore	18.39	3	3
7	Gigabux	20.00	1	1
6	Gigabux	20.00	1	2
4	Gigabux	40.50	3	3
5	Gigabux	50.50	4	4

Example: RANK and DENSE_RANK

Question: What is the ranking of each product by price, within each brand?

- For demonstration, the following only shows the Gigabux brand

```
SELECT prod_id, brand, price,  
       RANK() OVER (PARTITION BY brand ORDER BY price)  
             AS rank,  
       DENSE_RANK() OVER (PARTITION BY brand ORDER BY price)  
             AS d_rank  
  FROM products  
 WHERE brand = 'Gigabux';
```

prod_id	brand	price	rank	d_rank
7	Gigabux	20.00	1	1
6	Gigabux	20.00	1	1
4	Gigabux	40.50	3	2
5	Gigabux	50.50	4	3

Example: NTILE(n)

Question: In what quartile (n=4) and 2-tile for a brand is each product's price?

```
SELECT prod_id, brand, price,  
       NTILE(4) OVER (PARTITION BY brand ORDER BY price) AS tile4,  
       NTILE(2) OVER (PARTITION BY brand ORDER BY price) AS tile2  
  FROM products;
```

prod_id	brand	price	tile4	tile2
3	Dualcore	1.99	1	1
2	Dualcore	11.99	2	1
1	Dualcore	18.39	3	2
7	Gigabux	20.00	1	1
6	Gigabux	20.00	2	1
4	Gigabux	40.50	3	2
5	Gigabux	50.50	4	2

PERCENT_RANK and CUME_DIST

- PERCENT_RANK and CUME_DIST are similar
 - $\text{PERCENT_RANK} = (\text{RANK} - 1)/(\text{number of rows} - 1)$
 - To make it a true percentage, multiply by 100
 - CUME_DIST is also a decimal
 - For ascending order: proportion of rows with values \leq current value

prod_id	brand	price	RANK	PERCENT_RANK	CUME_DIST
3	Dualcore	1.99	1	0	0.3333...
2	Dualcore	11.99	2	0.5	0.6666...
1	Dualcore	18.39	3	1	1
7	Gigabux	20.00	1	0	0.5
6	Gigabux	20.00	1	0	0.5
4	Gigabux	40.50	3	0.6666...	0.75
5	Gigabux	50.50	4	1	1

Offset Functions

Function	Returns
<code>FIRST_VALUE(column)</code>	The first value in the specified column
<code>LAST_VALUE(column)</code>	The last value in the specified column
<code>LEAD(column, n, default)*</code>	The value in the specified column in the <i>n</i> th following row
<code>LAG(column, n, default)*</code>	The value in the specified column in the <i>n</i> th preceding row

* Arguments `n` and `default` are optional.

LEAD and LAG

Question: What is the difference in price between consecutively ranked products, for each brand?

```
SELECT prod_id, brand, price,  
    price-LAG(price) OVER (PARTITION BY brand ORDER BY price) AS lag_diff,  
    price-LEAD(price) OVER (PARTITION BY brand ORDER BY price) AS lead_diff  
FROM products;
```

prod_id	brand	price	lag_diff	lead_diff
3	Dualcore	1.99	NULL	-10.00
2	Dualcore	11.99	10.00	-6.40
1	Dualcore	18.39	6.40	NULL
7	Gigabux	20.00	NULL	0.00
6	Gigabux	20.00	0.00	-20.50
4	Gigabux	40.50	20.50	-10.00
5	Gigabux	50.50	10.00	NULL

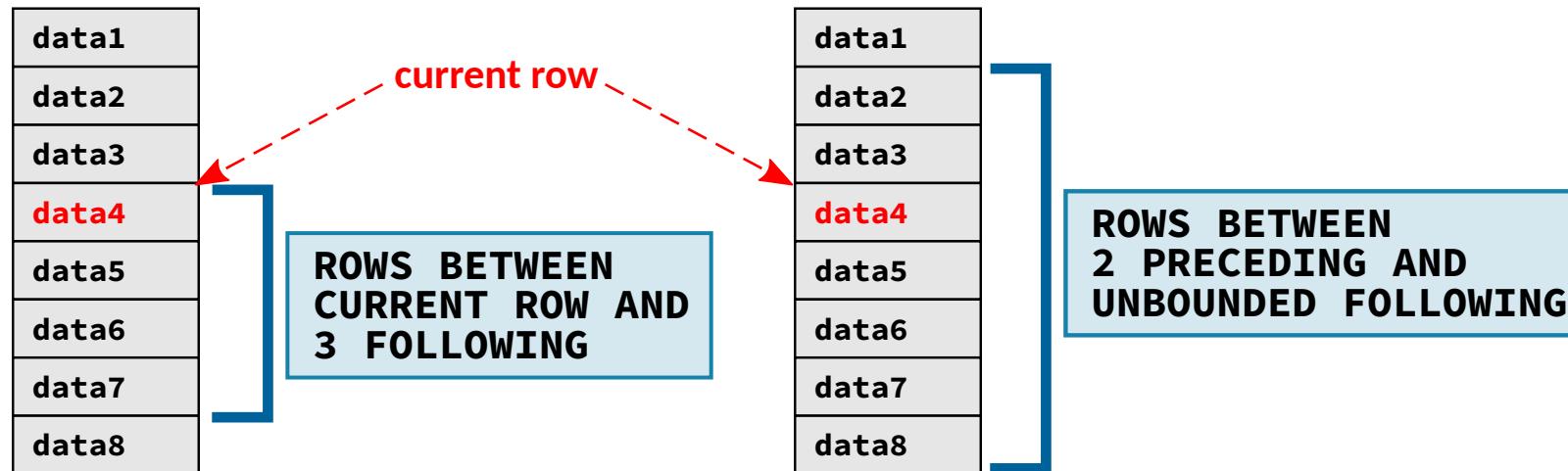
Chapter Topics

Analytic Functions and Windowing

- Using Analytic Functions
- Instructor-Led Demonstration: Answering Questions with Analytic Functions
- Other Analytic Functions
- **Sliding Windows**
- Essential Points
- Hands-On Exercise: Analytic Functions

Sliding Windows

- Some analytic functions accept a *window clause*
 - Specify a *sliding window* relative to the current row
 - Sometimes called *frame boundaries*
- Specifications for the sliding window
 - ROWS BETWEEN or RANGE BETWEEN
 - Lower bound
 - Upper bound



Window Bounds

- Specify upper bound and lower bound
- Upper and lower bounds have three options
 - CURRENT ROW
 - A number
 - UNBOUNDED
- For number or UNBOUNDED, include PRECEDING or FOLLOWING
- Examples:
 - ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING
 - ROWS BETWEEN CURRENT ROW AND 3 FOLLOWING
 - ROWS BETWEEN UNBOUNDED PRECEDING AND 3 FOLLOWING
 - ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
 - ROWS BETWEEN 2 PRECEDING AND UNBOUNDED FOLLOWING

Sliding Window Example

Question: What are the daily and three-day total number of ratings for product 1274348 in the ratings table, for ratings after April 2013?

```
SELECT this_date, daily,
       SUM(daily) OVER (ORDER BY this_date
                         ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS three_day
  FROM
    (SELECT to_date(posted) AS this_date, COUNT(rating) AS daily
       FROM ratings WHERE prod_id = 1274348
      GROUP BY this_date HAVING this_date > '2013-04-30') s;
```

this_date	daily	three_day
2013-05-01	6	6
2013-05-02	11	17
2013-05-03	13	30
2013-05-04	10	34
2013-05-05	8	31

this_date	daily	three_day
2013-05-06	14	32
2013-05-07	19	41
2013-05-08	19	52
2013-05-09	10	48
2013-05-10	16	45

*The results are split into two tables because of space limitations

Using RANGE

- Bounds must be between CURRENT ROW and UNBOUNDED (either end)
- Includes all rows with *values in the range of the sort values at the boundaries*
 - Combines rows that match the current row's ORDER BY value

```
SELECT month, amount,
       SUM(amount) OVER (ORDER BY month
                          ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS row_sum,
       SUM(amount) OVER (ORDER BY month
                          RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS range_sum
  FROM deposits;
```

id	month	amount	row_sum	range_sum
11938	12-2018	50	50	50
11940	01-2019	200	250	350
11939	01-2019	100	350	350
11941	02-2019	150	500	800
11942	02-2019	300	800	800

Supported Functions for Sliding Windows

- Supported functions

- AVG
- COUNT
- SUM
- FIRST_VALUE
- LAST_VALUE
- MAX
 - Impala only allows UNBOUNDED PRECEDING for lower bound
- MIN
 - Impala only allows UNBOUNDED PRECEDING for lower bound

Chapter Topics

Analytic Functions and Windowing

- Using Analytic Functions
- Instructor-Led Demonstration: Answering Questions with Analytic Functions
- Other Analytic Functions
- Sliding Windows
- **Essential Points**
- Hands-On Exercise: Analytic Functions

Essential Points

- **Analytic functions calculate over sets of rows without combining the rows**
 - Different from aggregate functions, which combine rows
 - Reduces the need for subqueries in some cases
- **Analytic functions include**
 - Common aggregate functions (COUNT, AVG, MAX, MIN, SUM)
 - Ranking functions (including ROW_NUMBER, RANK, NTILE(n))
 - Offset functions (FIRST_VALUE, LAST_VALUE, LEAD, LAG)
- **A window is a set of rows over which the analytic function is calculated**
 - Working with a window is called *windowing*
 - Specify using OVER (...) with optional clauses
 - PARTITION BY
 - ORDER BY
 - ROWS BETWEEN or RANGE BETWEEN

Bibliography

The following offer more information on topics discussed in this chapter

- **Hive Windowing and Analytic Functions**
 - <http://tiny.cloudera.com/hivewindow>
- **Impala Analytic Functions**
 - <http://tiny.cloudera.com/impalawindow>

Chapter Topics

Analytic Functions and Windowing

- Using Analytic Functions
- Instructor-Led Demonstration: Answering Questions with Analytic Functions
- Other Analytic Functions
- Sliding Windows
- Essential Points
- Hands-On Exercise: Analytic Functions

Hands-On Exercise: Analytic Functions

- In this exercise, you will use analytic functions to answer questions about data
- Please refer to the Hands-On Exercise Manual for instructions



Complex Data

Chapter 10

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- **Complex Data**
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Complex Data

In this chapter, you will learn

- What are the benefits of using complex data types
- How the complex data types are different from basic data types, and from each other
- What are the differences between using complex data with Apache Hive and with Apache Impala
- How to write queries for Hive and Impala that use complex data types

Chapter Topics

Complex Data

- **Complex Data with Hive**
- Complex Data with Impala
- Essential Points
- Hands-On Exercise: Complex Data

Complex Data Types

- Hive has support for complex data types
 - Represent multiple values within a single row/column 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 '|';
```

Data File

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

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)

- Another complex data type is MAP
 - Key-value pairs

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
 - $\text{MAP} < \text{KEY-TYPE}, \text{VALUE-TYPE} \rangle$
- 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 ':';
```

Data File

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

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 (1)

- A STRUCT stores a fixed number of named 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}

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

Query results

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



Using STRUCT Columns with Hive (2)

- STRUCT items have names and types

```
CREATE TABLE customers_addr
(cust_id STRING,
 name STRING,
 address STRUCT<street:STRING,
           city:STRING,
           state:STRING,
           zipcode:STRING>)
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  COLLECTION ITEMS TERMINATED BY '|';
```

Data File

```
a,Alice,742 Evergreen Terrace|Springfield|OR|97477
b,Bob,1600 Pennsylvania Ave NW|Washington|DC|20500
c,Carlos,342 Gravelpit Terrace|Bedrock
```

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}



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

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

```
SELECT phones  
FROM customers_phones;
```

Query results

phones
[555-1111, 555-2222, 555-3333]
[555-4444]
[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	Carlos	[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;
```

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

Chapter Topics

Complex Data

- Complex Data with Hive
- **Complex Data with Impala**
- Essential Points
- Hands-On Exercise: Complex Data

Complex Data with Impala

Impala provides limited support for ARRAY, MAP, and STRUCT types:

- **Tables using these types must be Parquet tables**
- **CREATE TABLE syntax for complex types is almost the same as HiveQL**
 - The collection item and map key terminators cannot be specified
- **SELECT syntax for complex types is different from HiveQL**
 - Does not use lateral views or table-generating functions like `explode`
 - Does not use collection functions like `size`
- **Impala cannot INSERT complex data into a table**
 - Use Hive to INSERT data into a Parquet table
 - Or use existing Parquet files created by another process



Querying ARRAY Columns with Impala (1)

- Impala lets you reference an ARRAY column as if it were a separate table
 - With one row for each ARRAY element
 - Containing the pseudocolumns `item` and `pos`

```
SELECT item, pos  
FROM cust_phones_parquet.phones;
```

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

Query results	
item	pos
555-1111	0
555-2222	1
555-3333	2
555-4444	0
555-5555	0
555-6666	1



Querying ARRAY Columns with Impala (2)

- Use implicit join notation to join ARRAY elements with rows of the table
 - Returns ARRAY elements with scalar column values from same rows

```
SELECT name, phones.item AS phone  
FROM cust_phones_parquet, cust_phones_parquet.phones;
```

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

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



Querying MAP Columns with Impala (1)

- You can reference a MAP column in Impala as if it were a separate table
 - With one row for each MAP element
 - Containing the pseudocolumns key and value

```
SELECT key, value  
FROM cust_phones_parquet.phones;
```

cust_phones_parquet 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}

Query results	
key	value
home	555-1111
work	555-2222
mobile	555-3333
mobile	555-4444
work	555-5555
home	555-6666



Querying MAP Columns with Impala (2)

- Use implicit join notation to join MAP elements with rows of the table
 - Can use key and value in the SELECT list and WHERE clause

```
SELECT name, phones.value AS home  
  FROM cust_phones_parquet, cust_phones_parquet.phones  
 WHERE phones.key = 'home';
```

cust_phones_parquet 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 }

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



Querying STRUCT Columns with Impala

- The Impala query syntax for STRUCT columns is the same as with Hive

cust_addr_parquet 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}

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

Query results

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



Complex Columns in the SELECT List in Impala Queries

- In Impala queries, complex columns are not allowed in the SELECT list



```
SELECT phones FROM cust_phones_parquet;
```

- You can issue SELECT * queries on tables with complex columns
 - But Impala omits the complex columns from the results

```
SELECT * FROM cust_phones_parquet;
```

cust_phones_parquet table

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

Query results

cust_id	name
a	Alice
b	Bob
c	Carlos

Returning the Number of Items in a Collection with Impala



- Impala does not support the `size` function or other collection functions
 - Use `COUNT` and `GROUP BY` to count the items in an `ARRAY` or `MAP`

```
SELECT name, COUNT(*) AS num
  FROM cust_phones_parquet, cust_phones_parquet.phones
 GROUP BY cust_id, name;
```

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

Query results	
name	phone
Alice	3
Bob	1
Carlos	2

Loading Data Containing Complex Types

- Impala supports querying complex types only in Parquet tables
- Impala cannot **INSERT** data containing complex types
- Workaround: Use Hive to **INSERT** data into Parquet 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  
STORED AS PARQUET  
AS  
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;
```

Querying the Denormalized Table

- You can query the table efficiently in Impala
- Example: Show the 30 highest-rated products with number of ratings and average rating*

```
SELECT brand, name,
       COUNT(ratings.rating) num_ratings,
       AVG(ratings.rating) average_rating
  FROM rated_products, rated_products.ratings
 GROUP BY brand, name
 ORDER BY nvl(average_rating, 0) DESC
 LIMIT 30;
```

*The `nvl` function in the `ORDER BY` clause returns the first argument (`average_rating`) if it is not `NULL` and the second argument (`0`) if the first is `NULL`

Chapter Topics

Complex Data

- Complex Data with Hive
- Complex Data with Impala
- **Essential Points**
- Hands-On Exercise: Complex Data

Essential Points

- **Complex types support efficient storage and querying of nested structures**
 - Represent multiple values within a single row/column position
- **Hive and Impala provide capabilities for working with complex data**
 - ARRAY stores a list of ordered elements
 - MAP stores a set of key-value pairs
 - STRUCT stores a set of named values
- **Impala support for complex types is limited**
 - Only with Parquet tables
 - No INSERT capability
- **Hive and Impala use different query syntax for ARRAY and MAP types**

Chapter Topics

Complex Data

- Complex Data with Hive
- Complex Data with Impala
- Essential Points
- **Hands-On Exercise: Complex Data**

Hands-On Exercise: Complex Data

- In this exercise, you will load and query data with complex columns related to a customer loyalty program
- Please refer to the Hands-On Exercise Manual for instructions



Analyzing Text

Chapter 11

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- **Analyzing Text**
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Analyzing Text

In this chapter, you will learn

- How to use functions for working with text, including using Regular Expressions
- What is the role of SerDes in Hive
- How to employ SerDes to process semi-structured and unstructured data in Hive
- How to analyze text data using special functions for Hive

Chapter Topics

Analyzing Text

- **Using Regular Expressions with Hive and Impala**
- Processing Text Data with SerDes in Hive
- Sentiment Analysis and n-grams in Hive
- Essential Points
- Hands-On Exercise: Analyzing Text with Hive

Matching Patterns in Text

- Hive and Impala support the LIKE operator
 - Basic pattern matching with wildcards (%)

```
SELECT * FROM products WHERE brand LIKE 'Dual%';
```

- LIKE is not flexible enough for sophisticated pattern matching
 - Regular expressions are more powerful

Regular Expression Matching

- The REGEXP operator performs regular expression (regex) matching
 - Returns true if the text matches the pattern

```
SELECT * FROM products WHERE brand REGEXP '^Dual';
```

Regular Expression	Description of Matching String
Dualcore	Contains the literal string Dualcore
^Dual	Starts with Dual
core\$	Ends with core
^Dualcore\$	Is the literal string Dualcore with no other characters
^Dual.*\$	Is Dual followed by zero or more other characters
^[A-Za-z]+\$	Is one or more uppercase or lowercase letters
^\w{8}\$	Is exactly eight word characters ([0-9A-Za-z_])
^\w{5,9}\$	Is between five and nine word characters (inclusive)

Regular Expression Case Sensitivity

- Regular expression matching is case-sensitive in Hive and Impala
 - Apply `lower` or `upper` to normalize the case for case-insensitive comparison

```
SELECT * FROM products  
WHERE lower(brand) REGEXP '^dual';
```

- Or use the Impala-only operator `IREGEXP` in CDH 5.7 and higher



```
SELECT * FROM products  
WHERE brand IREGEXP '^dual';
```

Extracting and Replacing Text

- Hive and Impala include basic functions for extracting or replacing text
 - Including `substring` and `translate`
- These examples assume that `txt` has the following value
 - It's on Oak St. or Maple St in 90210

```
SELECT substring(txt, 32, 5) FROM message;  
90210
```

```
SELECT translate(txt, '.', '') FROM message;  
It's on Oak St or Maple St in 90210
```

- These are not flexible enough for working with free-form text fields
 - Instead use regular expressions

Regular Expression Extract and Replace Functions

- Hive and Impala have functions for regular expression extract and replace
 - `regexp_extract` returns matched text
 - `regexp_replace` substitutes another value for matched text
- These examples assume that `txt` has the following value^{*}
 - It's on Oak St. or Maple St in 90210

```
SELECT regexp_extract(txt, '(\d{5})', 1)
  FROM message;
90210
```

```
SELECT regexp_replace(txt, 'St\\\\.?\\s+', 'Street ')
  FROM message;
It's on Oak Street or Maple Street in 90210
```

^{*}Single quotes in strings, like the one in `It's`, can cause problems. Use double quotes around the string; in Impala (before 2.12.0), also escape the single quote with a backslash: "`It\'s`"

Matched Portions

- Regular expressions can also be used to extract or replace matched text
 - REGEXP operator returns true if the pattern is *somewhere* in the full text
 - When extracting or replacing, the exact portion being matched is important

Regular Expression	String (matched portion in blue)
Dualcore	I wish Dualcore had 2 stores in 90210.
\d	I wish Dualcore had 2 stores in 90210.
\d{5}	I wish Dualcore had 2 stores in 90210.
\d\s\w+	I wish Dualcore had 2 stores in 90210.
\w{5,9}	I wish Dualcore had 2 stores in 90210.
.?\\".	I wish Dualcore had 2 stores in 90210.
.*\".	I wish Dualcore had 2 stores in 90210.
2[^]	I wish Dualcore had 2 stores in 90210.

Chapter Topics

Analyzing Text

- Using Regular Expressions with Hive and Impala
- Processing Text Data with SerDes in Hive
- Sentiment Analysis and n-grams in Hive
- Essential Points
- Hands-On Exercise: Analyzing Text with Hive

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" =
    "([^\n]*) ([^\n]*) ([^\n]*) ([^\n]*) \"([^\"]*)\"");
```

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

10309296107596201608290122150	akland	CA94618
-------------------------------	--------	---------

Below the table, a horizontal line is divided into seven segments by vertical tick marks. Below each segment is a label: cust_id, order_id, order_dt, order_tm, city, state, and zip.

10309296107596201608290122150akland CA94618

cust_id order_id order_dt order_tm city state zip

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



Fixed-Width Format Example

10309296107596201608290122150oakland

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	city	state	zip
1030929	6107596	20160829	012215	Oakland	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 a SerDe for processing CSV data
 - OpenCSVSerde available in CDH 5.4 and later
 - 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

Analyzing Text

- Using Regular Expressions with Hive and Impala
- Processing Text Data with SerDes in Hive
- **Sentiment Analysis and n-grams in Hive**
- Essential Points
- Hands-On Exercise: Analyzing Text with Hive

Sentiment Analysis

- **Sentiment analysis is an application of text analytics**
 - Classification and measurement of opinions
 - Frequently used for social media analysis
- **Context is essential for human languages**
 - Which word combinations appear together?
 - How frequently do these combinations appear?
- **Hive offers functions that help answer these questions**



Parsing Sentences into Words

- Hive's sentences function parses supplied text into words
- Input is a string containing one or more sentences
- Output is an ARRAY holding ARRAYS of STRINGS
 - Outer array contains one element per sentence
 - Inner array contains one element per word in that sentence

```
SELECT txt FROM phrases WHERE id=12345;  
I bought this computer and I love it. It's super fast.
```

```
SELECT sentences(txt) FROM phrases WHERE id=12345;  
[["I","bought","this","computer","and","I","love","it"],  
 ["It's","super","fast"]]
```



n-grams

- An **n-gram** is a word combination ($n=\text{number of words}$)
 - Bigram is a sequence of two words ($n=2$)
- **n-gram frequency analysis is an important step in many applications**
 - Suggesting spelling corrections in a search query
 - Finding the most important topics in a body of text
 - Identifying trending topics in social media messages



Calculating n -grams in Hive (1)

- Hive offers the `ngrams` function for calculating n -grams
- The function requires three input parameters
 - ARRAY of strings (sentences), each containing an ARRAY (words)
 - Number of words in each n -gram
 - Desired number of results (top-N, based on frequency)
- Output is an ARRAY of STRUCT with two fields
 - `ngram`: the n -gram itself (an ARRAY of words)
 - `estfrequency`: estimated frequency at which this n -gram appears



Calculating *n*-grams in Hive (2)

- The `ngrams` function is often used with the `sentences` function
 - Use `lower` to normalize case
 - Use `explode` to convert the resulting `ARRAY` to a set of rows

```
> SELECT txt FROM phrases;  
I bought this computer and I love it. It's super fast.  
New computer has arrived. I just started it. It's nice!  
This new computer is expensive, but I love it.  
I can't believe her new computer failed already.  
  
> SELECT explode(ngrams(sentences(lower(txt)), 2, 4))  
    FROM phrases;  
{"ngram": ["new", "computer"], "estfrequency": 3.0}  
{"ngram": ["i", "love"], "estfrequency": 2.0}  
 {"ngram": ["it", "it's"], "estfrequency": 2.0}  
 {"ngram": ["love", "it"], "estfrequency": 2.0}
```



Finding Specific *n*-grams in Text

- **context_ngrams** is similar, but considers only specific combinations
 - Additional input parameter: ARRAY of words used for filtering
 - Any NULL values in the ARRAY are treated as placeholders

```
> SELECT txt FROM phrases
    WHERE lower(txt) LIKE '%new computer%';
New computer has arrived. I just started it. It's nice!
This new computer is expensive, but I need it now.
I can't believe her new computer failed already.

> SELECT explode(context_ngrams(sentences(lower(txt)),
    ARRAY("new", "computer", NULL, NULL), 3))
    FROM phrases;
>{"ngram": ["has", "arrived"], "estfrequency": 1.0}
>{"ngram": ["failed", "already"], "estfrequency": 1.0}
>{"ngram": ["is", "expensive"], "estfrequency": 1.0}
```



Other Functions for Text

- Hive provides other functions that are useful for working with text
- The functions `split` and `explode` work together to generate individual rows for each word in a string

```
SELECT names FROM people;  
Amy, Sam, Ted
```

```
SELECT split(names, ',') FROM people;  
["Amy", "Sam", "Ted"]
```

```
SELECT explode(split(names, ',')) FROM people;  
Amy  
Sam  
Ted
```

Chapter Topics

Analyzing Text

- Using Regular Expressions with Hive and Impala
- Processing Text Data with SerDes in Hive
- Sentiment Analysis and n-grams in Hive
- **Essential Points**
- Hands-On Exercise: Analyzing Text with Hive

Essential Points

- **Hive and Impala support regular expressions for working with strings**
 - Compare
 - Extract
 - Replace
- **Hive uses SerDes to read and write data**
 - Specified (or defaulted) when creating a table
 - Enable Hive to process unstructured or semi-structured text data
 - RegexSerDe supports data lacking consistent delimiters
 - OpenCSVSerDe supports data in CSV format
- **Hive provides commands for working with *n*-grams**
 - Use ngrams and context_ngrams to find *n*-grams and their frequencies

Chapter Topics

Analyzing Text

- Using Regular Expressions with Hive and Impala
- Processing Text Data with SerDes in Hive
- Sentiment Analysis and n-grams in Hive
- Essential Points
- **Hands-On Exercise: Analyzing Text with Hive**

Hands-On Exercise: Analyzing Text with Hive

- In this exercise, you will use Hive to process and analyze web server log data and to analyze product ratings
- Please refer to the Hands-On Exercise Manual for instructions



Apache Hive Optimization

Chapter 12

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization**
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Apache Hive Optimization

In this chapter, you will learn

- How the parts of the Hive query process interact
- How to use execution plans and job details to target parts of a query to optimize
- When Hive on Spark is a good choice
- How to use bucketing to divide data for sampling purposes

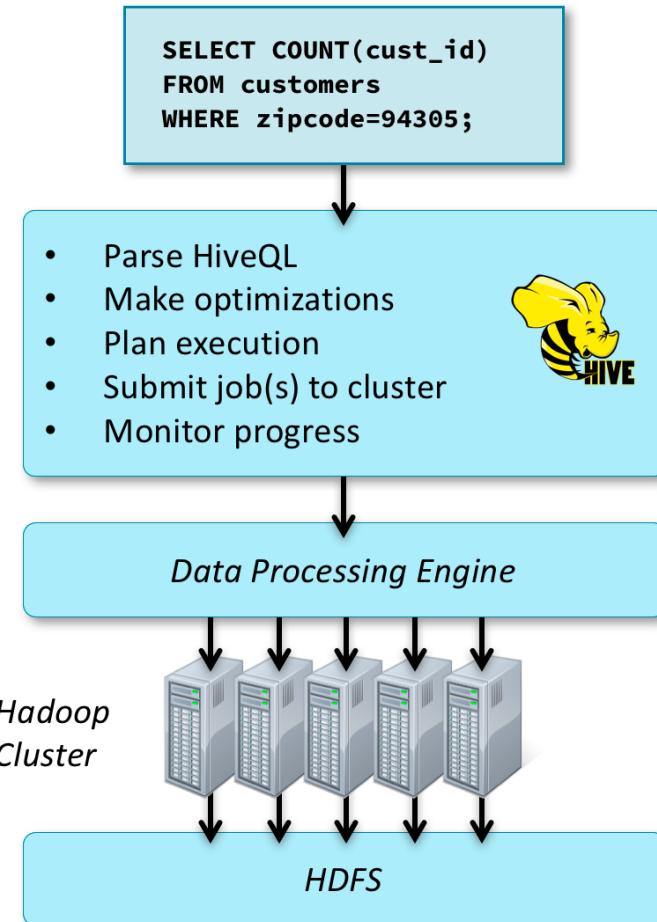
Chapter Topics

Apache Hive Optimization

- **Understanding Query Performance**
- Bucketing
- Hive on Spark
- Essential Points
- Hands-On Exercise: Hive Optimization

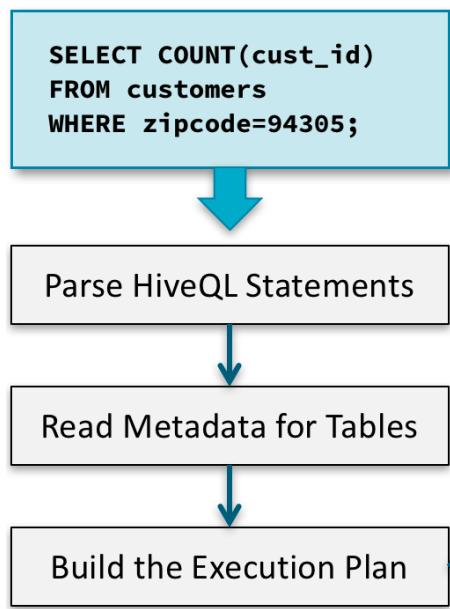
Hive Query Processing

- Recall that Hive generates jobs that are then executed by the underlying data processing engine
 - MapReduce or Spark
- To optimize Hive queries, you need to understand how queries are processed



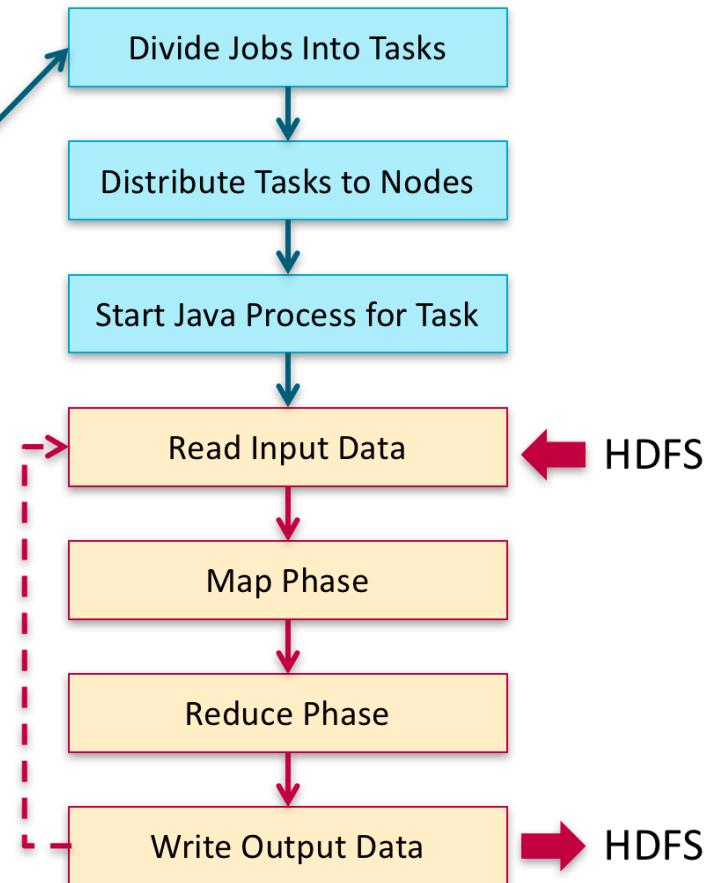
How Hive on MapReduce Processes Data

Steps Run by Hive Server



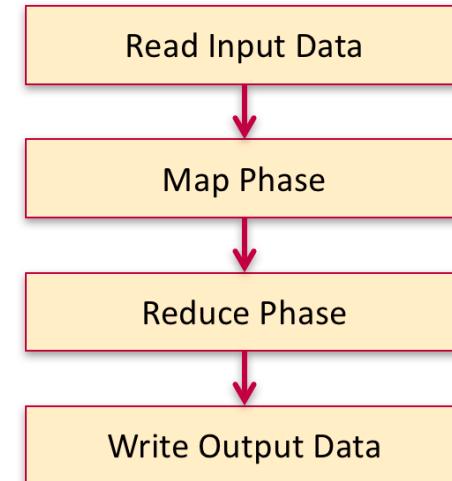
Submit Jobs to Hadoop Cluster

Steps Run on Hadoop Cluster



Understanding Map and Reduce

- A MapReduce job consists of two phases: map and reduce
 - The output from map becomes the input to reduce
- The *map phase* runs first
 - Used to filter, transform, or parse data
 - Each row is processed one at a time
- The *reduce phase* runs for some jobs
 - Used to summarize data from the map phase
 - Aggregates multiple rows
 - Not always needed—some jobs are *map-only*



MapReduce Example

- The following slides show how a query executes as a MapReduce job
 - Example query:

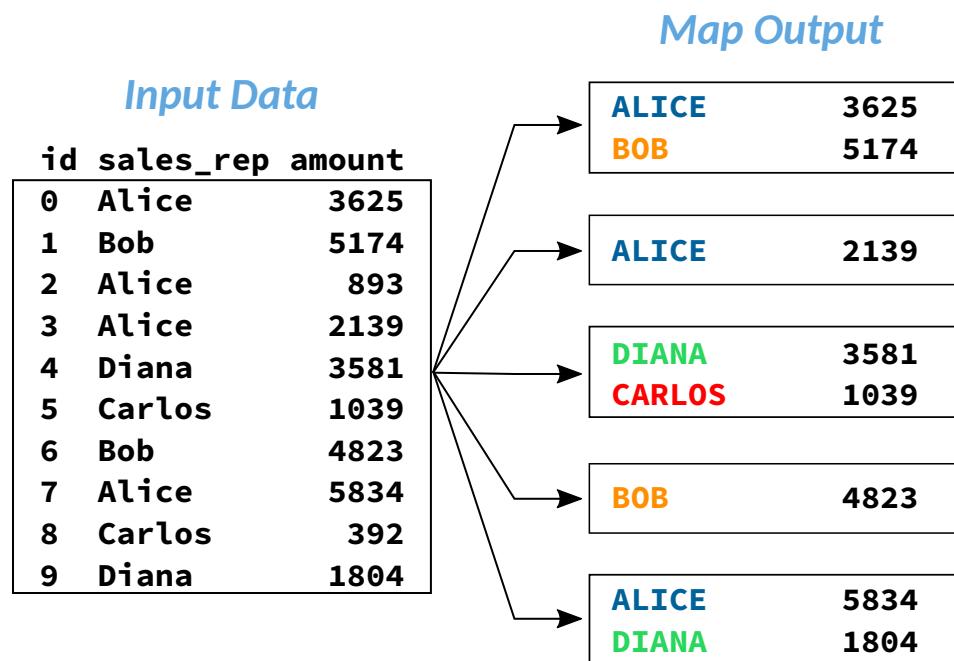
```
SELECT upper(sales_rep), SUM(amount) AS high_sales
  FROM order_info
 WHERE amount > 1000
 GROUP BY upper(sales_rep);
```

Input Data

	id	sales_rep	amount
0	Alice	3625	
1	Bob	5174	
2	Alice	893	
3	Alice	2139	
4	Diana	3581	
5	Carlos	1039	
6	Bob	4823	
7	Alice	5834	
8	Carlos	392	
9	Diana	1804	

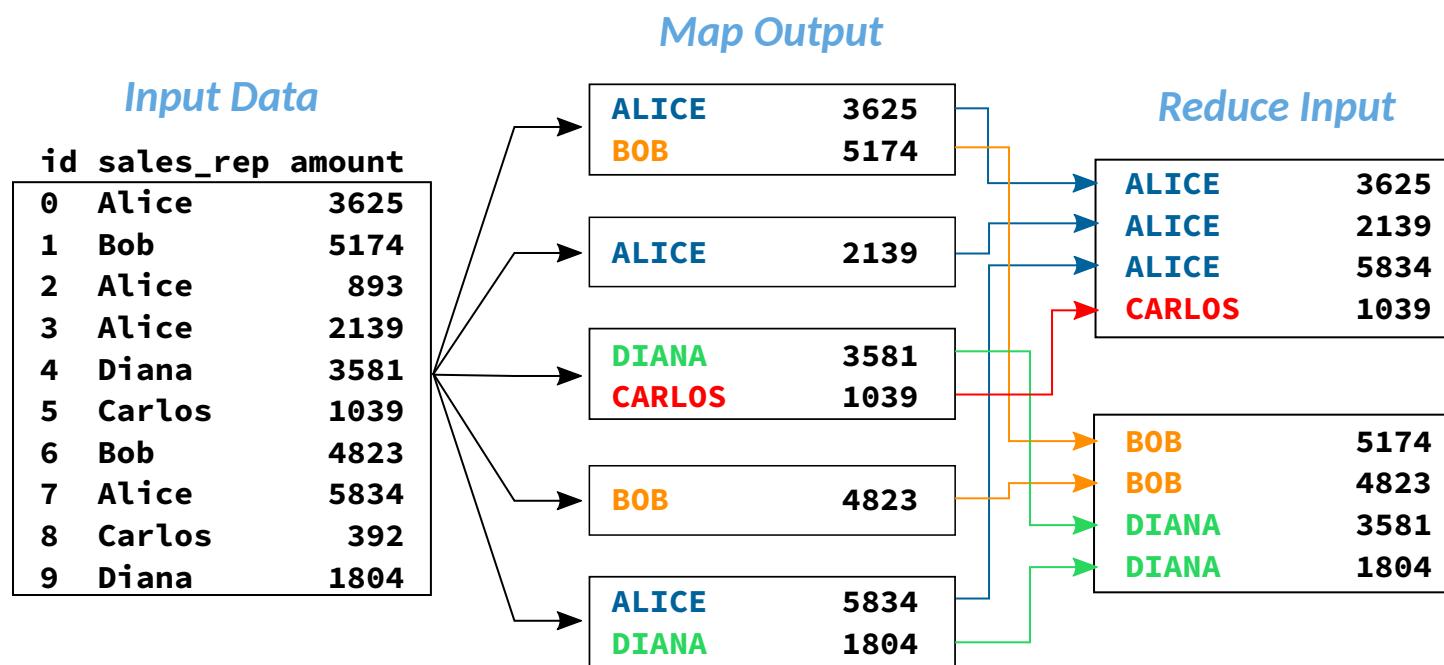
Map Phase

- Hadoop splits the job into many individual *map tasks*
 - Number of map tasks is determined by how much input data exists
 - Each map task receives a portion of the overall job input to process
- In this example, each map task selects records with amount > 1000
 - And then emits the name in upper case and the amount field for each record as output



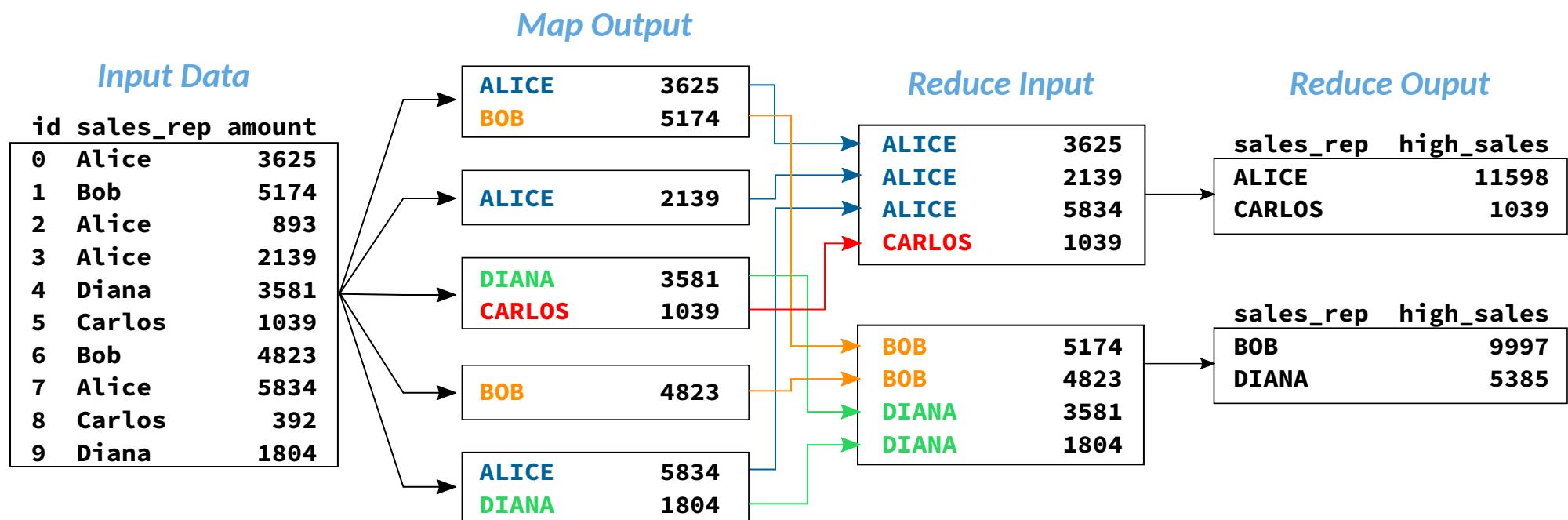
Shuffle and Sort

- Hadoop automatically sorts and merges output from all map tasks
 - This intermediate process is known as *shuffle and sort*
 - The result is the input to the reduce phase
 - In this example, the data is sorted by sales rep name, because that is how our query aggregates the data



Reduce Phase

- Input to the reduce phase comes from the shuffle and sort process
 - Reduce tasks process multiple records
 - In this example, each reduce task processes all the records with the same sales rep name
 - The reduce task aggregates by computing the sum of all the order amounts for the grouped rows





Hive Fetch Task

- Not all **SELECT queries in Hive execute as MapReduce jobs**
 - Hive executes simple queries as *fetch tasks*
 - The Hive server fetches data directly from HDFS and processes it
 - Avoids overhead of starting a MapReduce job
 - Reduces query latency
- To execute as a fetch task, a **SELECT statement must have**
 - No DISTINCT
 - No aggregation or windowing
 - No joins
 - Input data smaller than 1GB
- You can change these requirements using **Hive configuration properties**



Hive Query Performance Patterns (1)

- The fastest type of query involves only metadata

```
DESCRIBE customers;
```



- The next fastest executes as a fetch task

```
SELECT * FROM customers LIMIT 10;
```



- Then the type of query that requires only a map phase

```
INSERT INTO TABLE ny_customers  
SELECT * FROM customers  
WHERE state = 'NY';
```





Hive Query Performance Patterns (2)

- The next slowest type of query requires both map and reduce phases

```
SELECT COUNT(cust_id)
  FROM customers
 WHERE zipcode=94305;
```



- The slowest type of query requires multiple map and reduce phases

```
SELECT zipcode, COUNT(cust_id) AS num
  FROM customers
 GROUP BY zipcode
 ORDER BY num DESC
 LIMIT 10;
```



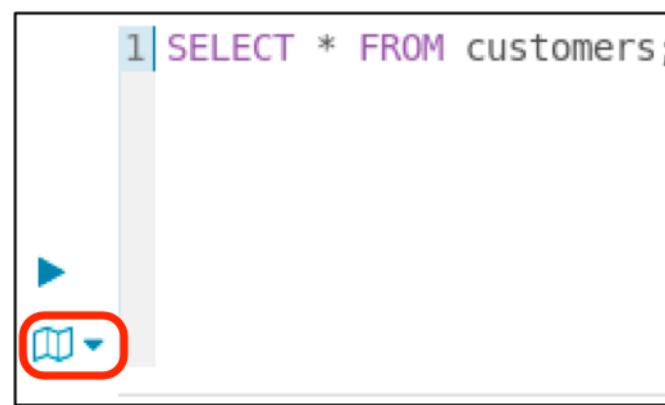


Viewing the Execution Plan

- How can you tell how Hive will execute a query?
 - Does it read only metadata?
 - Can it execute the query as a fetch task?
 - Will it require a reduce phase or multiple MapReduce jobs?
- To view Hive's execution plan, prefix your query with EXPLAIN or use the Explain button in Hue

```
EXPLAIN SELECT *  
FROM customers;
```

```
1| SELECT * FROM customers;
```



The screenshot shows a query editor window with a light gray background. On the left, there is a code editor containing the SQL command 'EXPLAIN SELECT * FROM customers;'. On the right, there is a results pane showing the query number '1' and the query itself: 'SELECT * FROM customers;'. At the bottom left of the editor area, there is a small icon of a blue book with a downward arrow pointing from it, which is the 'Explain' button. This button is highlighted with a red oval.

- The output of EXPLAIN can be long and complex
 - Fully understanding it requires in-depth knowledge of MapReduce
 - We will cover the basics here



Viewing a Query Plan with EXPLAIN (1)

- The query plan contains two main parts
 - Dependencies between stages
 - Description of the stages

```
> EXPLAIN CREATE TABLE cust_by_zip AS  
    SELECT zipcode, COUNT(cust_id) AS num  
    FROM customers GROUP BY zipcode;
```

STAGE DEPENDENCIES:

... (excerpt shown on next slide)

STAGE PLANS:

... (excerpt shown on upcoming slide)



Viewing a Query Plan with EXPLAIN (2)

- Our query has four stages
- Dependencies define order
 1. Stage-1 (first)
 2. Stage-0
 3. Stage-3
 4. Stage-2 (last)

STAGE DEPENDENCIES:

Stage-1 is a root stage
Stage-0 depends on stages: Stage-1
Stage-3 depends on stages: Stage-0
Stage-2 depends on stages: Stage-3

STAGE PLANS:

... (shown on next slide)



Viewing a Query Plan with EXPLAIN (3)

- Stage-1: MapReduce job
- Map phase
 - Read `customers` table
 - Select `zipcode` and `cust_id` columns
- Reduce phase
 - Group by `zipcode`
 - Count `cust_id`

STAGE PLANS:

Stage: Stage-1
Map Reduce

Map Operator Tree:

TableScan
alias: `customers`
Select Operator
`zipcode, cust_id`

Reduce Operator Tree:

Group By Operator
aggregations:
`count(cust_id)`
keys:
`zipcode`



Viewing a Query Plan with EXPLAIN (4)

- **Stage-0: HDFS action**
 - Move previous stage's output to Hive's warehouse directory

STAGE PLANS:

Stage: Stage-1 (covered earlier)...

Stage: Stage-0

Move Operator
files:

hdfs directory: true
destination: (HDFS path...)



Viewing a Query Plan with EXPLAIN (5)

- **Stage-3: Metastore action**
 - Create new table
 - Has two columns
- **Stage-2: Collect statistics**

STAGE PLANS:

Stage: Stage-1 (covered earlier) ...

Stage: Stage-0 (covered earlier) ...

Stage: Stage-3

Create Table Operator:

Create Table

columns: `zipcode` string,

`num` bigint

name: default.`cust_by_zip`

Stage: Stage-2

Stats-Aggr Operator



Viewing a Job in Hue (1)

- The Hue Job Browser displays running and recent jobs

The screenshot shows the Hue Job Browser interface. At the top, there is a search bar labeled "Search data and saved documents...". To the right of the search bar, there is a "Jobs" button with a red circle around it containing the number "1", and a "Job Mini Browser (pop-up)" button. Below the search bar, there are tabs for "Job Browser", "Jobs", "Workflows", and "Schemas". The "Job Browser" tab is active and highlighted with a blue box. On the left, there is a sidebar with a user dropdown set to "user:training" and filter buttons for "Succeeded", "Running", and "Pending". The main area displays a table of jobs:

Id	Name	User	Type	Status	Progress	Group
<input checked="" type="checkbox"/> application_1512414608093_0040	SELECT brand, SUM(price) AS total FROM ...10(Stage-15)	training	MAPREDUCE	RUNNING	0%	default
<input type="checkbox"/> application_1512414608093_0039	SELECT brand, SUM(price) AS total FROM ...10(Stage-5)	training	MAPREDUCE	SUCCEEDED	100%	default
<input type="checkbox"/> application_1512414608093_0038	SELECT brand, SUM(price) AS total FROM ...10(Stage-4)	training	MAPREDUCE	SUCCEEDED	100%	default
<input type="checkbox"/> application_1512414608093_0037	SELECT brand, SUM(price) AS total FROM ...10(Stage-9)	training	MAPREDUCE	SUCCEEDED	100%	default
<input type="checkbox"/> application_1512414608093_0036	SELECT brand, SUM(price) AS total FROM ...10(Stage-8)	training	MAPREDUCE	SUCCEEDED	100%	default

A "Job Mini Browser (pop-up)" window is open for the first job, showing its details. The window has a close button ("x Kill") and a "Kill" button.



Viewing a Job in Hue (2)

- View job details including tasks, metadata, and counters

The screenshot shows the Hue Job Browser interface for a completed MapReduce job. The job ID is job_15124005... and the name is SELECT term, COUNT(term) AS num FROM ...3(Stage-1). The job type is MAPREDUCE, status is SUCCEEDED, and user is training. The progress bar shows 100% completion for both MAP and REDUCE stages.

The Tasks tab is selected, displaying a table of task details:

Type	Id	Elapsed Time	Progress	State	Start Time
MAP	task_1512400517273_0003_m_000000	17.60s	100	SUCCEEDED	December 4, 2017 8:09 AM

A modal window is open over the Tasks table, showing the Counter details for the org.apache.hadoop.mapreduce.FileSystemCounter category:

Name	Maps total	Reduces total	Total
FILE_BYTES_READ	0	2220	2220
FILE_BYTES_WRITTEN	281906	281851	563757
FILE_READ_OPS	0	0	0
FILE_LARGE_READ_OPS	0	0	0
FILE_WRITE_OPS	0	0	0
HDFS_BYTES_READ	106344035	3570	106347605
HDFS_BYTES_WRITTEN	0	2754	2754
HDFS_READ_OPS	3	3	6
HDFS_LARGE_READ_OPS	0	0	0



Hive Server Web UI

- The Hive server provides a web UI
 - At `http://hostname:10002/hiveserver2.jsp`
 - Displays query plan and performance information

HiveServer2

Active Sessions

User Name	IP Address	Operation Count
training	127.0.0.1	54
training	127.0.0.1	0

Total number of sessions: 2

Open Queries

User Name	Query	Execution Engine	State	Opened Timestamp	Opened (s)	Latency (s)	Drilldown Link
training	CREATE FUNCTION url_decode AS 'stewi.hive.udf.URLDecode' USING JAR 'hdfs:/stewi-hive-udfs-1.0.2.jar'	mr	FINISHED	Tue Dec 05 07:58:11 PST 2017	1487524	4	Drilldown

Query Information: CREATE FUNCTION url_decode AS 'stewi.hive.udf.URLDecode' USING JAR 'hdfs:/stewi-hive-udfs-1.0.2.jar'

Base Profile	Stages	Query Plan	Performance Logging
User Name	training		
Query String		CREATE FUNCTION url_decode AS 'stewi.hive.udf.URLDecode' USING JAR 'hdfs:/stewi-hive-udfs-1.0.2.jar'	
Id		hive_20171205075858_199084e8-355b-4410-bc8a-994c8d4c2d12	
Execution Engine	mr		
State	FINISHED		
Opened Timestamp	Tue Dec 05 07:58:11 PST 2017		

CLOUDERA
Educational Services

Copyright © 2010–2019 Cloudera. All rights reserved. Not to be reproduced or shared without prior written consent from Cloudera.

12-23

The Hadoop Web UI

- You can also find information about Hive jobs in the Hadoop web UI
- The ResourceManager offers the most useful of Hadoop's web UIs
 - Access at `http://hostname:8088/`
 - In pseudo-distributed mode, the hostname is `localhost`
 - Includes job status information for the Hadoop cluster

Cluster Metrics							
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Mem	Mem
3	0	1	2	1	1 GB	4 GB	4 GB
Cluster Nodes Metrics							
Active Nodes		Decommissioning Nodes		Decommissioned Nodes		Lost Nodes	
1	0	0		0		0	
User Metrics for dr.who							
Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Mem
0	0	0	0	0	0	0	0 B
Scheduler Metrics							
Scheduler Type		Scheduling Resource Type			Minimum Allocation		
Fair Scheduler		[memory-mb (unit=Mi), vcores]			<memory:1024, vCores:1>		
Show 20 ▾ entries							
ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime
application_1544192593675_0002	training	SELECT AVG(order_profit) AS avg_profit_p...x (Stage-10)	MAPREDUCE	root.users.training	0	Fri Dec 7 08:28:25 -0800 2018	Fri Dec 7 08:29:04 -0800 2018
application_1544192593675_0001	training	SELECT AVG(order_profit)	MAPREDUCE	root.users.training	0	Fri Dec 7 08:28:03	Fri Dec 7 08:28:09



Parallel Execution

- Stages in Hive's execution plan often lack dependencies
- Hive supports parallel execution in such cases
 - However, this feature is disabled by default
- Enable this by setting the `hive.exec.parallel` property to `true`



Using Hive with Hadoop Standalone Mode

- **Running MapReduce jobs on the cluster has significant overhead**
 - Must divide work, assign tasks, start processes, collect results, and so on
 - Required to process large amounts of data in Hive
 - Possibly inefficient with small amount of data
- **Processing data in *standalone mode* can speed up smaller jobs**
 - Also called *local mode*

```
SET mapreduce.framework.name=local;
```

- **Runs the job in a single Java Virtual Machine (JVM) on the Hive server**
 - Significantly reduces query latency
 - But only appropriate to use with small amounts of data

Chapter Topics

Apache Hive Optimization

- Understanding Query Performance
- **Bucketing**
- Hive on Spark
- Essential Points
- Hands-On Exercise: Hive Optimization



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
- **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 codes to assign records to a bucket
- **Goal: Distribute rows across a predefined number of buckets**
 - Useful for jobs that need samples of data
 - Joins may be faster if all tables are bucketed on the join column



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);
```

- It is possible to use `TABLESAMPLE` on a non-bucketed table
 - However, this requires a full scan of the table

Chapter Topics

Apache Hive Optimization

- Understanding Query Performance
- Bucketing
- **Hive on Spark**
- Essential Points
- Hands-On Exercise: Hive Optimization

MapReduce and Spark

- MapReduce used to be the exclusive execution engine for Hadoop data tools
- MapReduce has excellent reliability and scalability
- But MapReduce has performance disadvantages
 - High latency even with small amounts of data
 - Speed limited by frequent disk reads and writes
- Apache Spark has emerged as a successor to MapReduce
 - General-purpose data processing engine
 - Uses in-memory processing for faster performance



Hive on Spark

- ***Hive on Spark uses Spark instead of MapReduce as Hive's execution engine***
 - Available in CDH 5.7 and higher
 - Fully compatible with existing Hive queries
 - Typically three times faster than Hive on MapReduce
- **Hive's execution engine is configurable using `hive.execution.engine`**
 - For Hive on MapReduce, set to `mr` (the default value)

```
SET hive.execution.engine=mr;
```
 - For Hive on Spark, set to `spark`

```
SET hive.execution.engine=spark;
```
- **Expect a long delay as Spark initializes after you submit the first query**
 - Subsequent queries run without a delay
- **Hive on Spark requires more memory on cluster than Hive on MapReduce**
 - Uses memory even when queries are not running



When to Use Hive on Spark

- **Hive on Spark is a good choice for**
 - Complex, slow Hive queries that are incompatible with Impala
 - For example, queries for which fault tolerance is essential
 - Data preparation and extract, transform, and load (ETL) jobs
 - Batch processing
- **Consider reliability and scalability needs before choosing Hive on Spark**
 - Spark may perform poorly when memory is limited
 - Spark parameters must be tuned and adjusted as workloads change
 - Problems scaling to very large clusters, data sizes, numbers of users
- **Impala is typically faster than Hive on Spark**
 - Impala remains a better choice for most interactive queries



Viewing a Hive-on-Spark Job

- The Hive on Spark web UI displays running and recent jobs and job details
 - Access from the Hadoop web UI at <http://hostname:8088/>

The screenshot shows the Spark Web UI interface. At the top, there's a navigation bar with tabs: Jobs (selected), Stages, Storage, Environment, Executors, and Hive on Spark application UI. Below the navigation bar, the main area is divided into two sections: "Spark Jobs" on the left and "Details for Job 0" on the right.

Spark Jobs section:

- Total Uptime: 1.4 min
- Scheduling Mode: FIFO
- Completed Jobs: 1
- [Event Timeline](#)
- Completed Jobs (1)**

Job Id	Description	Submitted
0	foreachAsync at RemoteHiveSparkClient.java:340	2016/10/19 09:21:49

Details for Job 0 section:

- Status: SUCCEEDED
- Completed Stages: 3
- [Event Timeline](#)
- [DAG Visualization](#)

The DAG visualization shows three stages:

- Stage 0:** Contains an "hadoopRDD" node with a "mapPartitions" operation.
- Stage 1:** Contains a "groupByKey" node with a "mapValues" operation, which has a dependency on the "mapPartitions" operation from Stage 0. It also contains a "mapPartitions" operation.
- Stage 2:** Contains a "sortByKey" node with a "mapPartitions" operation, which has dependencies on the "mapPartitions" operations from both Stage 0 and Stage 1.

```
graph TD; subgraph Stage0 [Stage 0]; A[hadoopRDD] -- mapPartitions --> B[ ]; end; subgraph Stage1 [Stage 1]; C[groupByKey] -- mapValues --> D[ ]; C -- mapPartitions --> E[ ]; end; subgraph Stage2 [Stage 2]; F[sortByKey] -- mapPartitions --> G[ ]; F -- mapPartitions --> H[ ]; end;
```

- Past jobs can be found at the Spark History Server
 - Access at <http://hostname:18088/>

Chapter Topics

Apache Hive Optimization

- Understanding Query Performance
- Bucketing
- Hive on Spark
- **Essential Points**
- Hands-On Exercise: Hive Optimization

Essential Points

- **In the Hive query process, Hive will**
 - Parse the query and make optimizations
 - Plan the execution and submit the jobs to the cluster
 - Monitors the progress of the jobs as its processed
- **The EXPLAIN command shows a query's execution plan**
 - Understanding the execution plan helps you understand query performance
- **Hive on Spark is faster than Hive on MapReduce but requires more memory**
 - Still a good choice for
 - Complex, slow Hive queries that are incompatible with Impala
 - Data preparation and extract, transform, and load (ETL) jobs
 - Batch processing
- **You can use the CLUSTERED BY clause to create a table with buckets**
 - Subdivides data in an effectively random way
 - Useful for creating samples of the data

Bibliography

The following offer more information on topics discussed in this chapter

- Hive manual for the EXPLAIN command
 - <http://tiny.cloudera.com/dac13a>
- Hive manual for bucketed tables
 - <http://tiny.cloudera.com/dac13b>
- Faster Batch Processing with Hive on Spark
 - http://tiny.cloudera.com/hive_on_spark

Chapter Topics

Apache Hive Optimization

- Understanding Query Performance
- Bucketing
- Hive on Spark
- Essential Points
- **Hands-On Exercise: Hive Optimization**

Hands-On Exercise: Hive Optimization

- In this exercise, you will practice techniques to improve Hive query performance
- Please refer to the Hands-On Exercise Manual for instructions



Apache Impala Optimization

Chapter 13

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- **Apache Impala Optimization**
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Apache Impala Optimization

In this chapter, you will learn

- How Impala processes queries
- What are the categories of factors that affect Impala performance
- How table statistics help improve Impala performance

Chapter Topics

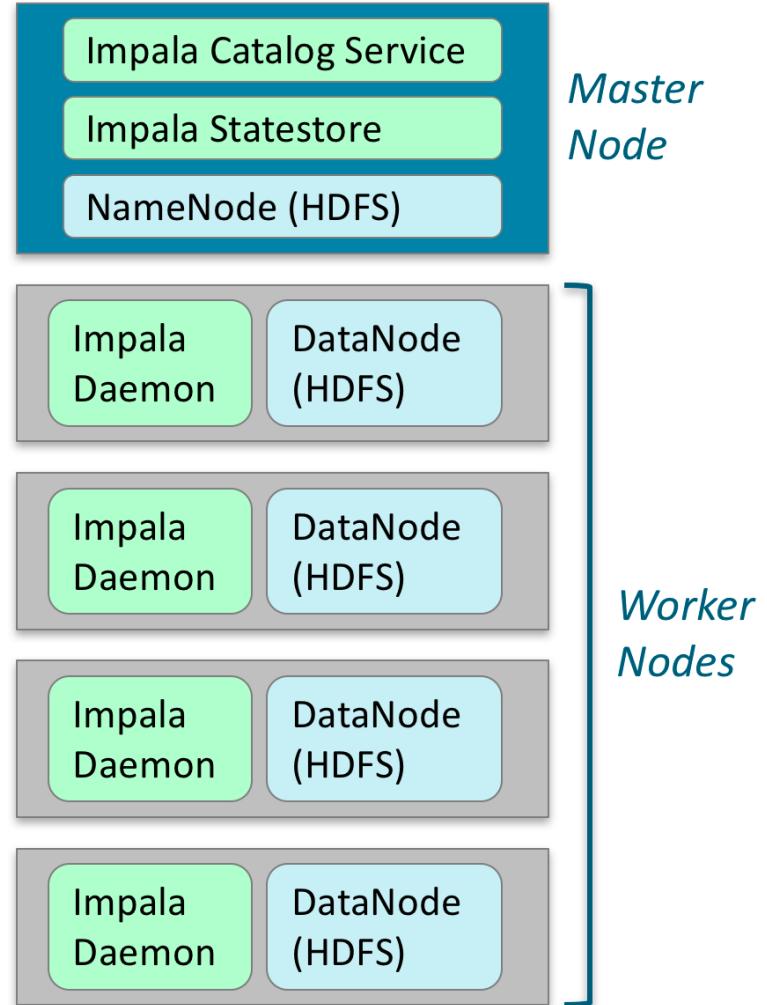
Apache Impala Optimization

- **How Impala Executes Queries**
- Improving Impala Performance
- Essential Points
- Hands-On Exercise: Impala Optimization



Impala in the Cluster

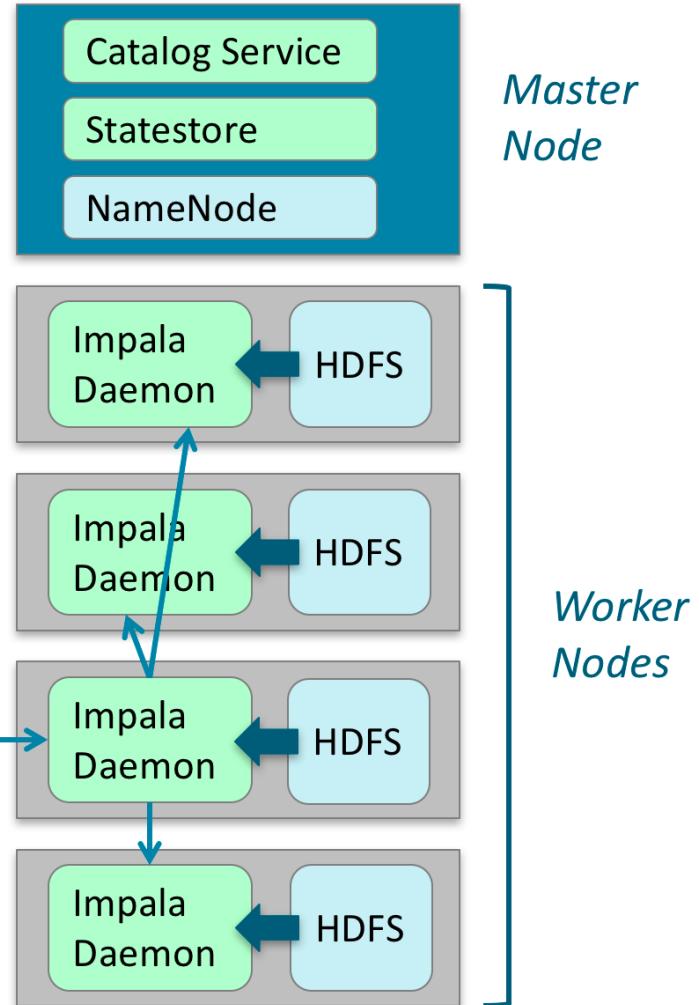
- Each worker node in the cluster runs an *Impala daemon*
 - Colocated with the HDFS worker daemon (DataNode)
- Two other daemons running on master nodes support Impala query execution
 - The **statestore**
 - Periodically checks status of Impala daemons
 - Relays status of daemons to other daemons
 - The **catalog service**
 - Relays metadata changes to all the Impala daemons in a cluster





How Impala Executes a Query

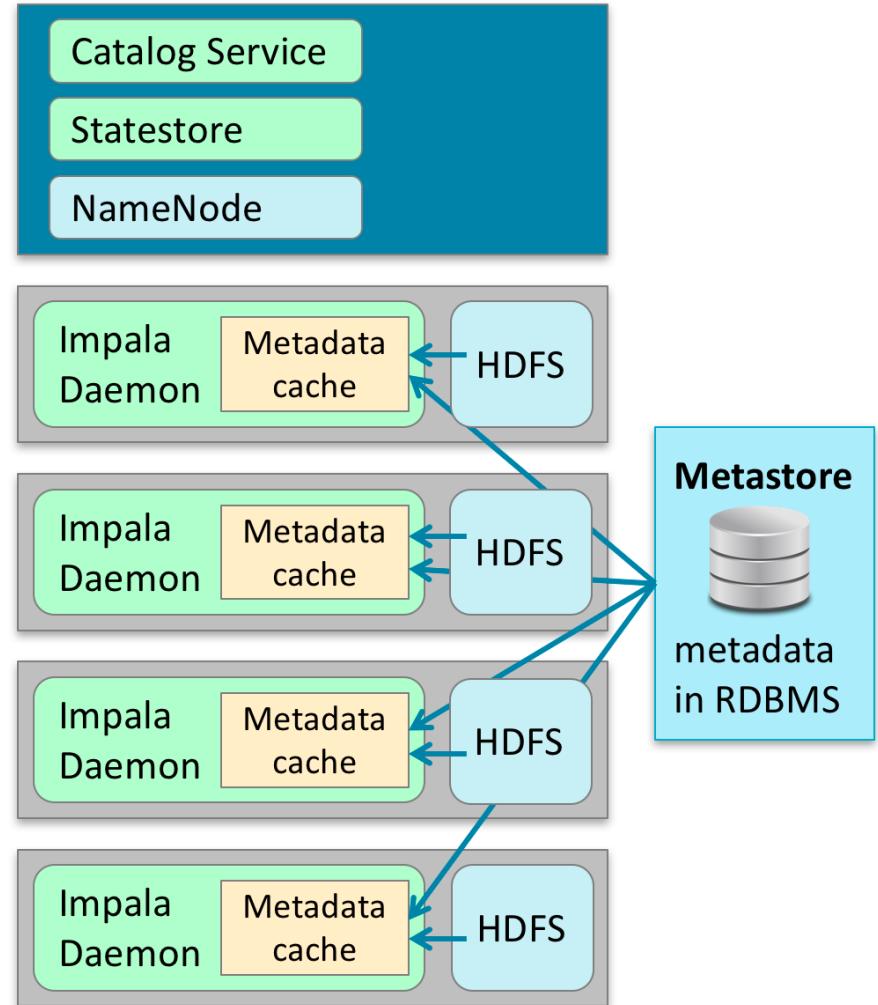
- **Impala daemon plans the query**
 - Client (Impala shell or Hue) connects to an Impala daemon
 - This is the *coordinator*
 - Coordinator requests a list of other Impala daemons in the cluster from the statestore
 - Coordinator distributes the query across other Impala daemons
 - Coordinator streams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





Metadata Caching (1)

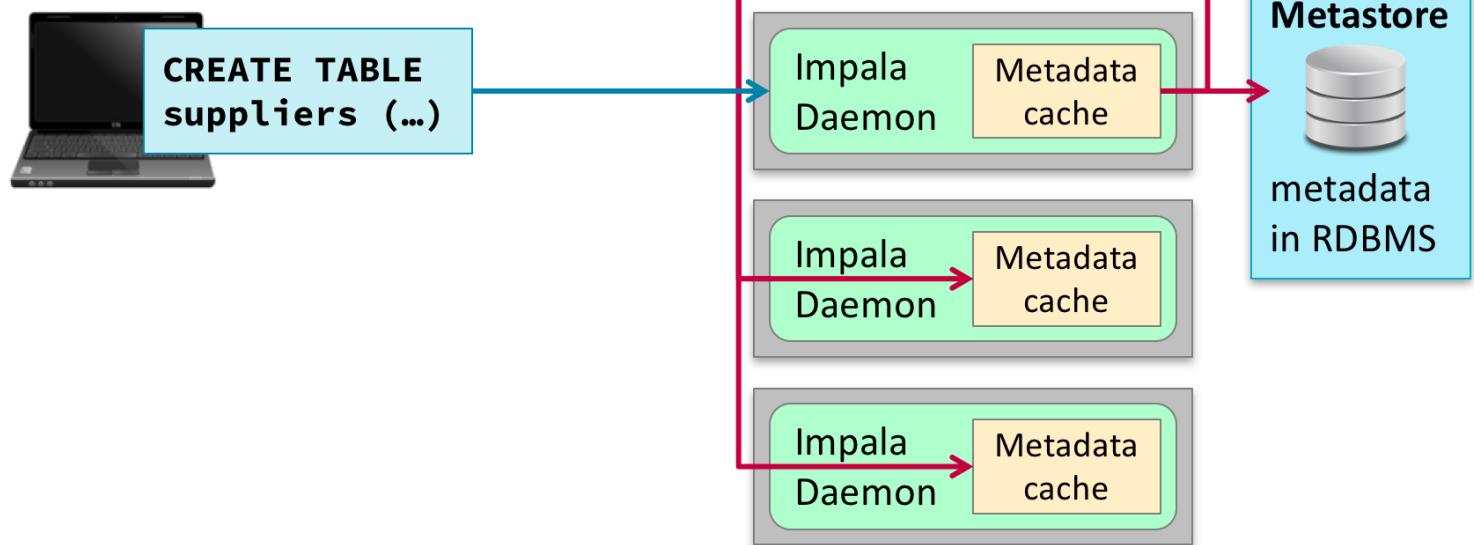
- Impala daemons cache three kinds of metadata
 - Table schema definitions
 - Locations of HDFS blocks containing table data
 - Table and column statistics (when available)
- Metadata is cached from the metastore and HDFS at startup
- This reduces query latency
 - Retrieving metadata can take significant time





Metadata Caching (2)

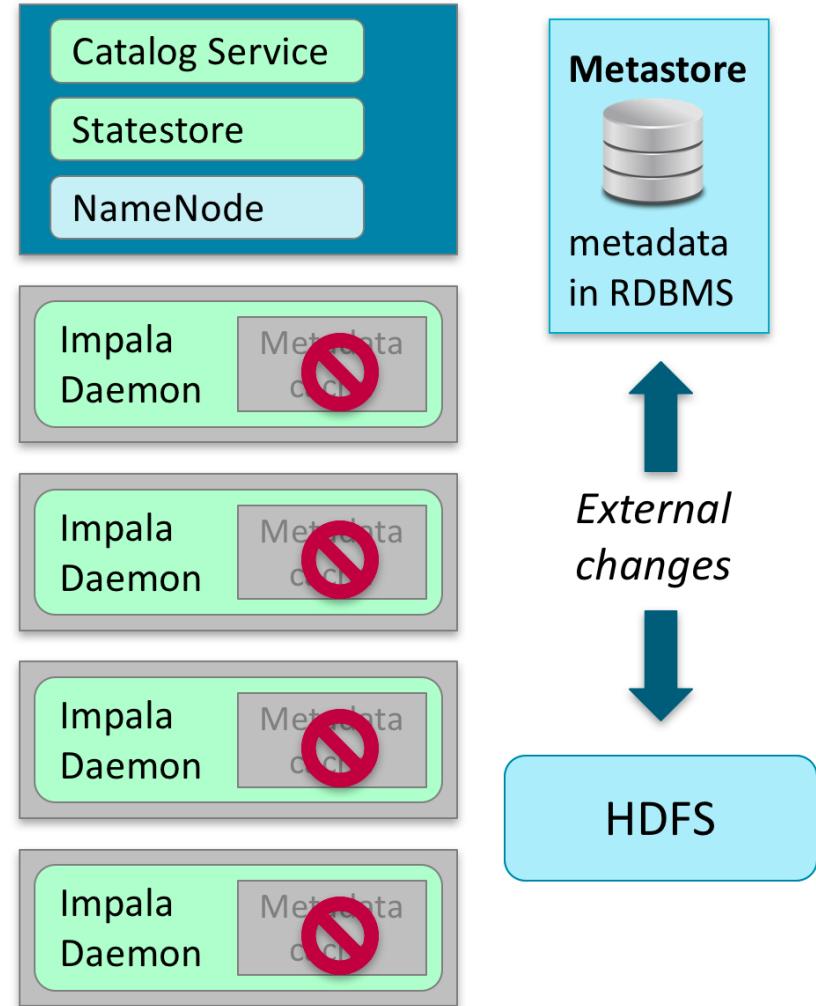
- When one Impala daemon changes the metastore or the location of HDFS blocks, it notifies the catalog service
- The catalog service notifies all Impala daemons to update their cache





External Changes and Metadata Caching

- Changes to the metastore and to table data *from outside of Impala* are not known to Impala
 - Changes made in Hive
 - Changes made with HCatalog
 - Changes made using the Hue Metadata Manager
 - Data added directly to HDFS
- Therefore the Impala metadata cache will be stale
- You must manually refresh or invalidate Impala's metadata cache
 - REFRESH *tablename* for new data or altered schema
 - INVALIDATE METADATA *tablename* for new table



Chapter Topics

Apache Impala Optimization

- How Impala Executes Queries
- **Improving Impala Performance**
- Essential Points
- Hands-On Exercise: Impala Optimization



Impala Performance Overview

- **Impala query performance is affected by three broad categories of factors**
 - The characteristics of the data being queried (format, type, and size)
 - Computing statistics on tables before running joins
 - The hardware and configuration of your cluster
- **Impala provides information to help you understand query performance**



Impala Memory Usage

- **Impala uses in-memory processing for faster performance**
 - Avoids writing intermediate data to disk
 - The more memory Impala can use, the better the performance
- **Impala stores intermediate data on disk when memory limits are reached**
 - Known as *spilling to disk*
 - Avoids query failures due to out-of-memory errors
 - Tradeoff: Decreased performance



Query Performance Optimization (1)

- Impala uses statistics about tables to optimize joins and similar functions
- You should compute statistics for tables with COMPUTE STATS
 - After you load a table initially
 - When the amount of data in a table changes substantially

```
COMPUTE STATS orders;
COMPUTE STATS order_details;
SELECT COUNT(o.order_id)
    FROM orders o
        JOIN order_details d
            ON (o.order_id = d.order_id)
WHERE YEAR(o.order_date) = 2008;
```

- Table statistics are stored in the metastore database
 - Can also be accessed and used by Hive



Query Performance Optimization (2)

- View using SHOW TABLE STATS and SHOW COLUMN STATS

```
SHOW TABLE STATS orders;
```

#Rows	#Files	Size	Bytes cached	Format
1662951	4	60.26MB	NOT CACHED	TEXT

```
SHOW COLUMN STATS orders;
```

Column	Type	#Distinct Values	#Nulls	Max Size	Avg Size
order_id	INT	1741201	-1	4	4
cust_id	INT	195884	-1	4	4
order_date	TIMESTAMP	1671984	-1	16	16



Viewing the Query Execution Plan

- To view Impala's execution plan, prefix your query with EXPLAIN or use the Explain button in Hue

```
> EXPLAIN SELECT *\n    FROM customers\n    WHERE state='NY';
```

Explain String
Estimated Per-Host Requirements: Memory=48.00MB Vcores=1 ...

The screenshot shows the Hue interface with a query editor and a results panel. In the query editor, a SELECT statement is typed:

```
1 | SELECT *\n2 |   FROM customers\n3 | WHERE state='NY';
```

A red box highlights the 'Explain' button in the toolbar, which is also indicated by a red arrow pointing from the left side of the screen towards it. The results panel displays the execution plan:

Max Per-Host Resource Reservation: Memory=0B
Per-Host Resource Estimates: Memory=16.00MB
WARNING: The following tables are missing relevant table column statistics.
default.customers

PLAN-ROOT SINK
|
01:EXCHANGE [UNPARTITIONED]



Example: Execution Plan (1)

```
SELECT COUNT(o.order_id)
  FROM orders o
    JOIN order_details d
      ON (o.order_id = d.order_id)
 WHERE YEAR(o.order_date) = 2008;
```



```
Max Per-Host Resource Reservation: Memory=4.75MB
Per-Host Resource Estimates: Memory=72.75MB

PLAN-ROOT SINK
|
06:AGGREGATE [FINALIZE]
|   output: count:merge(o.order_id)
|
05:EXCHANGE [UNPARTITIONED]
|
03:AGGREGATE
|   output: count(o.order_id)
|
02:HASH JOIN [INNER JOIN, BROADCAST]
|   hash predicates: d.order_id = o.order_id
|   runtime filters: RF000 <- o.order_id
|
|--04:EXCHANGE [BROADCAST]
|   |
```

Lines omitted



Example: Execution Plan (2)

Requirements for the whole query

Read query stages from the bottom up

Max Per-Host Resource Reservation: Memory=4.75MB
Per-Host Resource Estimates: Memory=72.75MB

```
PLAN-ROOT SINK
|
| 06:AGGREGATE [FINALIZE]
|   output: count:merge(o.order_id)
|
| 05:EXCHANGE [UNPARTITIONED]
|
| 03:AGGREGATE
|   output: count(o.order_id)
|
| 02:HASH JOIN [INNER JOIN, BROADCAST]
|   hash predicates: d.order_id = o.order_id
|   runtime filters: RF000 <- o.order_id
|
| --04:EXCHANGE [BROADCAST]
|
|   00:SCAN HDFS [default.orders o]
|     partitions=1/1 files=4 size=60.26MB
|     predicates: year(o.order_date) = 2008
|
|   01:SCAN HDFS [default.order_details d]
|     partitions=1/1 files=4 size=50.86MB
|     runtime filters: RF000 -> d.order_id
```



Example: Execution Plan (3)

Joins require scans
of both tables

```
...
02:HASH JOIN [INNER JOIN, BROADCAST]
|   hash predicates: d.order_id = o.order_id
|   runtime filters: RF000 <- o.order_id

--04:EXCHANGE [BROADCAST]
|
00:SCAN HDFS [default.orders o]
    partitions=1/1 files=4 size=60.26MB
    predicates: year(o.order_date) = 2008

01:SCAN HDFS [default.order_details d]
    partitions=1/1 files=4 size=50.86MB
    runtime filters: RF000 -> d.order_id
```



Setting the Explain Level

- Setting EXPLAIN_LEVEL controls the amount of query plan detail shown

Value	Name	Description
0	MINIMAL	Useful for checking the join order in very long queries
1	STANDARD	Shows the logical way that work is split up (default)
2	EXTENDED	Detail about how the query planner uses statistics
3	VERBOSE	Primarily used by Impala developers

```
> SET EXPLAIN_LEVEL=0;
> EXPLAIN SELECT COUNT(o.order_id) FROM orders o
   JOIN order_details d ON (o.order_id = d.order_id)
   WHERE YEAR(o.order_date) = 2008;
Estimated Per-Host Requirements: Memory=85.49MB Vcores=1
06:AGGREGATE [FINALIZE]
05:EXCHANGE [UNPARTITIONED]
03:AGGREGATE
02:HASH JOIN [INNER JOIN, BROADCAST]
...
```



Query Details after Execution

- The Impala shell provides commands to show details after running a query
 - SUMMARY: Overview of timings for query phases
 - PROFILE: Detailed report of query execution

```
SELECT COUNT(o.order_id) FROM orders o
  JOIN order_details d ON (o.order_id = d.order_id)
 WHERE YEAR(o.order_date) = 2008;
...
SUMMARY;

Operator      #Hosts  Avg Time    Max     #Rows  Est.    Peak   Est.          Detail
                           Time                  #Rows  Mem    Peak  Mem
-----
06:AGGREGATE    1    142.597ms  142.597ms  1     1    16.00 KB -1.00 B  FINALIZE
05:EXCHANGE     1    116.213us  116.213us  1     1     0        -1.00 B  UNPARTITIONED
03:AGGREGATE    1    148.320ms  148.320ms  1     1    10.71 MB 10.00 MB
02:HASH JOIN    1    68.250ms   68.250ms  52.14K 3.33M  6.52 MB  3.49 MB INNER JOIN, BROADCAST ...
```

Chapter Topics

Apache Impala Optimization

- How Impala Executes Queries
- Improving Impala Performance
- **Essential Points**
- Hands-On Exercise: Impala Optimization

Essential Points

- **Impala provides a high-performance SQL engine that distributes queries across a cluster**
 - Does not rely on MapReduce or Spark
 - Caches metadata from the metastore and HDFS
- **Impala query performance is affected by three broad categories of factors**
 - The characteristics of the data being queried (format, type, and size)
 - Computing statistics on tables before running joins
 - The hardware and configuration of your cluster
- **Impala's query planner uses table and column statistics to optimize join operations**
 - Use COMPUTE STATS to calculate stats before querying

Bibliography

The following offer more information on topics discussed in this chapter

- Impala Frequently Asked Questions
 - http://tiny.cloudera.com/impala_faq
- Tuning Impala for Performance
 - <http://tiny.cloudera.com/dac16e>
- Impala Concepts and Architecture
 - <http://tiny.cloudera.com/dac16f>
- Scalability Considerations for Impala
 - <http://tiny.cloudera.com/dac16g>

Chapter Topics

Apache Impala Optimization

- How Impala Executes Queries
- Improving Impala Performance
- Essential Points
- **Hands-On Exercise: Impala Optimization**

Hands-On Exercise: Impala Optimization

- In this exercise, you will explore query execution plans for various types of queries
- Please refer to the Hands-On Exercise Manual for instructions



Extending Apache Hive and Impala

Chapter 14

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- **Extending Apache Hive and Impala**
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Extending Apache Hive and Impala

In this chapter, you will learn

- What are some uses for parameterized queries
- How to use variable substitution to create and run parameterized queries
- How to employ and use user-defined functions
- How to implement a custom file format using a custom SerDe
- How to transform data in Apache Hive using an external program

Chapter Topics

Extending Apache Hive and Impala

- **Custom SerDes and File Formats in Hive**
- Data Transformation with Custom Scripts in Hive
- User-Defined Functions
- Parameterized Queries
- Essential Points
- Hands-On Exercise: Data Transformation with Hive



Recap: Hive File Formats and SerDes

- **Hive supports different file formats for data storage**
 - Including TEXTFILE, SEQUENCEFILE, AVRO, and PARQUET
 - Specified when creating a table, with STORED AS
- **Hive supports different row formats using SerDes**
 - SerDe stands for *serializer/deserializer*
 - Hive has the Regex SerDe and SerDes for delimited, CSV, and JSON data
 - Specify when creating a table, implicitly or with ROW FORMAT SERDE
- **A SerDe is a Java class**
 - You can specify a SerDe by specifying its fully qualified Java class name

```
CREATE TABLE people(fname STRING, lname STRING)
ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe';
```



File Format Java Classes

- **File formats are also defined by Java classes**
 - Previously, we specified these implicitly, using `STORED AS format`
 - It is also possible to specify file format Java classes explicitly
- **A file format consists of two Java classes**
 - `InputFormat` for reading data
 - `OutputFormat` for writing data



Specifying a File Format in Hive

- Previously, we specified the file format using `STORED AS format`

```
CREATE TABLE people(fname STRING, lname STRING)
  STORED AS TEXTFILE;
```

- You can also specify `InputFormat` and `OutputFormat` explicitly
 - Using fully qualified Java class names

```
CREATE TABLE people(fname STRING, lname STRING)
  STORED AS
    INPUTFORMAT
      'org.apache.hadoop.mapred.TextInputFormat'
    OUTPUTFORMAT
      'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat';
```



How Are File Formats Related to SerDes?

- **SerDes and file formats can be specified independently**
 - The SerDe specifies the record format
 - The `InputFormat` and `OutputFormat` specify the file format
- **But a SerDe only works with certain file formats**
 - For example, `LazySimpleSerde` works with text files and `SequenceFiles`, but not with Avro or Parquet files
 - Hive's CSV, JSON, and Regex SerDes are for use with text files
 - Columnar file formats require specialized columnar SerDes which are implicitly specified when you specify the file format



Custom File Formats and SerDes in Hive

- **Hive allows creation of custom SerDes and file formats using its Java APIs**
 - You can find open source Hive SerDes and file formats on the web
 - Writing your own is seldom necessary
- **Next we show how to use a custom SerDe and InputFormat in Hive**
 - To read data in XML format
 - Using JAR file from http://tiny.cloudera.com/xml_serde



Adding a JAR File to Hive

- First, copy the JAR file to HDFS

```
$ hdfs dfs -put hivexmlserde.jar /dualcore/scripts/
```

- Then register the JAR file with Hive

- Ensures Hive can find the JAR file at runtime

```
ADD JAR hdfs:/dualcore/scripts/hivexmlserde.jar;
```

- Remains in effect only during the current Beeline session

- Your system administrator can add the JAR permanently



Example: Custom XML SerDe and InputFormat (1)

Input Data

```
<records>
  <record customer_id="1">
    <firstname>Seo-yeon</firstname>
    <lastname>Lee</lastname>
  </record>
  <record customer_id="2">
    <firstname>Xavier</firstname>
    <lastname>Gray</lastname>
  </record>
</records>
```

Resulting Table

cust_id	fname	lname
1	Seo-yeon	Lee
2	Xavier	Gray



Example: Custom XML SerDe and InputFormat (2)

Specify SerDe and InputFormat

```
CREATE TABLE xml_customers
  (cust_id INT,
   fname STRING,
   lname STRING)
ROW FORMAT SERDE 'com.ibm.spss.hive.serde2.xml.XmlSerDe'
  WITH SERDEPROPERTIES
    ('column.xpath.cust_id'=' /record/@customer_id',
     'column.xpath.fname'=' /record/firstname/text()',
     'column.xpath.lname'=' /record/lastname/text()')
STORED AS
  INPUTFORMAT
    'com.ibm.spss.hive.serde2.xml.XmlInputFormat'
  OUTPUTFORMAT
    'org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat'
TBLPROPERTIES
  ('xmlinput.start'='<record customer',
   'xmlinput.end'='</record>');
```

Chapter Topics

Extending Apache Hive and Impala

- Custom SerDes and File Formats in Hive
- **Data Transformation with Custom Scripts in Hive**
- User-Defined Functions
- Parameterized Queries
- Essential Points
- Hands-On Exercise: Data Transformation with Hive



Using TRANSFORM to Process Data Using External Scripts

- You are not limited to manipulating data exclusively in HiveQL
 - Hive allows you to transform data through external scripts or programs
 - These can be written in nearly any language
- This is done with HiveQL's TRANSFORM . . . USING construct
 - One or more fields are supplied as arguments to TRANSFORM()
 - Copy the script to HDFS, and identify it with USING
 - It receives each record, processes it, and returns the result

```
SELECT TRANSFORM(product_name, price)
      USING 'hdfs:/myscripts/tax_calculator.py'
     FROM products;
```

- Note: TRANSFORM is not allowed when SQL authorization is enabled in Hive
 - Due to security risks, Hive will not execute a script on a secure cluster



Data Input and Output with TRANSFORM

- Your external program will receive one record per line on standard input
 - Each field in the supplied record will be a tab-separated string
 - NULL values are converted to the literal string \N
- You may need to convert values to appropriate types within your program
 - For example, converting to numeric types for calculations
- Your program must return tab-delimited fields on standard output
 - Output fields can optionally be named and cast using the syntax below

```
SELECT TRANSFORM(product_name, price)
    USING 'hdfs:/myscripts/tax_calculator.py'
        AS (item_name STRING, tax INT)
FROM products;
```



Hive TRANSFORM Example (1)

- Here is a complete example of using TRANSFORM in Hive
 - The Perl script parses an email address, determines to which country it corresponds, and then returns an appropriate greeting
 - A sample of input data is followed by the corresponding HiveQL code

employees table

fname	email
Antoine	antoine@example.fr
Kai	kai@example.de
Pedro	pedro@example.mx

```
SELECT TRANSFORM(fname, email)
  USING 'hdfs:/dualcore/scripts/greeting.pl'
    AS greeting
  FROM employees;
```



Hive TRANSFORM Example (2)

- The Perl script for this example is shown below
 - A complete explanation of this script follows on the next few slides

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<STDIN>) {
    ($name, $email) = split /\t/;
    ($suffix) = $email =~ /\.( [a-z]+ )$/;
    $greeting = $greetings{$suffix};
    $greeting = 'Hello' unless defined($greeting);
    print "$greeting $name\n";
}
```



Hive TRANSFORM Example (3)

```
#!/usr/bin/env perl ①

%greetings = ('de' => 'Hallo', ②
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<STDIN>) {
    ($name, $email) = split /\t/;
    ($suffix) = $email =~ /\.( [a-z]+ )$/;
    $greeting = $greetings{$suffix};
    $greeting = 'Hello' unless defined($greeting);
    print "$greeting $name\n";
}
```

- ① The first line tells the system to use the Perl interpreter when running this script.
- ② The next line defines the greetings using an associative array keyed by the country codes from the email addresses.



Hive TRANSFORM Example (4)

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<STDIN>) {
    ($name, $email) = split /\t/; ①
    ($suffix) = $email =~ /\.( [a-z]+ )$/;
    $greeting = $greetings{$suffix};
    $greeting = 'Hello' unless defined($greeting);
    print "$greeting $name\n";
}
```

- ① Read each record from standard input within the loop, and then split them into fields based on tab characters.



Hive TRANSFORM Example (5)

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<STDIN>) {
    ($name, $email) = split /\t/;
    ($suffix) = $email =~ /\.( [a-z]+ )$/; ①
    $greeting = $greetings{$suffix};
    $greeting = 'Hello' unless defined($greeting); ②
    print "$greeting $name\n";
}
```

- ① Extract the country code from the email address (the pattern matches any letters following the final dot).
- ② Use that to look up a greeting, but default to 'Hello' if none is found.



Hive TRANSFORM Example (6)

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<STDIN>) {
    ($name, $email) = split /\t/;
    ($suffix) = $email =~ /\.( [a-z]+ )$/;
    $greeting = $greetings{$suffix};
    $greeting = 'Hello' unless defined($greeting);
    print "$greeting $name\n"; ①
}
```

- ① Finally, return the greeting as a single field by printing this value to standard output. (For multiple fields, separate them with tab characters when printing them here.)



Hive TRANSFORM Example (7)

- Finally, here's the result of our transformation

```
> SELECT TRANSFORM(fname, email)
   USING 'hdfs:/dualcore/scripts/greeting.pl'
   AS greeting
  FROM employees;
```

Bonjour Antoine
Hallo Kai
Hola Pedro

Chapter Topics

Extending Apache Hive and Impala

- Custom SerDes and File Formats in Hive
- Data Transformation with Custom Scripts in Hive
- **User-Defined Functions**
- Parameterized Queries
- Essential Points
- Hands-On Exercise: Data Transformation with Hive

Overview of User-Defined Functions

- **User-defined functions** are custom functions
 - Invoked with the same syntax as built-in functions

```
SELECT calc_shipping_cost(weight, zipcode, '2-DAY')
      FROM shipments WHERE order_id=5742354;
```

- Hive supports three types of user-defined functions
 - “Standard” user-defined functions (UDFs)
 - User-defined aggregate functions (UDAFs)
 - User-defined table-generating functions (UDTFs)
- Impala supports UDFs and UDAFs
 - But not UDTFs



Developing Hive User-Defined Functions

- **Hive user-defined functions are written in Java**
 - Currently no support for writing them in other languages
 - Using TRANSFORM may be an alternative
- **Open source user-defined functions are plentiful on the web**
- **There are three steps for using a user-defined function in Hive**
 1. Copy the function's JAR file to HDFS
 2. Register the function
 3. Use the function in your query



Example: Using a UDF in Hive (1)

- The example UDF on the next slides was compiled from sources found on GitHub
 - Popular website for many open source software projects
 - Project URL: <http://tiny.cloudera.com/dac17a>
- The UDF is packaged into a JAR file on the hands-on environment
- The example is the URLDecode UDF in that JAR file
 - Converts URLs with encoding into a readable format



Example: Using a UDF in Hive (2)

- First, copy the JAR file to HDFS
 - Same step as with a custom SerDe

```
$ hdfs dfs -put url-decode-udf.jar /myscripts/
```

- Next, register the function and assign an alias
 - The quoted value is the fully qualified Java class for the UDF

```
CREATE FUNCTION url_decode
  AS 'stewi.hive.udf.URLDecode'
  USING JAR 'hdfs:/myscripts/url-decode-udf.jar';
```

- Hive persists the function in the metastore database
- To remove the function, use `DROP FUNCTION url_decode;`



Example: Using a UDF in Hive (3)

- You may then use the function in your query

```
> SELECT url_decode('http://example.com/this%2Bis%2Bcommon.pdf');
    http://example.com/this+is+common.pdf

> SELECT url_decode(parse_url(
    'http://example.com/email?recipient=whoever%40example.com',
    'QUERY', 'recipient'));
    whoever@example.com

> SELECT url_decode(
    'http://fr.wikipedia.org/wiki/%C3%A9l%C3%A9phant');
    http://fr.wikipedia.org/wiki/Éléphant
```

- The function is available in the database where you register it
 - Prepend the database name if you want to use it outside that database

```
> SELECT analyst.url_decode(...);
```



Overview of Impala User-Defined Functions

- **Impala also supports user-defined functions**
 - “Standard” UDFs and user-defined aggregate functions (UDAFs) are supported
 - User-defined table-generating functions (UDTFs) are not supported
- **Native Impala user-defined functions are written in C++**
 - These C++ functions cannot be used in Hive
- **Impala also supports Java UDFs developed for Hive**
 - Java UDAFs and UDTFs are not supported in Impala

Type	Hive	Impala
UDF	Java	C++ or Java*
UDAF	Java	C++
UDTF	Java	Not supported

*With limitations described at http://tiny.cloudera.com/impala_java_udf



Using a Java UDF in Impala

- **Java UDFs registered in Hive are available in Impala**
 - Refresh the cache with `REFRESH FUNCTIONS database;` first
 - Usage may differ slightly in Impala
 - Caused by differences between Hive and Impala
 - For example, Impala requires explicit casting of data types
- **Or you can use Impala to register a Java UDF**
 - The syntax is different than in Hive

```
CREATE FUNCTION url_decode
  LOCATION 'hdfs:/myscripts/url-decode-udf.jar'
  SYMBOL='stewi.hive.udf.URLDecode';
```



Using a C++ UDF in Impala

- Register the function with Impala
 - Specify argument data types and return data type

```
CREATE FUNCTION count_vowels(STRING)
    RETURNS INT
    LOCATION '/user/hive/udfs/sampleudfs.so'
    SYMBOL='CountVowels';
```

- You can then use the function in a query

```
SELECT count_vowels(email_address) FROM employees;
```

Chapter Topics

Extending Apache Hive and Impala

- Custom SerDes and File Formats in Hive
- Data Transformation with Custom Scripts in Hive
- User-Defined Functions
- **Parameterized Queries**
- Essential Points
- Hands-On Exercise: Data Transformation with Hive

Parameterized Queries

- Hive and Impala support variable substitution in queries
 - Enables using parameters for repetitive queries
- In Hue, use \${variable} for both Hive and Impala
 - Put the variable value in the boxes that appear below

```
1| SELECT * FROM products WHERE brand = ${brand} AND price > ${price}
2|
```

brand

'Gigabux'

price

1000



- This feature is implemented differently in Beeline and Impala Shell



Hive Variables (1)

- You can use Hive to set a variable

- Name is case-sensitive and must be prefixed with `hivevar:`

```
> SET hivevar:mystate=CA;
```

- Variables are set for the duration of the current session

- You can then use the variable in a Hive query

- This will be replaced with the variable's value at runtime

```
SELECT * FROM customers  
WHERE state = 'hivevar:mystate';
```

- Run `SET hivevar:mystate;` to see its current value



Hive Variables (2)

- You can also set variables when you start Beeline from the command line
- For example, the following query counts the unique customers in a state
 - This HiveQL is saved in the file `state.hql`

```
SELECT COUNT(DISTINCT cust_id) FROM customers  
WHERE state = '${hivevar:mystate}';
```

state.hql

- Using variables makes it easy to create per-state reports
 - Enclose values in quotes when you set them from the command line

```
$ beeline -u ... --hivevar mystate="CA" -f state.hql  
$ beeline -u ... --hivevar mystate="NY" -f state.hql
```

Command line



Impala Variables (1)

- You can use the Impala shell to set a variable
 - In Impala 2.5/CDH 5.7 and higher
 - Name is case-sensitive and must be prefixed with var:

```
> SET var:mystate=CA;
```

- Variables are set for the duration of the current session
- You can then use the variable in an Impala query
 - This will be replaced with the variable's value at runtime

```
SELECT * FROM customers  
WHERE state = '${var:mystate}';
```



Impala Variables (2)

- You can set variables when you start Impala shell from the command line
 - The syntax differs slightly from Beeline

```
SELECT COUNT(DISTINCT cust_id) FROM customers  
WHERE state = '${var:mystate}';
```

state.sql

```
$ impala-shell --var=mystate="CA" -f state.sql  
$ impala-shell --var=mystate="NY" -f state.sql
```

Command line

Chapter Topics

Extending Apache Hive and Impala

- Custom SerDes and File Formats in Hive
- Data Transformation with Custom Scripts in Hive
- User-Defined Functions
- Parameterized Queries
- **Essential Points**
- Hands-On Exercise: Data Transformation with Hive

Essential Points

- **Hive and Impala support variable substitution in queries**
 - Good for repetitive queries
 - Use the variable in a query or script
 - Set the variable with SET or pass at the command line when running a script
 - Hive and Impala syntax are slightly different
- **Hive and Impala both allow user-defined functions**
 - Hive can use Java UDFs, UDAFs, or UDTFs
 - Impala can use Java UDFs, or C++ UDFs and UDAFs
 - With both engines, copy and register before using
- **Hive allows implementing custom SerDes and file formats using its Java APIs**
 - Copy and register the JAR file
 - Use ROW FORMAT SERDE for SerDes
 - Use STORED AS INPUTFORMAT (or OUTPUTFORMAT) for file formats
- **Use TRANSFORM with an external program to process records in Hive**

Chapter Topics

Extending Apache Hive and Impala

- Custom SerDes and File Formats in Hive
- Data Transformation with Custom Scripts in Hive
- User-Defined Functions
- Parameterized Queries
- Essential Points
- **Hands-On Exercise: Data Transformation with Hive**

Hands-On Exercise: Data Transformation with Hive

- In this exercise, you will use a Hive transform script and a UDF to estimate shipping costs on abandoned orders
- Please refer to the Hands-On Exercise Manual for instructions



Choosing the Best Tool for the Job

Chapter 15

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- **Choosing the Best Tool for the Job**
- Conclusion
- Appendix: Apache Kudu

Choosing the Best Tool for the Job

In this chapter, you will learn

- What are the major differences among Hive, Impala, and relational databases
- What are the types of tasks for which each tool is best suited
- How to choose the best tool or mix of tools for a given task or workflow

Chapter Topics

Choosing the Best Tool for the Job

- **Comparing Hive, Impala, and Relational Databases**
- Which to Choose?
- Essential Points
- Optional Hands-On Exercise: Analyzing Abandoned Carts

Recap of Data Analysis and Processing Tools

- **Hive**
 - SQL-based queries executed using MapReduce or Spark
- **Impala**
 - High-performance SQL-based queries using an execution engine optimized for speed

Comparing Hive and Impala

Feature	Hive	Impala
SQL-based query language	Yes	Yes
Schemas optional	No	No
User-defined functions (UDFs)	Yes	Yes
Process data with external scripts	Yes	No
Extensible record and file formats	Yes	No
Complex data types	Yes	Limited
Query latency	High	Low
Fault tolerance	Yes	No
Built-in data partitioning	Yes	Yes
Accessible with ODBC / JDBC	Yes	Yes

Do These Replace an RDBMS?

- **Probably not if the RDBMS is used for its intended purpose**
- **Relational databases are optimized for**
 - Relatively small amounts of data
 - Immediate results
 - In-place modification of data (UPDATE and DELETE)
- **Hive and Impala are optimized for**
 - Large amounts of read-only data
 - Extensive scalability at low cost
- **Hive is better suited for batch processing**
 - Impala and RDBMSs are better for interactive use

Comparing an RDBMS to Hive and Impala

Feature	RDBMS	Hive	Impala
Insert records	Yes	Yes	Yes
Update and delete records	Yes	No*	Not
Transactions	Yes	No*	No
Role-based authorization	Yes	Yes	Yes
Stored procedures	Yes	No	No
Index support	Extensive	Limited	No
Latency	Very low	High	Low
Data size	Terabytes	Petabytes	Petabytes
Complex data types	No	Yes	Limited
Storage cost	Very high	Very low	Very low

* Hive now has limited, experimental support for UPDATE, DELETE, and transactions.

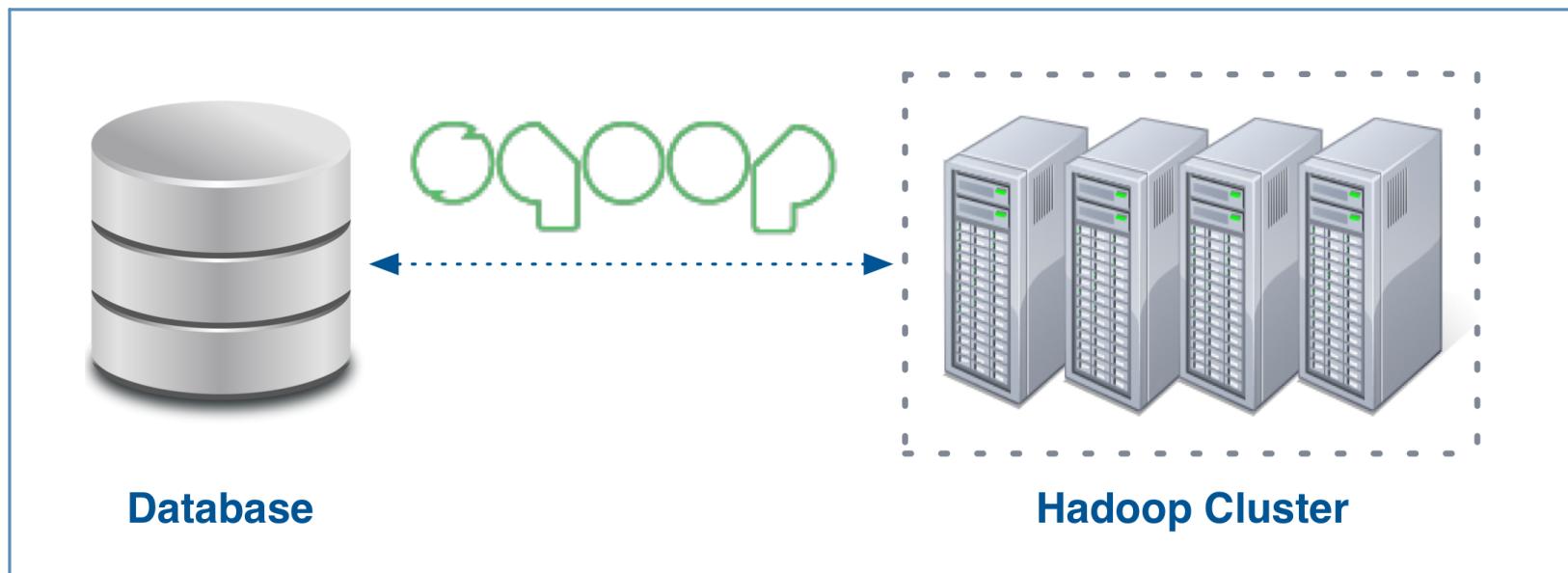
† Using Impala, you can update or delete individual rows in Kudu tables.

Hive Features Currently Unsupported in Impala

- **Impala does not currently support some features found in Hive**
 - Complex data types with table formats other than Parquet
 - BINARY data type
 - Bucketing
 - Unstructured text processing and analysis
 - Custom SerDes
 - Custom file formats
 - External transformations
- **Many of these are being considered for future Impala releases**

Apache Sqoop

- Sqoop helps you integrate Hadoop tools with relational databases
- It exchanges data between RDBMSs and Hadoop
 - Can import all tables, a single table, or a portion of a table into HDFS
 - Supports incremental imports
 - Can also export data from HDFS to a database



Chapter Topics

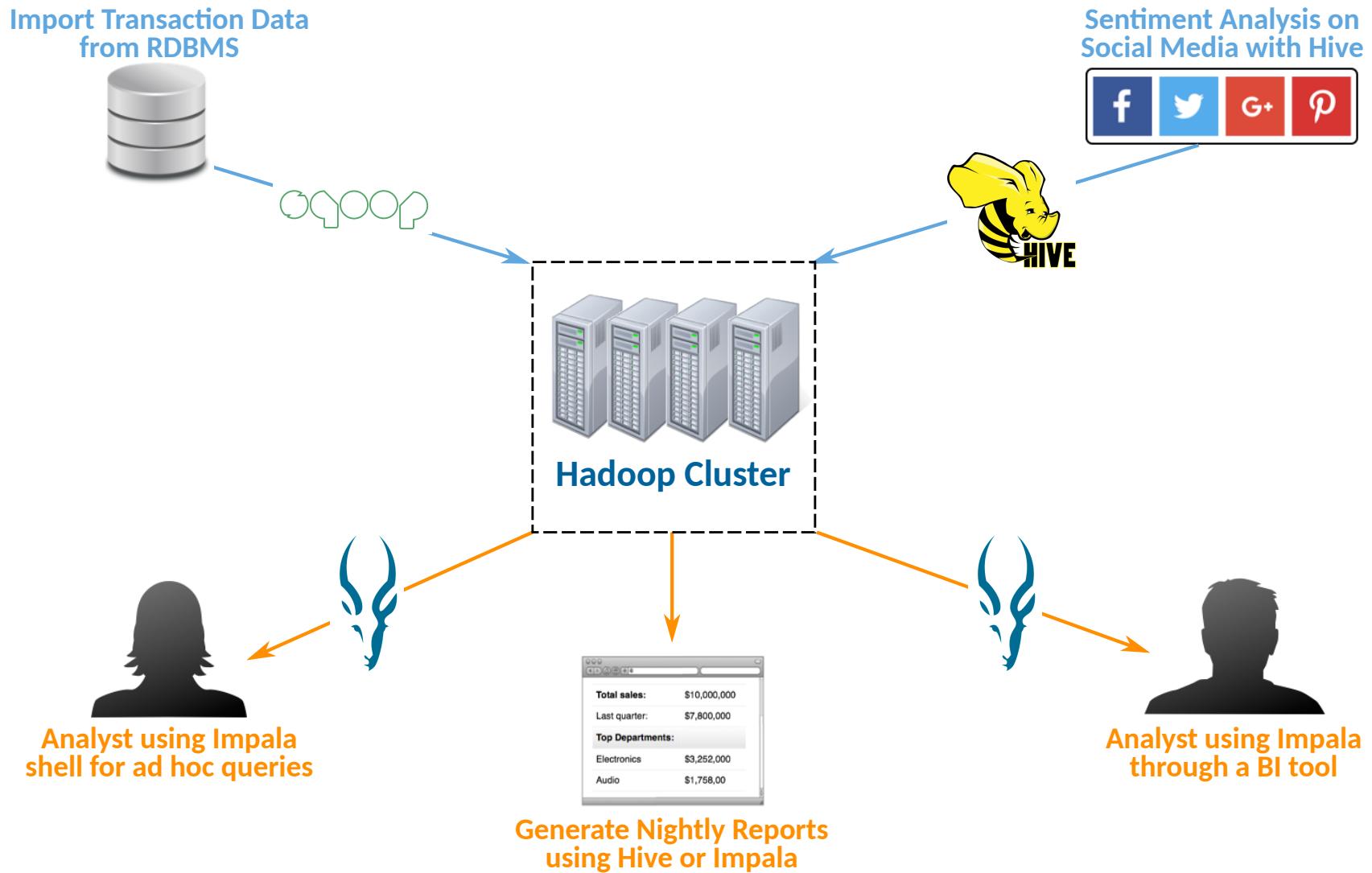
Choosing the Best Tool for the Job

- Comparing Hive, Impala, and Relational Databases
- **Which to Choose?**
- Essential Points
- Optional Hands-On Exercise: Analyzing Abandoned Carts

Which to Choose?

- **Hive turns SQL queries into MapReduce or Spark jobs**
 - Good choice for batch processing and ETL using SQL
 - Includes some features not supported in Impala
- **Impala is a high-performance SQL engine that runs on a Hadoop cluster**
 - Good choice for interactive and ad-hoc analysis
- **Choose the best one for a given task**
 - Mix and match as needed

Analysis Workflow Example



When to Use MapReduce or Spark?

- **MapReduce and Spark are powerful data processing engines**
 - Provide APIs for writing custom data processing code
 - Require programming skills
 - More time-consuming and error-prone to write
 - Best when control matters more than productivity
- **Hive and Impala offer greater productivity**
 - Faster to learn, write, test, and deploy than MapReduce or Spark
 - Complexities of lower-level code are abstracted from the user
 - Better choice for many data analysis and processing tasks

Chapter Topics

Choosing the Best Tool for the Job

- Comparing Hive, Impala, and Relational Databases
- Which to Choose?
- **Essential Points**
- Optional Hands-On Exercise: Analyzing Abandoned Carts

Essential Points

- **Criteria for choosing the best tool include scale, speed, control, and productivity**
 - Relational databases offer speed but not scalability
 - Hive offers productivity, scalability, and reliability but not necessarily speed
 - Impala offers scalability and speed but has some limitations
 - In addition, MapReduce and Spark offer control at the cost of productivity
- **This means each tool is better for some types of tasks than the others; for example:**
 - Relational databases are good for smaller datasets
 - Hive is good for batch processes and big data operations for which Impala is limited
 - Impala is good for ad hoc queries on large data
 - Some complex tasks may require a mix of these and other tools (such as Sqoop)

Chapter Topics

Choosing the Best Tool for the Job

- Comparing Hive, Impala, and Relational Databases
- Which to Choose?
- Essential Points
- **Optional Hands-On Exercise: Analyzing Abandoned Carts**

Optional Hands-On Exercise: Analyzing Abandoned Carts

- In this optional exercise, you will use your preferred tool to analyze data about abandoned orders to determine if a free shipping promotion would be profitable
- Please refer to the Hands-On Exercise Manual for instructions



Conclusion

Chapter 16

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- Appendix: Apache Kudu

Course Objectives (1)

During this course, you have learned

- How the open source ecosystem of big data tools addresses challenges not met by traditional RDBMSs
- How Apache Hive and Apache Impala are used to provide SQL access to data
- How Hive and Impala syntax and data formats, including functions and subqueries, help answer questions about data
- How to create, modify, and delete tables, views, and databases; load data; and store results of queries
- How to create and use partitions and different file formats

Course Objectives (2)

- How to combine two or more datasets using JOIN or UNION, as appropriate
- What analytic and windowing functions are, and how to use them
- How to store and query complex or nested data structures
- How to process and analyze semi-structured and unstructured data
- Different techniques for optimizing Hive and Impala queries
- How to extend the capabilities of Hive and Impala using parameters, custom file formats and SerDes, and external scripts
- How to determine whether Hive, Impala, an RDBMS, or a mix of these is best for a given task

Which Course to Take Next?

Cloudera offers a range of training courses for you and your team

- **For developers**
 - *Cloudera Search Training*
 - *Cloudera Training for Apache HBase*
 - *Developer Training for Apache Spark and Hadoop*
- **For system administrators**
 - *Cloudera Administrator Training for Apache Hadoop*
- **For data scientists**
 - *Cloudera Data Scientist Training*
- **For architects, managers, CIOs, and CTOs**
 - *Cloudera Essentials for Apache Hadoop*

Cloudera Certified Associate (CCA) Data Analyst

- **Prove your expertise with the most sought-after technical skills**
 - Cloudera certification holders have a unique license that displays, promotes, and verifies their certification record
- **The CCA Data Analyst exam tests foundational data analyst skills**
 - ETL processes
 - Data definition language
 - Query language
- **This course is excellent preparation for the exam**
- **Remote-proctored exam available anywhere at any time**
- **Register at http://tiny.cloudera.com/cca_data_analyst**



Apache Kudu

Appendix A

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Common Operators and Built-In Functions
- Data Management
- Data Storage and Performance
- Working with Multiple Datasets
- Analytic Functions and Windowing
- Complex Data
- Analyzing Text
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion
- **Appendix: Apache Kudu**

Apache Kudu

In this chapter, you will learn

- How Apache Kudu stores and delivers data
- What are the differences in table design between Kudu tables and other Apache Impala tables
- How to create schema designs, including a partitioning strategy, for Kudu tables
- How to create and delete Kudu tables using Impala
- How to insert and update rows in Kudu tables using Impala

Chapter Topics

Apache Kudu

- **What Is Kudu?**
- Kudu Tables
- Using Impala with Kudu
- Essential Points
- Hands-On Exercise: Using Apache Kudu with Apache Impala

What Is Kudu?

- **Open source distributed storage engine**
 - Designed for big data—terabytes or petabytes
- **Standalone storage system**
 - Built directly on underlying OS filesystem
- **Designed for random access and analytical queries on structured data**
 - Organizes data into tables
 - Supports scans, random lookups, and updates

Key Kudu Features

- **High throughput for big scans (like HDFS)**
- **Low latency for short accesses (like HBase)**
- **NoSQL-style scan, insert, and update operations**
 - Using Java, C++, and Python clients
- **Database-like transaction semantics**
 - Single-row atomic write operations
- **Integration into Apache Hadoop ecosystem**
 - Such as MapReduce, Apache Spark, and Apache Impala

Filling the Storage System Gap

- HBase and HDFS are optimized for different things
 - HDFS provides high throughput sequential reads
 - HBase provides low-latency random reads and writes
- Some applications require *both*
 - Apache Kudu fills this gap
- Kudu storage is for *structured data*

Capability	HDFS	HBase	Kudu
Fast sequential access	✓	✗	✓
Fast random access	✗	✓	✓
Data update	✗	✓	✓
Structured data	✓	✓	✓
Unstructured data	✓	✓	✗

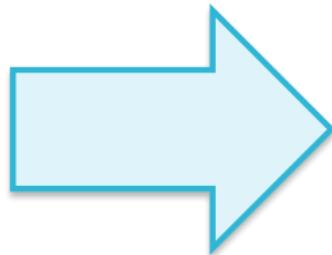
Distributed Tables

- **Kudu is a distributed, scalable system**
 - A *Kudu cluster* stores and manages tables
 - Highly available and fault tolerant
 - Scale the cluster by adding hosts as data grows
- **Table data is distributed across the cluster**
 - Tables are partitioned into *tablets*
 - Each tablet stores a subset of a table's rows
 - Tablets are stored on and served from multiple hosts

Columnar Data Storage in Kudu

- Kudu maintains data in a **columnar storage format**
 - Values from the same columns in many rows are stored contiguously on disk for faster access

A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3



Row-based store

A1	B1	C1	A2	B2	C2	A3	B3	C3
----	----	----	----	----	----	----	----	----

Column-based store

A1	A2	A3	B1	B2	B3	C1	C2	C3
----	----	----	----	----	----	----	----	----

Chapter Topics

Apache Kudu

- What Is Kudu?
- **Kudu Tables**
- Using Impala with Kudu
- Essential Points
- Hands-On Exercise: Using Apache Kudu with Apache Impala

Kudu Tables

- A Kudu table has rows of structured data
 - Structure is defined by a *schema*
- Each Kudu table must have one **primary key**
 - Comprised of one or more columns
 - Must be unique (when composed if composite) and not NULL
 - Must be defined as the first column or columns of the table schema
 - Cannot be updated
- Kudu tables are partitioned into *tablets*
 - Distributed to and replicated across tablet servers
 - Supports parallelism for better performance
 - Rows within a tablet are sorted lexicographically by primary key

Schema Design

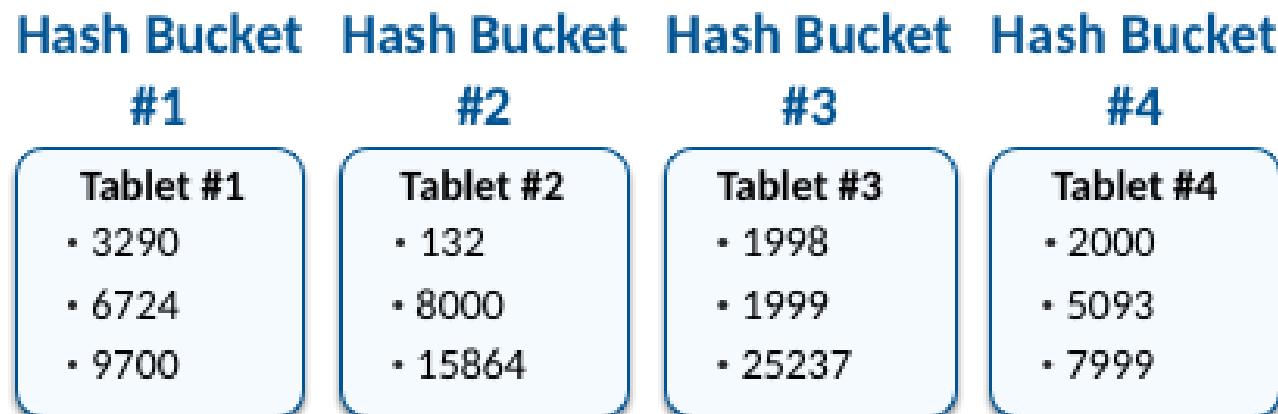
- **Data access is implemented through**
 - Primary key design
 - Good design makes key-based lookup more efficient
 - Partitioning design
 - Base your design on the type of workload
- **Factors to consider when designing primary key and partitioning**
 - The data model
 - The nature of the workload that will access the table

Partitioning Overview

- **Rows are assigned to tablets according to a partitioning strategy**
 - Goal: Distribute reads and writes across tablet servers
 - Rows are partitioned based on the value of one or more *partition keys*
 - A partition key column must be one of the primary key columns
- **The partitioning strategy is defined on a per-table basis**
 - The default is no partitioning
 - All rows stored in the same partition
 - Very inefficient for large amounts of data
- **For write-heavy workloads, spread writes across tablets**
 - This prevents overloading a single tablet server
- **For workloads involving many short scans, keep data together**
 - Performance is best if all data for the scan is in the same tablet

Partitioning Options (1)

- **Hash partitioning:** Uses a hash code calculated from the partition column
- **Example: customers_hash_id table**
 - Columns
 - customer_id (integer)
 - name (string)
 - ...
 - Primary key column: customer_id
 - Partitioning strategy
 - Hash partitioning on customer_id column into four buckets



Partitioning Options (2)

- **Range partitioning:** Uses lexicographical ordering of the partition column
- **Example: customers_range_date table**
 - Columns
 - customer_id (integer) name (string)
 - created (string) ...
 - Primary key columns: customer_id, created
 - Partitioning strategy
 - Initially partition on created column into three partitions by month
 - Add new partitions for subsequent months

Range < 2018-01 Range = 2018-01 Range = 2018-02

Tablet #1

- 2001, 2016-11
- 7623, 2017-08

Tablet #2

- 2000, 2018-01
- 4922, 2018-01
- 8051, 2018-01

Tablet #3

- 1182, 2018-02
- 7722, 2018-02
- 9541, 2018-02

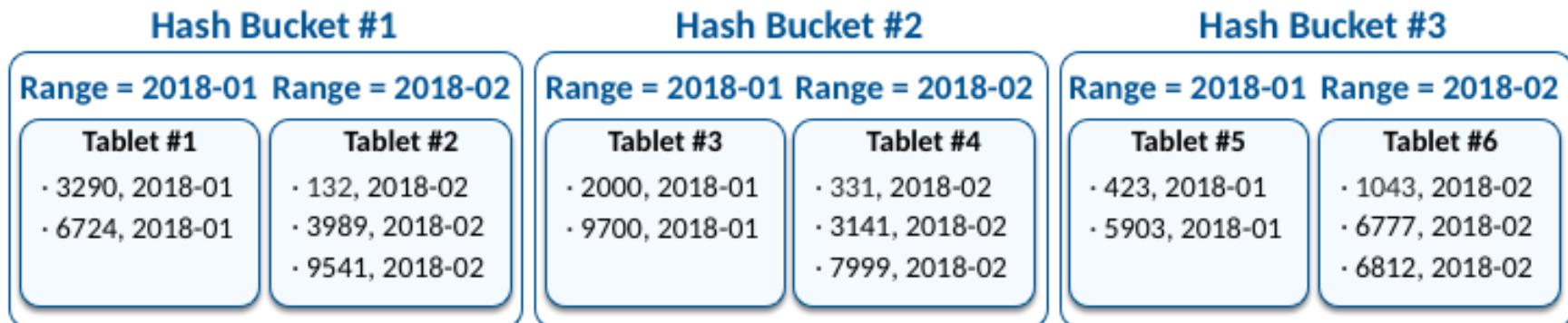
Partitioning Options (3)

- **Multilevel partitioning:** Offers the benefits of both
- **Example: customers_multi table**

- Columns

customer_id (integer)	name (string)
created (string)	...

- Primary key columns: `customer_id, created`
 - Partitioning strategy
 - Hash partitioning on `customer_id` column into three buckets
 - Initial range partitioning on `created` column into two ranges by month
 - Add new ranges for subsequent months



Chapter Topics

Apache Kudu

- What Is Kudu?
- Kudu Tables
- **Using Impala with Kudu**
- Essential Points
- Hands-On Exercise: Using Apache Kudu with Apache Impala

Apache Impala with Kudu

- **Impala and Kudu are tightly integrated**
 - Kudu tables are *not* accessible using Apache Hive
- **Impala SQL is a simpler alternative to the Kudu C++, Java, or Python APIs**
- **You can issue Impala SQL commands to**
 - Create and modify Kudu tables
 - Insert, update, and delete data in Kudu tables
 - Query Kudu tables

Creating an Impala Kudu Table

- **Use SQL command CREATE TABLE ... STORED AS KUDU to create an Impala Kudu table**
 - Creates an Impala table and, if needed, a corresponding Kudu table
- **Command must always specify a primary key based on one or more columns**
 - Use PRIMARY KEY(*column*, *column*,...)
- **Command typically includes a partitioning strategy**
 - Use PARTITION BY
 - If omitted, only one partition is created
 - This is inefficient and not recommended

Example: Creating an Impala Kudu Table

- By default, creating an Impala table creates a Kudu table at the same time
- Example: Create a table for makes and models of mobile devices

```
CREATE TABLE analyst_loudacre.devices
(
    devnum INTEGER NOT NULL,
    released STRING,
    make STRING,
    model STRING,
    dev_type STRING,
    PRIMARY KEY(devnum)
)
PARTITION BY HASH(devnum) PARTITIONS 4
STORED AS KUDU;
```

Impala Kudu Table Names (1)

- Impala allows tables to be grouped together in a namespace called a database
 - Tables in different databases can have the same name
- Kudu does not have a similar notion of a database or namespace
 - Kudu table names must be unique
- By default, Kudu table names are constructed based on the Impala database and table names
- Example

```
CREATE TABLE analyst_loudacre.devices
...
STORED AS KUDU;
```

- Impala database: `analyst_loudacre`
- Impala table: `devices`
- Kudu table: `impala::analyst_loudacre.devices`

Impala Kudu Table Names (2)

- Set the table property `kudu.table_name` to give it a name different from the generated default

```
CREATE TABLE analyst_loudacre.devices
  ...
  STORED AS KUDU
  TBLPROPERTIES(
    'kudu.table_name' = 'devices_kudu'
  );
```

- Impala database: `analyst_loudacre`
- Impala table: `devices`
- Kudu table: `devices_kudu`

Setting Table Replication Level

- Set the `kudu.num_tablet_replicas` property to set table replication level
 - Each tablet will be replicated on specified number of servers
 - Provides fault-tolerance
 - Improves performance

```
CREATE TABLE analyst_loudacre.devices
...
STORED AS KUDU
TBLPROPERTIES(
  'kudu.num_tablet_replicas' = '5'
);
```

Creating an Impala Table from an Existing Kudu Table

- You can create a new Impala table to query data in an existing Kudu table
 - Set `kudu.table_name` property to Kudu table's name
- Impala table's schema is inferred from the Kudu table
- Example

```
CREATE EXTERNAL TABLE analyst_loudacre.devices_external
  STORED AS KUDU
  TBLPROPERTIES(
    'kudu.table_name' = 'devices_spark'
  );
```

Creating New Kudu Impala Table from Existing Impala Table

- Use `CREATE TABLE ... AS SELECT` to create a new Impala table and the corresponding Kudu table
 - Commonly known as CTAS
- New table's schema based on `SELECT` result set
- New table is loaded with `SELECT` result set data from existing Impala table
- Example

```
CREATE TABLE analyst_loudacre.devices_ctas
  PRIMARY KEY (devnum)
  PARTITION BY HASH(devnum) PARTITIONS 3
  STORED AS KUDU
  AS SELECT devnum,make,model FROM analyst_loudacre.devices;
```

Impala Data Types Not Supported by Kudu

- These Impala data types are not currently supported by Kudu
 - CHAR
 - VARCHAR
 - REAL
 - ARRAY
 - MAP
 - STRUCT

Selecting Data from a Table

- Use standard SELECT statements in Impala to query a Kudu table
- Impala automatically “pushes down” supported *predicates* to Kudu
 - Example: WHERE clause predicates filter out rows from a query

```
SELECT * FROM devices WHERE id > 1000;
```

Kudu filters out unneeded rows before returning data to Impala

- Use supported predicates where possible to improve performance
- Supported predicates include
 - Integer comparison operators: =, >, <, >=, <=, BETWEEN
 - String comparison operators: =, >, <, >=, <=
 - IS NULL and IS NOT NULL
 - IN LIST

Deleting Kudu Tables

- **Use DROP TABLE as you normally would in Impala**
- **You need to delete a Kudu table manually if**
 - You deleted an external Impala table and want to delete the corresponding Kudu table
 - You want to delete a table created using the Kudu API
- **Use the Kudu command-line tool to delete Kudu tables**
 - Will not delete corresponding Impala table (if it still exists)

```
$ kudu table delete kudu-master:port table_name
```

Inserting New Rows

- Use standard SQL **INSERT** command to add one or more rows to a table
- Example: Insert a single row

```
INSERT INTO analyst_loudacre.devices VALUES  
(140,'2010-05-25','Titanic','2200','phone');
```

- Example: Insert multiple rows

```
INSERT INTO analyst_loudacre.devices VALUES  
(150,'2012-08-10','iFruit','4','phone'),  
(151,'2012-08-10','iFruit','4t','tablet');
```

- All rows must have unique keys
 - Kudu will ignore inserted rows with existing keys
 - Operation will give a warning message

Updating Existing Rows

- Use standard SQL UPDATE command to change values in existing rows

```
UPDATE analyst_loudacre.devices SET model='5a' WHERE  
model='5';
```

- Use UPSERT to change values in existing rows or to add a new row
 - All values in the row must be specified

```
UPSERT INTO analyst_loudacre.devices  
VALUES(152,'2017-01','Titanic','Model D','phone');
```

- You cannot change the values of primary key columns
 - Or have NULL values for primary keys

Deleting Rows

- Use standard SQL DELETE command to delete a row or rows from a table

```
DELETE FROM analyst_loudacre.devices  
WHERE make = 'Ronin'
```

- Truncate a Kudu table with an unqualified DELETE:

```
DELETE FROM analyst_loudacre.devices
```

Bulk Insert

- **Approaches to inserting rows in bulk**
 - Impala `INSERT` or `UPSERT` statements with multiple `VALUES` clauses
 - Loading data from an existing table
 - `INSERT INTO ... SELECT`
 - `CREATE TABLE ... AS SELECT`
 - Programmatic batch insert
 - Use Spark, Python, C++, or Java API

Bulk Insert: INSERT and UPSERT

- **Use INSERT or UPSERT command**
 - INSERT will insert any rows with new primary keys and ignore existing rows
 - UPSERT will insert new rows with new primary keys and change the values for existing rows
- **Avoid commands that insert a single row**
 - Latency cost is high when executing multiple single commands
- **Impala batches new records before sending the requests to Kudu**
 - Batch size is 1024 records by default
 - You can change the `batch_size` property for an Impala session

```
SET batch_size = 10000;
```

Bulk Insert: Using SELECT

- Insert values into an existing Impala Kudu table by querying another existing Impala table
 - Use a SELECT clause with INSERT or UPSERT

```
INSERT INTO kudu_accounts
SELECT * FROM impala_accounts;
```

- Create a new table and insert data from an existing table using CREATE TABLE ... AS SELECT
- Advantages
 - Best performance
 - Select only needed columns and rows

Specifying Hash Partitioning

- Hash partitioning assigns rows to tablets using a hash code
- Use PARTITION BY HASH(*hash-column-name*)
 - Multiple hash columns are allowed
 - Hash columns must be primary key columns
 - If you do not specify hash column(s), it will use all primary key columns
- Set number of buckets using PARTITIONS *num-buckets*

```
CREATE TABLE analyst_loudacre.devices (
    devnum INTEGER,
    released STRING,
    make STRING,
    ...
    PRIMARY KEY(devnum)
)
PARTITION BY HASH(devnum) PARTITIONS 3
STORED AS KUDU;
```

Specifying Range Partitioning

- Range partitioning distributes sequential rows to tablets based on key value
- Use PARTITION BY RANGE(*range-column-name*)
 - Range column must be a primary key column
- Specify PARTITION clauses with value comparison operators

```
CREATE TABLE analyst_loudacre.devices_range (
    devnum INTEGER,
    released STRING,
    ...
    PRIMARY KEY(devnum, released)
)
PARTITION BY RANGE(released) (
    PARTITION '2010-01' <= VALUES < '2017-01',
    PARTITION VALUE = '2017-01',
    PARTITION VALUE = '2017-02',
    PARTITION VALUE = '2017-03'
)
STORED AS KUDU;
```

Specifying Multilevel Partitioning

- Multilevel partitioning combines one or more hashes with one optional range
- Use HASH and RANGE clauses together

```
CREATE TABLE analyst_loudacre.devices_multi (
    devnum INTEGER,
    released STRING,
    ...
    PRIMARY KEY(devnum,released)
)
PARTITION BY HASH(devnum) PARTITIONS 3, RANGE(released) (
    PARTITION '2010-01' <= VALUES < '2017-01',
    PARTITION VALUE = '2017-01',
    PARTITION VALUE = '2017-02',
    PARTITION VALUE = '2017-03'
)
STORED AS KUDU;
```

Adding and Removing Range Partitions

- Ranges can be added or removed after table creation
 - Add range partitions to hold new data outside existing ranges
 - Remove partitions containing unneeded data
- Warning:** Data in the dropped partitions will be lost

```
ALTER TABLE devices_range  
ADD RANGE PARTITION VALUE = '2017-04';
```

```
ALTER TABLE devices_range  
DROP RANGE PARTITION VALUE = '2017-01';
```

Chapter Topics

Apache Kudu

- What Is Kudu?
- Kudu Tables
- Using Impala with Kudu
- **Essential Points**
- Hands-On Exercise: Using Apache Kudu with Apache Impala

Essential Points

- **Apache Kudu uses a cluster to store and manage data**
 - Tables distributed across tablets, partitioned by required primary keys
 - Table schema design requires specifying primary key and partition option
 - Options are hash, range, or multilevel (uses both hash and range)
- **Create Kudu tables in Impala**
 - Use `STORED AS KUDU` with `CREATE TABLE`
 - Specify `PRIMARY KEY` and `PARTITION BY`
- **Delete Kudu tables from Impala using `DROP TABLE`**
 - Might also need to delete Kudu table manually
- **Insert and update rows in Kudu tables using `INSERT`, `UPDATE`, `UPSERT`**

Bibliography

The following offer more information on topics discussed in this chapter

- **Official Kudu website (with documentation)**
 - <https://kudu.apache.org/>
- **Apache Kudu Overview**
 - http://tiny.cloudera.com/Kudu_Overview
- **Kudu: Storage for Fast Analytics on Fast Data**
 - http://tiny.cloudera.com/Kudu_Paper
- **Documentation for using Impala with Kudu**
 - http://tiny.cloudera.com/Kudu_Impala_Integration
- ***Impala Guide: Using Impala to Query Kudu Tables***
 - http://tiny.cloudera.com/Kudu_Impala_Query
- **Impala SQL language reference (includes Kudu extensions)**
 - http://tiny.cloudera.com/Kudu_Impala_SQL

Chapter Topics

Apache Kudu

- What Is Kudu?
- Kudu Tables
- Using Impala with Kudu
- Essential Points
- **Hands-On Exercise: Using Apache Kudu with Apache Impala**

Hands-On Exercise: Using Apache Kudu with Apache Impala

- In this exercise, you will use Impala to create, populate, and query Impala Kudu tables
- Please refer to the Hands-On Exercise Manual for instructions