



Cloudera Data Analyst Training





Introduction

Chapter 1



Course Chapters

- **Introduction**

- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

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 Avro
 - Apache Bigtop
 - Apache Crunch
 - Apache Flume
 - Apache Hadoop
 - Apache HBase
 - Apache HCatalog
 - Apache Hive
 - Apache Impala (incubating)
 - Apache Kafka
 - Apache Kudu
 - Apache Lucene
 - Apache Mahout
 - Apache Oozie
 - Apache Parquet
 - Apache Pig
 - Apache Sentry
 - Apache Solr
 - Apache Spark
 - Apache Sqoop
 - Apache Tika
 - Apache Whirr
 - Apache ZooKeeper
- All other product names, logos, and brands cited herein are the property of their respective owners

Chapter Topics

Introduction

- **About this Course**
- About Cloudera
- Course Logistics
- Introductions

Course Objectives (1)

During this course, you will learn

- **The purpose of Apache Hadoop and its related tools**
- **The features that Apache Pig, Apache Hive, and Apache Impala (incubating) offer for data acquisition, storage, and analysis**
- **How to identify typical use cases for large-scale data analysis**
- **How to load data from relational databases and other sources**
- **How to manage data in HDFS and export it for use with other systems**
- **How Pig, Hive, and Impala improve productivity for typical analysis tasks**
- **The language syntax and data formats supported by these tools**

Course Objectives (2)

- How to design and execute queries on data stored in HDFS
- How to join diverse datasets to gain valuable business insight
- How Hive and Impala can be extended with custom functions and scripts
- How to store and query complex or nested data structures
- How to analyze structured, semi-structured, and unstructured data
- How to store and query data for better performance
- How to determine which tool is the best choice for a given task

Chapter Topics

Introduction

- About this Course
- **About Cloudera**
- Course Logistics
- Introductions



- **The leader in Apache Hadoop-based software and services**
 - Our customers include many key users of Hadoop
- **Founded by Hadoop experts from Facebook, Yahoo, Google, and Oracle**
- **Provides support, consulting, training, and certification for Hadoop users**
- **Staff includes committers to virtually all Hadoop projects**
- **Many authors of authoritative books on Apache Hadoop projects**

Cloudera Training

- **We have a variety of training courses:**
 - *Developer Training for Apache Spark and Hadoop*
 - *Cloudera Administrator Training for Apache Hadoop*
 - *Cloudera Essentials for Apache Hadoop*
 - *Cloudera Search Training*
 - *Cloudera Training for Apache HBase*
 - *Data Science at Scale using Spark and Hadoop*
- **We offer courses online OnDemand and in instructor-led physical and virtual classrooms**
- **We also offer private courses:**
 - Can be delivered on-site, virtually, or online OnDemand
 - Can be tailored to suit customer needs

Cloudera Certification

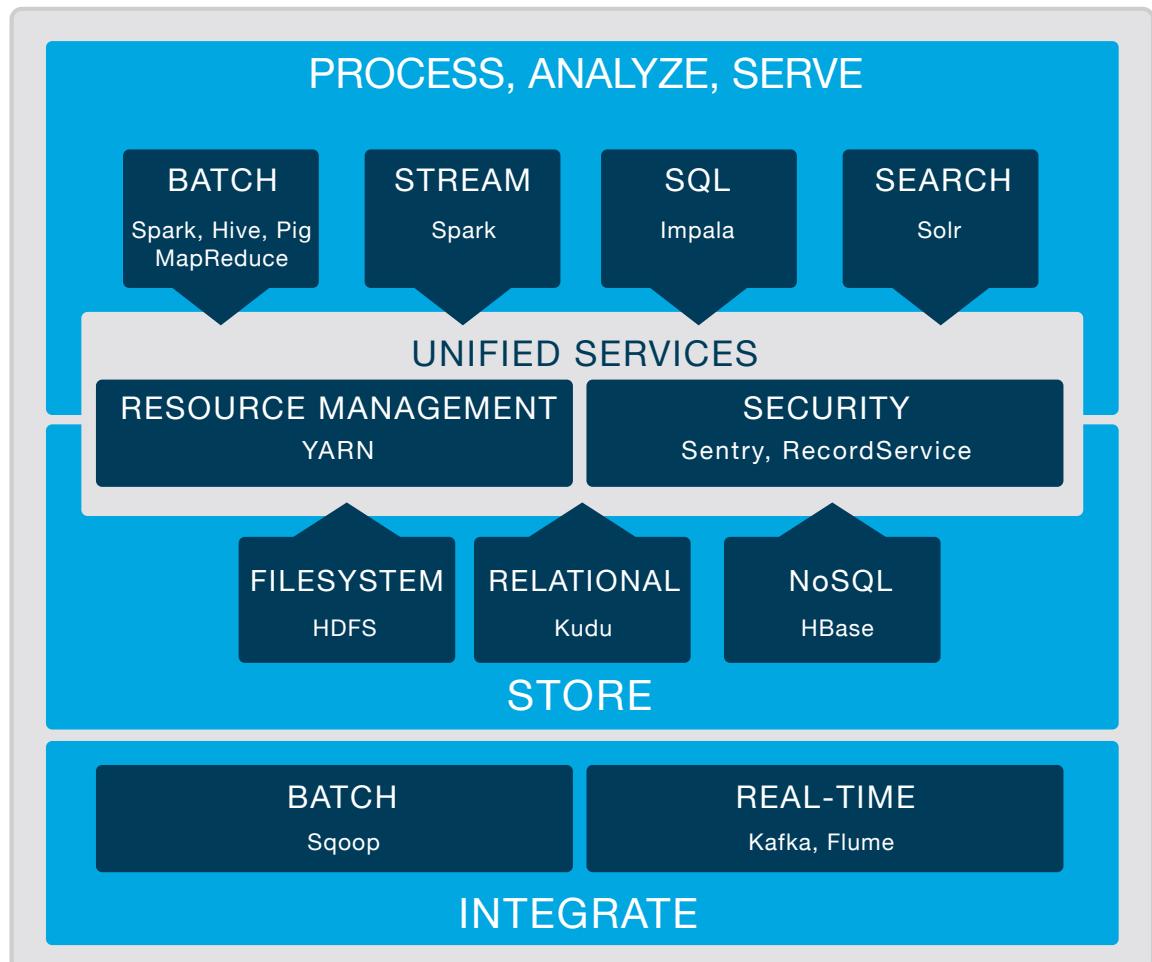
- In addition to our public training courses, Cloudera offers two levels of certifications
- **Cloudera Certified Professional (CCP)**
 - The industry's most demanding performance-based certification, CCP evaluates and recognizes a candidate's mastery of the technical skills most sought-after by employers
 - CCP Data Engineer
 - CCP Data Scientist
- **Cloudera Certified Associate (CCA)**
 - CCA exams validate foundational skills and provide the groundwork for a candidate to achieve mastery under the CCP program
 - CCA Data Analyst
 - CCA Spark and Hadoop Developer
 - Cloudera Certified Administrator for Apache Hadoop (CCAH)

Cloudera Certified Associate (CCA) Data Analyst

- **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**

CDH (Cloudera's Distribution including Apache Hadoop)

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



Cloudera Express

- **Cloudera Express**

- Completely free to download and use

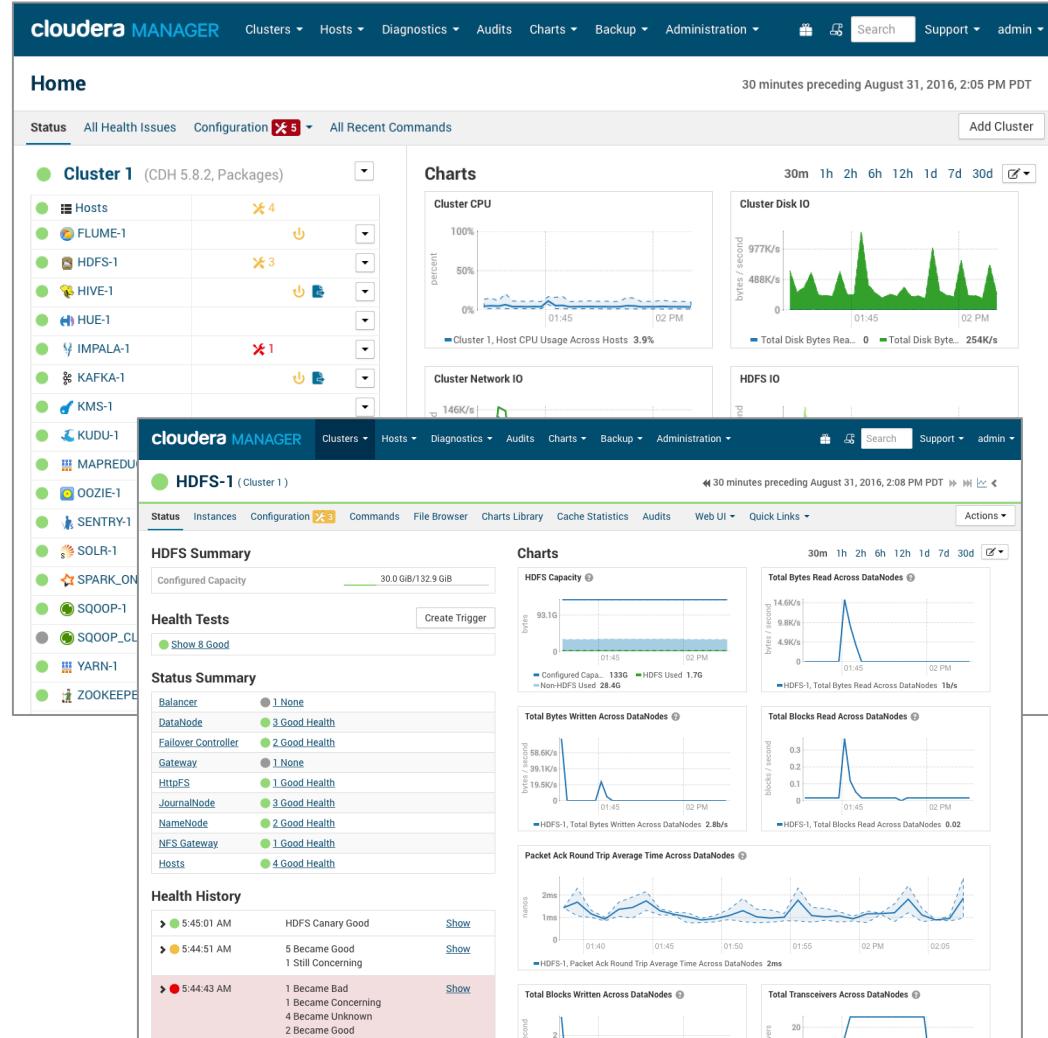
- **The best way to get started with Hadoop**

- **Includes CDH**

- **Includes Cloudera Manager**

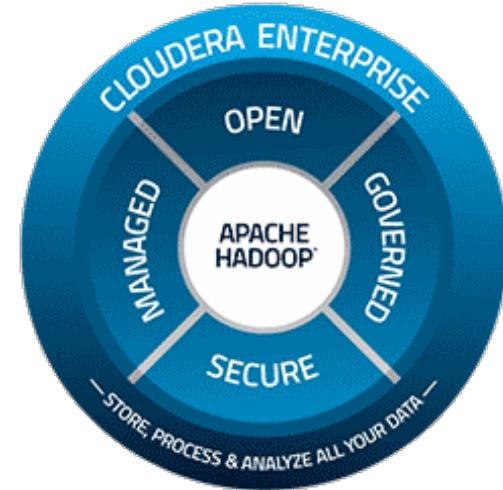
- End-to-end administration for Hadoop

- Deploy, manage, and monitor your cluster



Cloudera Enterprise

- Subscription product including CDH and Cloudera Manager
- Provides advanced features, such as
 - Operational and utilization reporting
 - Configuration history and rollbacks
 - Rolling updates & service restarts
 - External authentication (LDAP/SAML)
 - Automated backup and disaster recovery
- Specific editions offer additional capabilities, such as
 - Governance and data management (Cloudera Navigator)
 - Active data optimization (Cloudera Navigator Optimizer)
 - Comprehensive encryption (Cloudera Navigator Encrypt)
 - Key management (Cloudera Navigator Key Trustee)



Chapter Topics

Introduction

- About this Course
- About Cloudera
- **Course Logistics**
- Introductions

Logistics

- Class start and finish times
- Lunch
- Breaks
- Restrooms
- Wi-Fi access
- Hands-on environment

Your instructor will give you details on how to access the course materials and exercise instructions for the course

Chapter Topics

Introduction

- About this Course
- About Cloudera
- Course Logistics
- **Introductions**

Introductions

- **About your instructor**

- **About you**

- Where do you work? What do you do there?
- Which database(s) and platform(s) do you use?
- Do you have experience with UNIX or Linux?
- How much experience do you have with Hadoop or related tools?
- Any experience as a developer?
 - What programming languages do you use?
- What do you expect to gain from this course?



Apache Hadoop Fundamentals

Chapter 2



Course Chapters

- Introduction
- **Apache Hadoop Fundamentals**
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Apache Hadoop Fundamentals

In this chapter, you will learn

- **Which factors led to the era of Big Data**
- **What Hadoop is and what significant features it offers**
- **How Hadoop offers reliable storage for massive amounts of data with HDFS**
- **How Hadoop supports large-scale data processing through MapReduce and Spark**
- **How Hadoop ecosystem tools can boost an analyst's productivity**
- **Several ways to integrate Hadoop into the modern data center**

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: Pig, 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 2.25 billion shares are traded on the New York Stock Exchange
 - Facebook stores 4.5 billion “Likes”
 - Google processes about 24 petabytes of data
- **Every minute...**
 - Facebook users share nearly 2.5 million pieces of content
 - Email users send 204,000,000 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**
 - Automating many processes
 - Interconnecting numerous systems
 - Interacting with each other online

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
 - Predicting demand
 - 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

We Need a System that Scales

- **We're generating too much data to process with traditional tools**
- **Two key problems to address**
 - How can we reliably store large amounts of data at a reasonable cost?
 - How can we analyze all the data we have stored?

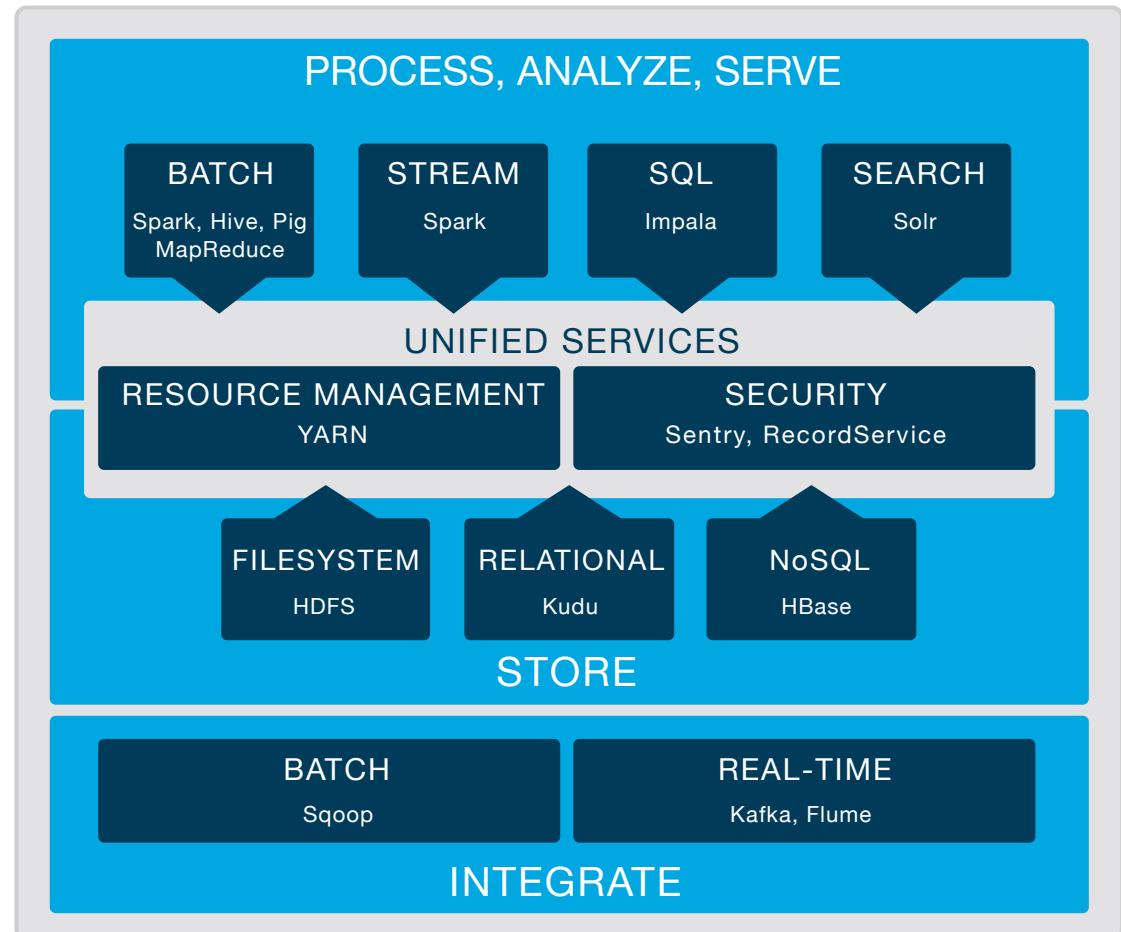
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- **Hadoop Overview**
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Pig, 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

- **Paradox: Adding nodes increases the chance that any one of them will fail**
 - Solution: Build redundancy into the system and handle it automatically
- **Files loaded into HDFS 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**

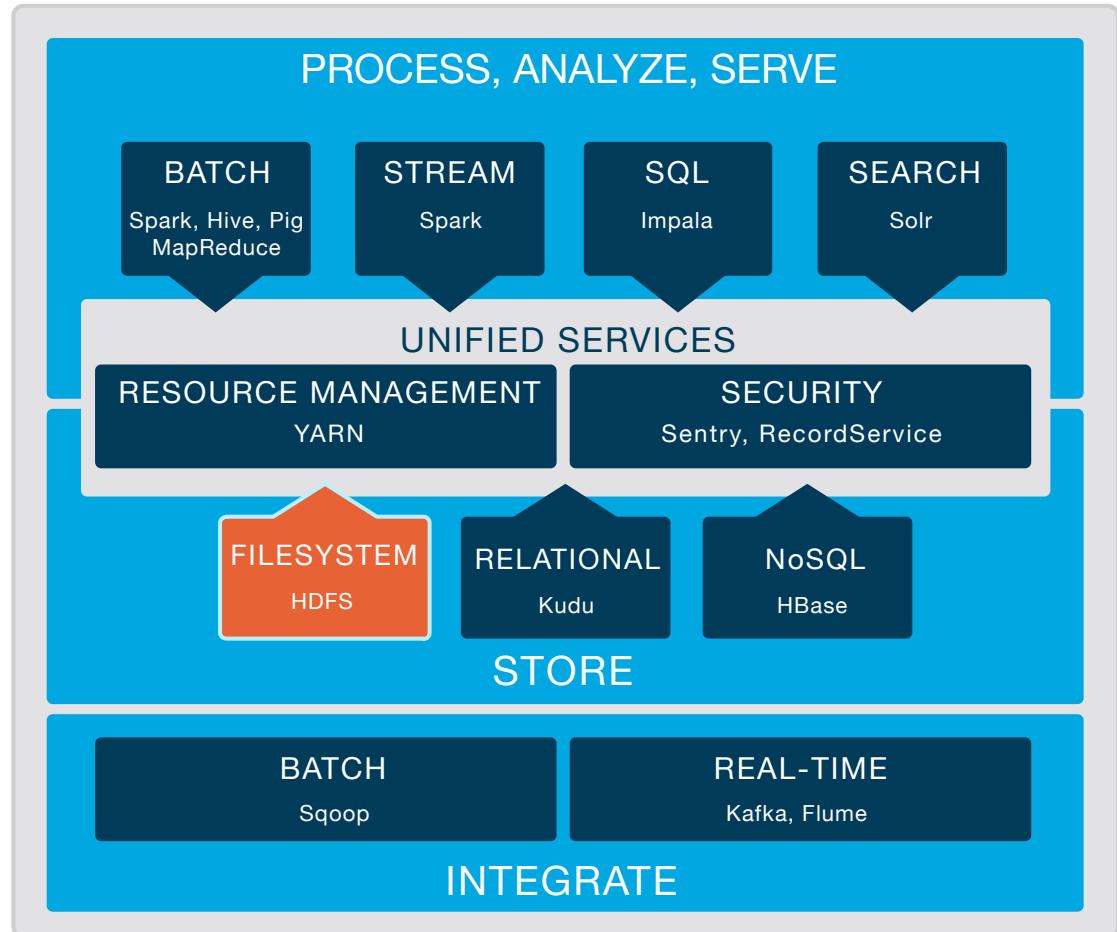
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- **Data Storage: HDFS**
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Pig, 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 provides the storage layer for Hadoop data processing**
- **Provides inexpensive and reliable storage for massive amounts of data**
- **Other Hadoop components work with data in HDFS**
 - Such as MapReduce, Hive, Impala, Pig, and Spark



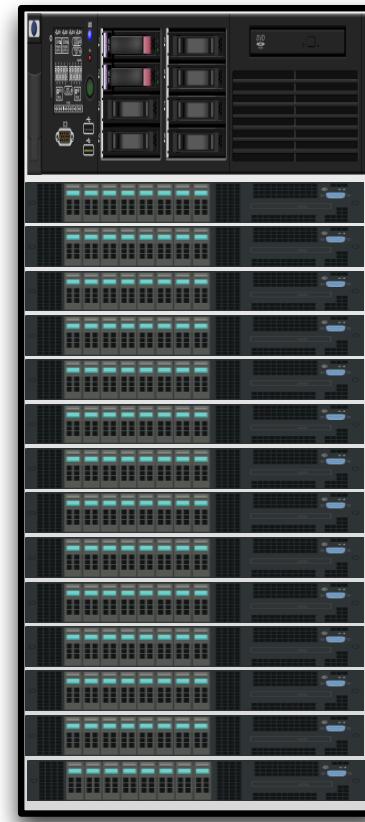
HDFS Characteristics

- **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

A Small Hadoop Cluster



← Master
HDFS NameNode

Workers
HDFS DataNodes

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**

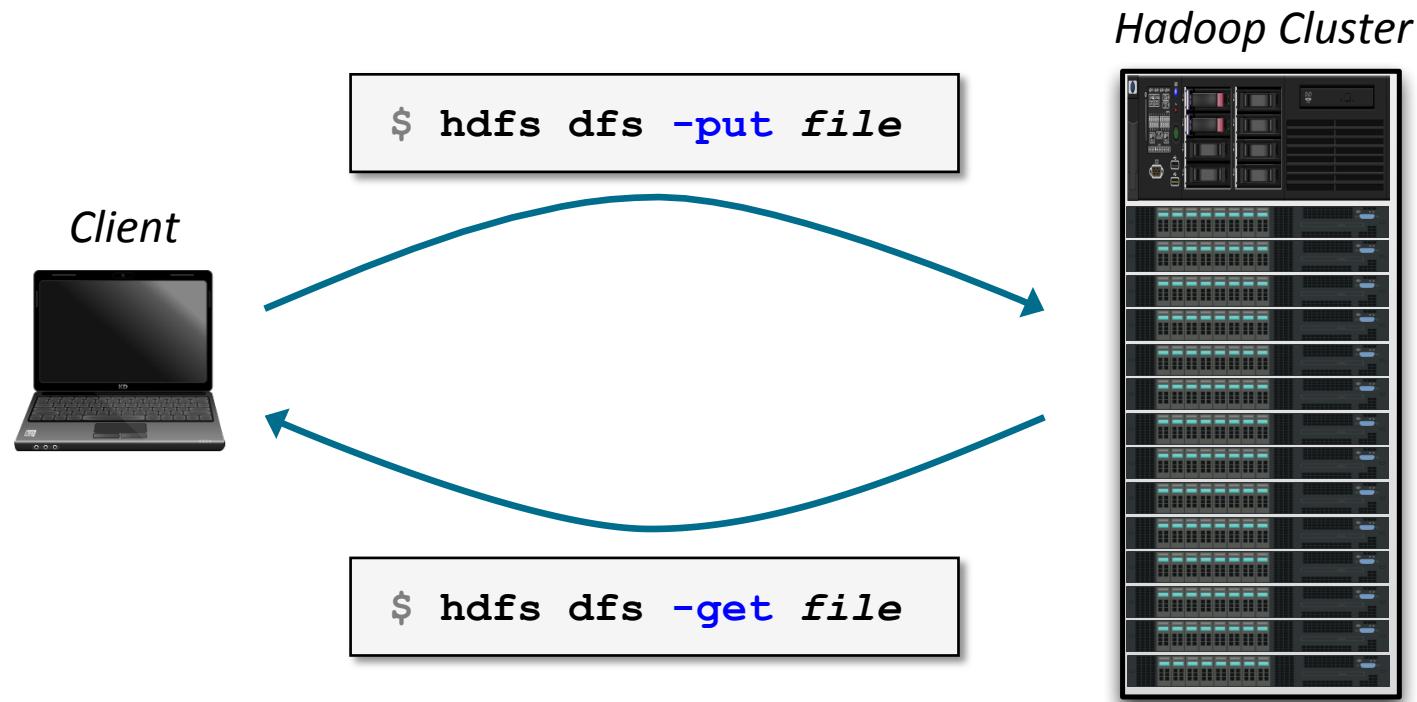
```
$ hdfs dfs -cat /user/fred/sales.txt
```

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

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

- This will copy the file to `/user/username/input.txt`
- 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/`

The screenshot shows the Hue HDFS File Manager interface. At the top, there's a navigation bar with links for Home, Query Editors, Data Browsers, Workflows, Security, and a File Browser icon (circled in red). Below the navigation bar, the title "File Browser" is displayed. On the left, there's a sidebar with a search bar, an "Actions" dropdown, and a "Move to trash" button. The main area shows a breadcrumb path: Home / dualcore. A "Manage Files" button is highlighted with a blue box and a callout. On the right, there's a "File Browser" section with "Upload" and "New" buttons, and a "History" and "Trash" link. Another "Upload Files" button is highlighted with a blue box and a callout. At the bottom, there's a table listing file and directory entries, and a footer with page navigation controls.

Name	Size	User	Group	Permissions	Last Modified
↑		hdfs	supergroup	drwxr-xr-x	August 17, 2016 02:14 PM
.		training	supergroup	drwxrwxrwx	August 29, 2016 12:36 PM
ad_data1		training	supergroup	drwxrwxrwx	August 29, 2016 12:34 PM
ad_data1.txt	30.1 MB	training	supergroup	-rw-rw-rw-	August 29, 2016 12:30 PM
ad_data2		training	supergroup	drwxrwxrwx	August 29, 2016 12:36 PM

Show 45 of 9 items

Page 1 of 1

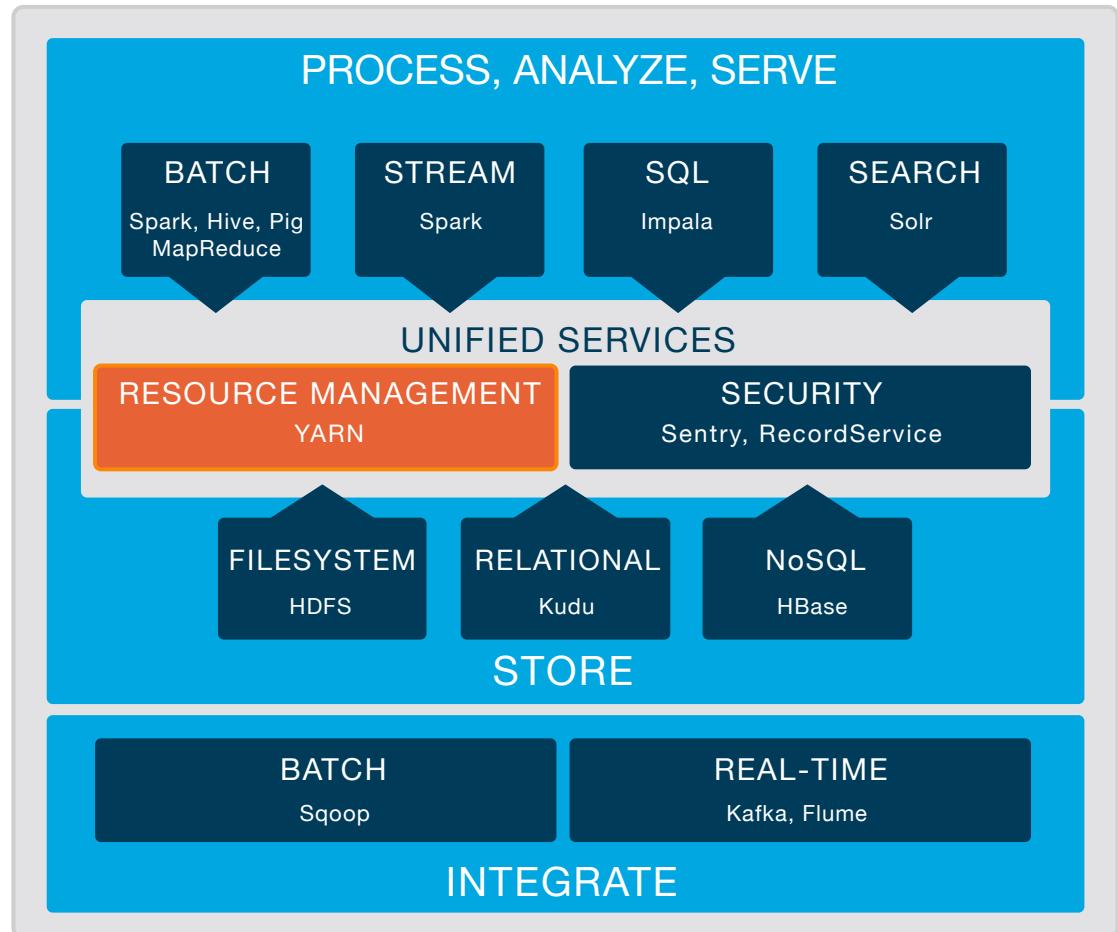
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- **Distributed Data Processing: YARN, MapReduce, and Spark**
- Data Processing and Analysis: Pig, 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 Hadoop tools work with data in a Hadoop cluster
- Distributing and monitoring work across the cluster requires workload management
- YARN (Yet Another Resource Negotiator) allocates cluster resources to workloads and monitors resource usage



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

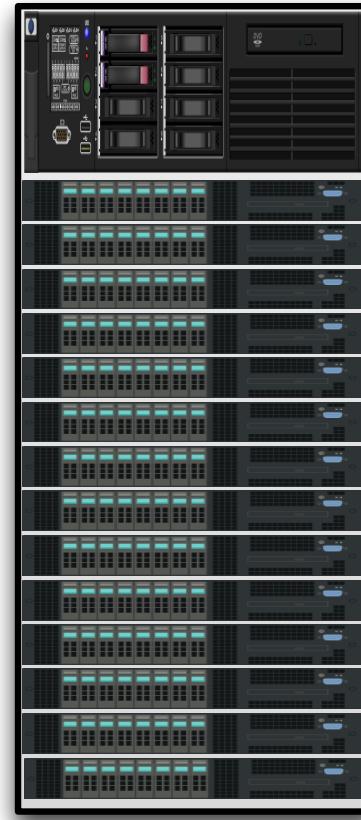
- **YARN worker daemon: NodeManager**

- Launches tasks to run on worker node, reports status back to master

- **HDFS and YARN are colocated**

- Worker nodes run both HDFS and YARN on the same machines

A Small Hadoop Cluster

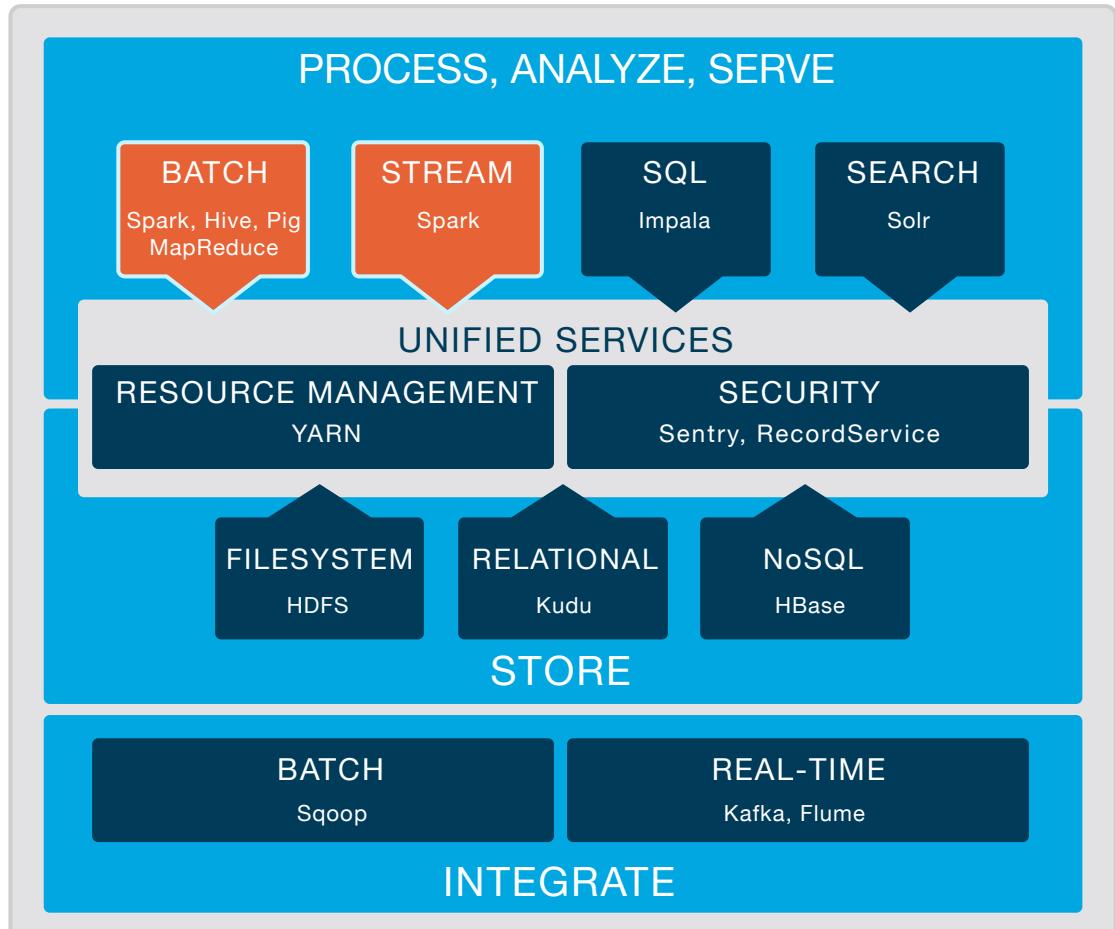


← **Master
YARN
ResourceManager**
HDFS NameNode

**Workers
YARN
NodeManagers**
HDFS DataNodes

General Data Processing

- Hadoop includes two general data processing engines
 - MapReduce
 - Spark
- Higher-level tools like Hive and Pig 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 Analysis and Processing: Pig, Hive, and Impala**
- Database Integration: Sqoop
- Other Hadoop Data Tools
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

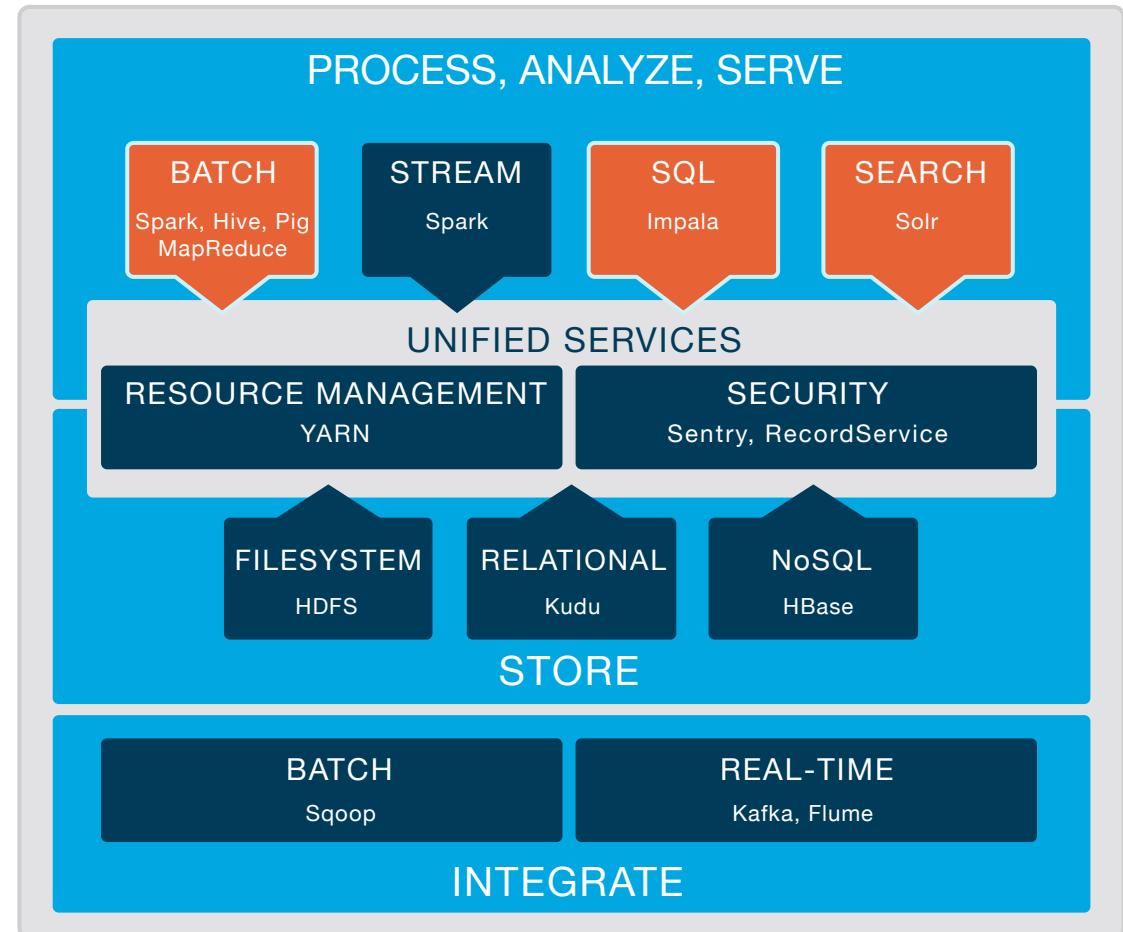
Data Processing and Analysis with Hadoop (1)

- **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

Data Processing and Analysis with Hadoop (2)

- **Higher-level abstractions for general data processing**
 - Such as Pig and Hive

- **Specialized processing engines for interactive analysis**
 - Such as Impala and Cloudera Search (based on Apache Solr)



Apache Pig

- **Pig uses Hadoop to offer high-level data processing**
 - This is an alternative to writing low-level MapReduce code
 - Pig is especially good at joining and transforming data

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

- **The Pig interpreter runs on the client machine**
 - Turns Pig Latin scripts into MapReduce jobs
 - Submits those jobs to the cluster



Apache Hive

- **Hive is another abstraction on top of Hadoop**
 - Like Pig, it also reduces development time
 - Hive 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 (Incubating)

- **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



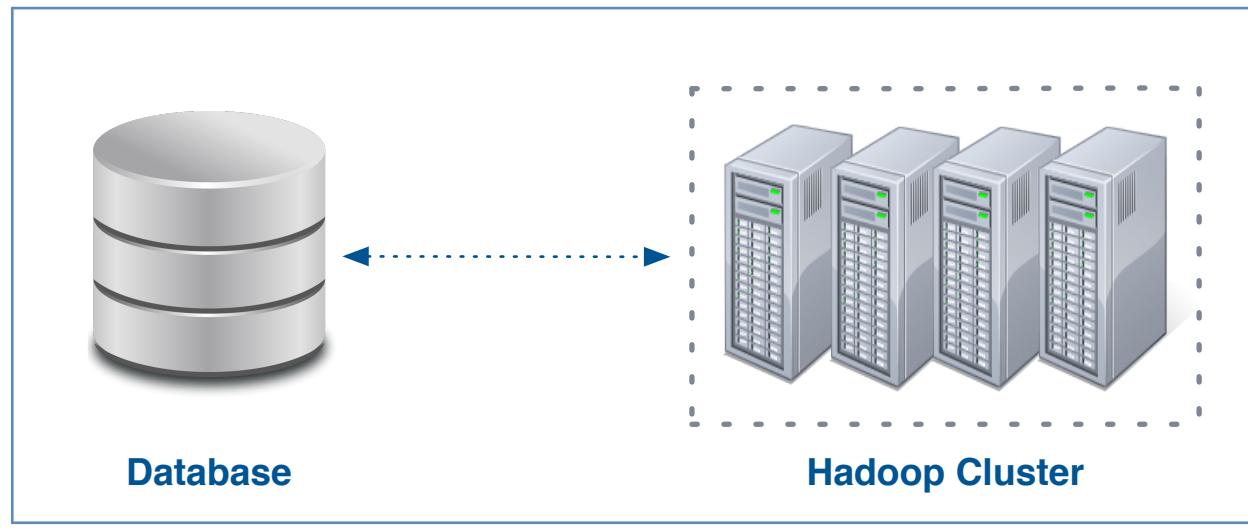
Chapter Topics

Apache Hadoop Fundamentals

- The Motivation for Hadoop
- Hadoop Overview
- Data Storage: HDFS
- Distributed Data Processing: YARN, MapReduce, and Spark
- Data Processing and Analysis: Pig, 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**
- **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

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: Pig, Hive, and Impala
- Database Integration: Sqoop
- **Other Hadoop Data Tools**
- Exercise Scenario Explanation
- Essential Points
- Hands-On Exercise: Data Ingest with Hadoop Tools

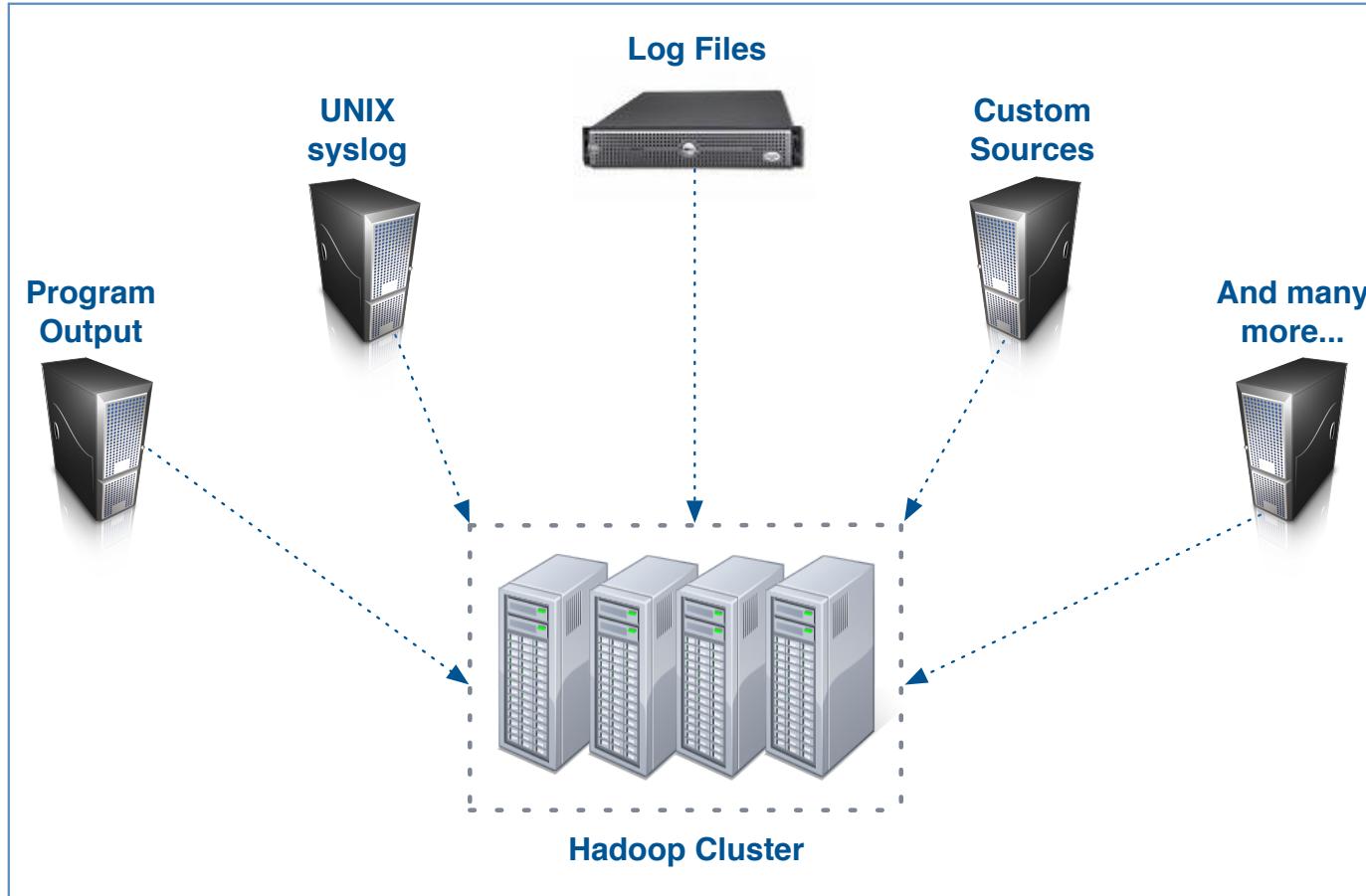
Apache HBase

- **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

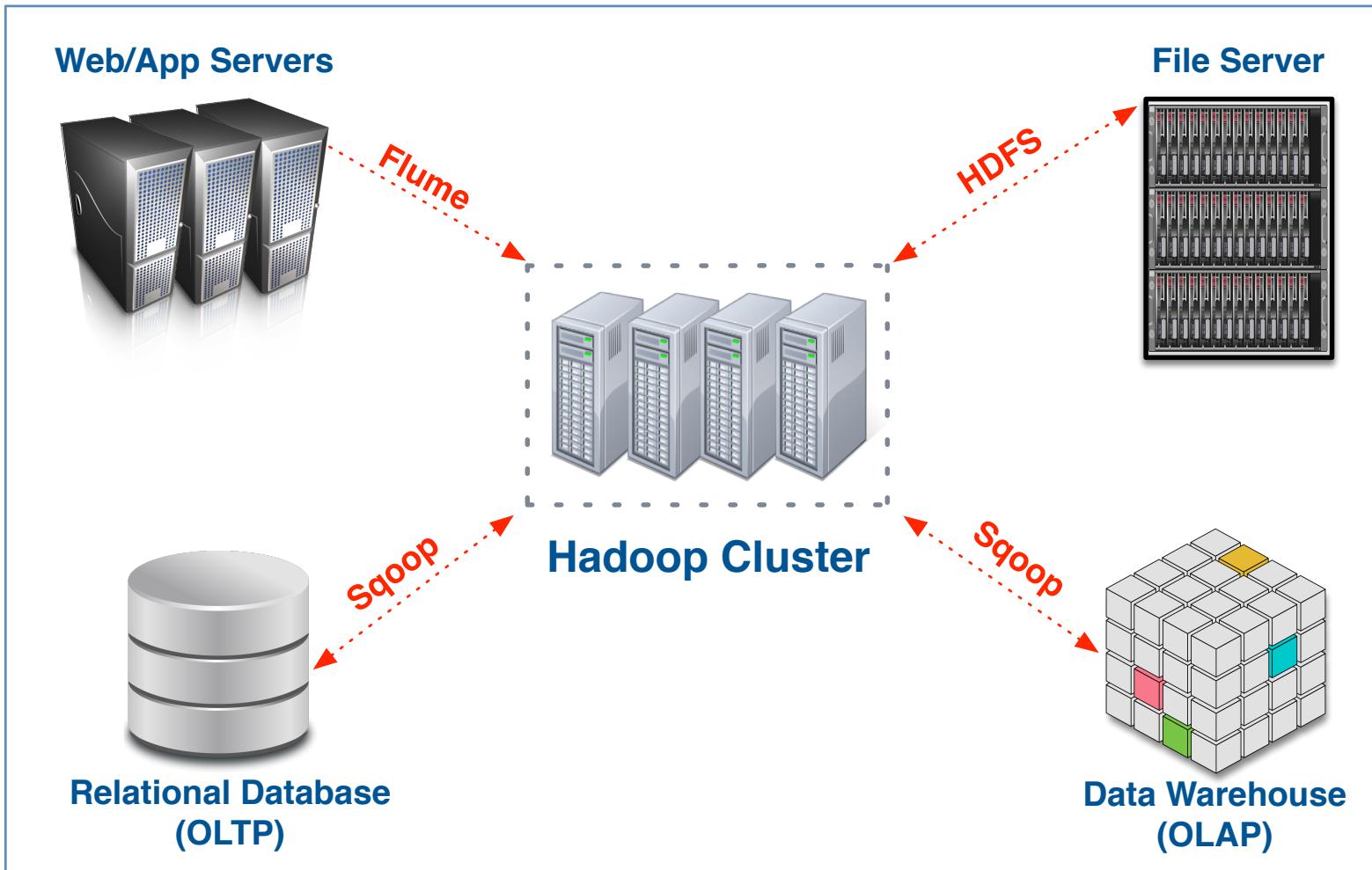


Apache Flume

- 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: Pig, 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 the kind of tasks often performed using 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: Pig, 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: Pig, 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 Exercise: Data Ingest with Hadoop Tools

- **In this Hands-On 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

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



Introduction to Apache Pig

Chapter 3



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig**
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Introduction to Apache Pig

In this chapter, you will learn

- **The key features Pig offers**
- **How organizations use Pig for data processing and analysis**
- **How to use Pig interactively and in batch mode**

Chapter Topics

Introduction to Apache Pig

- **What Is Pig?**
- Pig Features
- Pig Use Cases
- Interacting with Pig
- Essential Points

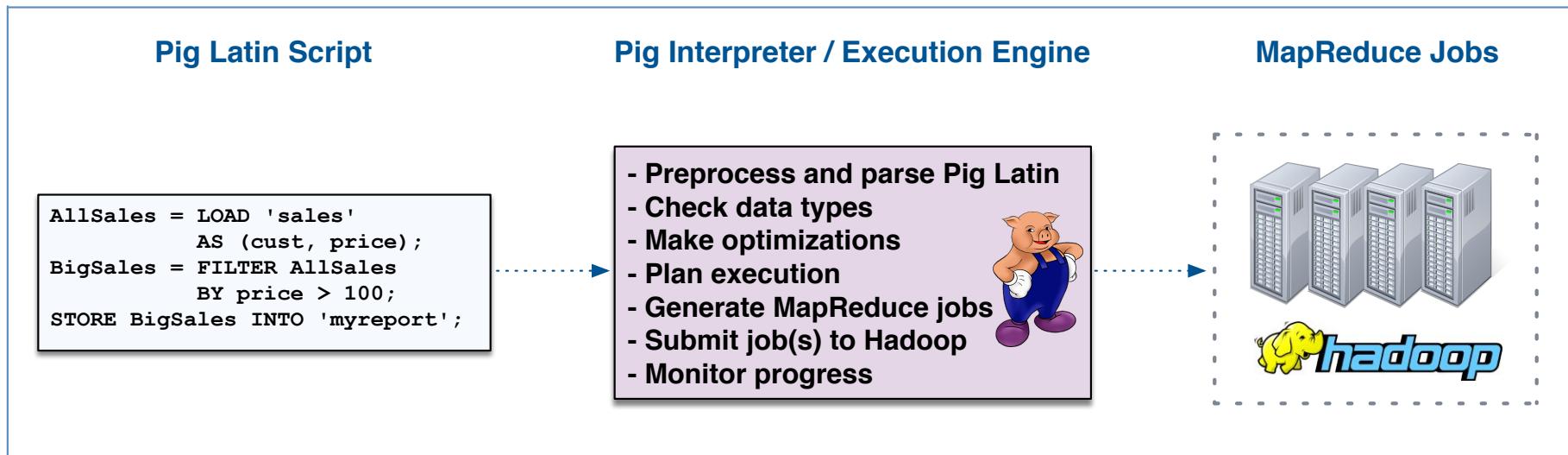
Apache Pig Overview

- **Apache Pig is a platform for data analysis and processing on Hadoop**
 - It offers an alternative to writing MapReduce code directly
- **Originally developed as a research project at Yahoo**
 - Goals: flexibility, productivity, and maintainability
 - Now an open source Apache project

The Anatomy of Pig

■ Main components of Pig

- The data flow language (Pig Latin)
- The interactive shell (Grunt) where you can type Pig Latin statements
- The Pig interpreter and execution engine



Chapter Topics

Introduction to Apache Pig

- What Is Pig?
- **Pig Features**
- Pig Use Cases
- Interacting with Pig
- Essential Points

Pig Features

- **Pig is an alternative to writing low-level MapReduce code in Java**
- **Many features enable sophisticated analysis and processing**
 - HDFS manipulation
 - UNIX shell commands
 - Relational operations
 - Positional references for fields
 - Common mathematical functions
 - Support for custom functions and data formats
 - Complex data structures

Chapter Topics

Introduction to Apache Pig

- What Is Pig?
- Pig Features
- **Pig Use Cases**
- Interacting with Pig
- Essential Points

How Are Organizations Using Pig?

- **Many organizations use Pig for *data analysis***
 - Finding relevant records in a massive dataset
 - Querying multiple datasets
 - Calculating values from input data
- **Pig is also frequently used for *data processing***
 - Reorganizing an existing dataset
 - Joining data from multiple sources to produce a new dataset

Use Case: Web Log Sessionization

- Pig can help you extract valuable information from web server log files

Web Server Log Data

```
...
10.174.57.241 - - [03/May/2016:17:57:41 -0500] "GET /s?q=widget HTTP/1.1" 200 3617 "http://www.hotbot.com/find/dualcore" "WebTV 1.2" "U=129"
10.218.46.19 - - [03/May/2016:17:57:43 -0500] "GET /de.html HTTP/1.1" 404 955 "http://www.example.com/s?q=JBuilder" "Mosaic/3.6 (X11;SunOS)"
10.174.57.241 - - [03/May/2016:17:58:03 -0500] "GET /wres.html HTTP/1.1" 200 5741 "http://www.example.com/s?q=widget" "WebTV 1.2" "U=129"
10.32.51.237 - - [03/May/2016:17:58:04 -0500] "GET /os.html HTTP/1.1" 404 955 "http://www.example.com/s?q=VMS" "Mozilla/1.0b (Win3.11)"
10.174.57.241 - - [03/May/2016:17:58:25 -0500] "GET /detail?w=41 HTTP/1.1" 200 8584 "http://www.example.com/wres.html" "WebTV 1.2" "U=129"
10.157.96.181 - - [03/May/2016:17:58:26 -0500] "GET /mp3.html HTTP/1.1" 404 955 "http://www.example.com/s?q=Zune" "Mothra/2.77" "U=3622"
10.174.57.241 - - [03/May/2016:17:59:36 -0500] "GET /order.do HTTP/1.1" 200 964 "http://www.example.com/detail?w=41" "WebTV 1.2" "U=129"
10.174.57.241 - - [03/May/2016:17:59:47 -0500] "GET /confirm HTTP/1.1" 200 964 "http://www.example.com/order.do" "WebTV 1.2" "U=129"
...
```

Process Logs



Clickstream Data for User Sessions

Recent Activity for John Smith

May 3, 2016

Search for 'Widget'

Widget Results

Details for Widget X

Order Widget X

May 12, 2016

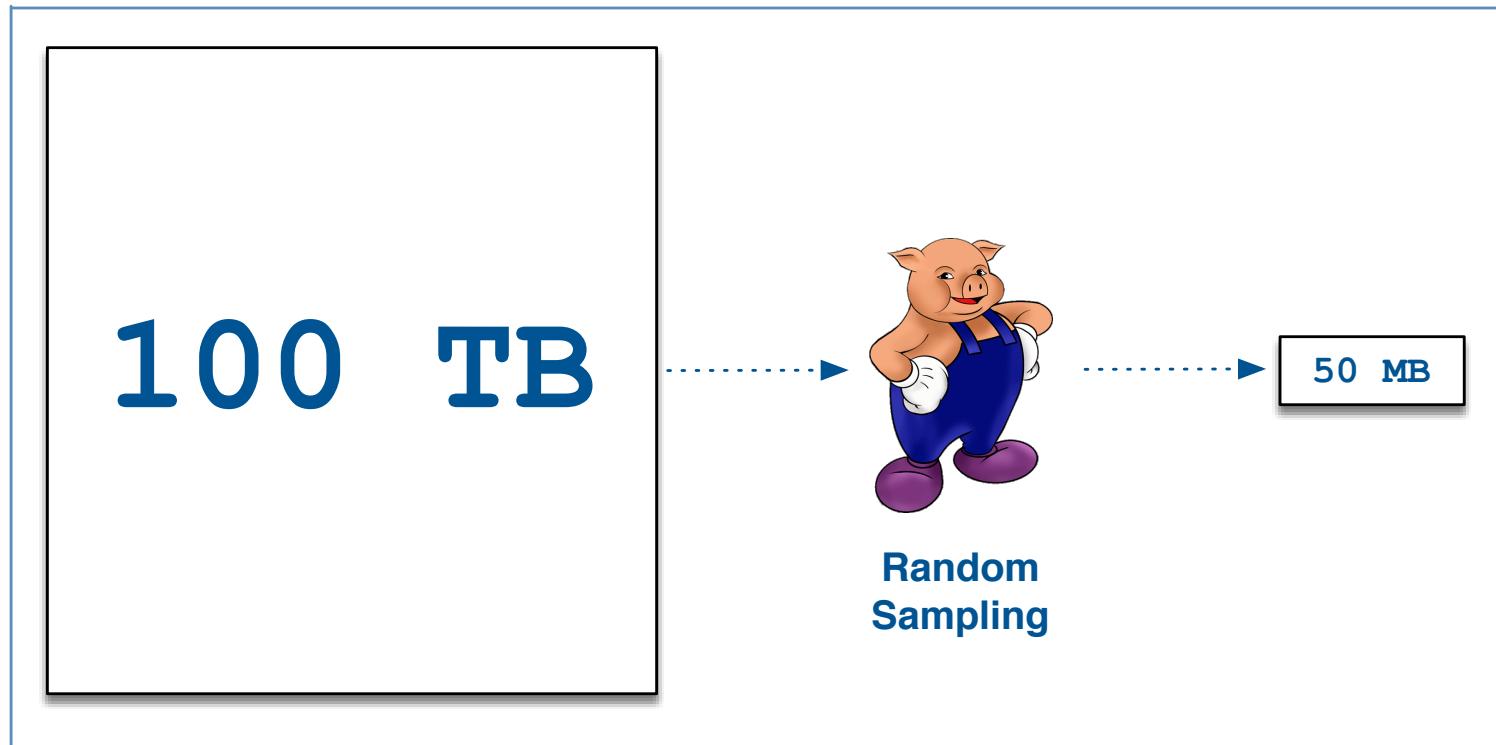
Track Order

Contact Us

Send Complaint

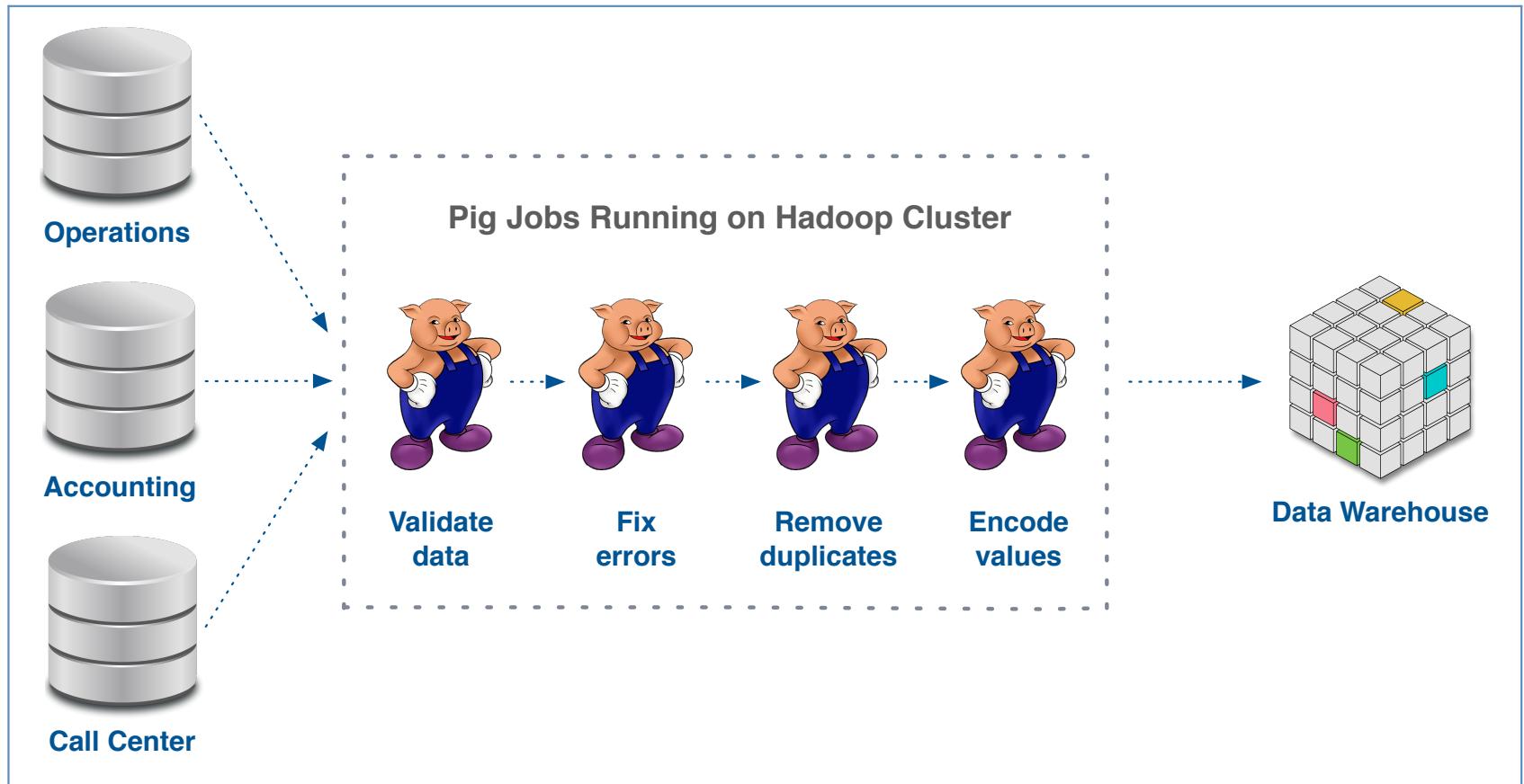
Use Case: Data Sampling

- Sampling can help you explore a representative portion of a large dataset
 - Allows you to examine this portion with tools that do not scale as well
 - Supports faster iterations during development of analysis jobs



Use Case: ETL Processing

- Pig is also widely used for Extract, Transform, and Load (ETL) processing



Chapter Topics

Introduction to Apache Pig

- What Is Pig?
- Pig Features
- Pig Use Cases
- **Interacting with Pig**
- Essential Points

Using Pig Interactively

- You can use Pig interactively in the Grunt shell
 - Pig interprets each Pig Latin statement as you type it
 - Execution is delayed until output is required
 - Very useful for ad hoc data inspection
- Example of how to start, use, and exit Grunt

```
$ pig
grunt> allsales = LOAD 'sales' AS (name, price);
grunt> bigsales = FILTER allsales BY price > 999;
grunt> STORE bigsales INTO 'myreport';
grunt> quit;
```

- Use **pig -e** to execute a Pig Latin statement from the UNIX shell

Interacting with HDFS

- You can manipulate HDFS with Pig, using the `fs` command

```
grunt> fs -mkdir sales/;
grunt> fs -put europe.txt sales/;
grunt> allsales = LOAD 'sales' AS (name, price);
grunt> bigsales = FILTER allsales BY price > 999;
grunt> STORE bigsales INTO 'myreport';
grunt> fs -getmerge myreport/ bigsales.txt;
```

Interacting with UNIX

- The **sh** command lets you run UNIX programs from Pig

```
grunt> sh date;  
Wed Nov 12 06:39:13 PST 2016  
grunt> sh ls;          -- lists local files  
grunt> fs -ls;        -- lists HDFS files
```

Running Pig Scripts

- A Pig script is simply Pig Latin code stored in a text file
 - By convention, these files have the `.pig` extension
- You can run a Pig script from within the Grunt shell using `run`
 - This is useful for automation and batch execution

```
grunt> run salesreport.pig;
```

- It is common to run a Pig script directly from the UNIX shell

```
$ pig salesreport.pig
```

MapReduce and Local Modes

- As described earlier, Pig turns Pig Latin into MapReduce jobs
 - Pig submits those jobs for execution on the Hadoop cluster
- It is also possible to run Pig in “local mode” using the **-x** flag
 - This runs jobs on the *local machine* instead of the cluster
 - Local mode uses the local filesystem instead of HDFS
 - Can be helpful for testing before deploying a job to production

```
$ pig -x local          -- interactive  
$ pig -x local salesreport.pig -- batch
```

Client-Side Log Files

- **If a job fails, Pig may produce a log file to explain why**
 - These log files are typically produced in your current working directory on the local (client) machine

Chapter Topics

Introduction to Apache Pig

- What Is Pig?
- Pig Features
- Pig Use Cases
- Interacting with Pig
- **Essential Points**

Essential Points

- **Pig offers an alternative to writing MapReduce code directly**
 - Pig interprets Pig Latin code in order to create MapReduce jobs
 - It then submits these jobs to the Hadoop cluster
- **You can execute Pig Latin code interactively through the Grunt shell**
 - Pig delays job execution until output is required
- **It is also common to store Pig Latin code in a script for batch execution**
 - Allows for automation and code reuse

Bibliography

The following offer more information on topics discussed in this chapter

- Apache Pig website
 - <http://pig.apache.org/>
- Process a Million Songs with Apache Pig
 - <http://tiny.cloudera.com/dac03a>
- Powered by Pig
 - <http://tiny.cloudera.com/poweredbypig>
- LinkedIn: User Engagement Powered By Apache Pig and Hadoop
 - <http://tiny.cloudera.com/dac03c>
- *Programming Pig* (O'Reilly book)
 - <http://tiny.cloudera.com/programmingpig>



Basic Data Analysis with Apache Pig

Chapter 4



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- **Basic Data Analysis with Apache Pig**
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Basic Data Analysis with Apache Pig

In this chapter, you will learn

- **The basic syntax of Pig Latin**
- **How to load and store data using Pig**
- **Which simple data types Pig uses to represent data**
- **How to sort and filter data in Pig**
- **How to use many of Pig's built-in functions for data processing**

Chapter Topics

Basic Data Analysis with Apache Pig

- **Pig Latin Syntax**
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly Used Functions
- Essential Points
- Hands-On Exercise: Using Pig for ETL Processing

Pig Latin Overview

- Pig Latin is a *data flow* language
 - The flow of data is expressed as a sequence of statements
- The following is a simple Pig Latin script to load, filter, and store data

```
allsales = LOAD 'sales' AS (name, price);  
  
bigsales = FILTER allsales BY price > 999; -- in US cents  
  
/*  
 * Save the filtered results into a new  
 * directory, below my home directory.  
 */  
STORE bigsales INTO 'myreport';
```

Pig Latin Grammar: Keywords

- Pig Latin keywords are highlighted here in blue text
 - Keywords are reserved; you cannot use them to name things

```
allsales = LOAD 'sales' AS (name, price);  
  
bigsales = FILTER allsales BY price > 999; -- in US cents  
  
/*  
 * Save the filtered results into a new  
 * directory, below my home directory.  
 */  
STORE bigsales INTO 'myreport';
```

Pig Latin Grammar: Identifiers (1)

- **Identifiers** are the names assigned to fields and other data structures

```
allsales = LOAD 'sales' AS (name, price);

bigsales = FILTER allsales BY price > 999; -- in US cents

/*
 * Save the filtered results into a new
 * directory, below my home directory.
 */
STORE bigsales INTO 'myreport';
```

Pig Latin Grammar: Identifiers (2)

- Identifiers must conform to Pig's naming rules
- An identifier must always begin with a letter
 - This may only be followed by letters, numbers, or underscores

Valid	Invalid
x	4
Q1	price\$
q1_2016	profit%
myData	_sale

Pig Latin Grammar: Comments

- Pig Latin supports two types of comments

- Single line comments begin with `--`
- Multi-line comments begin with `/*` and end with `*/`

```
allsales = LOAD 'sales' AS (name, price);

bigsales = FILTER allsales BY price > 999; -- in US cents

/*
 * Save the filtered results into a new
 * directory, below my home directory.
 */

STORE bigsales INTO 'myreport';
```

Case Sensitivity in Pig Latin

- Whether case is significant in Pig Latin depends on context
- Keywords (shown here in blue text) *are not* case-sensitive
 - Neither are operators (such as **AND**, **OR**, or **IS NULL**)
 - But the convention is to use uppercase for keywords and operators
- Identifiers and paths (shown here in red text) *are* case-sensitive
 - So are function names (such as **SUM** or **COUNT**) and constants

```
allsales = LOAD 'sales' AS (name, price);  
  
bigsales = FILTER allsales BY price > 999;  
  
STORE bigsales INTO 'myreport';
```

Common Operators in Pig Latin

- Many commonly used operators in Pig Latin are familiar to SQL users
 - Notable difference: Pig Latin uses `==` and `!=` for comparison

Arithmetic	Comparison	Null	Boolean
<code>+</code>	<code>==</code>	<code>IS NULL</code>	<code>AND</code>
<code>-</code>	<code>!=</code>	<code>IS NOT NULL</code>	<code>OR</code>
<code>*</code>	<code><</code>		<code>NOT</code>
<code>/</code>	<code>></code>		
<code>%</code>	<code><=</code>		
	<code>>=</code>		

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- **Loading Data**
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly Used Functions
- Essential Points
- Hands-On Exercise: Using Pig for ETL Processing

Basic Data Loading in Pig

- **Pig's default loading function is called `PigStorage`**
 - The name of the function is implicit when calling `LOAD`
 - `PigStorage` assumes text format with tab-separated columns
- **Consider the following file in HDFS called `sales`**
 - The two fields are separated by tab characters

Alice	2999
Bob	3625
Carlos	2764

- **This example loads data from the above file**

```
allsales = LOAD 'sales' AS (name, price);
```

Data Sources: Files and Directories

- The previous example loads data from a file named sales

```
allsales = LOAD 'sales' AS (name, price);
```

- Since this is not an absolute path, it is relative to your home directory
 - Your home directory in HDFS is typically /user/*youruserid*/
 - Can also specify an absolute path like /dept/sales/2016/q4
- The path can also refer to a directory
 - In this case, Pig will recursively load all files in that directory
 - File patterns (*globs*) are also supported

```
allsales = LOAD 'sales_200[5-9]' AS (name, price);
```

Specifying Column Names During Load

- The previous example also assigns names to each column

```
allsales = LOAD 'sales' AS (name, price);
```

- Assigning column names is not required

- Omitting column names can be useful when exploring a new dataset
 - Refer to fields by position (\$0 is first, \$1 is second, and so on)

```
allsales = LOAD 'sales';
bigsales = FILTER allsales BY $1 > 999;
```

Using Alternative Column Delimiters

- You can specify an alternative delimiter as an argument to `PigStorage`
- This example shows how to load comma-delimited data
 - Note that this is a single statement

```
allsales = LOAD 'sales.csv' USING PigStorage(',')  
AS (name, price);
```

- This example shows how to load pipe-delimited data without specifying column names

```
allsales = LOAD 'sales.txt' USING PigStorage('|');
```

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- **Simple Data Types**
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly Used Functions
- Essential Points
- Hands-On Exercise: Using Pig for ETL Processing

Simple Data Types in Pig

- Pig supports several simple data types
 - Similar to those in most databases and programming languages

Name	Description	Example Value
int	Whole numbers	2016
long	Large whole numbers	5,365,214,142L
float	Decimals	3.14159F
double	Very precise decimals	3.14159265358979323846
boolean	True or false values	true
datetime	Date and time	2016-05-30T14:52:39.000-04:00
chararray	Text strings	Alice
bytearray	Raw bytes (any data)	N/A

Specifying Data Types in Pig

- Pig treats a field of unspecified type as an array of bytes
 - Called the **bytearray** type in Pig

```
allsales = LOAD 'sales' AS (name, price);
```

- However, it is better to specify data types explicitly when possible
 - Helps with error checking and optimizations
 - Easiest to do this upon load using the format *fieldname : type*

```
allsales = LOAD 'sales' AS (name:chararray, price:int);
```

- Choosing the right data type is important to avoid loss of precision
- Important: Avoid using floating-point numbers to represent money!

How Pig Handles Invalid Data

- When encountering invalid data, Pig substitutes **NULL** for the value
 - For example, an **int** field containing the value **Q4**
- The **IS NULL** and **IS NOT NULL** operators test for null values
 - Note that **NULL** is not the same as the empty string ''
- You can use these operators to filter out bad records

```
has_prices = FILTER records BY price IS NOT NULL;
```

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- **Field Definitions**
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly Used Functions
- Essential Points
- Hands-On Exercise: Using Pig for ETL Processing

Key Data Concepts in Pig

- Relational databases have tables, rows, columns, and fields
- We will use the following data to illustrate Pig's equivalents
 - Assume this data was loaded from a tab-delimited text file as before

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Fields

- A **field** is a single element of data
 - For example, one of Pig's data types seen earlier

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Tuples

- A **tuple** is a collection of fields
 - Fields within a tuple are ordered, but need not all be of the same type

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Bags

- A *bag* is a collection of tuples
- Tuples within a bag are unordered by default
 - The field count and types may vary between tuples in a bag

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

Pig Data Concepts: Relations

- A ***relation*** is simply a bag with an assigned name (alias)
 - Most Pig Latin statements create a new relation
- A typical script loads one or more datasets into relations
 - Processing creates new relations instead of modifying existing ones
 - The final result is usually also a relation, stored as output

```
allsales = LOAD 'sales' AS (name, price);  
bigsales = FILTER allsales BY price > 999;  
STORE bigsales INTO 'myreport';
```

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- **Data Output**
- Viewing the Schema
- Filtering and Sorting Data
- Commonly Used Functions
- Essential Points
- Hands-On Exercise: Using Pig for ETL Processing

Data Output in Pig

- The command used to handle output depends on its destination
 - **DUMP** sends output to the screen
 - **STORE** sends output to disk (HDFS)
- Example of **DUMP** output, using data from the file shown earlier
 - The parentheses and commas indicate tuples with multiple fields

```
(Alice,2999,us)
(Bob,3625,ca)
(Carlos,2764,mx)
(Dieter,1749,de)
(Étienne,2368,fr)
(Fredo,5637,it)
```

Storing Data with Pig

- **STORE is used to store data to HDFS**
 - Similar to **LOAD**, but *writes* data instead of reading it
 - The output path is the name of a directory
 - The directory must not yet exist
- **As with LOAD, the use of PigStorage is implicit**
 - The field delimiter also has a default value (tab)

```
STORE bigsales INTO 'myreport';
```

- You may also specify an alternative delimiter

```
STORE bigsales INTO 'myreport' USING PigStorage( , );
```

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- **Viewing the Schema**
- Filtering and Sorting Data
- Commonly Used Functions
- Essential Points
- Hands-On Exercise: Using Pig for ETL Processing

Viewing the Schema with DESCRIBE

- DESCRIBE shows the structure of the data, including names and types
- The following Grunt session shows an example

```
grunt> allsales = LOAD 'sales' AS (name:chararray,  
      price:int);  
grunt> DESCRIBE allsales;  
  
allsales: {name: chararray,price: int}
```

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- **Filtering and Sorting Data**
- Commonly Used Functions
- Essential Points
- Hands-On Exercise: Using Pig for ETL Processing

Filtering in Pig

- **FILTER** extracts tuples matching the specified criteria

```
bigsales = FILTER allsales BY price > 3000;
```

allsales

name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it

bigsales

name	price	country
Bob	3625	ca
Fredo	5637	it



Price is greater than 3000

Filtering by Multiple Criteria

- You can combine criteria with AND and OR

```
somesales = FILTER allsales BY name == 'Dieter' OR (price > 3500 AND price < 4000);
```

allsales		
name	price	country
Alice	2999	us
Bob	3625	ca
Carlos	2764	mx
Dieter	1749	de
Étienne	2368	fr
Fredo	5637	it



somesales		
name	price	country
Bob	3625	ca
Dieter	1749	de

Name is Dieter, or price is greater than 3500 and less than 4000

Aside: String Comparisons in Pig

- The == operator is supported for *any* type in Pig Latin
 - This operator is used for exact comparisons

```
alices = FILTER allsales BY name == 'Alice';
```

- Pig Latin supports pattern matching through Java's *regular expressions*
 - This is done with the MATCHES operator

```
a_names = FILTER allsales BY name MATCHES 'A.*';
```

```
spammers = FILTER senders BY email_addr  
MATCHES '.*@example\\.com$';
```

Field Selection in Pig

- Filtering extracts tuples, but sometimes we need to extract columns
 - This is done in Pig Latin using FOREACH and GENERATE

```
twofields = FOREACH allsales GENERATE amount, trans_id;
```

allsales

salesperson	amount	trans_id
Alice	2999	107546
Bob	3625	107547
Carlos	2764	107548
Dieter	1749	107549
Étienne	2368	107550
Fredo	5637	107550

twofields

amount	trans_id
2999	107546
3625	107547
2764	107548
1749	107549
2368	107550
5637	107550



Generating New Fields in Pig

- **FOREACH** and **GENERATE** can also be used to *create* fields
 - For example, you could create a new field based on price

```
t = FOREACH allsales GENERATE price * 0.07;
```

- It is possible to name such fields

```
t = FOREACH allsales GENERATE price * 0.07 AS tax;
```

- And you can also specify the data type

```
t = FOREACH allsales GENERATE price * 0.07 AS tax:float;
```

Changing Data Types in Pig

- Convert fields from one data type to another using *cast operators*
 - Useful if schema is unspecified when data is loaded
 - Conversion to some data types is not supported
 - Data may be lost or truncated

```
allsales = LOAD 'sales';
sales_typed = FOREACH allsales GENERATE (chararray) $0
AS name, (int)$1 AS price;
```

Eliminating Duplicates

- **DISTINCT** eliminates duplicate records in a bag
 - All *fields* must be equal to be considered a duplicate

```
unique_records = DISTINCT all_alices;
```

all_alices

firstname	lastname	country
Alice	Smith	us
Alice	Jones	us
Alice	Brown	us
Alice	Brown	us
Alice	Brown	ca



unique_records

firstname	lastname	country
Alice	Smith	us
Alice	Jones	us
Alice	Brown	us
Alice	Brown	ca

Controlling Sort Order

- Use **ORDER . . . BY** to sort the records in a bag in ascending order
 - Add **DESC** to sort in descending order instead
 - Take care to specify a schema; data type affects how data is sorted!

```
sortedsales = ORDER allsales BY country DESC;
```

allsales

name	price	country
Alice	29.99	us
Bob	36.25	ca
Carlos	27.64	mx
Dieter	17.49	de
Étienne	23.68	fr
Fredo	56.37	it



sortedsales

name	price	country
Alice	29.99	us
Carlos	27.64	mx
Fredo	56.37	it
Étienne	23.68	fr
Dieter	17.49	de
Bob	36.25	ca

LIMITing Results

- As in SQL, you can use LIMIT to reduce the number of output records

```
somesales = LIMIT allsales 10;
```

- Beware: Record ordering is random unless specified with ORDER . . . BY
 - Use ORDER . . . BY and LIMIT together to find top N results

```
sortedsales = ORDER allsales BY price DESC;  
top_five = LIMIT sortedsales 5;
```

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- **Commonly Used Functions**
- Essential Points
- Hands-On Exercise: Using Pig for ETL processing

Built-In Functions

- **FOREACH** and **GENERATE** can be used to apply functions to data
 - For example, apply the **ROUND** function to a floating-point number

```
rounded = FOREACH allsales GENERATE ROUND(price) ;
```

- Pig includes many built-in functions
 - Math functions
 - String functions
 - Date and time functions
- Function names are case-sensitive

Built-In Math Functions

- This is a sample of Pig's built-in math functions

Function Description	Example Invocation	Input	Output
Return absolute value of a number	<code>ABS(amount)</code>	-24	24
Round up to closest whole number	<code>CEIL(price)</code>	37.19	38.0
Round down to closest whole num	<code>FLOOR(price)</code>	37.19	37.0
Return a random number	<code>RANDOM()</code>		0.481613 26652569
Round to closest whole number	<code>ROUND(price)</code>	37.19	37

Built-In String Functions

- This is a sample of Pig's built-in string functions

Function Description	Example Invocation	Input	Output
Concatenate two values	<code>CONCAT(lname, fname)</code>	<code>John, Doe</code>	<code>JohnDoe</code>
Convert to lowercase	<code>LOWER(country)</code>	<code>UK</code>	<code>uk</code>
Replace characters in a string	<code>REPLACE(date, '-' , '/')</code>	<code>12-21-16</code>	<code>12/21/16</code>
Return chars between two positions	<code>SUBSTRING(name, 1, 4)</code>	<code>Diana</code>	<code>ian</code>
Remove leading/trailing spaces	<code>TRIM(name)</code>	<code>_Bob_</code>	<code>Bob</code>
Convert to uppercase	<code>UPPER(country)</code>	<code>uk</code>	<code>UK</code>

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly Used Functions
- **Essential Points**
- Hands-On Exercise: Using Pig for ETL processing

Essential Points

- **Pig Latin supports many of the same operations as SQL**
 - Though Pig's approach is quite different
 - Pig Latin loads, transforms, and stores data in a series of steps
- **The default delimiter for both input and output is the tab character**
 - You can specify an alternative delimiter as an argument to **PigStorage**
- **Specifying the names and types of fields is not required**
 - But it can improve performance and readability of your code

Bibliography

The following offer more information on topics discussed in this chapter

- Pig Latin Basics
 - <http://tiny.cloudera.com/piglatinbasics>
- Pig Latin built-in functions
 - <http://tiny.cloudera.com/piglatinbuiltin>
- Documentation for Java regular expression patterns
 - <http://tiny.cloudera.com/javaregex>

Chapter Topics

Basic Data Analysis with Apache Pig

- Pig Latin Syntax
- Loading Data
- Simple Data Types
- Field Definitions
- Data Output
- Viewing the Schema
- Filtering and Sorting Data
- Commonly Used Functions
- Essential Points
- **Hands-On Exercise: Using Pig for ETL processing**

Hands-On Exercise: Using Pig for ETL processing

- In this Hands-On Exercise, you will write Pig Latin code to perform basic ETL processing tasks on data related to Dualcore's online advertising campaigns
 - Please refer to the Hands-On Exercise Manual for instructions



Processing Complex Data with Apache Pig

Chapter 5



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- **Processing Complex Data with Apache Pig**
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Processing Complex Data with Apache Pig

In this chapter, you will learn

- How Pig uses bags, tuples, and maps to represent complex data
- What techniques Pig provides for grouping and ungrouping data
- How to use aggregate functions in Pig Latin
- How to iterate through records in complex data structures

Chapter Topics

Processing Complex Data with Apache Pig

- **Storage Formats**
- Complex/Nested Data Types
- Grouping
- Built-In Functions for Complex Data
- Iterating Grouped Data
- Essential Points
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Storage Formats

- We have seen that **PigStorage** loads and stores data
 - Uses a delimited text file format

```
allsales = LOAD 'sales' AS (name, price);
```

- The default delimiter (tab) can easily be changed

```
allsales = LOAD 'sales' USING PigStorage(' , ')  
AS (name, price);
```

- Sometimes you need to load or store data in other formats

Other Supported Formats

- Here are a few of Pig's built-in functions for loading data in other formats
 - It is also possible to implement a custom loader by writing Java code

Name	Loads Data from
TextLoader	Text files (entire line as one field)
JsonLoader	Text files containing JSON-formatted data
BinStorage	Files containing binary data
HBaseStorage	HBase, a scalable NoSQL database built on Hadoop

Load and Store Functions

- Some functions load data, some store data, and some do both

Load Function	Equivalent Store Function
PigStorage	PigStorage
TextLoader	None
JsonLoader	JsonStorage
BinStorage	BinStorage
HBaseStorage	HBaseStorage

Chapter Topics

Processing Complex Data with Apache Pig

- Storage Formats
- **Complex/Nested Data Types**
- Grouping
- Built-In Functions for Complex Data
- Iterating Grouped Data
- Essential Points
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Pig's Complex Data Types: Tuple and Bag

- We have already seen two of Pig's three complex data types
 - A tuple is a collection of fields
 - A bag is a collection of tuples

trans_id	total	salesperson
107546	2999	Alice
107547	3625	Bob
107548	2764	Carlos
107549	1749	Dieter
107550	2368	Étienne
107551	5637	Fredo

A diagram illustrating complex data types in Pig. A blue rectangular box encloses the first five rows of the table, representing a bag. A red rectangular box encloses the third row (trans_id 107548, total 2764, salesperson Carlos), representing a tuple. A red arrow points from the word "tuple" to the red box. A blue arrow points from the word "bag" to the blue box.

Pig's Complex Data Types: Map

- Pig also supports another complex type: *map*
 - A map associates a **chararray** (key) to another data element (value)

trans_id	amount	salesperson	sales_details
107546	2999	Alice	date → 12-02-2016 SKU → 40155 store → MIA01
107547	3625	Bob	date → 12-02-2016 SKU → 3720 store → STL04 coupon → DEC13
107548	2764	Carlos	date → 12-03-2016 SKU → 76102 store → NYC15

Representing Complex Types in Pig

- It is important to know how to define and recognize these types in Pig

Type	Definition
Tuple	Comma-delimited list inside parentheses: <code>('107546', 2999, 'Alice')</code>
Bag	Braces surround comma-delimited list of tuples: <code>{ ('107546', 2999, 'Alice'), ('107547', 3625, 'Bob') }</code>
Map	Brackets surround comma-delimited list of pairs; keys and values separated by #: <code>['store'#'MIA01','location'#'Coral Gables']</code>

Loading and Using Complex Types (1)

- Complex data types can be used in any Pig field
- The following example shows how a bag is stored in a text file

Example: Transaction ID, amount, items sold (a bag of tuples)

```
107550      2368      { (40120,1999), (37001,369) }
```

Field 1

TAB

Field 2

TAB

Field 3

- Here is the corresponding LOAD statement specifying the schema

```
details = LOAD 'sales' AS (
    trans_id:chararray, amount:int,
    items_sold:bag
        {item:tuple (SKU:chararray, price:int)});
```

Loading and Using Complex Types (2)

- The following example shows how a map is stored in a text file

Example: Customer name, credit account details (map) , year account opened

Eva	[creditlimit#5000,creditused#800]	2015
-----	-----------------------------------	------

Field 1

TAB

Field 2

TAB

Field 3

- Here is the corresponding LOAD statement specifying the schema

```
credit = LOAD 'customer_accounts' AS (
    name:chararray, account:map[], year:int);
```

Referencing Map Data

- Consider a file with the following data

```
Bob      [salary#52000, age#52]
```

- And loaded with the following schema

```
details = LOAD 'data' AS (name:chararray, info:map[]);
```

- Here is the syntax for referencing data within the map and bag

```
salaries = FOREACH details GENERATE info#salary';
```

Chapter Topics

Processing Complex Data with Apache Pig

- Storage Formats
- Complex/Nested Data Types
- **Grouping**
- Built-In Functions for Complex Data
- Iterating Grouped Data
- Essential Points
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Grouping Records by a Field (1)

- Sometimes you need to group records by a given field
 - For example, so you can calculate commissions for each employee

```
Alice      729
Bob        3999
Alice     27999
Carol      32999
Carol      4999
```

- Use GROUP BY to do this in Pig Latin
 - The new relation has one record per unique value in the specified field

```
byname = GROUP sales BY name;
```

Grouping Records by a Field (2)

- The new relation always contains two fields

```
grunt> byname = GROUP sales BY name;
grunt> DESCRIBE byname;
byname: {group: chararray,sales: { (name:
chararray,price: int) } }
```

- The first field is *literally named group* in all cases
 - Contains the value from the field specified in GROUP BY
- The second field is named after the relation specified in GROUP BY
 - It is a bag containing one tuple for each corresponding value

Grouping Records by a Field (3)

- The example below shows the data after grouping

```
grunt> byname = GROUP sales BY name;  
grunt> DUMP byname;  
(Bob, { (Bob,3999) })  
(Alice, { (Alice,729), (Alice,27999) })  
(Carol, { (Carol,32999), (Carol,4999) })
```

Input Data (**sales**)

Alice	729
Bob	3999
Alice	27999
Carol	32999
Carol	4999

group
field

sales
field

Using GROUP BY to Aggregate Data

- **Aggregate functions** create one output value from multiple input values
 - For example, to calculate total sales by employee
 - Usually applied to grouped data

```
grunt> byname = GROUP sales BY name;
grunt> DUMP byname;
(Bob,{(Bob,3999)})
(Alice,{(Alice,729),(Alice,27999)})
(Carol,{(Carol,32999),(Carol,4999)})  
  
grunt> totals = FOREACH byname GENERATE
          group, SUM(sales.price);
grunt> DUMP totals;
(Bob,3999)
(Alice,28728)
(Carol,37998)
```

Grouping Everything into a Single Record

- We just saw that GROUP BY creates one record for each unique value
- GROUP ALL puts *all* data into one record

```
grunt> grouped = GROUP sales ALL;
grunt> DUMP grouped;
(all,{(Alice,729),(Bob,3999),(Alice,27999),
(Carol,32999),(Carol,4999)})
```

Using GROUP ALL to Aggregate Data

- Use GROUP ALL when you need to aggregate one or more columns
 - For example, to calculate total sales for all employees

```
grunt> grouped = GROUP sales ALL;
grunt> DUMP grouped;
(all,{(Alice,729),(Bob,3999),(Alice,27999),(Carol,32999),
(Carol,4999)})  
  
grunt> totals = FOREACH grouped GENERATE SUM(sales.price);
grunt> DUMP totals;
(70725)
```

Removing Nesting in Data

- Some operations in Pig, like grouping, produce nested data structures

```
grunt> byname = GROUP sales BY name;  
grunt> DUMP byname;  
(Bob,{(Bob,3999)})  
(Alice,{(Alice,729),(Alice,27999)})  
(Carol,{(Carol,32999),(Carol,4999)})
```

- Grouping can be useful to supply data to aggregate functions
- However, sometimes you want to work with a “flat” data structure
 - The **FLATTEN** operator removes a level of nesting in data

An Example of FLATTEN

- The following shows the nested data and what FLATTEN does to it

```
grunt> byname = GROUP sales BY name;
grunt> DUMP byname;
(Bob,{(Bob,3999)})
(Alice,{(Alice,729),(Alice,27999)})
(Carol,{(Carol,32999),(Carol,4999)})

grunt> flat = FOREACH byname GENERATE group,
        FLATTEN(sales.price);
grunt> DUMP flat;
(Bob,3999)
(Alice,729)
(Alice,27999)
(Carol,32999)
(Carol,4999)
```

Chapter Topics

Processing Complex Data with Apache Pig

- Storage Formats
- Complex/Nested Data Types
- Grouping
- **Built-In Functions for Complex Data**
- Iterating Grouped Data
- Essential Points
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Pig's Built-In Aggregate Functions

- Pig has built-in support for other aggregate functions besides SUM
- Examples:
 - AVG: Calculates the average (mean) of all values in the bag
 - MIN: Returns the smallest value in the bag
 - MAX: Returns the largest value in the bag
- Pig has two built-in functions for counting records
 - COUNT: Returns the number of non-null elements in the bag
 - COUNT_STAR: Returns the number of all elements in the bag

Other Notable Built-In Functions

- Here are some other useful Pig functions
 - See the Pig documentation for a complete list

Function	Description
BagToString	Concatenates the elements of a bag into a string (chararray)
BagToTuple	Un-nests the elements of a bag into a tuple
DIFF	Finds tuples that appear in only one of two supplied bags
IsEmpty	Used with FILTER to match bags or maps that contain no data
SIZE	Returns the size of the field (definition of size varies by data type)
TOKENIZE	Splits a text string (chararray) into a bag of individual words

Chapter Topics

Processing Complex Data with Apache Pig

- Storage Formats
- Complex/Nested Data Types
- Grouping
- Built-In Functions for Complex Data
- **Iterating Grouped Data**
- Essential Points
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Record Iteration

- We have seen that FOREACH . . . GENERATE iterates through records
- The goal is to transform records to produce a new relation
 - Sometimes to select only certain columns

```
price_column_only = FOREACH sales GENERATE price;
```

- Sometimes to create new columns

```
taxes = FOREACH sales GENERATE price * 0.07;
```

- Sometimes to invoke a function on the data

```
totals = FOREACH grouped GENERATE SUM(sales.price);
```

Nested FOREACH

- A variation on FOREACH applies a set of operations to each record
 - This is often used to apply a series of transformations in a group
- This is called a nested FOREACH
 - Allows only relational operations (for example, LIMIT, FILTER, ORDER BY)
 - GENERATE must be the last line in the block

Nested FOREACH Example (1)

- Our input data contains a list of employee job titles and corresponding salaries
- Goal: identify the three highest salaries within each title

Input Data

President	192000
Director	152500
Director	161000
Director	167000
Director	165000
Director	147000
Engineer	92300
Engineer	85000
Engineer	83000
Engineer	81650
Engineer	82100
Engineer	87300
Engineer	76000
Manager	87000
Manager	81000
Manager	75000
Manager	79000
Manager	67500

Nested FOREACH Example (2)

- First, load the data from the file
- Next, group employees by title
 - Assigned to new relation `title_group`

Input Data (excerpt)

President	192000
Director	152500
Director	161000
...	
Engineer	92300
...	
Manager	67500

```
employees = LOAD 'data' AS (title:chararray, salary:int);
title_group = GROUP employees BY title;

top_salaries = FOREACH title_group {
    sorted = ORDER employees BY salary DESC;
    highest_paid = LIMIT sorted 3;
    GENERATE group, highest_paid.salary;
};
```

Nested FOREACH Example (3)

- The nested FOREACH iterates through every record in the group (in this case, each job title)
 - It sorts each record in that group in descending order of salary
 - It then selects the top three
 - GENERATE** outputs the title and salaries

Input Data (excerpt)

President	192000
Director	152500
Director	161000
...	
Engineer	92300
...	
Manager	67500

```
employees = LOAD 'data' AS (title:chararray, salary:int);
title_group = GROUP employees BY title;

top_salaries = FOREACH title_group {
    sorted = ORDER employees BY salary DESC;
    highest_paid = LIMIT sorted 3;
    GENERATE group, highest_paid.salary;
};
```

Nested FOREACH Example (4)

Code (**LOAD** statement removed for brevity)

```
title_group = GROUP employees BY title;  
  
top_salaries = FOREACH title_group {  
    sorted = ORDER employees BY salary DESC;  
    highest_paid = LIMIT sorted 3;  
    GENERATE group, highest_paid.salary;  
};
```

Input Data (excerpt)

President	192000
Director	152500
Director	161000
...	
Engineer	92300
...	
Manager	67500

Output produced by **DUMP top_salaries;**

```
(Director,{(167000),(165000),(161000)})  
(Engineer,{(92300),(87300),(85000)})  
(Manager,{(87000),(81000),(79000)})  
(President,{(192000)})
```

Chapter Topics

Processing Complex Data with Apache Pig

- Storage Formats
- Complex/Nested Data Types
- Grouping
- Built-In Functions for Complex Data
- Iterating Grouped Data
- **Essential Points**
- Hands-On Exercise: Analyzing Ad Campaign Data with Pig

Essential Points

- **Pig has three complex data types: tuple, bag, and map**
 - A map is simply a collection of key/value pairs
- **These structures can contain simple types like int or chararray**
 - But they can also contain complex data types
 - Nested data structures are common in Pig
- **Pig provides methods for grouping and ungrouping data**
 - You can remove a level of nesting using the **FLATTEN** operator
- **Pig offers several built-in aggregate functions**

Chapter Topics

Processing Complex Data with Apache Pig

- Storage Formats
- Complex/Nested Data Types
- Grouping
- Built-In Functions for Complex Data
- Iterating Grouped Data
- Essential Points
- **Hands-On Exercise: Analyzing Ad Campaign Data with Pig**

Hands-On Exercise: Analyzing Ad Campaign Data with Pig

- In this Hands-On Exercise, you will analyze data from Dualcore's online ad campaign
 - Please refer to the Hands-On Exercise Manual for instructions



Multi-Dataset Operations with Apache Pig

Chapter 6



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- **Multi-Dataset Operations with Apache Pig**
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Multi-Dataset Operations with Apache Pig

In this chapter, you will learn

- **How we can use grouping to combine data from multiple sources**
- **What types of join operations Pig supports and how to use them**
- **How to concatenate records to produce a single dataset**
- **How to split a single dataset into multiple relations**

Chapter Topics

Multi-Dataset Operations with Apache Pig

- **Techniques for Combining Datasets**
- Joining Datasets in Pig
- Set Operations
- Splitting Datasets
- Essential Points
- Hands-On Exercise: Analyzing Disparate Datasets with Pig

Overview of Combining Datasets

- **So far, we have concentrated on processing single datasets**
 - Valuable insight often results from combining multiple datasets
- **Pig offers several techniques for achieving this**
 - Using the **GROUP** operator with multiple relations
 - Joining the data as you would in SQL
 - Performing set operations like **CROSS** and **UNION**

Example Datasets (1)

- Most examples in this chapter will involve the same two datasets
- The first is a file containing information about Dualcore's stores
- There are two fields in this relation
 1. **store_id:chararray** (unique key)
 2. **name:chararray** (name of the city in which the store is located)

stores	
A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

Example Datasets (2)

- Our other dataset is a file containing information about Dualcore's salespeople
- This relation contains three fields
 1. `person_id:int` (unique key)
 2. `name:chararray` (salesperson name)
 3. `store_id:chararray` (refers to store)

stores	
A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

salespeople		
1	Alice	B
2	Bob	D
3	Carlos	F
4	Dieter	A
5	Étienne	F
6	Fredo	C
7	George	D
8	Hannah	B
9	Irina	C
10	Jack	

Grouping Multiple Relations

- We previously learned about the **GROUP** operator
 - Groups values in a relation based on the specified field(s)
- The **GROUP** operator can also group *multiple* relations
 - In this case, using the synonymous **COGROUP** operator is preferred

```
grouped = COGROUP stores BY store_id, salespeople BY store_id;
```

- This collects values from both datasets into a new relation
 - As before, the new relation is keyed by a field named **group**
 - This **group** field is associated with one bag for each input

```
(group, {bag of records}, {bag of records})
```

↑
store_id

↑
records from **stores**

↑
records from **salespeople**

Example of COGROUP

stores

A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

salespeople

1	Alice	B
2	Bob	D
3	Carlos	F
4	Dieter	A
5	Étienne	F
6	Fredo	C
7	George	D
8	Hannah	B
9	Irina	C
10	Jack	

```
grunt> grouped = COGROUP stores BY store_id,  
salespeople BY store_id;  
  
grunt> DUMP grouped;  
(A, { (A,Anchorage) }, {(4,Dieter,A)})  
(B, { (B,Boston) }, {(1,Alice,B), (8,Hannah,B)})  
(C, { (C,Chicago) }, {(6,Fredo,C), (9,Irina,C)})  
(D, { (D,Dallas) }, {(2,Bob,D), (7,George,D)})  
(E, { (E,Edmonton) }, {})  
(F, { (F,Fargo) }, {(3,Carlos,F), (5,Étienne,F)} )  
(, {}, {(10,Jack,)})
```

Chapter Topics

Multi-Dataset Operations with Apache Pig

- Techniques for Combining Datasets
- **Joining Datasets in Pig**
- Set Operations
- Splitting Datasets
- Essential Points
- Hands-On Exercise: Analyzing Disparate Datasets with Pig

Join Overview

- **The COGROUP operator creates a nested data structure**
- **Pig Latin's JOIN operator creates a flat data structure**
 - Similar to joins in a relational database
- **A JOIN is similar to doing a COGROUP followed by a FLATTEN**
 - But with more options for handling non-matching records

Key Fields

- Like COGROUP, joins rely on a field shared by each relation

```
joined = JOIN stores BY store_id, salespeople BY store_id;
```

- Joins can also use multiple key fields

```
joined = JOIN customers BY (name, phone_number),  
accounts BY (name, phone_number);
```

Inner Joins

- The default join type in Pig Latin is an inner join

```
joined = JOIN stores BY store_id, salespeople BY store_id;
```

- An inner join excludes records with non-matching key values
 - Stores with no salespeople, salespeople with no store
- You can do an inner join on more than two relations in a single statement
 - But you must use the same key to join them

Inner Join Example

stores

A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

salespeople

1	Alice	B
2	Bob	D
3	Carlos	F
4	Dieter	A
5	Étienne	F
6	Fredo	C
7	George	D
8	Hannah	B
9	Irina	C
10	Jack	

```
grunt> joined = JOIN stores BY store_id,  
salespeople BY store_id;
```

```
grunt> DUMP joined;  
(A,Anchorage,4,Dieter,A)  
(B,Boston,1,Alice,B)  
(B,Boston,8,Hannah,B)  
(C,Chicago,6,Fredo,C)  
(C,Chicago,9,Irina,C)  
(D,Dallas,2,Bob,D)  
(D,Dallas,7,George,D)  
(F,Fargo,3,Carlos,F)  
(F,Fargo,5,Étienne,F)
```

Eliminating Duplicate Fields (1)

- As with COGROUP, the new relation still contains duplicate fields

```
grunt> joined = JOIN stores BY store_id,  
salespeople BY store_id;  
  
grunt> DUMP joined;  
(A,Anchorage,4,Dieter,A)  
(B,Boston,1,Alice,B)  
(B,Boston,8,Hannah,B)  
(C,Chicago,6,Fredo,C)  
(C,Chicago,9,Irina,C)  
(D,Dallas,2,Bob,D)  
(D,Dallas,7,George,D)  
(F,Fargo,3,Carlos,F)  
(F,Fargo,5,Étienne,F)
```

Eliminating Duplicate Fields (2)

- Use FOREACH . . . GENERATE to retain only the fields you need
 - However, it is now slightly more complex to refer to fields
 - You must *disambiguate* any fields with names that are not unique

```
grunt> DESCRIBE joined;
joined: {stores::store_id: chararray,stores::name:
chararray,salespeople::person_id: int,salespeople::name:
chararray,salespeople::store_id: chararray}

grunt> cleaned = FOREACH joined GENERATE stores::store_id,
stores::name, person_id, salespeople::name;

grunt> DUMP cleaned;
(A,Anchorage,4,Dieter)
(B,Boston,1,Alice)
(B,Boston,8,Hannah)
... (additional records omitted for brevity) ...
```

Outer Joins

- **Pig Latin allows you to specify the type of join following the field name**
 - Inner joins do not specify a join type

```
joined = JOIN relation1 BY field [LEFT|RIGHT|FULL] OUTER,  
relation2 BY field;
```

- **An outer join includes records with non-matching key values**
 - From one or both relations, depending on the type of outer join
- **Outer joins require Pig to know the schema for at least one relation**
 - Which relation requires schema depends on the join type
 - Full outer joins require schema for both relations

Left Outer Join Example

stores

A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

salespeople

1	Alice	B
2	Bob	D
3	Carlos	F
4	Dieter	A
5	Étienne	F
6	Fredo	C
7	George	D
8	Hannah	B
9	Irina	C
10	Jack	

- **Left Outer Join:** Result contains *all* records from the relation specified on the left, but only *matching* records from the one specified on the right

```
grunt> joined = JOIN stores BY store_id  
LEFT OUTER, salespeople BY store_id;
```

```
grunt> DUMP joined;  
(A,Anchorage,4,Dieter,A)  
(B,Boston,1,Alice,B)  
(B,Boston,8,Hannah,B)  
(C,Chicago,6,Fredo,C)  
(C,Chicago,9,Irina,C)  
(D,Dallas,2,Bob,D)  
(D,Dallas,7,George,D)  
(E,Edmonton,,,)  
(F,Fargo,3,Carlos,F)  
(F,Fargo,5,Étienne,F)
```

Right Outer Join Example

stores

A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

salespeople

1	Alice	B
2	Bob	D
3	Carlos	F
4	Dieter	A
5	Étienne	F
6	Fredo	C
7	George	D
8	Hannah	B
9	Irina	C
10	Jack	

- Right Outer Join: Result contains *all* records from the relation specified on the right, but only *matching* records from the one specified on the left

```
grunt> joined = JOIN stores BY store_id  
RIGHT OUTER, salespeople BY store_id;
```

```
grunt> DUMP joined;  
(A,Anchorage,4,Dieter,A)  
(B,Boston,1,Alice,B)  
(B,Boston,8,Hannah,B)  
(C,Chicago,6,Fredo,C)  
(C,Chicago,9,Irina,C)  
(D,Dallas,2,Bob,D)  
(D,Dallas,7,George,D)  
(F,Fargo,3,Carlos,F)  
(F,Fargo,5,Étienne,F)  
(,,10,Jack,)
```

Full Outer Join Example

stores

A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

salespeople

1	Alice	B
2	Bob	D
3	Carlos	F
4	Dieter	A
5	Étienne	F
6	Fredo	C
7	George	D
8	Hannah	B
9	Irina	C
10	Jack	

- Full Outer Join: Result contains *all* matching and non-matching records from *both* relations

```
grunt> joined = JOIN stores BY store_id  
      FULL OUTER, salespeople BY store_id;
```

```
grunt> DUMP joined;  
(A,Anchorage,4,Dieter,A)  
(B,Boston,1,Alice,B)  
(B,Boston,8,Hannah,B)  
(C,Chicago,6,Fredo,C)  
(C,Chicago,9,Irina,C)  
(D,Dallas,2,Bob,D)  
(D,Dallas,7,George,D)  
(E,Edmonton,,,)  
(F,Fargo,3,Carlos,F)  
(F,Fargo,5,Étienne,F)  
(,,10,Jack,)
```

Chapter Topics

Multi-Dataset Operations with Apache Pig

- Techniques for Combining Datasets
- Joining Datasets in Pig
- **Set Operations**
- Splitting Datasets
- Essential Points
- Hands-On Exercise: Analyzing Disparate Datasets with Pig

Crossing Datasets

- **JOIN finds records in one relation that match records in another**
- **Pig's CROSS operator creates the *cross product* of both relations**
 - Combines all records in both tables regardless of matching
 - In other words, all possible combinations of records

```
crossed = CROSS sizes, colors;
```

- **Careful: This can generate huge amounts of data!**

Cross Product Example

sizes

S	3.5 ml
M	7.5 ml
L	12.0 ml

colors

C	Cyan
M	Magenta
Y	Yellow
K	Black

- Generates every possible combination of records in the **sizes** and **colors** relations

```
grunt> crossed = CROSS sizes, colors;
```

```
grunt> DUMP crossed;  
(S,3.5 ml,C,Cyan)  
(S,3.5 ml,M,Magenta)  
(S,3.5 ml,Y,Yellow)  
(S,3.5 ml,K,Black)  
(M,7.5 ml,C,Cyan)  
(M,7.5 ml,M,Magenta)  
(M,7.5 ml,Y,Yellow)  
(M,7.5 ml,K,Black)  
(L,12.0 ml,C,Cyan)  
(L,12.0 ml,M,Magenta)  
(L,12.0 ml,Y,Yellow)  
(L,12.0 ml,K,Black)
```

Concatenating Datasets

- **We have explored several techniques for combining datasets**
 - They have had one thing in common: they combine horizontally
- **The UNION operator combines records vertically**
 - It adds data from input relations into a new single relation
 - Pig does not require these inputs to have the same schema
 - It does not eliminate duplicate records nor preserve order
- **This is helpful for incorporating new data into your processing**

```
both = UNION june_items, july_items;
```

UNION Example

june_items

Adapter	549
Battery	349
Cable	799
DVD	1999
HDTV	79999

july_items

Fax	17999
GPS	24999
HDTV	65999
Ink	3999

- Concatenates all records from both relations

```
grunt> both = UNION june_items, july_items;  
  
grunt> DUMP both;  
(Fax,17999)  
(GPS,24999)  
(HDTV,65999)  
(Ink,3999)  
(Adapter,549)  
(Battery,349)  
(Cable,799)  
(DVD,1999)  
(HDTV,79999)
```

Chapter Topics

Multi-Dataset Operations with Apache Pig

- Techniques for Combining Datasets
- Joining Datasets in Pig
- Set Operations
- **Splitting Datasets**
- Essential Points
- Hands-On Exercise: Analyzing Disparate Datasets with Pig

Splitting Datasets

- You have learned several ways to combine datasets into a single relation
- Sometimes you need to split a dataset into multiple relations
 - Server logs by date range
 - Customer lists by region
 - Product lists by vendor
- Pig Latin supports this with the **SPLIT** operator

```
SPLIT relation INTO relationA IF expression1,  
                    relationB IF expression2,  
                    relationC IF expression3, ...  
                    relationZ OTHERWISE;
```

- Expressions need not be mutually exclusive
- Optional keyword **OTHERWISE** designates a default relation

SPLIT Example

- Split customers into groups for a rewards program, based on lifetime value

customers	
Aiyana	9700
Bruce	23500
Charles	17800
Dustin	8900
Eva	29100
Fatma	9300
Glynn	27800
Harsha	21250
Isabel	43800
Jeff	8500
Kai	34000
Laura	7800
Mirko	24200

```
grunt> SPLIT customers INTO
          gold_program IF ltv >= 25000,
          silver_program IF ltv >= 10000
          AND ltv < 25000;

grunt> DUMP gold_program;
(Eva,29100)
(Glynn,27800)
(Isabel,43800)
(Kai,34000)

grunt> DUMP silver_program;
(Bruce,23500)
(Charles,17800)
(Harsha,21250)
(Mirko,24200)
```

Chapter Topics

Multi-Dataset Operations with Apache Pig

- Techniques for Combining Datasets
- Joining Datasets in Pig
- Set Operations
- Splitting Datasets
- **Essential Points**
- Hands-On Exercise: Analyzing Disparate Datasets with Pig

Essential Points

- **You can use COGROUP to group multiple relations**
 - This creates a nested data structure
- **Pig supports common SQL join types**
 - Inner, left outer, right outer, and full outer
 - You may need to disambiguate field names when using joined data
- **Pig's CROSS operator creates every possible combination of input data**
 - This can create huge amounts of data; use it carefully!
- **You can use a UNION to concatenate datasets**
- **In addition to combining datasets, Pig supports splitting them too**

Chapter Topics

Multi-Dataset Operations with Apache Pig

- Techniques for Combining Datasets
- Joining Datasets in Pig
- Set Operations
- Splitting Datasets
- Essential Points
- **Hands-On Exercise: Analyzing Disparate Datasets with Pig**

Hands-On Exercise: Analyzing Disparate Datasets with Pig

- In this Hands-On Exercise, you will analyze multiple datasets with Pig
 - Please refer to the Hands-On Exercise Manual for instructions



Apache Pig Troubleshooting and Optimization

Chapter 7

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- **Apache Pig Troubleshooting and Optimization**
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Apache Pig Troubleshooting and Optimization

In this chapter, you will learn

- How to control the information that Pig and Hadoop write to log files
- How Hadoop's web UI can help you troubleshoot failed jobs
- How to use SAMPLE and ILLUSTRATE to test and debug Pig jobs
- How Pig creates MapReduce jobs from your Pig Latin code
- How several simple changes to your Pig Latin code can make it run faster
- Which resources are especially helpful for troubleshooting Pig errors

Chapter Topics

Apache Pig Troubleshooting And Optimization

- **Troubleshooting Pig**
- Logging
- Using Hadoop's Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Pig Jobs
- Essential Points

Troubleshooting Overview

- **We have now covered how to use Pig for data analysis**
 - Unfortunately, sometimes your code may not work as you expect
 - It is important to remember that Pig and Hadoop are intertwined
- **Here we will cover some techniques for isolating and resolving problems**
 - We will start with a few options to the `pig` command

Helping Yourself

- We will discuss some options for the **pig** command in this chapter
 - You can view all of them by using the **-h** (help) option
 - Keep in mind that many options are advanced or rarely used
- One useful option is **-c** (check), which validates the syntax of your code

```
$ pig -c myscript.pig  
myscript.pig syntax OK
```

Getting Help from Others

- **Sometimes you may need help from others**
 - Mailing lists or newsgroups
 - Forums and bulletin board sites
 - Support services
- **You may need to provide the version of Pig and Hadoop you are using**

```
$ pig -version
Apache Pig version 0.12.0-cdh5.8.0
```

```
$ hadoop version
Hadoop 2.6.0-cdh5.8.0
```

Chapter Topics

Apache Pig Troubleshooting And Optimization

- Troubleshooting Pig
- **Logging**
- Using Hadoop's Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Pig Jobs
- Essential Points

Customizing Log Messages

- You may wish to change how much information is logged
 - A change in Hadoop can cause lots of warnings when using Pig
- Pig and Hadoop use the Log4J library, which is easily customized
- Edit the `/etc/pig/conf/log4j.properties` file to include:

```
log4j.logger.org.apache.pig=ERROR  
log4j.logger.org.apache.hadoop.conf.Configuration=ERROR
```

- Edit the `/etc/pig/conf/pig.properties` file to set this property:

```
log4jconf=/etc/pig/conf/log4j.properties
```

Customizing Log Messages on a Per-Job Basis

- Often you just want to *temporarily* change the log level
 - Especially while trying to troubleshoot a problem with your script
- You can specify a Log4J properties file to use when you invoke Pig
 - This overrides the default Log4J configuration
- Create a `customlog.properties` file to include:

```
log4j.logger.org.apache.pig=DEBUG
```

- Specify this file using the `-log4jconf` argument

```
$ pig -log4jconf customlog.properties
```

Controlling Client-Side Log Files

- When a job fails, Pig may produce a log file to explain why
 - These are typically produced in your current directory
- To use a different location, use the **-l (log)** option when starting Pig

```
$ pig -l /tmp
```

- Or set it permanently by editing **/etc/pig/conf/pig.properties**
 - Specify a different directory using the **log.file** property

```
log.file=/tmp
```

Chapter Topics

Apache Pig Troubleshooting And Optimization

- Troubleshooting Pig
- Logging
- **Using Hadoop's Web UI**
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Pig Jobs
- Essential Points

The Hadoop Web UI

- Each Hadoop daemon has a corresponding web application
 - This allows us to easily see cluster and job status with a browser
 - In pseudo-distributed mode, the hostname is **localhost**

Service	Daemon Name	Address
HDFS	NameNode	<code>http://hostname:50070/</code>
	DataNode	<code>http://hostname:50075/</code>
YARN	ResourceManager	<code>http://hostname:8088/</code>
	NodeManager	<code>http://hostname:8042/</code>

The ResourceManager Web UI (1)

- The ResourceManager offers the most useful of Hadoop's web UIs
 - It displays job status information for the Hadoop cluster
 - You can click one of the links to see details for a particular job

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes
4	0	1	3	2	768 MB	3 GB	0 B	2	2	0	1

User Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Memory Used	Memory Pending	Memory Reserved	VCores Used	VCores Pending	VCores Reserved
0	0	0	0	0 B	0 B	0 B	0	0	0

Show 20 entries Search:

ID	User	Name	Application Type	StartTime	FinishTime	State	Progress	Tracking UI
application_1473355272956_0004	training	PigLatin:x.pig	MAPREDUCE	Thu Sep 8 11:54:23 -0700	N/A	RUNNING	<div style="width: 100px; height: 10px; background-color: #ccc;"></div>	ApplicationMaster
application_1473355272956_0003	training	PigLatin:x.pig	MAPREDUCE	Thu Sep 8 11:48:57 -0700	Thu Sep 8 11:49:34 -0700	FINISHED	<div style="width: 100px; height: 10px; background-color: #ccc;"></div>	History
application_1473355272956_0002	training	PigLatin:x.pig	MAPREDUCE	Thu Sep 8 11:46:55 -0700	Thu Sep 8 11:47:31 -0700	FINISHED	<div style="width: 100px; height: 10px; background-color: #ccc;"></div>	History
application_1473355272956_0001	training	PigLatin:x.pig	MAPREDUCE	Thu Sep 8 11:43:30 -0700	Thu Sep 8 11:44:07 -0700	FINISHED	<div style="width: 100px; height: 10px; background-color: #ccc;"></div>	History

Showing 1 to 4 of 4 entries

First Previous 1 Next Last

The ResourceManager Web UI (2)

- The job detail page can help you troubleshoot a problem

Job Overview

Job Name: PigLatin:x.pig
User Name: training
Queue: root.training
State: FAILED
Uberized: false
Submitted: Thu Sep 08 11:54:23 PDT 2016
Started: Thu Sep 08 11:54:33 PDT 2016
Finished: Thu Sep 08 11:54:59 PDT 2016
Elapsed: 24sec
Diagnostics: Task failed task_1472494653255_0154_m_000000 Job failed as tasks failed. failedMaps:1 failedReduces:0

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	Thu Sep 08 11:54:27 PDT 2016	localhost:8042	logs

Task Type	Total	Complete
Map	1	0
Reduce	0	0

Attempt Type	Failed	Killed	Successful
Maps	4	0	0
Reduces	0	0	0

Naming Your Job

- **Hadoop clusters are typically shared resources**
 - There might be dozens or hundreds of others using the cluster
 - As a result, sometimes it is hard to find *your* job in the web UI
- **We recommend setting a name in your scripts to help identify your jobs**
 - Set the `job.name` property, either in the Grunt shell or in your script

```
grunt> set job.name 'Q2 2016 Sales Reporter';
```

ID	User	Name	Application Type
application_1473355272956_0005	training	PigLatin:Q2 2016 Sales Reporter	MAPREDUCE

Killing a Job

- A job that processes a lot of data can take hours to complete
 - Sometimes you spot an error in your code just after submitting a job
 - Rather than wait for the job to complete, you can kill it
- First, find the job's ID on the main page of the ResourceManager web UI



- Then, use the `yarn application -kill` command with that ID
 - Use this to kill any Hadoop job, not just Pig jobs

```
$ yarn application -kill \
application_1473355272956_0005
```

Chapter Topics

Apache Pig Troubleshooting And Optimization

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- **Data Sampling and Debugging**
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Pig Jobs
- Essential Points

Using **SAMPLE** to Create a Smaller Dataset

- Your code might process terabytes of data in production
 - But it is convenient to test with smaller amounts during development
- Use **SAMPLE** to choose a random set of records from a dataset
- This example selects about 5% of records from **bigdata**
 - Stores them in a new directory called **mysample**

```
everything = LOAD 'bigdata' ;
subset = SAMPLE everything 0.05 ;
STORE subset INTO 'mysample' ;
```

Intelligent Sampling with ILLUSTRATE

- Sometimes a random sample may lack data needed for testing
 - For example, matching records in two datasets for a **JOIN** operation
- Pig's ILLUSTRATE keyword can do more intelligent sampling
 - Pig will examine the code to determine what data is needed
 - It picks a few records that properly exercise the code
- You should specify a schema when using ILLUSTRATE
 - Pig will generate records when yours do not suffice

Using ILLUSTRATE Helps You to Understand Data Flow

- Like DUMP and DESCRIBE, ILLUSTRATE aids in debugging
 - The syntax is the same for all three

```
grunt> allsales = LOAD 'sales' AS (name:chararray, price:int);  
grunt> bigsales = FILTER allsales BY price > 999;  
grunt> ILLUSTRATE bigsales;  
(Bob,3625)
```

allsales name:chararray price:int
Bob 3625
Bob 998
bigsales name:chararray price:int
Bob 3625

General Debugging Strategies

- **Use DUMP, DESCRIBE, and ILLUSTRATE often**
- **Look at a sample of the data**
 - Verify that it matches the fields in your **LOAD** specification
 - The data might not be what you think it is
- **Look at the logs, especially task logs available from the web UI**

Chapter Topics

Apache Pig Troubleshooting And Optimization

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Data Sampling and Debugging
- **Performance Overview**
- Understanding the Execution Plan
- Tips for Improving the Performance of Pig Jobs
- Essential Points

Performance Overview

- **We have discussed several techniques for finding errors in Pig Latin code**
 - Once you get your code working, you'll often want it to work *faster*
- **Performance tuning is a broad and complex subject**
 - Requires a deep understanding of Pig, Hadoop, Java, and Linux
 - Typically the domain of engineers and system administrators
- **Most of these topics are beyond the scope of this course**
 - We will cover the basics here, and offer several performance improvement tips
 - See *Programming Pig* for detailed coverage

Chapter Topics

Apache Pig Troubleshooting And Optimization

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Data Sampling and Debugging
- Performance Overview
- **Understanding the Execution Plan**
- Tips for Improving the Performance of Pig Jobs
- Essential Points

How Pig Latin Becomes a MapReduce Job

- **Pig Latin code ultimately runs as MapReduce jobs on the Hadoop cluster**
- **However, Pig does not translate your code into Java MapReduce code**
 - Much like relational databases don't translate SQL to C language code
 - Like a database, Pig interprets the Pig Latin to develop execution plans
 - Pig's execution engine uses these to submit MapReduce jobs to Hadoop
- **EXPLAIN describes Pig's three execution plans**
 - Logical
 - Physical
 - MapReduce
- **Seeing an example job will help us better understand EXPLAIN's output**

Description of Example Code and Data

- Goal is to produce a list of per-store sales

stores

A	Anchorage
B	Boston
C	Chicago
D	Dallas
E	Edmonton
F	Fargo

sales

A	1999
D	2399
A	4579
B	6139
A	2489
B	3699
E	2479
D	5799

```
grunt> stores = LOAD 'stores'  
      AS (store_id:chararray, name:chararray);  
grunt> sales = LOAD 'sales'  
      AS (store_id:chararray, price:int);  
grunt> groups = GROUP sales BY store_id;  
grunt> totals = FOREACH groups GENERATE group,  
      SUM(sales.price) AS amount;  
grunt> joined = JOIN totals BY group,  
      stores BY store_id;  
grunt> result = FOREACH joined  
      GENERATE name, amount;  
grunt> DUMP result;  
(Anchorage,9067)  
(Boston,9838)  
(Dallas,8198)  
(Edmonton,2479)
```

Using EXPLAIN

- Using EXPLAIN rather than DUMP will show the execution plans

```
grunt> DUMP result;
(Anchorage,9067)
(Boston,9838)
(Dallas,8198)
(Edmonton,2479)
```

```
grunt> EXPLAIN result;
#-----
# New Logical Plan:
#-----
result: (Name: LOStore Schema:
stores::name#49:chararray,totals::amount#70:long)
|
|---result: (Name: LOForEach Schema:
stores::name#49:chararray,totals::amount#70:long)
```

(other lines, including physical and MapReduce plans, would follow)

Chapter Topics

Apache Pig Troubleshooting And Optimization

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- **Tips for Improving the Performance of Pig Jobs**
- Essential Points

Pig's Runtime Optimizations

- Pig does not necessarily run statements exactly as written
- It may remove unnecessary operations

```
sales = LOAD 'sales' AS (store_id:chararray, price:int);  
unused = FILTER sales BY price > 789;  
DUMP sales;
```

- It may rearrange operations for efficiency



```
grouped = GROUP sales BY store_id;  
totals = FOREACH grouped GENERATE group, SUM(sales.price);  
joined = JOIN totals BY group, stores BY store_id;  
only_a = FILTER joined BY store_id == 'A';  
DUMP only_a;
```

Optimizations You Can Make in Pig Latin Code

- **Pig's optimizer does what it can to improve performance**
 - But you know your own code and data better than it does
 - A few small changes in your code can allow additional optimizations
- **On the next few slides, we will rewrite this Pig Latin code for performance**

```
stores = LOAD 'stores' AS (store_id, name, postcode, phone);
sales = LOAD 'sales' AS (store_id, price);
joined = JOIN sales BY store_id, stores BY store_id;
DUMP joined;
groups = GROUP joined BY sales::store_id;
totals = FOREACH groups GENERATE
          FLATTEN(joined.stores::name) AS name,
          SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
region = FILTER unique BY name == 'Anchorage' OR name == 'Edmonton';
sorted = ORDER region BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Don't Produce Output You Don't Really Need

- In this case, we forgot to remove the DUMP statement
 - Sometimes happens when moving from development to production
 - And it might go unnoticed if you're not watching the terminal

```
stores = LOAD 'stores' AS (store_id, name, postcode, phone);
sales = LOAD 'sales' AS (store_id, price);
joined = JOIN sales BY store_id, stores BY store_id;
DUMP joined;
groups = GROUP joined BY sales::store_id;
totals = FOREACH groups GENERATE
    FLATTEN(joined.stores::name) AS name,
    SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
region = FILTER unique BY name == 'Anchorage' OR name == 'Edmonton';
sorted = ORDER region BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Specify Schema Whenever Possible

- Specifying schema when loading data eliminates the need for Pig to guess
 - It may choose a “bigger” type than you need (like `long` instead of `int`)
- The `postcode` and `phone` fields in the `stores` dataset were never used
 - Eliminating them in the schema ensures they'll be omitted in `joined`

```
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales = LOAD 'sales' AS (store_id:chararray, price:int);
joined = JOIN sales BY store_id, stores BY store_id;
groups = GROUP joined BY sales::store_id;
totals = FOREACH groups GENERATE
          FLATTEN(joined.stores::name) AS name,
          SUM(joined.sales::price) AS amount;
unique = DISTINCT totals;
region = FILTER unique BY name == 'Anchorage' OR name == 'Edmonton';
sorted = ORDER region BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Filter Unwanted Data as Early as Possible

- We previously did the JOIN before the FILTER
 - This produced lots of data we ultimately discarded
 - Moving the FILTER operation up makes the script more efficient
 - Caveat: We now have to filter by store ID rather than store name

```
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales = LOAD 'sales' AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN regsales BY store_id, stores BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
          FLATTEN(joined.stores::name) AS name,
          SUM(joined.regsales::price) AS amount;
unique = DISTINCT totals;
sorted = ORDER unique BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Consider Adjusting the Parallelization

- Hadoop clusters scale by processing data in parallel
 - Pig chooses the number of parallel reduce tasks based on input data size
 - However, it is often beneficial to set a value explicitly in your script
 - Your system administrator can help you determine the best value

```
set default_parallel 5;
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales = LOAD 'sales' AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN regsales BY store_id, stores BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
          FLATTEN(joined.stores::name) AS name,
          SUM(joined.regsales::price) AS amount;
unique = DISTINCT totals;
sorted = ORDER unique BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Specify the Largest Dataset Last in a Join

- We can optimize joins by specifying the larger dataset last
 - Pig will “stream” the larger dataset instead of reading it into memory
 - In this case, there are far more records in **sales** than in **stores**
 - Changing the order in the **JOIN** statement can improve performance

```
set default_parallel 5;
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales = LOAD 'sales' AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN stores BY store_id, regsales BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
          FLATTEN(joined.stores::name) AS name,
          SUM(joined.regsales::price) AS amount;
unique = DISTINCT totals;
sorted = ORDER unique BY amount DESC;
topone = LIMIT sorted 1;
STORE topone INTO 'topstore';
```

Try Using Compression on Intermediate Data

- Pig scripts often yield jobs with both a map and a reduce phase
 - Output of the map phase becomes input to the reduce phase
 - Compressing this intermediate data is easy and can boost performance
 - Your system administrator may need to install a compression library

```
set mapreduce.map.output.compress true;
set mapred.map.output.compress.codec
    org.apache.hadoop.io.compress.SnappyCodec;
set default_parallel 5;
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales = LOAD 'sales' AS (store_id:chararray, price:int);
regsales = FILTER sales BY store_id == 'A' OR store_id == 'E';
joined = JOIN stores BY store_id, regsales BY store_id;
groups = GROUP joined BY regsales::store_id;
totals = FOREACH groups GENERATE
    FLATTEN(joined.stores::name) AS name,
    SUM(joined.regsales::price) AS amount;
```

(other lines unchanged, but removed for brevity)

A Few More Tips for Improving Performance

- **Main theme: Eliminate unnecessary data as early as possible**
 - Use **FOREACH . . . GENERATE** to select only the fields you need
 - Use **ORDER BY** and **LIMIT** when you only need a few records
 - Use **DISTINCT** when you don't need duplicate records
- **Dropping records with **NULL** keys before a join can boost performance**
 - These records will be eliminated in the final output anyway
 - But Pig doesn't discard them until after the join
 - Use **FILTER** to remove records with **NULL** keys before the join

```
stores = LOAD 'stores' AS (store_id:chararray, name:chararray);
sales  = LOAD 'sales'  AS (store_id:chararray, price:int);

nonnull_stores = FILTER stores BY store_id IS NOT NULL;
nonnull_sales = FILTER sales BY store_id IS NOT NULL;

joined = JOIN nonnull_stores BY store_id, nonnull_sales BY store_id;
```

Chapter Topics

Apache Pig Troubleshooting And Optimization

- Troubleshooting Pig
- Logging
- Using Hadoop's Web UI
- Data Sampling and Debugging
- Performance Overview
- Understanding the Execution Plan
- Tips for Improving the Performance of Pig Jobs
- **Essential Points**

Essential Points

- You can boost performance by eliminating unneeded data during processing
- Pig's error messages don't always clearly identify the source of a problem
 - We recommend testing your scripts with a small data sample
 - Looking at the web UI, and especially the log messages, can be helpful

Bibliography

The following offer more information on topics discussed in this chapter

- Pig Testing and Diagnostics
 - <http://tiny.cloudera.com/pigtesting>
- Mailing list for Pig users
 - <http://tiny.cloudera.com/piglist>
- Questions tagged “apache-pig” on Stack Overflow
 - <http://tiny.cloudera.com/dac08d>
- *Programming Pig* (O'Reilly book)
 - <http://tiny.cloudera.com/programmingpig>



Introduction to Apache Hive and Impala

Chapter 8



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- **Introduction to Apache Hive and Impala**
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Introduction to Apache Hive and Impala

In this chapter, you will learn

- **What Apache Hive is**
- **What Apache Impala (incubating) is**
- **How Hive and Impala differ from a relational database**
- **Ways in which organizations use Hive and Impala**

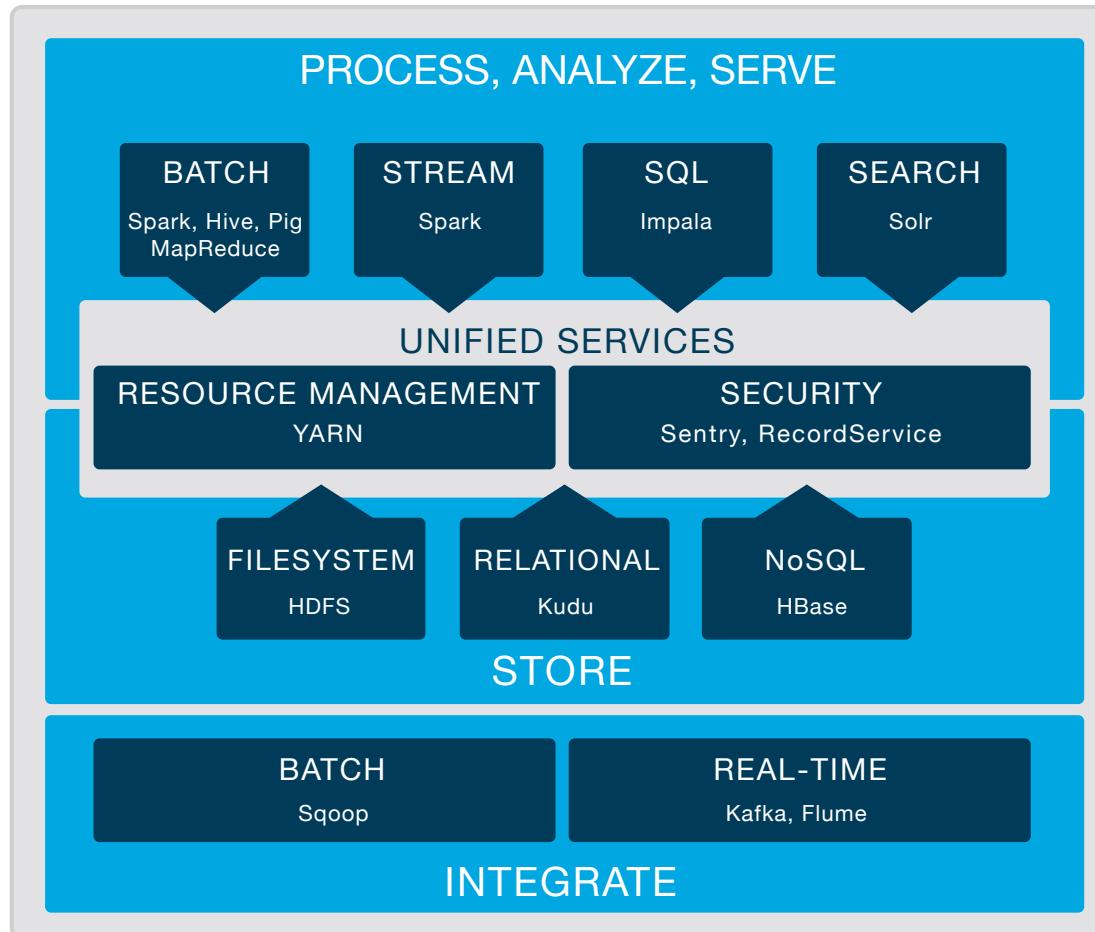
Chapter Topics

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

Review: Hadoop Data Processing and Analysis

- Hadoop includes many tools for data processing and analysis



What Is Apache Hive?

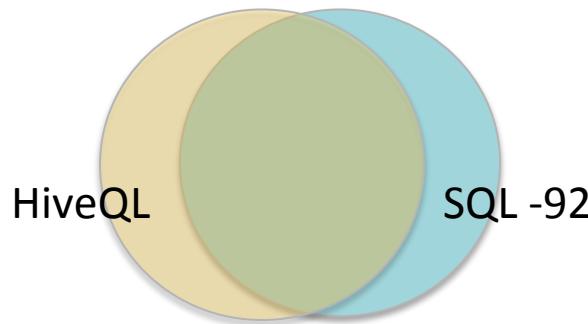
- **Hive is data warehouse infrastructure for Hadoop**
 - Alternative to writing low-level MapReduce code
 - Uses a SQL-like language called HiveQL
 - Generates jobs that run on the Hadoop cluster
 - Originally developed by Facebook
 - Now an open source Apache project



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 High-Level 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

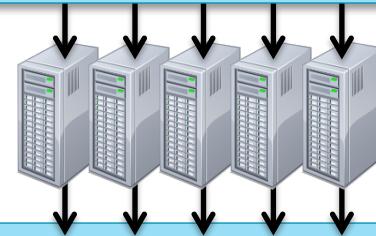
```
SELECT zipcode, SUM(cost) AS total  
  FROM customers  
  JOIN orders  
    ON (customers.cust_id = orders.cust_id)  
 WHERE zipcode LIKE '63%'  
 GROUP BY zipcode  
 ORDER BY total DESC;
```

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



Data Processing Engine

Hadoop Cluster



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 (incubating)?

- **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 Hadoop clusters**
 - Can query data stored in HDFS or HBase tables
 - Reads and writes data in common Hadoop file formats
- **Developed by Cloudera**
 - Donated to the Apache Software Foundation, where it is incubating
 - 100% open source, released under the Apache software license



Impala High-Level Overview

- Executes Impala SQL queries directly on the Hadoop 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

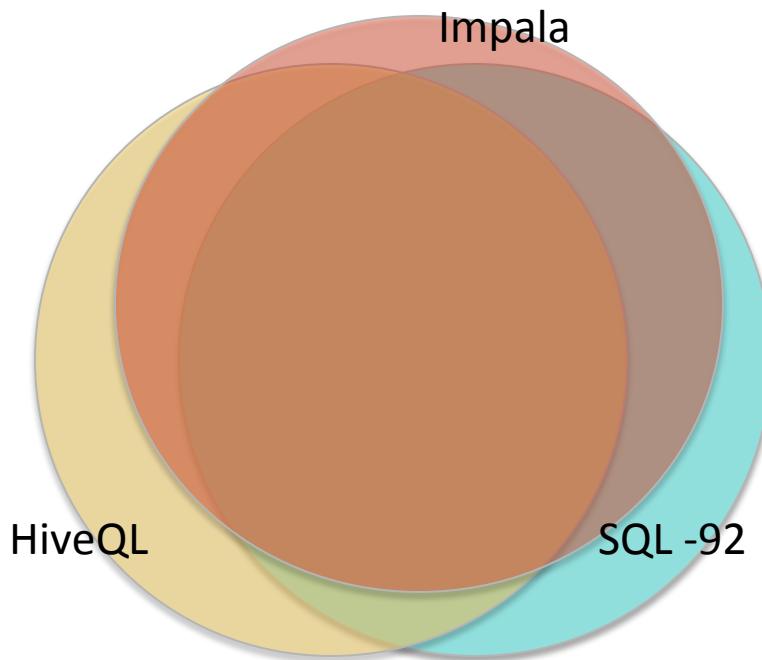


Hadoop Cluster

HDFS

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 and 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
- **Offers interoperability with other systems**
 - Extensible through Java and external scripts
 - 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

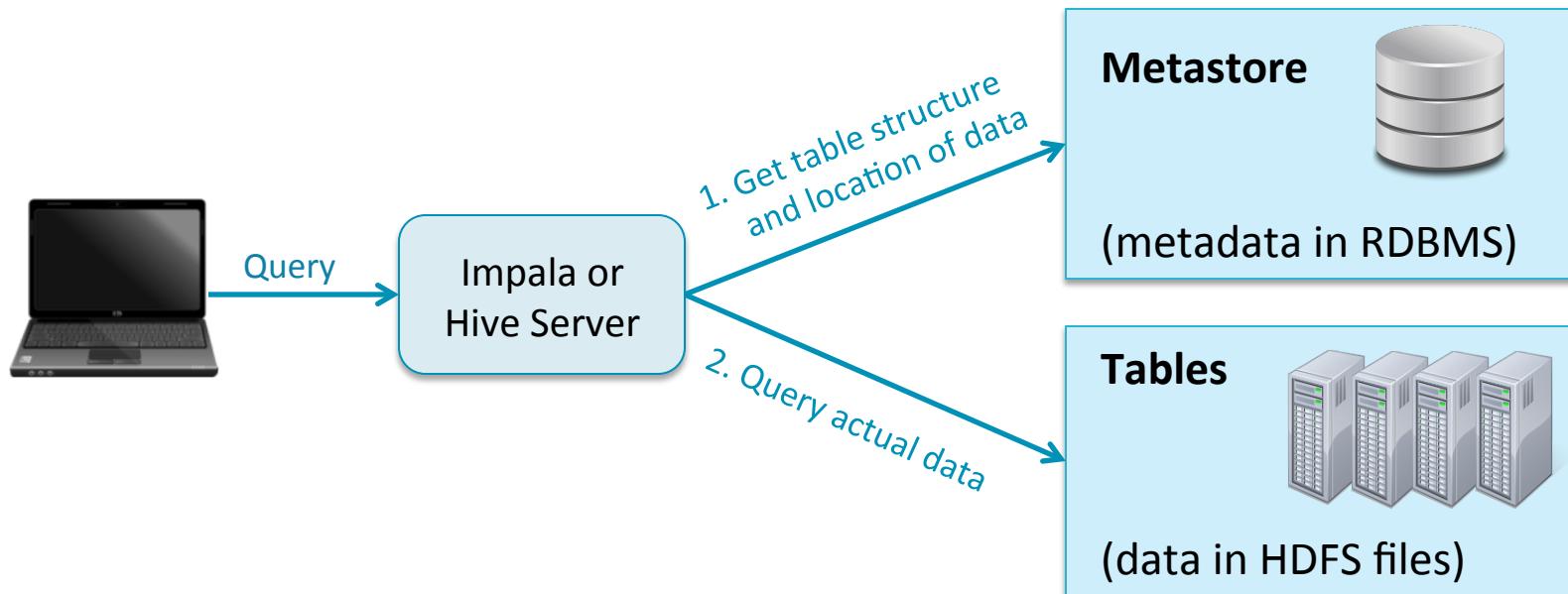
- What Is Hive?
- What Is Impala?
- Why Use Hive and Impala?
- **Schema and Data Storage**
- Comparing Hive and Impala to Traditional Databases
- Use Cases
- Essential Points

How Hive and Impala Load and Store Data (1)

- **Queries operate on tables, just like in an RDBMS**
 - A table is typically an HDFS directory containing one or more files
 - Default path: **/user/hive/warehouse/tablename**
 - Supports many formats for data storage and retrieval
- **You specify the structure and location of tables when creating them**
 - This metadata is stored in the *metastore*
 - Contained in an RDBMS such as MySQL
- **Hive and Impala work with the same data**
 - Tables in HDFS, metadata in the metastore

How Hive and Impala Load and Store Data (2)

- **Hive and Impala use the metastore to determine data format and location**
 - The query itself operates on data stored in 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*	No
Delete individual records	Yes	No*	No
Transactions	Yes	No*	No
Index support	Extensive	Limited	No
Latency	Very low	High	Low
Data size	Terabytes	Petabytes	Petabytes
Storage cost	Very high	Very low	Very low

* Hive now has limited, experimental support for **UPDATE**, **DELETE**, and transactions.
Cloudera neither recommends nor supports using these features in 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

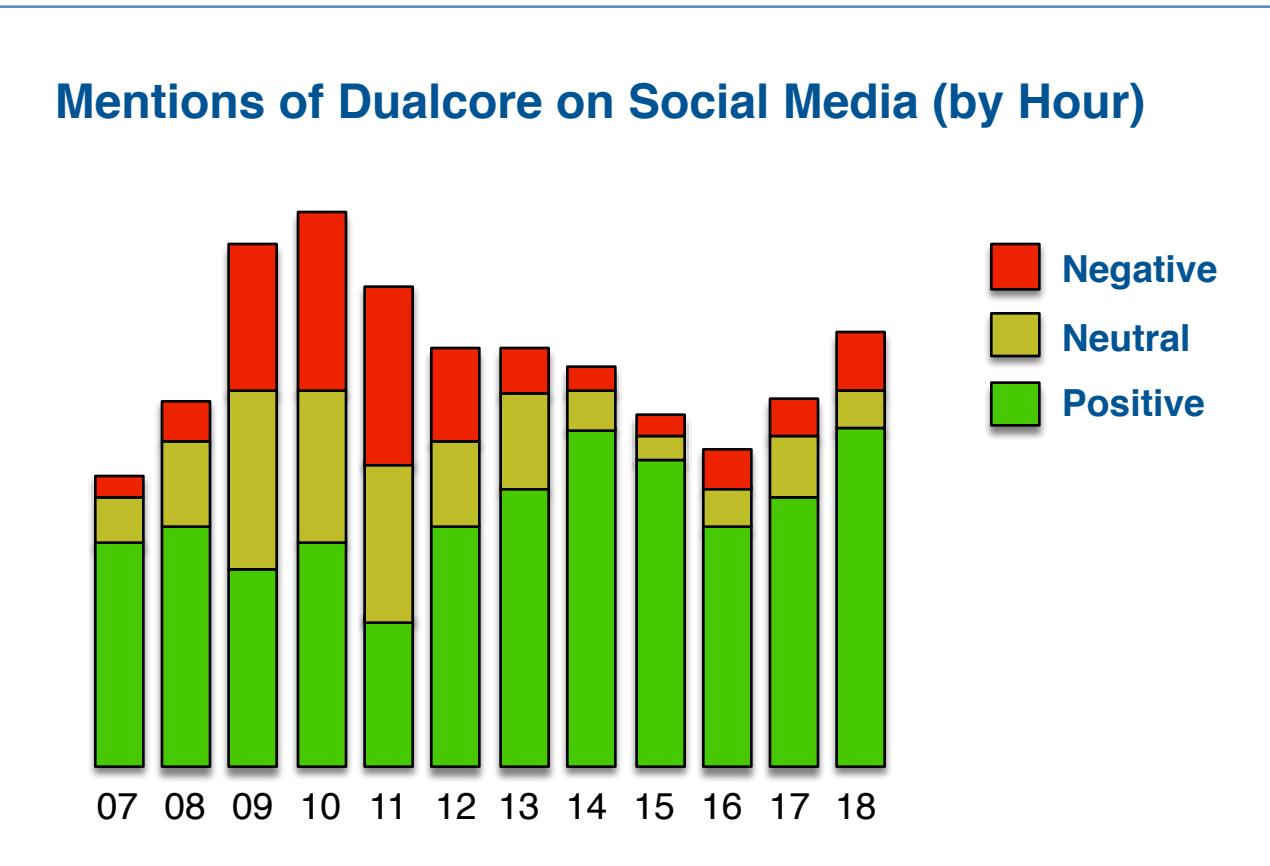
Use Case: Log File Analytics

- 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%

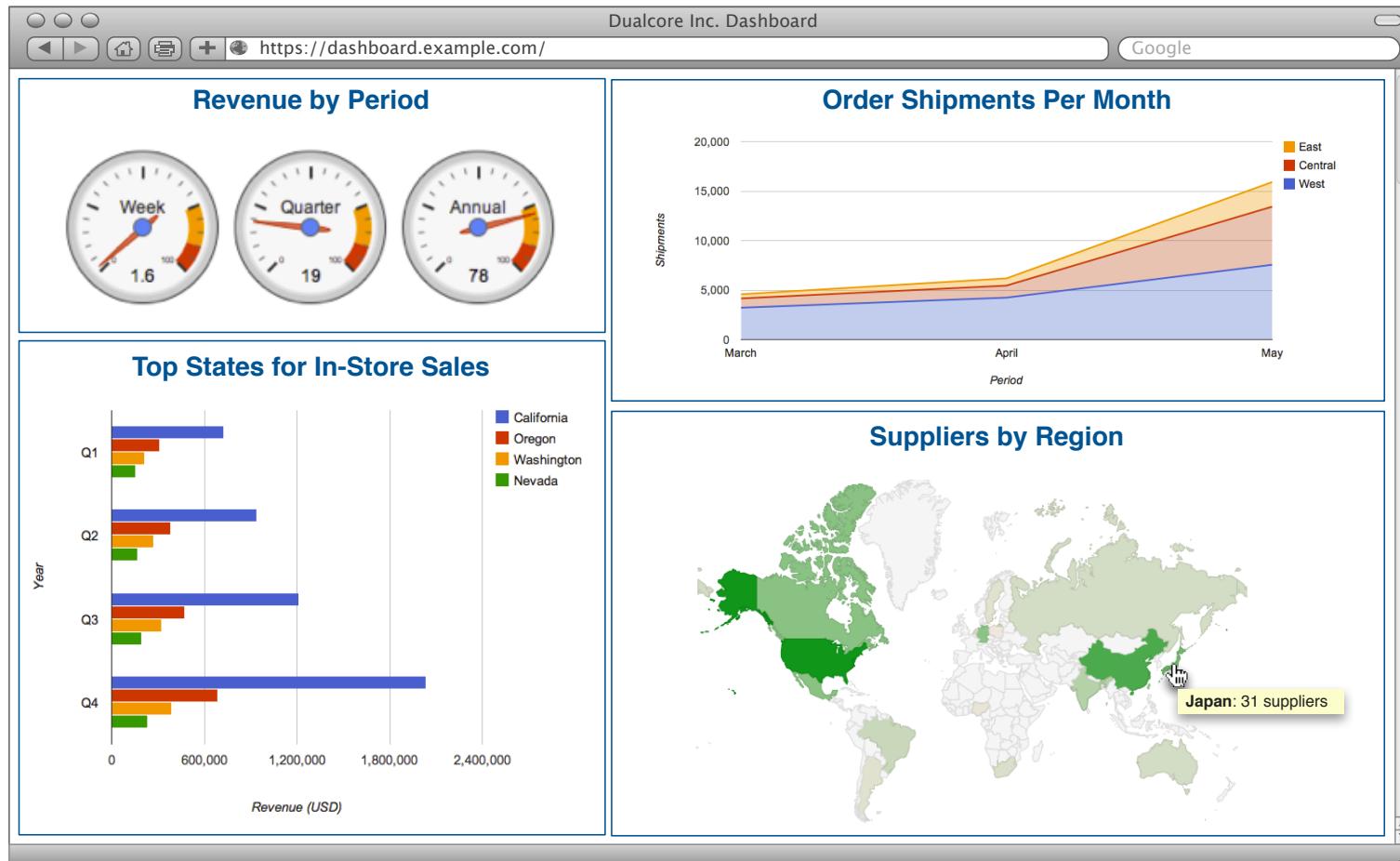
Use Case: Sentiment Analysis

- 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

- **Hive and Impala are tools for performing SQL queries on data in Hadoop**
- **HiveQL and Impala SQL are very similar to SQL-92**
 - Easy to learn for those with relational database experience
 - However, does *not* replace your RDBMS
- **Hive generates jobs that run on the Hadoop cluster data processing engine**
 - Runs MapReduce or Spark jobs based on HiveQL statements
- **Impala executes queries directly on the Hadoop cluster**
 - Uses a very fast specialized SQL engine, not MapReduce or Spark
- **Tables are typically directories of files in HDFS**
 - Information about those tables is kept in the metastore (in an RDBMS)

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 9

Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- **Querying with Apache Hive and Impala**
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Querying with Apache Hive and Impala

In this chapter, you will learn

- **How to explore databases and tables with HiveQL and Impala SQL**
- **How the query syntax for Hive and Impala compares to “standard” SQL**
- **Which data types HiveQL and Impala SQL support**

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 the Shell, Scripts, and Hue

Tables

- **Data for Hive and Impala tables is stored on the filesystem (typically 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
 - Unless the table is partitioned
- **The metastore gives context to this data**
 - Helps map raw data in HDFS 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;  
> USE sales;  
> SELECT * FROM customers;
```

Queries table in the
default database

Queries table in the
sales database

Exploring Databases and Tables (3)

- Use SHOW TABLES to list the tables in a database

Shows tables in the current database (**accounting**)

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

```
> 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 the Shell, 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 limitations compared to “standard” SQL**
 - Some features are not supported
 - Others are only partially implemented

Syntax Basics

- **Keywords are not case-sensitive**
 - Though they are often capitalized by convention
- **Statements are terminated by a semicolon**
 - A statement may span multiple lines
- **Comments begin with -- (*double hyphen*)**
 - Only supported in scripts
 - There are no multi-line comments (in Hive)
 - Impala supports /* */ multiline comment syntax

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

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;
```

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');
```

Common Operators in HiveQL and Impala SQL

- Commonly used operators are familiar to SQL users

Arithmetic	Comparison	Null	Logical
+	=	IS NULL	AND
-	!= or <>	IS NOT NULL	OR
*	<= >		NOT
/	<		
%	>		
	<=		
	>=		
	IN		
	BETWEEN		
	LIKE		

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' ;
```

Combining Query Results with a Union

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

```
SELECT cust_id, fname, lname
      FROM customers
     WHERE state='NY'
UNION ALL
SELECT cust_id, fname, lname
      FROM customers
     WHERE zipcode LIKE '073%'
```

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

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

- Recent versions of 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 now support *uncorrelated* subqueries
 - Both now support *correlated* subqueries, but Hive's support is limited
 - Impala supports *scalar* subqueries, but Hive does not

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 the Shell, 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 floating-point numbers**
 - Both positive and negative values allowed
 - Use **DECIMAL** when exact values are required!

Name	Description	Example Value
FLOAT	Decimals	3.14159
DOUBLE	More precise decimals	3.14159265358979323846
DECIMAL (p , s)	Exact precision	3.14 (p=3, s=2)

Character Types

- Character types are used to hold alphanumeric text values

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

Other Simple Types

- There are a few other data types

Name	Description	Example Value
BOOLEAN	True or false	<code>true</code>
TIMESTAMP	Instant in time	<code>2016-06-14 16:51:05</code>
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
Esma        129
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
29
17
NULL
NULL
127
NULL

age
29
17
NULL
127
-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 the Shell, 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

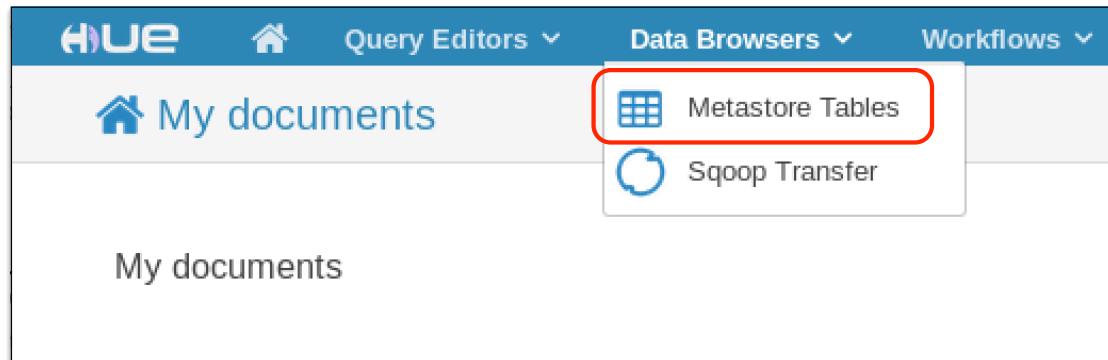
Using Hue with Hive and Impala

You can use Hue to...

Query data with
Hive or Impala



View and manage
the metastore



The Hue Query Editor

- The Hive and Impala query editors are nearly identical

The screenshot shows the Hue Query Editor interface for Impala. The top navigation bar includes links for HUE, Home, Query Editors, Data Browsers, Workflows, and Security. A search bar allows adding a name and description. The main area displays a query log and results table.

- Choose a database**: Points to the 'default' database selection in the sidebar.
- Explore schema and sample data**: Points to the 'Tables' section on the left, which lists 'customers' with columns: cust_id, fname, lname, address, city, state, zipcode, and sample data rows for order_details, orders, and products.
- Show logs**: Points to the 'Logs' tab in the top right.
- Enter, edit, save and execute queries**: Points to the query editor area where a simple SELECT statement is shown.
- Visualize or save query results**: Points to the results table below.
- View results and history**: Points to the 'Results' tab in the top right.

	cust_id	fname	lname	address	city	state	zipcode
1	1100688	David	Ball	4109 North 13th Street	Coloma	CA	95613
2	1100689	Wendy	Hairston	34264 East 5th Street	Angels Camp	CA	95222
3	1100691	Raymond	Saunders	20229 West 14th Street	Palo Alto	CA	94306
4	1100695	John	Stonge	2975 West 6th Street	Campionville	CA	95922
5	1100697	Wilbert	Ebert	1061 North 23rd Street	Williams	CA	95987
6	1100704	Thanh	Sampson	1463 North Madison Parkway	Tracy	CA	95376
7	1100705	Joe	Busch	1849 West 4th Street	San Fran		

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 the Shell, 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 HiveServer2
 - 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 1.1.0-cdh5.8.0)
Beeline version 1.1.0-cdh5.8.0 by Apache Hive
0: jdbc:hive2://localhost:10000>
```



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 the Shell, 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 2.6.0-cdh5.8.0
Welcome to the Impala shell.
[localhost.localdomain:21000] >
```

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

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

Note: Some log messages have been truncated to better fit the slide



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 May 31 17:30:27 CDT 2016
```

- No direct support for HDFS commands

- But you can run **hdfs dfs** commands using **shell**

```
> shell hdfs dfs -mkdir /reports/sales/2016;
```



Running Impala Queries from the Command Line

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

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

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

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

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

```
$ impala-shell -f myquery.hql \
--delimited \
--output_delimiter='\t' \
-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 the Shell, Scripts, and Hue

Essential Points

- **HiveQL and Impala SQL syntax are familiar to those who know SQL**
 - A subset of SQL-92, plus extensions
- **Every table belongs to exactly one database**
 - The **SHOW DATABASES** command lists databases
 - The **USE** command switches the active database
 - The **SHOW TABLES** command lists all tables in a database
- **Every column in a table has an associated data type**
 - Most simple column types are similar to SQL data types
- **Hive and Impala both have query editors in Hue and command line shells**

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 the Shell, Scripts, and Hue**

Hands-on Exercise: Running Queries from the Shell, Scripts, and Hue

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



Apache Hive and Impala Data Management

Chapter 10



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- **Apache Hive and Impala Data Management**
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Apache Hive and Impala Data Management

In this chapter you will learn

- How Hive and Impala encode and store data
- How to create databases, tables, and views
- How to load data into tables
- How to alter and remove databases, tables, and views
- How to save query results into tables and files

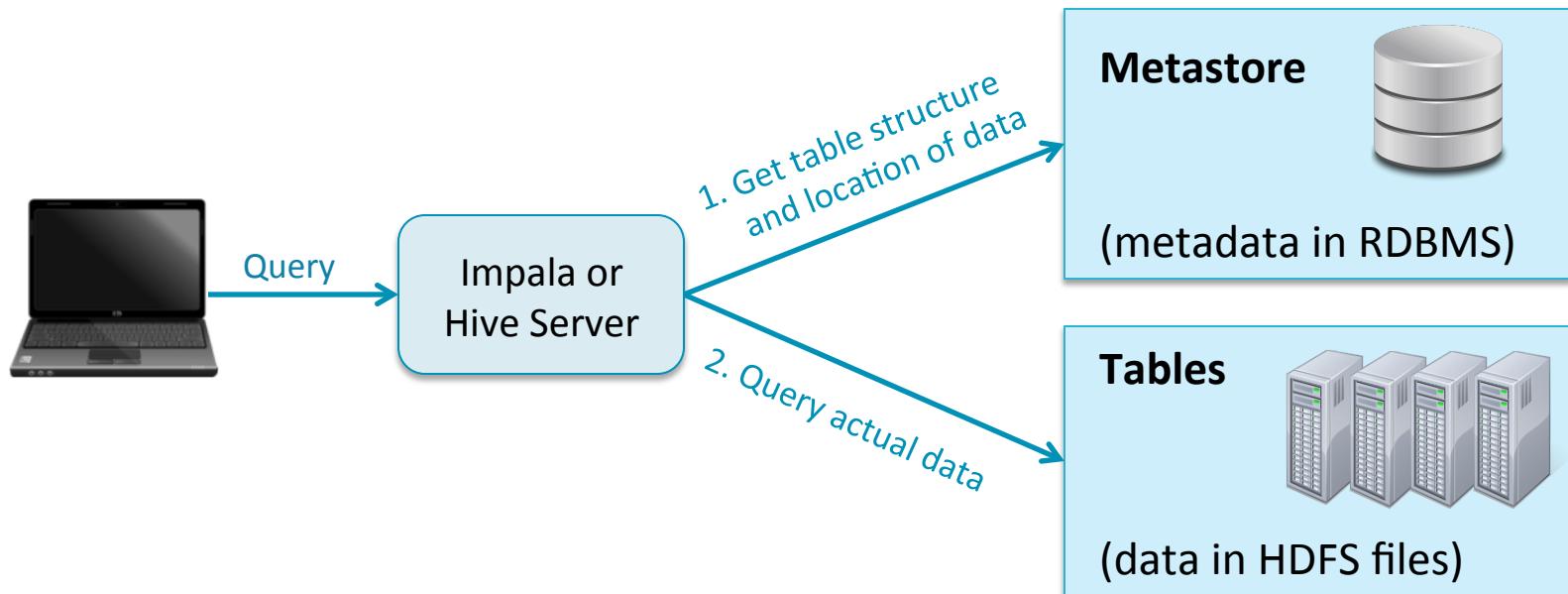
Chapter Topics

Apache Hive and Impala 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 Load and Store 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)

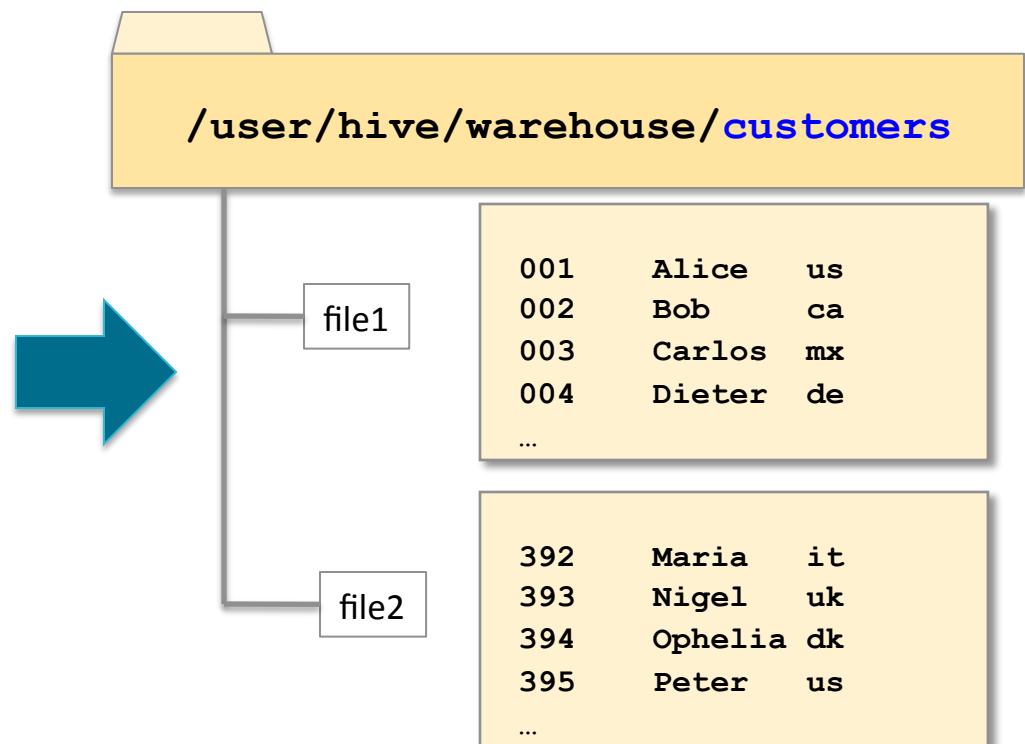


The Warehouse Directory

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

customers table

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



Impala Metadata Cache Invalidation

- **Impala caches metadata to reduce query latency**
 - Table structure and data location from the metastore
 - File information from HDFS
- **Changes made to tables outside Impala require cache invalidation**
 - Creating tables with Hive
 - Importing table data with Sqoop
 - Adding table data in HDFS



```
INVALIDATE METADATA;
```

Using the Hue Metastore Manager

■ The Hue Metastore Manager

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

The image shows two screenshots of the Hue Metastore Manager. The left screenshot shows a sidebar with 'Data Browsers' and 'Workflows' tabs. The 'Metastore Tables' item is highlighted with a red box and has a red arrow pointing to the right screenshot. The right screenshot shows the 'Metastore Manager' interface with a sidebar showing 'Tables' with entries: 'customers', 'order_details', 'orders', and 'products'. The main area shows 'Databases > default'. Under 'STATS', it says 'Default Hive database' with 'public (ROLE)' and 'Location'. Under 'TABLES', there is a search bar 'Search for a table...' and a table with columns: 'Table Name', 'Comment', and 'Type'. The table contains four rows: 'customers', 'order_details', 'orders', and 'products', each with a small icon and a grid icon.

Accessing the Metastore with HCatalog

- **Hive and Impala have built-in capability to directly access the metastore**
- **Other tools like Pig 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

Apache Hive and Impala 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 simply namespaces**
 - Helps to organize your 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 conditionally create a new database**

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

Creating a Table (2)

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

```
ROW FORMAT DELIMITED
```

```
FILEFORMAT
```

```
STORED AS
```

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  
  FIELDS TERMINATED BY char  
STORED AS format
```

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  
STORED AS {TEXTFILE|SEQUENCEFILE|...};
```

...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|...};
```

Finally, you may declare the file format. **STORED AS TEXTFILE** is the default and does not need to be specified.

Example Table Definition

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

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

- Example of corresponding record for the table above

```
1,Data Analyst,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 and names are derived from the existing table
 - New table will contain no data

Controlling Table Data Location

- By default, table data is stored in the warehouse directory
- 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 '/dualcore/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 '/dualcore/ad_data' ;
```

Creating Tables Using the Metastore Manager

- The Metastore Manager provides a table creation wizard
 - Supports most, but not all, table options

The screenshot shows the Metastore Manager interface. At the top, there are navigation links: Home, Query Editors, Data Browsers, Workflows, and Security. Below the header, the title "Metastore Manager" is displayed next to a database icon. On the left, a sidebar lists the "default" database with its tables: "customers", "order_details", "orders", and "products". The main area shows the "Databases > default" structure. Below this, a "STATS" section displays information: "Default Hive database", "public (ROLE)", and "Location". On the right side of the main area, there are several icons: a refresh symbol, a file icon with a plus sign, and a large red-outlined plus sign. The red-outlined plus sign is specifically highlighted, indicating it is the button used for creating a new table.

Chapter Topics

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

Loading Data from HDFS Files

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

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

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

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

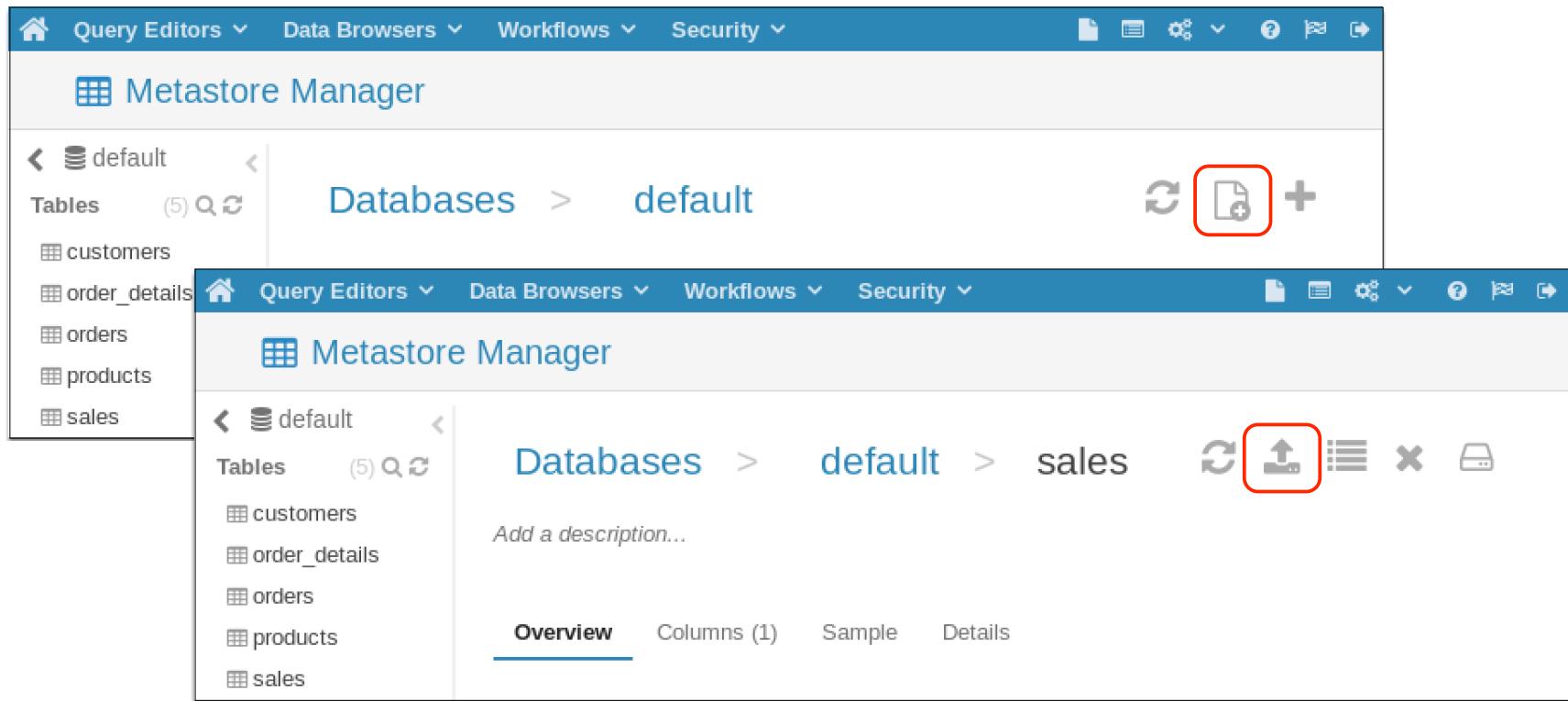
Overwriting Data from Files

- 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 Metastore Manager

- The Metastore Manager provides two wizards for loading data into tables
 - Create a new table from a file
 - Import data into an existing table



Loading Data from a Relational Database

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

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

Chapter Topics

Apache Hive and Impala 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 is similar 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 customers;
```

```
DROP TABLE IF EXISTS 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;
```

Old Name New Name Type

- 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 jobs CHANGE salary salary BIGINT;
```

Old Name New Name Type



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

Chapter Topics

Apache Hive and Impala 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

- Views are conceptually like a table, but backed by a query
 - You cannot directly add data to a view

```
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;
```

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

Apache Hive and Impala 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 to HDFS in Hive

- Hive also lets you save output to a directory in HDFS
 - *Caution: Hive does not delete existing contents of the directory!*

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



Saving to Multiple Directories in Hive (1)

- We just saw that you can save output to an HDFS file

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

- This query 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)

- We sometimes need to extract data to multiple files
 - Hive **SELECT** queries can take a long time to complete
- Hive allows us to do 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

Apache Hive and Impala 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

- **Impala caches metadata**
 - **INVALIDATE METADATA** after making changes outside Impala
- **Each table maps to a directory in HDFS**
 - Table data is stored as one or more files
 - Default format: plain text with delimited fields
- **The Hue Metastore Manager can create and load simple tables**
- **Dropping managed (internal) tables deletes data in HDFS**
 - Unmanaged (external) tables require manual data deletion
- **ALTER TABLE is used to add, modify, and remove columns**
- **Views can help to simplify complex and repetitive queries**

Chapter Topics

Apache Hive and Impala 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 Hands-On Exercise, you will create and load tables using the Hue Metastore Manager, Impala SQL, and Sqoop
 - Please refer to the Hands-On Exercise Manual for instructions



Data Storage and Performance

Chapter 11



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- **Data Storage and Performance**
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Data Storage and Performance

In this chapter you will learn

- How partitioning improves query performance
- How to create and load data into partitioned tables
- When to use partitioning
- Which file formats Hive and Impala support
- How to choose the best file format for your use case
- How to use the Avro and Parquet file formats

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**

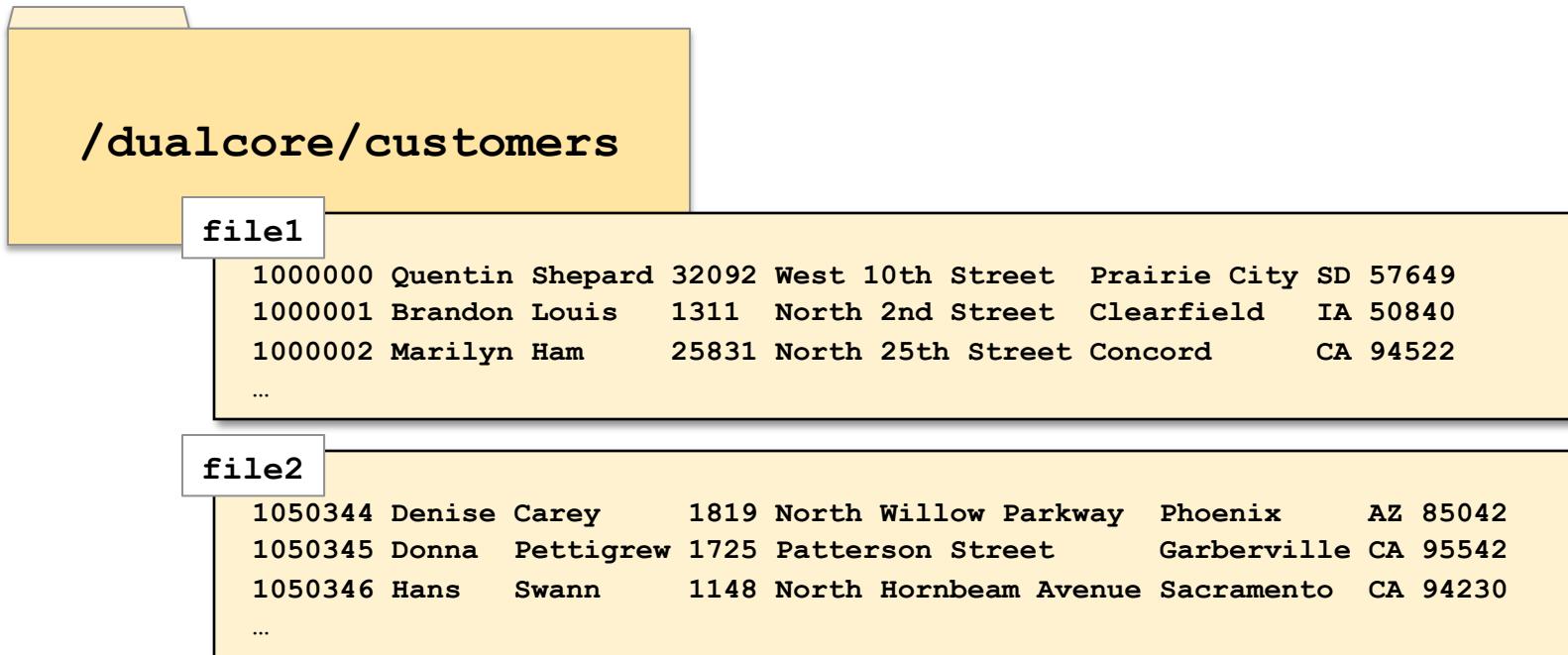
Example: Partitioning Customers by State (1)

- 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



Example: Partitioning Customers by State (3)

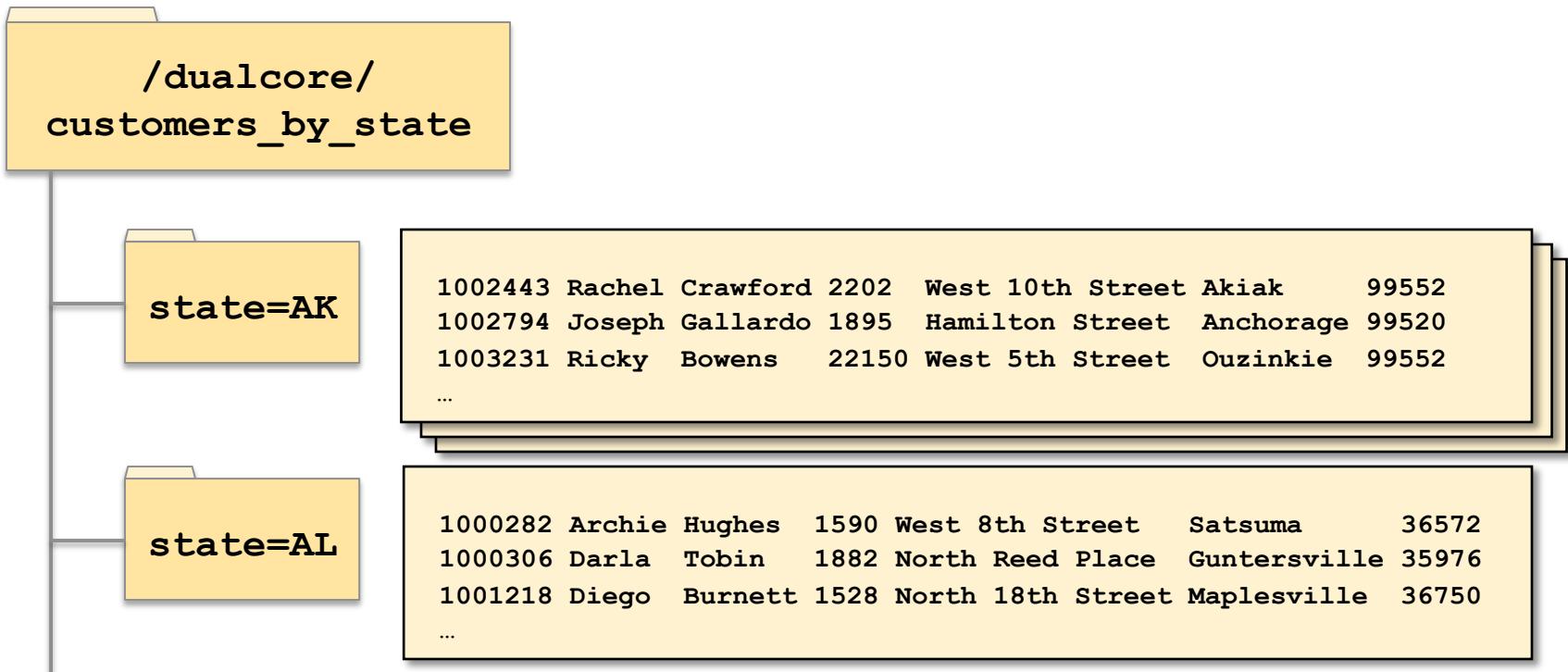
- What if most of Dualcore's analysis on the customer table was done by state? For example:

```
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 files by state
 - Queries that filter by state scan only the relevant files

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

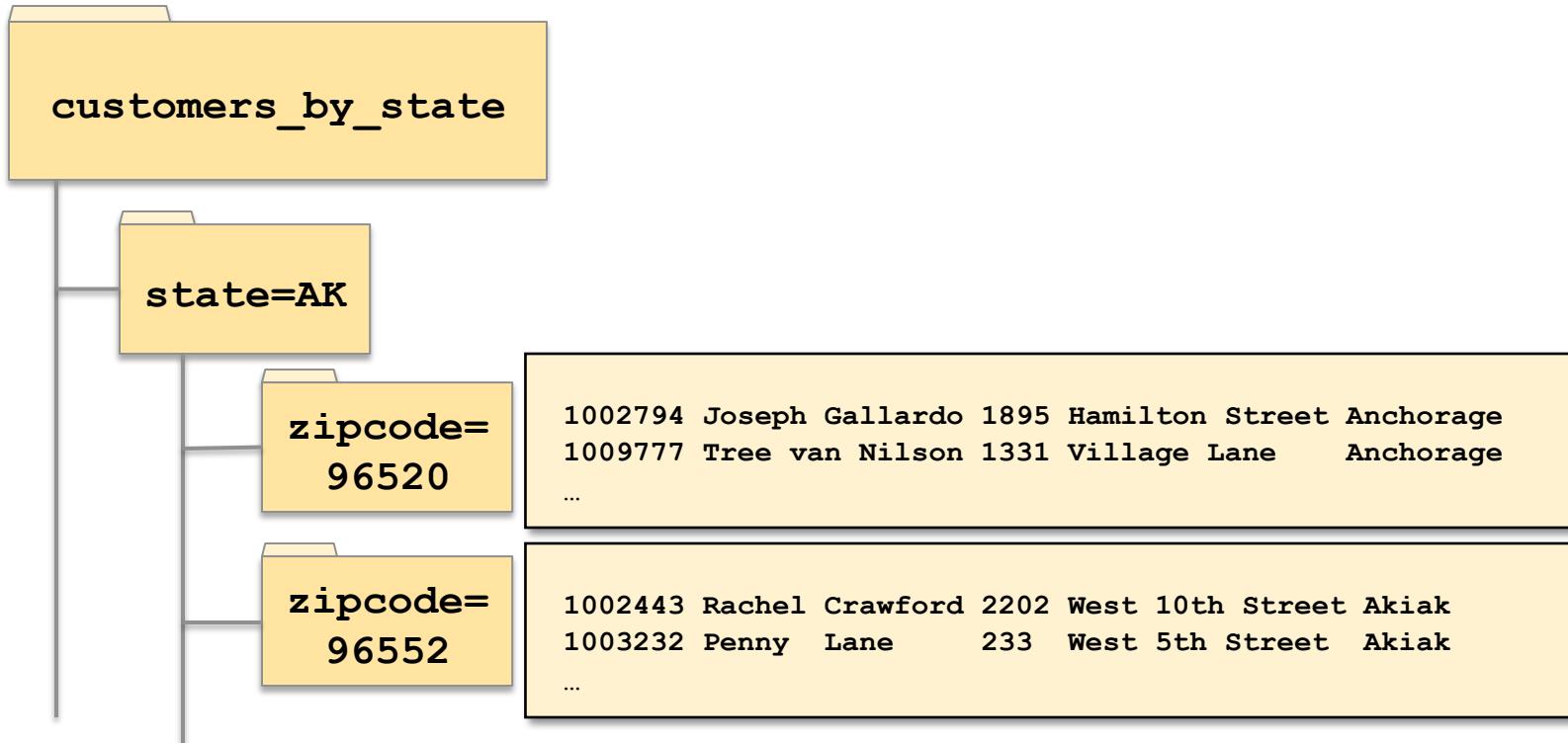
```
DESCRIBE customers_by_state;
+-----+-----+-----+
| name      | type      | comment |
+-----+-----+-----+
| cust_id   | int       |          |
| fname     | string    |          |
| lname     | string    |          |
| address   | string    |          |
| city      | string    |          |
| zipcode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

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)
```



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

- **Dynamic partitioning**
 - Hive/Impala automatically creates partitions
 - Inserted data is stored in the correct partitions based on column values
- **Static partitioning**
 - You manually create new partitions using **ADD PARTITION**
 - When loading data, you specify which partition to store it in

Dynamic Partitioning

- With dynamic partitioning, you use an **INSERT** statement to load data
 - The partition column(s) must be included in the **PARTITION** clause
 - The partition column(s) must be specified *last* in the **SELECT** list

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

- Hive or Impala automatically creates partitions and inserts data into them based on the values of the specified partition column
 - The values of the partition column(s) are not included in the files

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 Only: Shortcut for Loading Data into 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

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



Partitioning in Hive (1)

- By default, Hive requires at least one partition to be static
 - 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

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 by column, rather than by row
 - Very efficient when selecting only a subset of a table's columns

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
6	David	Sunnyvale	Mechanic

*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
6	David	Sunnyvale	Mechanic

*Organization of data in
columnar formats*

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
- An Avro table must also 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)

- Like Avro, Parquet embeds a schema definition in the file itself
- 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

- **Partitioned tables store data in subdirectories**
- **Partitioning improves performance of queries that filter on partition columns**
- **Dynamic partitioning and static partitioning are different techniques for loading data into partitioned tables**
- **A variety of file formats are supported by Hive and Impala**
- **Consider the data and the use case when choosing a file format**
 - Text files: basic, convenient, human-readable
 - Avro: best choice for general-purpose storage
 - Parquet: best choice for columnar storage

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 Hands-On 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



Relational Data Analysis with Apache Hive and Impala

Chapter 12



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- **Relational Data Analysis with Apache Hive and Impala**
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Relational Data Analysis with Apache Hive and Impala

In this chapter, you will learn

- **How to join data from different datasets**
- **How to use many of Hive's and Impala's built-in functions**
- **How to group rows to aggregate data**
- **How to use window functions to group data**

Chapter Topics

Relational Data Analysis with Apache Hive and Impala

- **Joining Datasets**
- Common Built-In Functions
- Aggregation and Windowing
- Essential Points
- Hands-On Exercise: Relational Data Analysis

Joins

- Joining disparate datasets is a common operation
- Hive and Impala support several types of joins
 - Inner joins
 - Outer joins (left, right, and full)
 - Cross joins
 - Left semi-joins
- Hive only supports equality conditions in joins
 - 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)
- *Implicit joins* do the same thing
 - Supported in Impala, and in Hive 0.13/CDH 5.2 and later

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

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

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

```
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;
```

orders table

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

Result of query

cust_id	name	total
d	Dieter	NULL
NULL	NULL	2137

NULL Values in Join Key Columns

customers_with_null table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de
NULL	Unknown	NULL

orders_with_null table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
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

customers_with_null table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de
NULL	Unknown	NULL

orders_with_null table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
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



Non-Equijoin Example (Impala only)

employees table

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

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

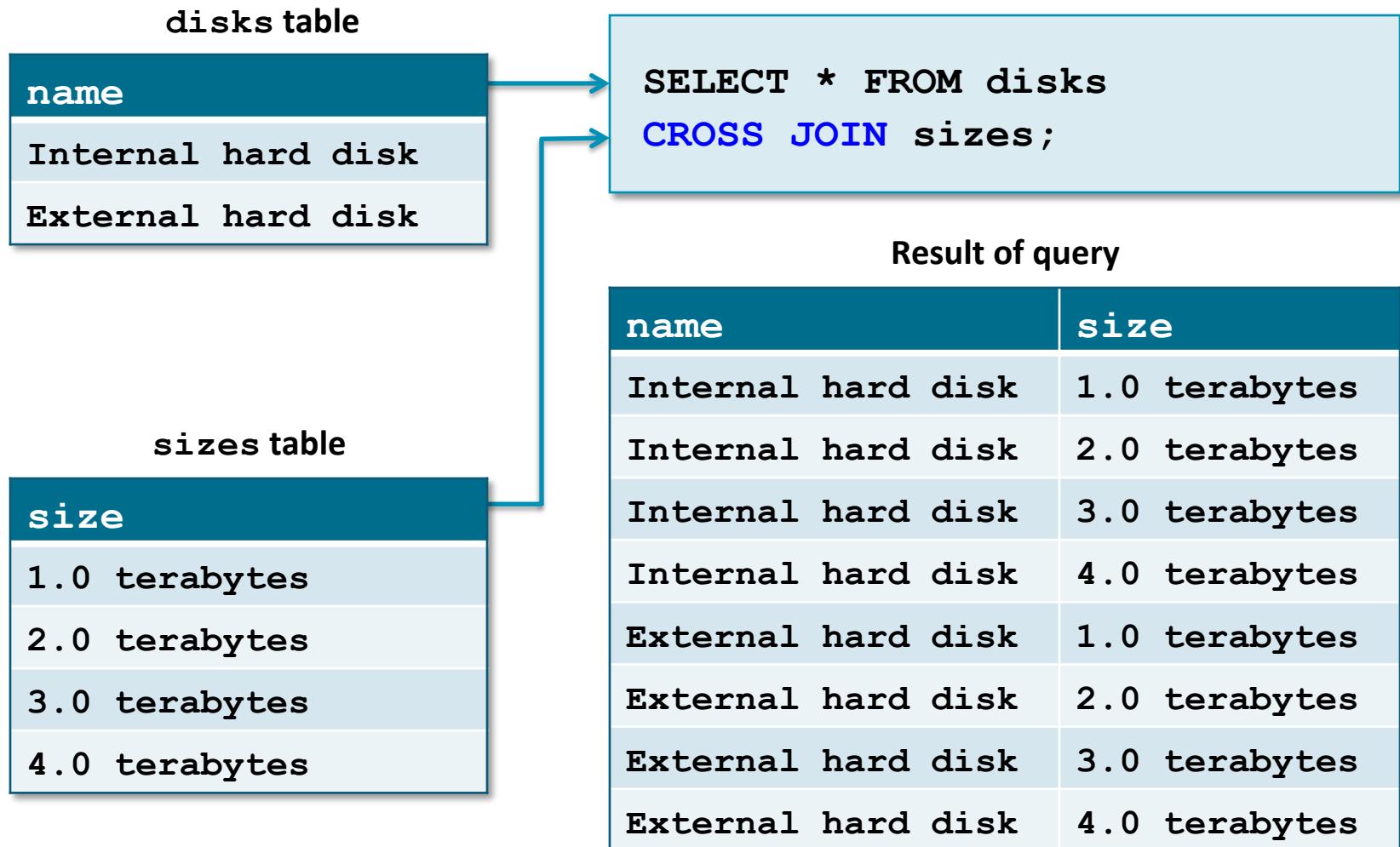
salary_grades table

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

Result of query

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

Cross Join Example



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
 - For which there is 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

Relational Data Analysis with Apache Hive and Impala

- Joining Datasets
- **Common Built-In Functions**
- Aggregation and Windowing
- Essential Points
- Hands-On Exercise: Relational Data Analysis

Built-In Functions

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

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



Information about Built-In Functions in Hive

- To see a list of all functions and operators (Hive only)

```
> SHOW FUNCTIONS;
```

```
...
```

```
abs
```

```
acos
```

```
add_months
```

```
...
```

- 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
```

Built-In Mathematical Functions

- These functions perform numeric calculations

Function Description	Example Invocation	Input	Output
Rounds to specified # of decimals	<code>round(total_price, 2)</code>	23.492	23.49
Returns nearest integer above	<code>ceil(total_price)</code>	23.492	24
Returns nearest integer below	<code>floor(total_price)</code>	23.492	23
Return absolute value	<code>abs(temperature)</code>	-49	49
Returns square root	<code>sqrt(area)</code>	64	8
Returns 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>		2016-06-14 16:51:05.0
Convert to UNIX format	<code>unix_timestamp(order_dt)</code>	2016-06-14 16:51:05	1465923065
Convert to string format	<code>from_unixtime(mod_time)</code>	1465923065	2016-06-14 16:51:05
Extract date portion	<code>to_date(order_dt)</code>	2016-06-14 16:51:05	2016-06-14
Extract year portion	<code>year(order_dt)</code>	2016-06-14 16:51:05	2016
Return # of days between dates	<code>datediff(ship_dt, order_dt)</code>	2016-06-17, 2016-06-14	3

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
- But you can convert to and from different formats
 - `unix_timestamp` and `from_unixtime` accept a format parameter
 - For example, convert a date string from `MM/dd/yyyy` format

```
to_date(from_unixtime(  
    unix_timestamp(order_dt, 'MM/dd/yyyy')))
```

- Represent the current date and time as a formatted string

```
from_unixtime(unix_timestamp(current_timestamp()),  
    'MMM d, yyyy HH:mm')
```

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, 3, 4)</code>	Samuel	muel
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
Selectively return value	<code>if(price > 1000, 'A', 'B')</code>	1500	A
Convert to another type	<code>cast(weight AS INT)</code>	3.581	3

Chapter Topics

Relational Data Analysis with Apache Hive and Impala

- Joining Datasets
- Common Built-In Functions
- **Aggregation and Windowing**
- Essential Points
- Hands-On Exercise: Relational Data Analysis

Aggregation and Windowing

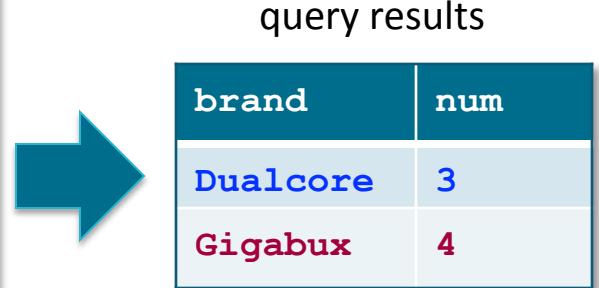
- The functions covered earlier work on data from a single row
- Some functions work by combining values from different rows
- Aggregation
 - Groups rows based on a column expression
 - GROUP BY *column*
 - Rows in the group are combined
 - Individual row values are not available
- Windowing
 - Calculates values within a “window” without combining rows

Example: Record Grouping and Aggregate Functions

- **GROUP BY** groups selected data by one or more columns
 - Caution: Columns not part of aggregation must be listed in **GROUP BY**

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



Built-in Aggregate Functions

- Hive and Impala offer many aggregate functions, including these

Function Description	Example Invocation
Counts all rows	<code>COUNT (*)</code>
Counts all rows where field is not NULL	<code>COUNT (fname)</code>
Counts all rows where field is unique and not NULL	<code>COUNT (DISTINCT fname)</code>
Returns the largest value	<code>MAX (price)</code>
Returns the smallest value	<code>MIN (price)</code>
Adds all supplied values and returns result	<code>SUM (price)</code>
Returns the average of all supplied values	<code>AVG (price)</code>

- You can also define custom User Defined Aggregate Functions (UDAFs)

Window Aggregation

- **Standard aggregation groups sets of rows together into single rows**
 - Individual rows are not preserved in the output
- **Windowing applies a function over sets of rows without combining them**
 - Individual rows are preserved
- **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)

Windows

- A **window** defines a set of rows in a table
- **OVER (*window-specification*)** specifies the windows over which to apply an aggregation or windowing function
- 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 the concept of windows in a database query. A vertical bracket on the right side of the table groups the first three rows (prod_id 1-3) under the label "Window", and the last four rows (prod_id 4-7) under the label "Window". This visualizes how the `OVER (PARTITION BY brand)` clause creates separate windows for each brand.

Example: Aggregation over a Window (1)

- Question: What is the price of the least expensive product in each brand?

```
SELECT prod_id, brand, price,  
       MIN(price) OVER(PARTITION BY brand) AS m  
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	m
1	Dualcore	18.39	1.99
2	Dualcore	11.99	1.99
3	Dualcore	1.99	1.99
4	Gigabux	40.50	20.00
5	Gigabux	50.50	20.00
6	Gigabux	20.00	20.00
7	Gigabux	20.00	20.00

Example: Aggregation over a Window (2)

- Question: For each product, how does the price compare to the minimum price for that brand?

```
SELECT prod_id, brand, price,  
       price - MIN(price) OVER(PARTITION BY brand) AS d  
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	d
1	Dualcore	18.39	16.40
2	Dualcore	11.99	10.00
3	Dualcore	1.99	0.00
4	Gigabux	40.50	20.50
5	Gigabux	50.50	30.50
6	Gigabux	20.00	0.00
7	Gigabux	20.00	0.00

Example: RANK and ROW_NUMBER

- Question: Rank the products 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
6	Gigabux	20.00	1	1
7	Gigabux	20.00	1	2
4	Gigabux	40.50	3	3
5	Gigabux	50.50	4	4

Example: Using Windowing for Subqueries

- Question: What is the least expensive product from each brand?

```
SELECT prod_id, brand, price FROM(
  SELECT prod_id, brand, price,
    RANK() OVER(PARTITION BY brand ORDER BY price) AS rank
  FROM products) p
WHERE rank=1;
```

subquery results

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



query results

prod_id	brand	price
3	Dualcore	1.99
6	Gigabux	20.00
7	Gigabux	20.00

Other Windowing Functions

Function	Description
DENSE_RANK	Rank of current value within the window (with consecutive rankings)
NTILE (n)	The n -tile (of n) within the window that the current value is in
PERCENT_RANK	Rank of current value within the window, expressed as a percentage
CUME_DIST	Cumulative distribution of current value within the window
LEAD (column, n, default)	The value in the specified column in the n th following row
LAG (column, n, default)	The value in the specified column in the n th preceding row

Example: **RANK** and **DENSE_RANK**

- Question: Rank the products by price within a 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
6	Gigabux	20.00	1	1
7	Gigabux	20.00	1	1
4	Gigabux	40.50	3	2
5	Gigabux	50.50	4	3

Time-Based Windows

- Analyzing data over time is a common use of windowing
- Example: Analyzing ad display data

ads table

campaign_id	display_date	display_site	cost
A1	2016-05-01	audiophile	76
A1	2016-05-01	photosite	64
A3	2016-05-01	photosite	68
A2	2016-05-02	audiophile	82
A3	2016-05-02	dvdreview	66
A3	2016-05-02	audiophile	82
A1	2016-05-02	audiophile	70
A3	2016-05-03	audiophile	66

Example: Time-Based Windows

- Question: How many ads were displayed per site per day?
- A non-windowed aggregation:

```
SELECT display_date, display_site, COUNT(display_date) AS n  
FROM ads GROUP BY display_date, display_site;
```

query results

display_date	display_site	n
2016-05-01	audiophile	1
2016-05-01	photosite	2
2016-05-02	audiophile	3
2016-05-02	dvdreview	1
2016-05-03	audiophile	1

Example: Rank over Time-Based Windows

- Question: How did each site rank by day in number of displays?

```
SELECT display_date, display_site, n,
       RANK() OVER (PARTITION BY display_date ORDER BY n DESC)
               AS dayrank
  FROM (
    SELECT display_date, display_site, COUNT(display_date) AS n
      FROM ads GROUP BY display_date, display_site) ads
 ORDER BY display_date, display_site;
```

display_date	display_site	n	dayrank
2016-05-01	audiophile	1	2
2016-05-01	photosite	2	1
2016-05-02	audiophile	3	1
2016-05-02	dvdreview	1	2
2016-05-03	audiophile	1	1

Example: Using LAG with Time-Based Windows

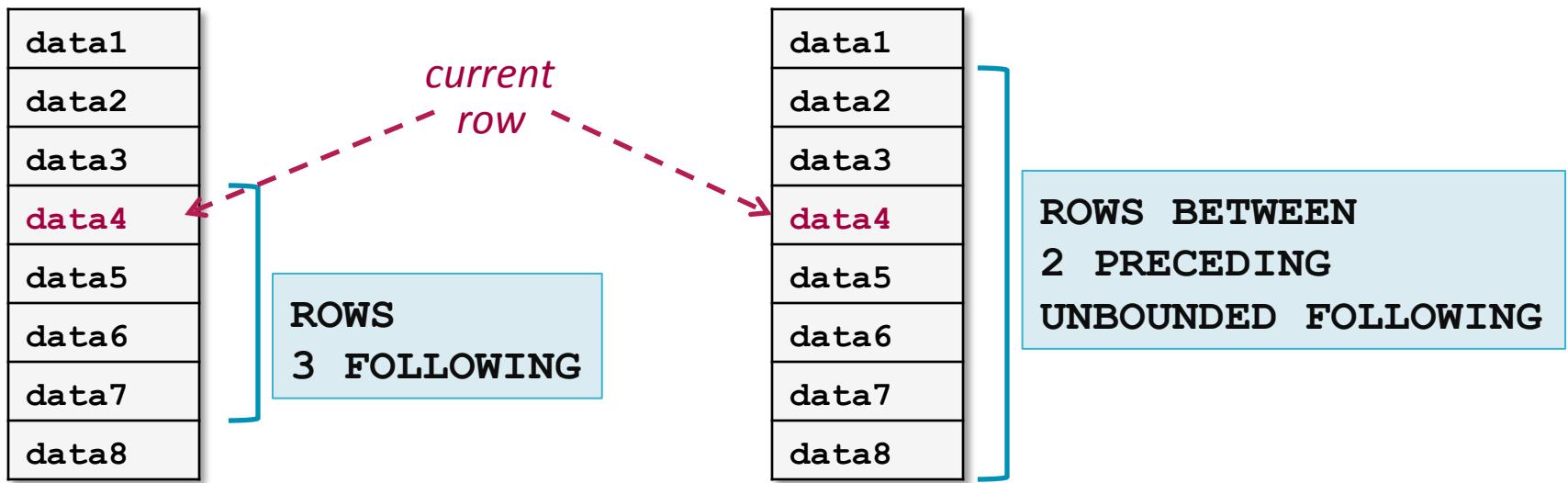
- Question: How did each site's daily count compare to the previous day?

```
SELECT display_date, display_site, n,
       LAG(n) OVER
          (PARTITION BY display_site ORDER BY display_date) AS nprev
FROM (
    SELECT display_date, display_site, COUNT(display_date) AS n
    FROM ads GROUP BY display_date, display_site) ads
ORDER BY display_date, display_site;
```

display_date	display_site	n	nprev
2016-05-01	audiophile	1	NULL
2016-05-01	photosite	2	NULL
2016-05-02	audiophile	3	1
2016-05-02	dvdreview	1	NULL
2016-05-03	audiophile	1	3

Sliding Windows

- There are three optional parts of a window specification
 - Partitioning – **PARTITION BY**
 - Ordering – **ORDER BY (ASC or DESC)**
 - Frame boundaries – **ROWS** or **RANGE**
- Frame boundaries specify a *sliding window* relative to the current row



Example: Time-Based Sliding Window

- Question: What is the per-site average count for the week ending today?

```
SELECT display_date, display_site, n,
       AVG(n) OVER
         (PARTITION BY display_site ORDER BY display_date
          ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS wavg
    FROM (
      SELECT display_date, display_site, COUNT(display_date) AS n
        FROM ads GROUP BY display_date, display_site) ads
   ORDER BY display_date, display_site;
```

display_date	display_site	n	wavg
2016-05-01	audiophile	1	1
2016-05-01	photosite	2	2
2016-05-02	audiophile	3	2
2016-05-02	dvdreview	1	1
2016-05-03	audiophile	1	1.66

Chapter Topics

Relational Data Analysis with Apache Hive and Impala

- Joining Datasets
- Common Built-In Functions
- Aggregation and Windowing
- **Essential Points**
- Hands-On Exercise: Relational Data Analysis

Essential Points

- **Hive and Impala offer many ways to join data from disparate datasets**
- **Many standard SQL functions are available**
 - Perform operations on strings, dates, URLs, and numeric values
- **Aggregation functions group values in multiple rows into single values**
- **Windowing functions calculate values based on sets of rows without combining the rows**

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>

- Hive Windowing and Analytic Functions

- <http://tiny.cloudera.com/hivewindow>

- Impala Analytic Functions

- <http://tiny.cloudera.com/impalawindow>

Chapter Topics

Relational Data Analysis with Apache Hive and Impala

- Joining Datasets
- Common Built-In Functions
- Aggregation and Windowing
- Essential Points
- **Hands-On Exercise: Relational Data Analysis**

Hands-On Exercise: Relational Data Analysis

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



Complex Data with Apache Hive and Impala

Chapter 13



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- **Complex Data with Apache Hive and Impala**
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Complex Data with Apache Hive and Impala

In this chapter, you will learn

- **How Hive and Impala store and query complex data**
- **What are some reasons for using complex types**
- **What are the differences between Hive and Impala when working with complex data**

Chapter Topics

Complex Data with Apache Hive and Impala

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

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 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

- How would you store multiple phone numbers for customers?
 - The traditional way uses two tables joined on a common key

customers table

cust_id	name
a	Alice
b	Bob
c	Carlos

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

phones table

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

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

<code>cust_id</code>	<code>name</code>	<code>phones</code>
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

<code>name</code>	<code>phones [0]</code>	<code>phones [1]</code>
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
```



Using MAP Columns with Hive (1)

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

`customers_phones` table

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

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

query results

<code>name</code>	<code>home</code>
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
```



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

<code>cust_id</code>	<code>name</code>	<code>address</code>
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

<code>name</code>	<code>state</code>	<code>zipcode</code>
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
```



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

<code>cust_id</code>	<code>name</code>	<code>phones</code>
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

<code>name</code>	<code>num</code>
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 with Apache Hive and Impala

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

Complex Data with Impala

- **Impala has limited support for complex data types**
 - Supported in CDH 5.5.0 and higher
 - Only with Parquet tables
 - **ARRAY**, **MAP**, and **STRUCT** are supported
- **Impala CREATE TABLE syntax for complex types is same as HiveQL**
 - Except the collection item and map key terminators cannot be specified
- **Impala 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

<code>cust_id</code>	<code>name</code>	<code>address</code>
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

<code>name</code>	<code>state</code>	<code>zipcode</code>
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	num
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;
```

Chapter Topics

Complex Data with Apache Hive and Impala

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

Essential Points

- **Complex types allow 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**
 - Requires CDH 5.5.0 and higher
 - Only with Parquet tables
 - No **INSERT** capability
- **Hive and Impala use different query syntax for **ARRAY** and **MAP** types**

Chapter Topics

Complex Data with Apache Hive and Impala

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

Hands-On Exercise: Complex Data with Hive and Impala

- 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 with Apache Hive and Impala

Chapter 14



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- **Analyzing Text with Apache Hive and Impala**
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Analyzing Text with Apache Hive and Impala

In this chapter, you will learn

- How to use regular expressions in Hive and Impala
- What role SerDes play in Hive
- How to process unstructured or semi-structured text data with Hive
- What n -grams are and why they are useful
- How to use Hive to estimate how often words or phrases occur in text

Chapter Topics

Analyzing Text with Apache Hive and Impala

- **Using Regular Expressions with Hive and Impala**
- Processing Text Data with SerDes in Hive
- Sentiment Analysis and n -grams
- 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

- A regular expression (*regex*) matches a pattern in text

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'
^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 Matching Operator

- The **REGEXP** operator performs regular expression matching

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

- Regular expression matching is case-sensitive in Hive and Impala
 - Apply **lower** or **upper** 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

- Regular expressions can also be used to extract or replace matched text

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.

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
```

Chapter Topics

Analyzing Text with Apache Hive and Impala

- Using Regular Expressions with Hive and Impala
- **Processing Text Data with SerDes in Hive**
- Sentiment Analysis and n -grams
- 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**
 - But extracting it requires a different approach



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)

Log excerpt

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

Regex SerDe

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

- **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)

Log excerpt

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

Regex SerDe

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

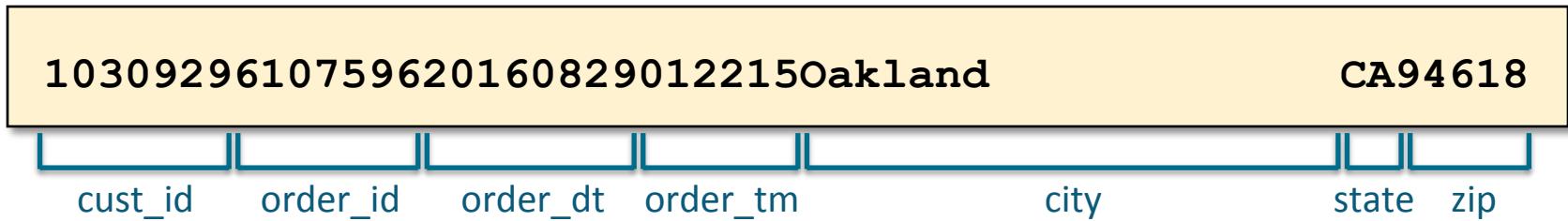
Table excerpt

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



Fixed-Width Formats

- Many older applications produce data in fixed-width formats



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



Fixed-Width Format Example

1030929610759620160829012215Oakland CA94618

Input data

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)
```

Regex SerDe

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

Chapter Topics

Analyzing Text with Apache Hive and Impala

- Using Regular Expressions with Hive and Impala
- Processing Text Data with SerDes in Hive
- **Sentiment Analysis and *n*-grams**
- 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**



Splitting a String into Records

- Hive provides useful functions to work 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



Parsing Sentences into Words

- Hive's **sentences** function parses supplied text into words
- Input is a string containing one or more sentences
- Output is a two-dimensional **ARRAY** of **STRING**
 - 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*=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 search results
 - 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

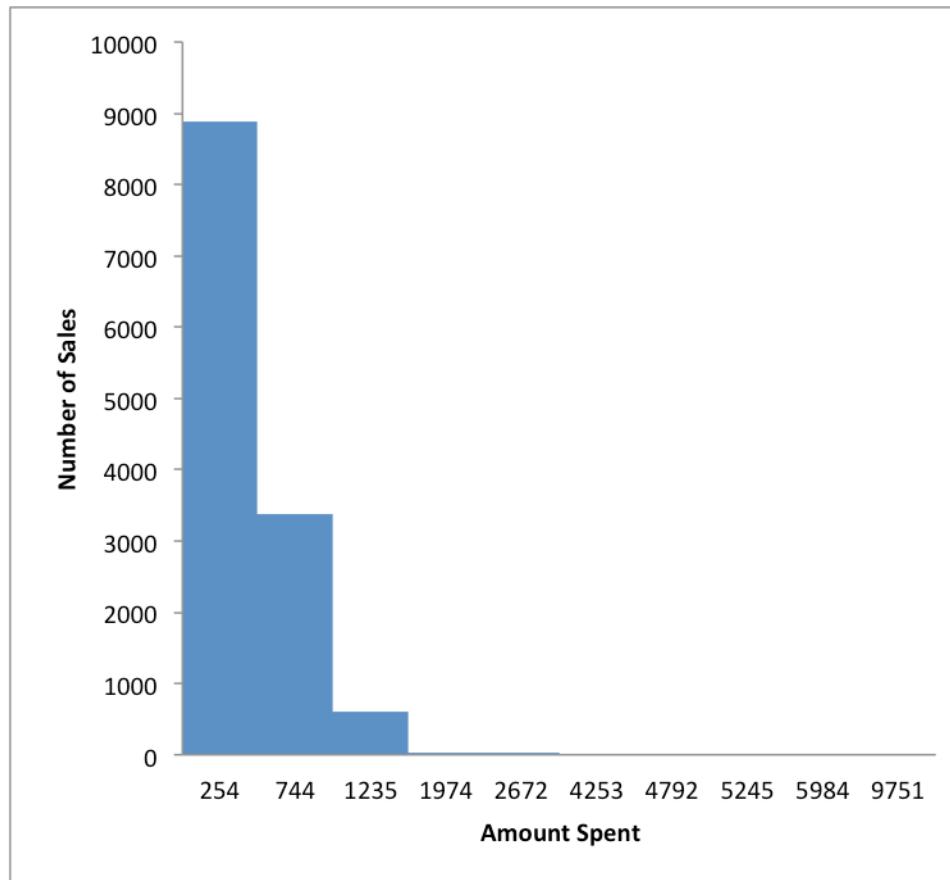
- **context_ngrams** is similar, but considers only specific combinations
 - Additional input parameter: **ARRAY** of words used for filtering
 - Any **NULL** values in the **ARRAY** are treated as placeholders

```
> 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}
```

Histograms

- **Histograms illustrate how values in the data are distributed**
 - This helps us estimate the overall shape of the data distribution





Calculating Data for Histograms

- **histogram_numeric** creates data needed for histograms
 - Input: column name and number of “bins” in the histogram
 - Output: coordinates representing bin centers and heights

```
> SELECT explode(histogram_numeric(total_price, 10))
   FROM cart_orders;
{"x":25417.336745023003,"y":8891.0}
{"x":74401.5041469194,"y":3376.0}
{"x":123550.04418985262,"y":611.0}
{"x":197421.12500000006,"y":24.0}
{"x":267267.53846153844,"y":26.0}
{"x":425324.0,"y":4.0}
{"x":479226.38461538474,"y":13.0}
{"x":524548.0,"y":6.0}
 {"x":598463.5,"y":2.0}
 {"x":975149.0,"y":2.0}
```

Import this data into charting software to produce a histogram

Chapter Topics

Analyzing Text with Apache Hive and Impala

- Using Regular Expressions with Hive and Impala
- Processing Text Data with SerDes in Hive
- Sentiment Analysis and n -grams
- **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
- **An n -gram is a sequence of words**
 - Use **ngrams** and **context_ngrams** in Hive to find their frequency

Chapter Topics

Analyzing Text with Apache Hive and Impala

- Using Regular Expressions with Hive and Impala
- Processing Text Data with SerDes in Hive
- Sentiment Analysis and n -grams
- 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 15



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- **Apache Hive Optimization**
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Apache Hive Optimization

In this chapter, you will learn

- Which factors help determine the performance of Hive queries
- What command displays Hive's execution plan for a query
- How to enable several useful Hive performance features
- How to use table bucketing to sample data
- How to create and rebuild indexes in Hive
- How to use Spark instead of MapReduce as Hive's execution engine

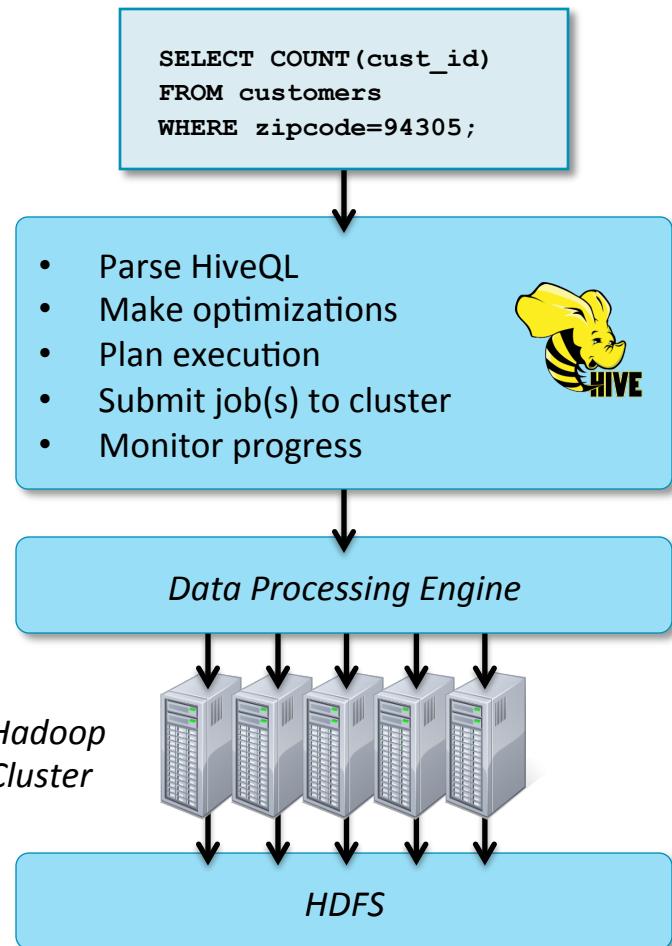
Chapter Topics

Apache Hive Optimization

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

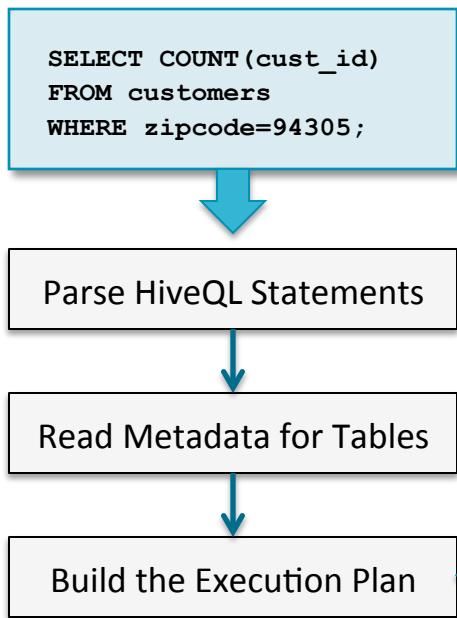
Hive Query Processing

- 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

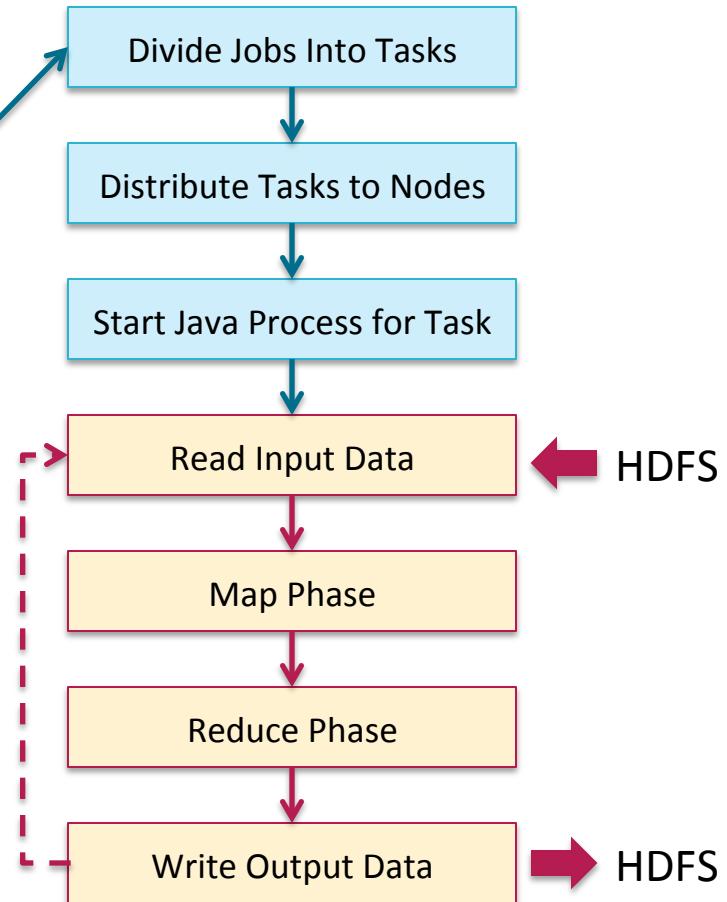


How Hive on MapReduce Processes Data

Steps Run by Hive Server

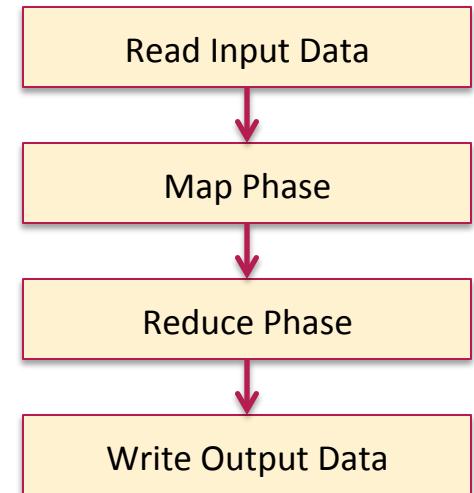


Steps Run on Hadoop Cluster



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 function
 - 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 to sum order totals by sales representative:

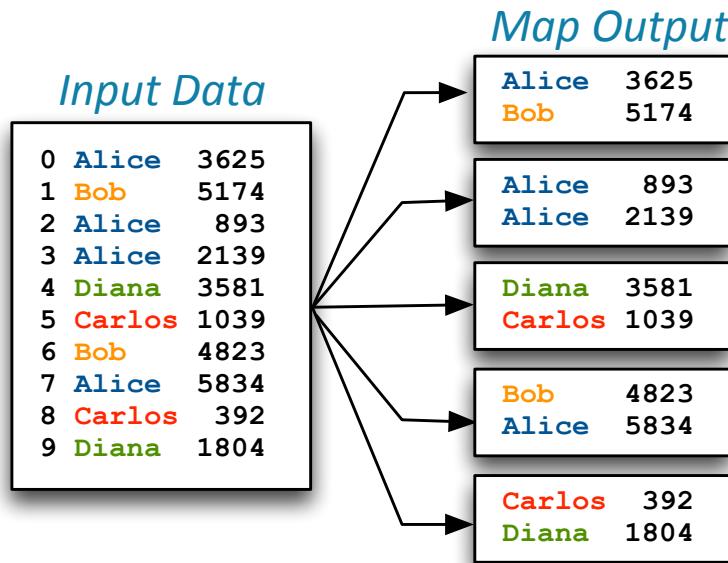
```
SELECT sales_rep, SUM(total)
  FROM order_info
 GROUP BY sales_rep;
```

Input Data

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

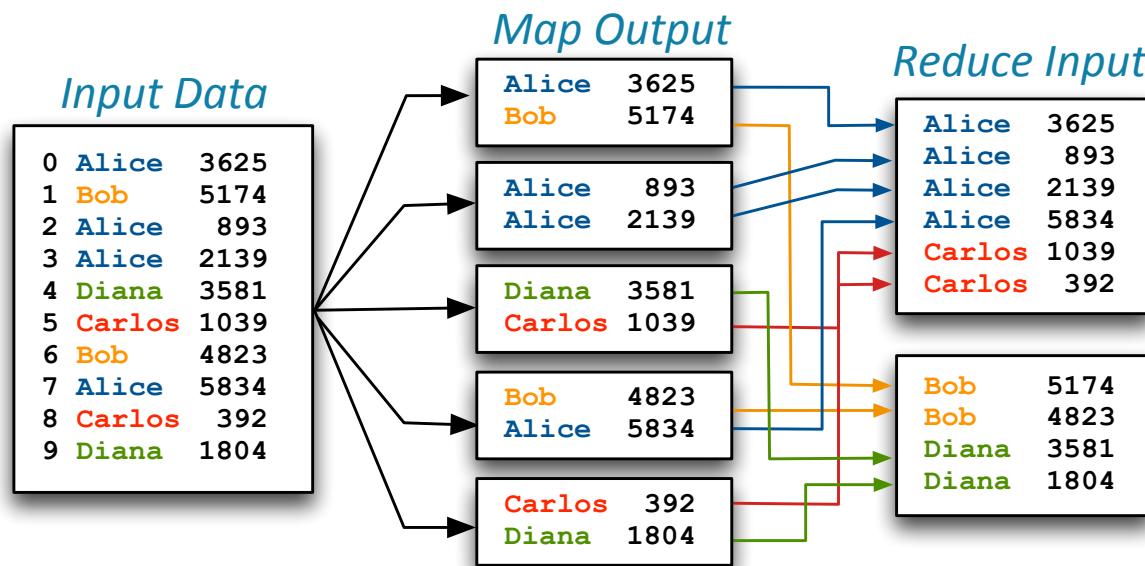
Map Phase

- Hadoop splits the job into many individual *map tasks*
 - Number of map tasks is determined by the amount of input data
 - Each map task receives a portion of the overall job input to process
- In this example, each map task simply reads input records
 - And then emits the name and price fields for each record as output



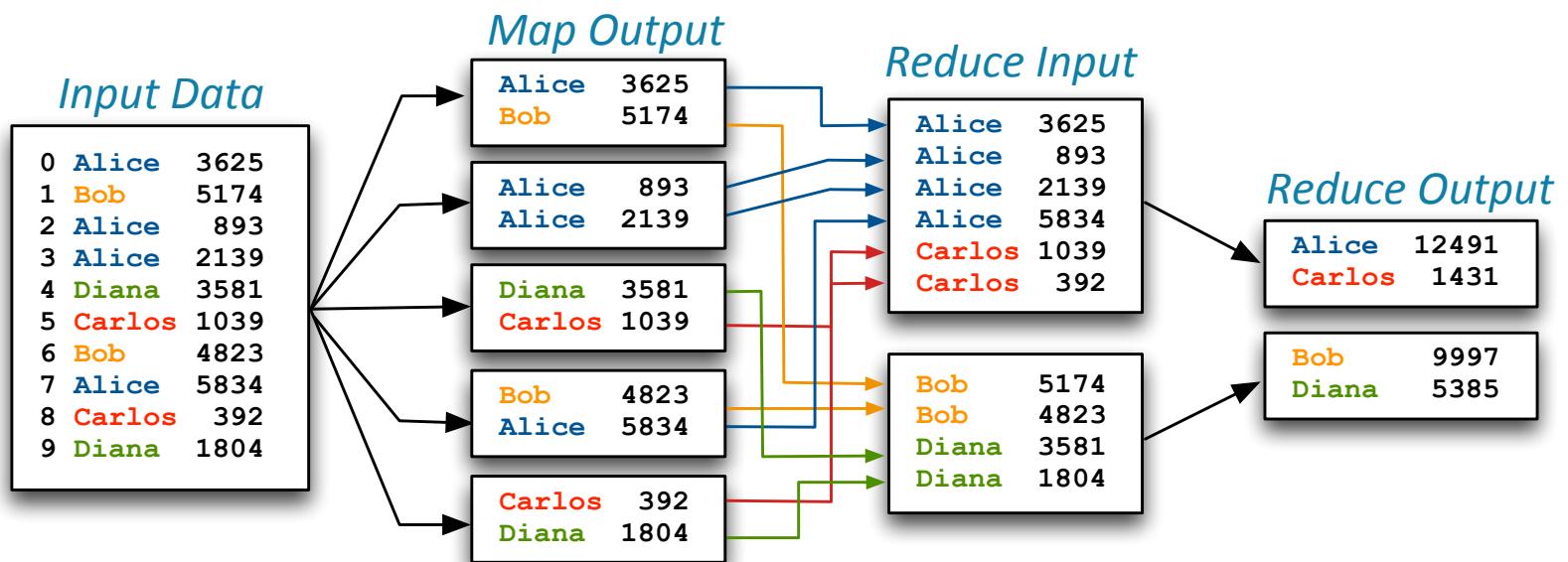
Shuffle and Sort

- 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 totals 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 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 navigation bar with links for 'Query Editors', 'Data Browsers', 'Workflows', and 'Security'. On the far right of the navigation bar, there is a toolbar with several icons, one of which is circled in red and has a black arrow pointing to it from the text 'Job Browser' located above the table. Below the navigation bar, the title 'Job Browser' is displayed next to a blue icon. There are search fields for 'Username' and 'Text', and buttons for filtering by 'Succeeded', 'Running', 'Failed', and 'Killed'. The main area contains a table listing four recent jobs. The columns are labeled 'Logs ID', 'Name', 'Application Type', 'Status', 'User', 'Maps', and 'Reduces'. Each job entry shows a small icon, the job ID, the query name, the application type (MAPREDUCE), the status (SUCCEEDED), the user (anonymous), and completion percentages for maps and reduces (both at 100%).

Logs ID	Name	Application Type	Status	User	Maps	Reduces
1475854045986_0067	SELECT sales_rep, total FROM order_i...total(Stage-1)	MAPREDUCE	SUCCEEDED	anonymous	100%	100%
1475854045986_0066	SELECT sales_rep, total FROM order_i...total(Stage-1)	MAPREDUCE	SUCCEEDED	anonymous	100%	100%
1475854045986_0065	SELECT sales_rep, total FROM ord...sales_rep(Stage-1)	MAPREDUCE	SUCCEEDED	anonymous	100%	100%
1475854045986_0064	SELECT sales_rep, total FROM order_i...total(Stage-1)	MAPREDUCE	SUCCEEDED	anonymous	100%	100%



Viewing a Job in Hue (2)

- View job details including tasks, metadata, and counters

Attempts Tasks Metadata Counters

Recent Tasks

Logs Tasks

- task_1475854045986_0061_m_000000
- task_1475854045986_0061_r_000000

Attempts Tasks Metadata Counters

Org.apache.hadoop.mapreduce.file System Counter

Name	Maps Total	Reduces Total	Total
File Bytes Read	0	234	234
File Bytes Written	233300	233245	466545
File Large Read Ops	0	0	0
File Read Ops	0	0	0
File Write Ops	0	0	0
Hdfs Bytes Read	3030	2789	5819
Hdfs Bytes Written	0	106	106
Hdfs Large Read Ops	0	0	0
Hdfs Read Ops	3	3	6
Hdfs Write Ops	0	2	2



Hive Server Web UI

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

Home Local logs Metrics Dump Hive Configuration Stack Trace

Base Profile Stages Query Plan Performance Logging

User Name	anonymous
Query String	SELECT zipcode, COUNT(cust_id) AS num FROM customers GROUP BY zipcode ORDER BY num DESC LIMIT 10
Query Id	hive_20161017132828_3d2a227a-a0b8-4fdd-9c30-547955ed6524
Execution Engine	mr
State	FINISHED
Begin Time	Mon Oct 17 13:28:03 PDT 2016
Elapsed Time (s)	52
End Time	Mon Oct 17 13:28:55 PDT 2016



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**
- Indexing Data
- 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 for values inserted into bucketed columns
 - Hash codes are used to assign new 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 Data into a Bucketed Table

- Bucketing is not automatically enforced when inserting data
- Set the `hive.enforce.bucketing` property to `true`
 - This sets the number of reduce tasks to the number of buckets in the table definition

```
SET hive.enforce.bucketing=true;
INSERT OVERWRITE TABLE orders_bucketed
    SELECT * FROM orders;
```



Sampling Data from a Bucketed Table

- Use the following syntax to sample data from a bucketed table
 - 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
- **Indexing Data**
- Hive on Spark
- Essential Points
- Hands-On Exercise: Hive Optimization



Indexes in Hive

- Tables in Hive also support indexes
 - Similar to indexes in RDBMSs, but much more limited
- Indexes can improve performance for certain types of queries
 - But maintaining them costs disk space and CPU time
- Syntax to create an index:

```
CREATE INDEX idx_orders_cust_id
  ON TABLE orders(cust_id)
  AS 'handler'
  WITH DEFERRED REBUILD;
```

- Supported handlers are **COMPACT** and **BITMAP**



Viewing and Building Indexes in Hive

- This command lists the indexes associated with the `orders` table

```
SHOW INDEX ON orders;
```

- Hive indexes are initially empty

- Building (and later rebuilding) indexes is a manual process
- Use the `ALTER INDEX` command to rebuild an index
- Caution: This can be a lengthy operation

```
ALTER INDEX idx_orders_cust_id ON orders REBUILD;
```

Chapter Topics

Apache Hive Optimization

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

MapReduce and Spark

- **MapReduce was the exclusive execution engine underlying Hive**
 - And many other 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
- **To use Spark as Hive's execution engine, set `hive.execution.engine`**
 - For Hive on MapReduce, set to `mr` (the default value)
 - 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**
 - And 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
 - 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 Hue Job Browser or the Hadoop web UI

Spark 1.6.0 Jobs Stages Storage Environment Executors **Hive on Spark application UI**

Spark Jobs (?)

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

Status: SUCCEEDED
Completed Stages: 3

▶ Event Timeline
▼ DAG Visualization

```
graph TD; subgraph Stage0 [Stage 0]; A[hadoopRDD] --> B[mapPartitions]; end; subgraph Stage1 [Stage 1]; C[groupByKey] --> D[mapValues]; D --> E[mapPartitions]; end; subgraph Stage2 [Stage 2]; F[sortByKey] --> G[mapPartitions]; G --> H[mapPartitions]; end; B --> C; E --> F;
```

Chapter Topics

Apache Hive Optimization

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

Essential Points

- **The EXPLAIN command shows a query's execution plan**
 - Understanding the execution plan helps you understand query performance
- **Bucketing subdivides a table's data**
 - Useful for jobs that need samples of data
 - Joins may be faster if all tables are bucketed on the join column
- **Hive's indexing feature can boost performance for certain queries**
 - But it comes at the cost of increased disk and CPU usage
- **Hive on Spark improves Hive's query performance**
 - Uses Spark instead of MapReduce as the underlying execution engine
 - You can set the Spark or MapReduce engine on a per-query basis
 - Fully compatible with HiveQL

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>
- Hive manual for indexes
 - <http://tiny.cloudera.com/dac13c>
- Faster Batch Processing with Hive on Spark
 - http://tiny.cloudera.com/hive_on_spark

Chapter Topics

Apache Hive Optimization

- Understanding Query Performance
- Bucketing
- Indexing Data
- 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 16



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- **Apache Impala Optimization**
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Apache Impala Optimization

In this chapter, you will learn

- **How Impala executes queries in a cluster**
- **How Impala caches metadata and how to refresh it**
- **What commands to use to see how Impala executes a query**
- **How to improve Impala performance**

Chapter Topics

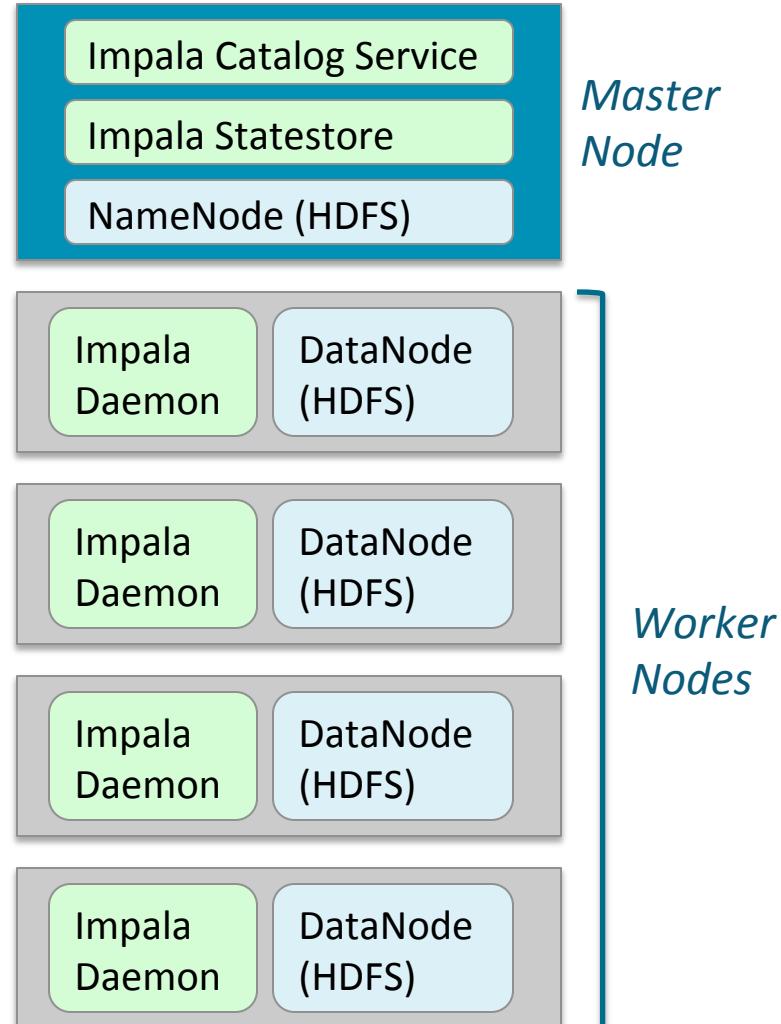
Apache Impala Optimization

- **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*
 - Provides lookup service for Impala daemons
 - Periodically checks status of Impala daemons
 - The *catalog service*
 - Relays metadata changes to all the Impala daemons in a cluster

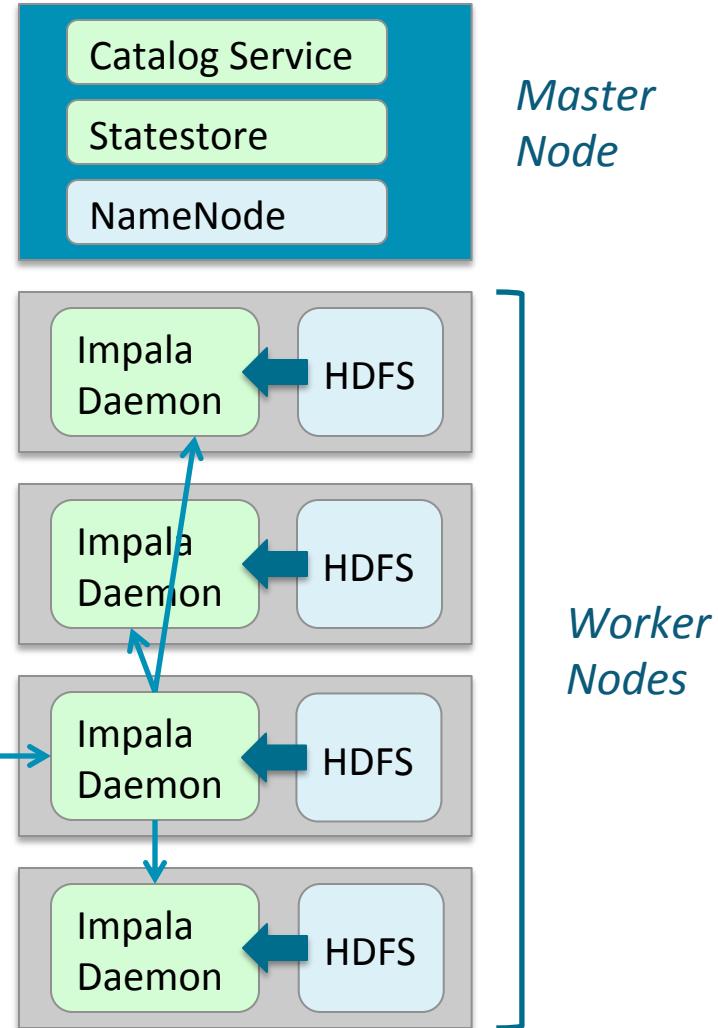




How Impala Executes a Query

■ Impala daemon plans the query

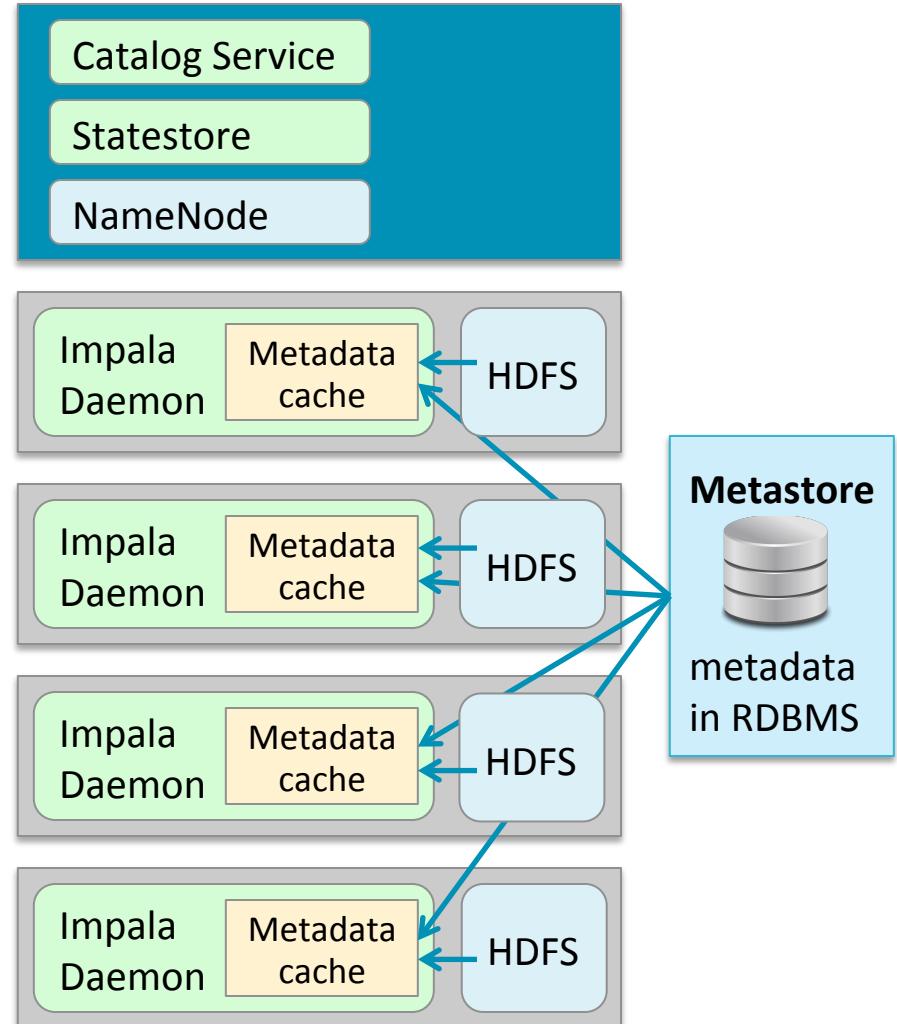
- Client (Impala shell or Hue) connects to an Impala daemon
 - This is the *coordinator*
- Coordinator requests a list of other Impala daemons in the cluster from the statestore
- Coordinator distributes the query across other Impala daemons
- Coordinator streams results to client





Metadata Caching (1)

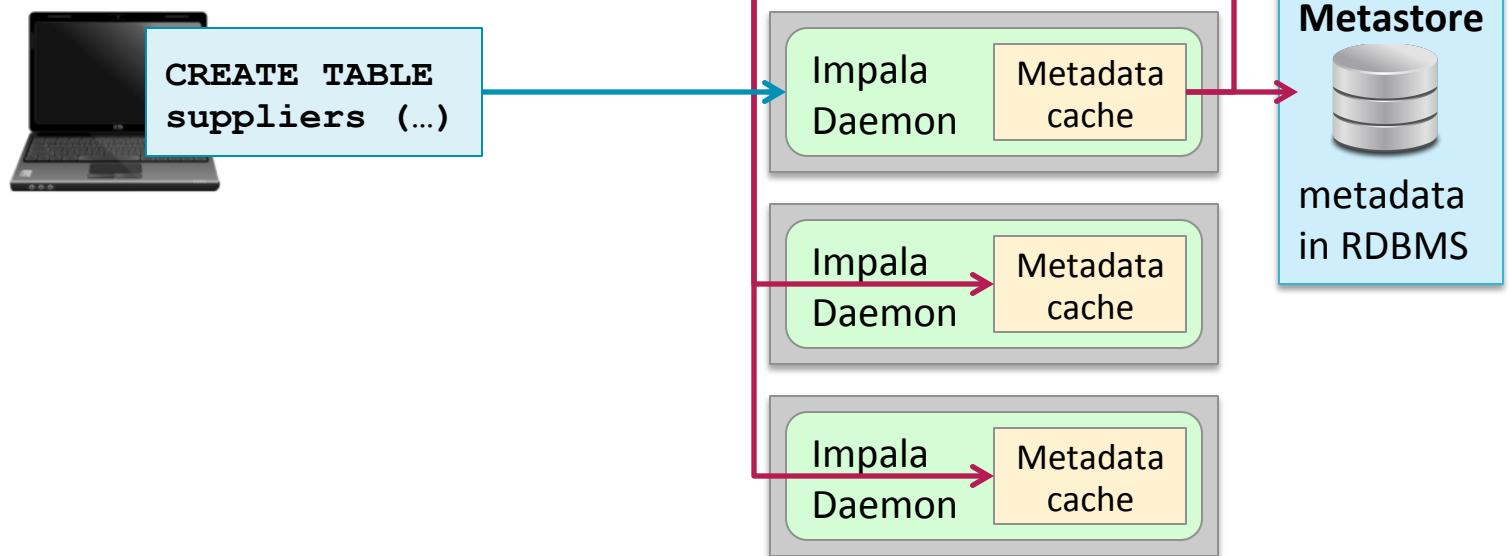
- **Impala daemons cache metadata**
 - Table schema definitions
 - Locations of HDFS blocks containing table data
- **Metadata is cached from the metastore and HDFS at startup**
- **This reduces query latency**
 - Retrieving metadata can take significant time





Metadata Caching (2)

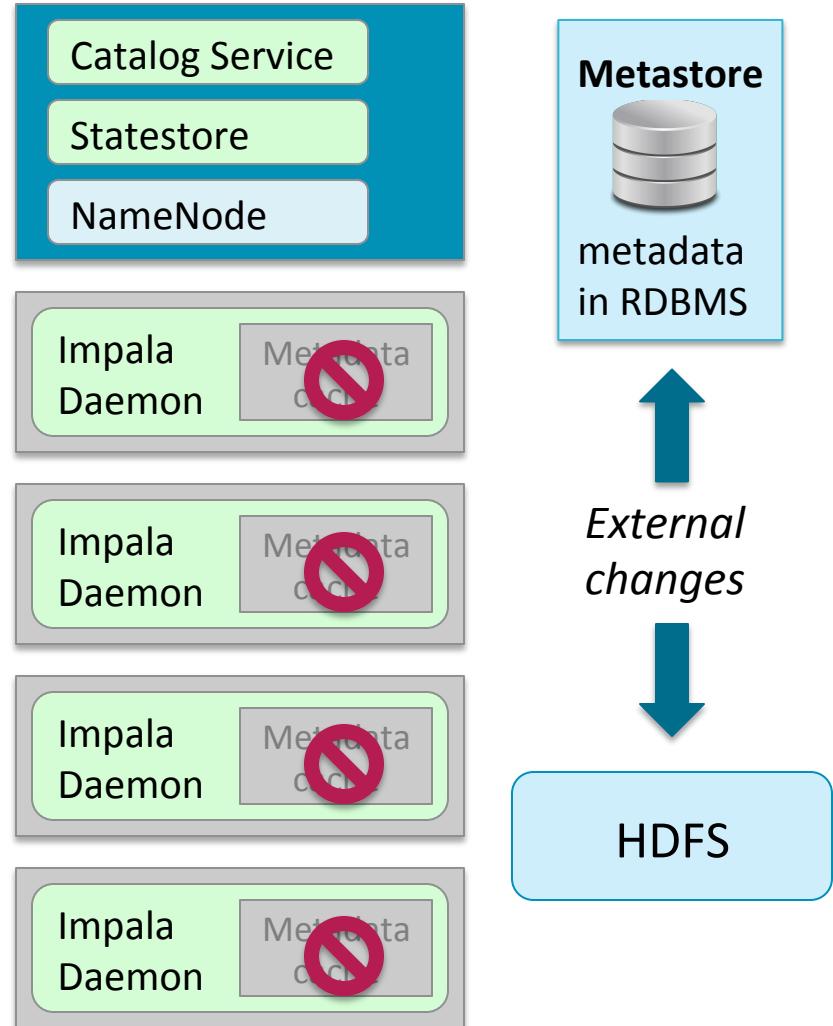
- When one Impala daemon changes the metastore or the location of HDFS blocks, it notifies the catalog service
- The catalog service notifies all Impala daemons to update their cache





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**





Updating the Impala Metadata Cache

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



Query Fault Tolerance in Impala

- **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

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
- **Earlier Impala versions required intermediate data to fit in cluster memory**
 - Versions prior to Impala 2.0/CDH 5.2



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 *  
  FROM customers  
 WHERE state='NY';
```

```
+-----+  
| Explain String |  
+-----+  
| Estimated Per-Host Requirements:  
Memory=48.00MB Vcores=1  
...  
...
```

The screenshot shows the Hue interface for viewing Impala queries. At the top, there is a code editor window containing the following SQL query:

```
1 SELECT *\n2   FROM customers\n3 WHERE state='NY';
```

Below the code editor, there is a toolbar with several icons. One icon, which looks like a document with a downward arrow, is highlighted with a red box and has a red arrow pointing to it from the left side of the slide. To the right of this icon is the word "Query History". Further to the right are "Saved Queries" and an "Explain" button, which is also highlighted with a red box and has a red arrow pointing to it from the bottom right corner of the slide.

The main pane below the toolbar displays the execution plan. It starts with the estimated per-host requirements:

Estimated Per-Host Requirements: Memory=48.00MB Vcores=1
WARNING: The following tables are missing relevant table
default.customers

Then it shows the execution steps:

```
01:EXCHANGE [UNPARTITIONED]  
|  
00:SCAN HDFS [default.customers]  
  partitions=1/1 files=4 size=11.99MB  
  predicates: state = 'NY'
```



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;
```



```
Estimated Per-Host Requirements: Memory=85.49MB
VCores=1

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 (2)

Requirements for
the whole query

Read query
stages from the
bottom up

Estimated Per-Host Requirements: Memory=85.49MB
vCores=1

```
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 (not available in Hue)
 - 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
- **Impala caches metadata from the metastore and HDFS**
 - Use **INVALIDATE METADATA** or **REFRESH** to update the cache following external changes
- **Impala's query planner uses table and column statistics to optimize join operations**
 - Use **COMPUTE STATS** to calculate stats before querying
- **Use EXPLAIN to understand a query plan**
 - Use **SUMMARY** or **PROFILE** to see how the query executed afterwards

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 Hands-On 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 17



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- **Extending Apache Hive and Impala**
- Choosing the Best Tool for the Job
- Conclusion

Extending Apache Hive and Impala

In this chapter, you will learn

- **How to use custom SerDes and custom file formats in Hive**
- **How to use TRANSFORM for custom record processing in Hive**
- **How to extend Hive and Impala with user-defined functions (UDFs)**
- **How to use variable substitution**

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)

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>');
```



Example: Custom XML SerDe and InputFormat (2)

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

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;
```



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

fname	email	employees table
Antoine	antoin@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

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');
```

```
while (<
    ($name = <$name>),
    ($suffix = <$suffix>),
    $greet = <$greet>,
    $greet = <$greet>,
    print <$print>
}
```

The first line tells the system to use the Perl interpreter when running this script.

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
```

```
%greeti:
```

Read each record from standard input within the loop, and then split them into fields based on tab characters.

```
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 (5)

```
#!/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);
}
```

- Extract the country code from the email address (the pattern matches any letters following the final dot). 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',
```

wl 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.)

```
    $greeting = $greetings{$name},
    $greeting = 'Hello' unless defined($greeting);
    print "$greeting $name\n";
}
```



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



Using Scripts in a Secure Cluster

- **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
 - Workaround: use Hadoop Streaming instead of Hive to invoke the script

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)

- **This example UDF was compiled from sources found on GitHub**
 - Popular website for many open source software projects
 - Project URL: <http://tiny.cloudera.com/dac14e>
- **The UDF is packaged into a JAR file on the hands-on environment**
- **This example shows the `date_format` UDF in that JAR file**
 - Allows great flexibility in formatting date fields in output



Example: Using a UDF in Hive (2)

- First, copy the JAR file to HDFS
 - Same step as with a custom SerDe

```
$ hdfs dfs -put date-format-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 date_format
  AS 'com.nexr.platform.hive.udf.UDFDateFormat'
  USING JAR 'hdfs:/myscripts/date-format-udf.jar';
```

- Hive persists the function in the metastore database
- To remove the function, use **DROP FUNCTION date_format;**



Example: Using a UDF in Hive (3)

- You may then use the function in your query

```
> SELECT order_date FROM orders LIMIT 1;  
2016-12-06 10:03:35  
  
> SELECT date_format(order_date, 'dd-MMM-yyyy')  
    FROM orders LIMIT 1;  
06-Dec-2016  
  
> SELECT date_format(order_date, 'dd/mm/yy')  
    FROM orders LIMIT 1;  
06/12/16  
  
> SELECT date_format(order_date, 'EEEE, MMMMM d, yyyy')  
    FROM orders LIMIT 1;  
Tuesday, December 6, 2016
```



Overview of Impala User-Defined Functions

- **Impala also supports user-defined functions**
 - “Standard” UDFs and UDAFs are supported
 - 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
 - After refreshing the metastore cache with **INVALIDATE METADATA**;
 - Usage may differ slightly in Impala

```
> SELECT date_format(cast(order_date AS STRING) ,  
    'dd-MMM-yyyy')  
    FROM orders LIMIT 1;  
06-Dec-2016
```

- Or you can use Impala to register a Java UDF
 - But the syntax is different than in Hive

```
CREATE FUNCTION date_format  
    LOCATION 'hdfs:/myscripts/date-format-udf.jar'  
    SYMBOL='com.nexr.platform.hive.udf.UDFDateFormat';
```



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 parameterization of repetitive queries
- **This feature is implemented differently in Hive and Impala**
 - In Impala, variable substitution only works using the Impala shell
 - The Hive and Impala syntax differ



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

state.sql

```
SELECT COUNT(DISTINCT cust_id) FROM customers  
WHERE state = '${var:mystate}';
```

Command line

```
$ impala-shell --var=mystate="CA" -f state.sql  
$ impala-shell --var=mystate="NY" -f state.sql
```

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

- **File formats and SerDes govern table storage in Hive**
 - Hive allows creation of custom file formats and SerDes
- **TRANSFORM processes records in Hive using an external program**
 - This can be written in nearly any language
- **Hive and Impala support user-defined functions**
 - Custom logic that can be invoked just like built-in functions
- **Hive and Impala can substitute variables with values you assign**
 - This is done when you execute the query
 - Especially helpful with repetitive queries

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 Hands-On 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 18



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- **Choosing the Best Tool for the Job**
- Conclusion

Choosing the Best Tool for the Job

In this chapter, you will learn

- **How Pig, Hive, Impala, and RDBMSs compare**
- **Why a workflow might involve several different tools**
- **How to select the best tool for a given job**

Chapter Topics

Choosing the Best Tool for the Job

- **Comparing Pig, Hive, Impala, and Relational Databases**
- Which to Choose?
- Essential Points
- Optional Hands-On Exercise: Analyzing Abandoned Carts

Recap of Data Analysis and Processing Tools

- **Pig**
 - Procedural data flow language executed using MapReduce
- **Hive**
 - SQL-based queries executed using MapReduce or Spark
- **Impala**
 - High-performance SQL-based queries using a custom execution engine

Comparing Pig, Hive, and Impala

Feature	Pig	Hive	Impala
SQL-based query language	No	Yes	Yes
Schemas optional	Yes	No	No
User-defined functions (UDFs)	Yes	Yes	Yes
Process data with external scripts	Yes	Yes	No
Extensible record and file formats	Yes	Yes	No
Complex data types	Yes	Yes	Limited
Query latency	High	High	Low
Built-in data partitioning	No	Yes	Yes
Accessible with ODBC / JDBC	No	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**)
- **Pig, Hive, and Impala are optimized for**
 - Large amounts of read-only data
 - Extensive scalability at low cost
- **Pig and Hive are 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*	No
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

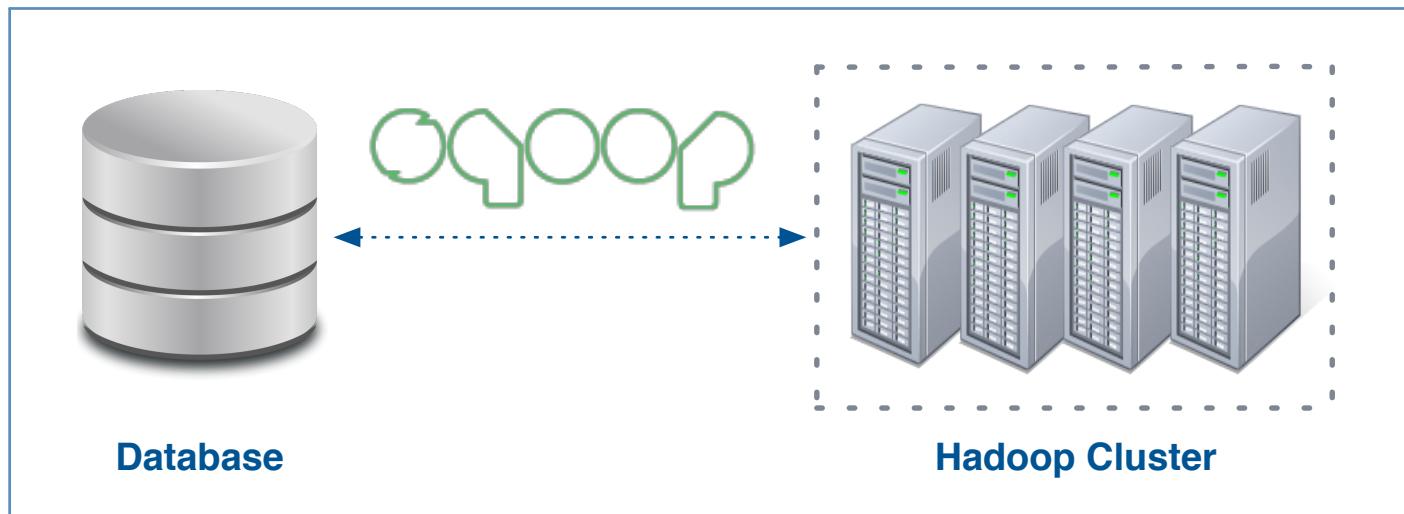
* Hive now has limited, experimental support for **UPDATE**, **DELETE**, and transactions.
Cloudera neither recommends nor supports using these features in Hive.

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
 - Indexing
 - Bucketing and table sampling
 - 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

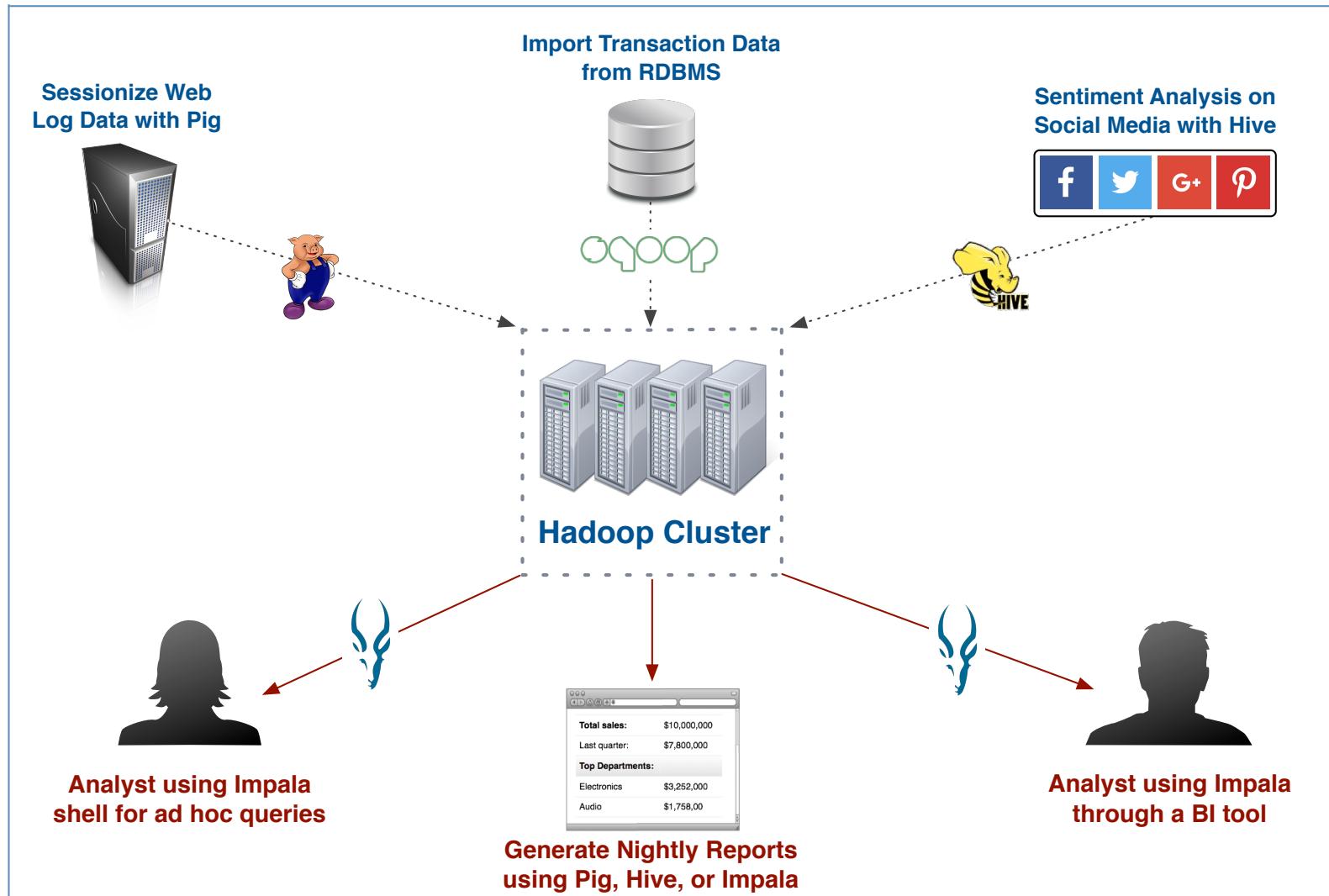
Choosing the Best Tool for the Job

- Comparing Pig, Hive, Impala, and Relational Databases
- **Which to Choose?**
- Essential Points
- Optional Hands-On Exercise: Analyzing Abandoned Carts

Which to Choose?

- **Pig turns data flow language scripts into MapReduce jobs**
 - Good choice for batch processing and ETL, or when schema is unknown
- **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

- **Pig, 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 Pig, Hive, Impala, and Relational Databases
- Which to Choose?
- **Essential Points**
- Optional Hands-On Exercise: Analyzing Abandoned Carts

Essential Points

- **You have learned about several tools for data processing and analysis**
 - Each is better at some tasks than others
 - Choose the best one for a given job
 - Workflows may involve exchanging data between them
- **Selection criteria include scale, speed, control, and productivity**
 - MapReduce and Spark offer control at the cost of productivity
 - Pig and Hive offer productivity but not necessarily speed
 - Relational databases offer speed but not scalability
 - Impala offers scalability and speed but has some limitations

Chapter Topics

Choosing the Best Tool for the Job

- Comparing Pig, 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 Hands-On 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 19



Course Chapters

- Introduction
- Apache Hadoop Fundamentals
- Introduction to Apache Pig
- Basic Data Analysis with Apache Pig
- Processing Complex Data with Apache Pig
- Multi-Dataset Operations with Apache Pig
- Apache Pig Troubleshooting and Optimization
- Introduction to Apache Hive and Impala
- Querying with Apache Hive and Impala
- Apache Hive and Impala Data Management
- Data Storage and Performance
- Relational Data Analysis with Apache Hive and Impala
- Complex Data with Apache Hive and Impala
- Analyzing Text with Apache Hive and Impala
- Apache Hive Optimization
- Apache Impala Optimization
- Extending Apache Hive and Impala
- Choosing the Best Tool for the Job
- Conclusion

Conclusion (1)

During this course, you have learned

- **The purpose of Apache Hadoop and its related tools**
- **The features that Apache Pig, Apache Hive, and Apache Impala (incubating) offer for data acquisition, storage, and analysis**
- **How to identify typical use cases for large-scale data analysis**
- **How to load data from relational databases and other sources**
- **How to manage data in HDFS and export it for use with other systems**
- **How Pig, Hive, and Impala improve productivity for typical analysis tasks**
- **The language syntax and data formats supported by these tools**

Conclusion (2)

- How to design and execute queries on data stored in HDFS
- How to join diverse datasets to gain valuable business insight
- How Hive and Impala can be extended with custom functions and scripts
- How to store and query complex or nested data structures
- How to analyze structured, semi-structured, and unstructured data
- How to store and query data for better performance
- How to determine which tool is the best choice 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**
 - *Data Science at Scale using Spark and Hadoop*
- **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**



Extending Apache Pig

Appendix A



Extending Apache Pig

In this appendix, you will learn

- How to use parameters in your Pig Latin code to increase its flexibility
- How to define and invoke macros to improve the reusability of your code
- How to call user-defined functions from your code
- How to use load and store functions to integrate Pig with Hive and Impala
- How to write user-defined functions in Python
- How to process data with external scripts

Chapter Topics

Extending Apache Pig

- **Adding Flexibility with Parameters**
- Macros and Imports
- User-Defined Functions
- Contributed Functions
- Load and Store Functions
- Using Other Languages to Process Data with Pig
- Essential Points
- Hands-On Exercise: Extending Pig with Streaming and UDFs

The Need for Parameters (1)

- Some processing is very repetitive
 - For example, creating sales reports

```
allsales = LOAD 'sales' AS (name, price);  
bigsales = FILTER allsales BY price > 999;  
  
bigsales_alice = FILTER bigsales BY name == 'Alice';  
STORE bigsales_alice INTO 'Alice';
```

The Need for Parameters (2)

- You may need to change the script slightly for each run
 - For example, to modify the paths or filter criteria

```
allsales = LOAD 'sales' AS (name, price);  
bigsales = FILTER allsales BY price > 999;  
  
bigsales_alice = FILTER bigsales BY name == 'Alice';  
STORE bigsales_alice INTO 'Alice';
```

Making the Script More Flexible with Parameters

- Instead of hardcoding values, Pig allows you to use parameters
 - These are replaced with specified values at runtime

reporter.pig

```
allsales = LOAD '$INPUT' AS (name, price);  
bigsales = FILTER allsales BY price > $MINPRICE;  
  
bigsales_name = FILTER bigsales BY name == '$NAME';  
STORE bigsales_name INTO '$NAME';
```

- Then specify the values on the command line

Command line

```
$ pig -p INPUT=sales -p MINPRICE=999 \  
-p NAME=Alice reporter.pig
```

Two Tricks for Specifying Parameter Values

- You can also specify parameter values in a text file
 - An alternative to typing each one on the command line

```
INPUT=sales
MINPRICE=999
# comments look like this
NAME='Alice'
```

- Use **-m filename** option to tell Pig which file contains the values
- Parameter values can be defined with the output of a shell command
 - For example, to set **MONTH** to the current month

```
MONTH=`date +'%m'`      # returns 03 for March, 05 for May
```

Chapter Topics

Extending Apache Pig

- Adding Flexibility with Parameters
- **Macros and Imports**
- User-Defined Functions
- Contributed Functions
- Load and Store Functions
- Using Other Languages to Process Data with Pig
- Essential Points
- Hands-On Exercise: Extending Pig with Streaming and UDFs

The Need for Macros

- Parameters simplify repetitive code by allowing you to pass in values
 - But sometimes you would like to reuse the actual code too

```
allsales = LOAD 'sales' AS (name, price);
byperson = FILTER allsales BY name == 'Alice';

SPLIT byperson INTO low IF price < 1000,
    high IF price >= 1000;

amt1 = FOREACH low GENERATE name, price * 0.07 AS amount;
amt2 = FOREACH high GENERATE name, price * 0.12 AS amount;

commissions = UNION amt1, amt2;
grpds = GROUP commissions BY name;

out = FOREACH grpds GENERATE SUM(commissions.amount) AS total;
```

Defining a Macro in Pig Latin

- Macros allow you to define a block of code to reuse easily
 - Similar (but not identical) to a function in a programming language

```
define calc_commission (NAME, SPLIT_AMT, LOW_PCT, HIGH_PCT)
returns result {
    allsales = LOAD 'sales' AS (name, price);
    byperson = FILTER allsales BY name == '$NAME';

    SPLIT byperson INTO low if price < $SPLIT_AMT,
        high IF price >= $SPLIT_AMT;

    amt1 = FOREACH low GENERATE name, price * $LOW_PCT AS amount;
    amt2 = FOREACH high GENERATE name, price * $HIGH_PCT AS amount;

    commissions = UNION amt1, amt2;
    grouped = GROUP commissions BY name;

    $result = FOREACH grouped GENERATE SUM(commissions.amount);
};
```

Invoking Macros

- To invoke a macro, call it by name and supply values in the correct order

```
define calc_commission (NAME, SPLIT_AMT, LOW_PCT, HIGH_PCT)
returns result {
    allsales = LOAD 'sales' AS (name, price);

    . . . (other code removed for brevity) . .

    $result = FOREACH grouped GENERATE SUM(commissions.amount);
};

alice_comm = calc_commission('Alice', 1000, 0.07, 0.12);
carlos_comm = calc_commission('Carlos', 2000, 0.08, 0.14);
```

Reusing Code with Imports

- After defining a macro, you may wish to use it in multiple scripts
- You can include one script within another
 - Use the `import` keyword and the path to the file being imported

```
-- Macro is defined in a file named commission_calc.pig  
  
import 'commission_calc.pig';  
  
alice_comm = calc_commission('Alice', 1000, 0.07, 0.12);
```

Chapter Topics

Extending Apache Pig

- Adding Flexibility with Parameters
- Macros and Imports
- **User-Defined Functions**
- Contributed Functions
- Load and Store Functions
- Using Other Languages to Process Data with Pig
- Essential Points
- Hands-On Exercise: Extending Pig with Streaming and UDFs

User-Defined Functions (UDFs)

- This course has covered many of Pig's built-in functions
- It is also possible to define your own functions
- Pig allows writing user-defined functions (UDFs) in several languages
 - Java
 - Python
 - JavaScript (experimental)
 - Ruby (experimental)
 - Groovy (experimental)
- In the next few slides, you will see how to use UDFs written in Java, and how to write and use UDFs in Python

Using UDFs Written in Java

- UDFs are packaged into JAR files
- There are only two required steps for using them
 - Register the JAR file(s) containing the UDF and its dependencies
 - Invoke the UDF using the fully qualified class name

```
REGISTER '/path/to/myudf.jar';
...
data = FOREACH allsales GENERATE com.example.MYFUNC(name);
```

- You can optionally define an alias for the function

```
REGISTER '/path/to/myudf.jar';
DEFINE FOO com.example.MYFUNC;
...
data = FOREACH allsales GENERATE FOO(name);
```

Writing UDFs in Python (1)

- Now you will see how to write a simple UDF in Python
- The data you want to process has inconsistent phone number formats

Alice	(314) 555-1212
Bob	212.555.9753
Carlos	405-555-3912
David	(202) 555.8471

- You need a UDF that can consistently extract the area code

Writing UDFs in Python (2)

- The Python code is straightforward
- The only unusual thing is the optional `@outputSchema` decorator
 - This tells Pig what data type to return
 - If not specified, Pig will assume `bytearray`

phonenumbers.py

```
@outputSchema("areacode:chararray")
def get_area_code(phone):
    areacode = "???" # return this for unknown formats

    if len(phone) == 12:
        # XXX-YYY-ZZZZ or XXX.YYY.ZZZZ format
        areacode = phone[0:3]
    elif len(phone) == 14:
        # (XXX) YYY-ZZZZ or (XXX) YYY.ZZZZ format
        areacode = phone[1:4]

    return areacode
```

Invoking Python UDFs from Pig

- Using this UDF in Pig Latin code is also easy
 - The Python code is in the file **phonenumer.py**
 - That file is in the current directory

```
REGISTER 'phonenumer.py' USING jython AS phoneudf;  
  
names = LOAD 'names' AS (name:chararray, phone:chararray);  
  
areacodes = FOREACH names GENERATE  
    phoneudf.get_area_code(phone) AS ac;
```

Chapter Topics

Extending Apache Pig

- Adding Flexibility with Parameters
- Macros and Imports
- User-Defined Functions
- **Contributed Functions**
- Load and Store Functions
- Using Other Languages to Process Data with Pig
- Essential Points
- Hands-On Exercise: Extending Pig with Streaming and UDFs

Open Source UDFs

- Pig ships with a set of community-contributed UDFs called Piggy Bank
- Another popular package of UDFs is Apache DataFu (incubating)

Piggy Bank

- **Piggy Bank ships with Pig**

- You need to register the file **piggybank.jar** to use its functions
- The location may vary depending on source and version
- In the provided hands-on environment, it is located at **/usr/lib/pig/piggybank.jar**

- **UDFs in Piggy Bank include**

Class Name	Description
ISOToUnix	Converts an ISO 8601 date/time format to UNIX format
UnixToISO	Converts a UNIX date/time format to ISO 8601 format
LENGTH	Returns the number of characters in the supplied string
HostExtractor	Returns the host name from a URL
DiffDate	Returns number of days between two dates

- Package names omitted for brevity

Apache DataFu (incubating)

- **DataFu does not ship with Pig, but is part of CDH**
 - You will need to register the DataFu JAR file
 - In the provided hands-on environment, it is located at
`/usr/lib/pig/`
- **UDFs in DataFu include**

Class Name	Description
Quantile	Calculates quantiles for a dataset
Median	Calculates the median for a dataset
Sessionize	Groups data into sessions based on a specified time window
HaversineDistInMiles	Calculates distance in miles between two points, given latitude and longitude

- Package names omitted for brevity

Using a Contributed UDF

- Here is an example of using a UDF from DataFu to calculate distance

37.789336 -122.401385 40.707555 -74.011679

Input data

```
REGISTER '/usr/lib/pig/datafu-* .jar';
DEFINE DIST datafu.pig.geo.HaversineDistInMiles;

places = LOAD 'data' AS (lat1:double, lon1:double,
                        lat2:double, lon2:double);

dist = FOREACH places GENERATE DIST(lat1, lon1, lat2, lon2);
DUMP dist;
```

Pig Latin

(2564.207116295711)

Output data

Chapter Topics

Extending Apache Pig

- Adding Flexibility with Parameters
- Macros and Imports
- User-Defined Functions
- Contributed Functions
- **Load and Store Functions**
- Using Other Languages to Process Data with Pig
- Essential Points
- Hands-On Exercise: Extending Pig with Streaming and UDFs

Recap: Load and Store Functions

- Pig includes built-in functions for loading and storing data
 - The default load and store function is **PigStorage**
 - Other built-in load and store functions include
 - **JsonLoader** and **JsonStorage**
 - **HBaseLoader** and **HBaseStorage**
- You can specify a load or store function with the USING keyword

```
all_products = LOAD 'products' USING PigStorage();
first_ten = LIMIT all_products 10;
STORE first_ten INTO 'ten_products' USING PigStorage();
```

- It is also possible to write custom load and store functions
 - And to use open source load and store functions contributed by others

Sharing Data Between Pig, Hive, and Impala

- The contributed load and store functions `HCatLoader` and `HCatStorer` allow Pig to read from and write to Hive/Impala tables
 - Pig lacks built-in direct metastore access
 - These functions are provided separately as part of HCatalog
- To use, specify fully qualified Java class names after the `USING` keyword

```
all_products = LOAD 'products'  
    USING org.apache.hive.hcatalog.pig.HCatLoader();  
first_ten = LIMIT all_products 10;  
STORE first_ten INTO 'ten_products'  
    USING org.apache.hive.hcatalog.pig.HCatStorer();
```

Chapter Topics

Extending Apache Pig

- Adding Flexibility with Parameters
- Macros and Imports
- User-Defined Functions
- Contributed Functions
- Load and Store Functions
- **Using Other Languages to Process Data with Pig**
- Essential Points
- Hands-On Exercise: Extending Pig with Streaming and UDFs

Processing Data with an External Script

- While Pig Latin is powerful, some tasks are easier in another language
- Pig allows you to stream data through another language for processing
 - This is done using **STREAM**
- Similar concept to Hadoop Streaming and HiveQL's **TRANSFORM**
 - Data is supplied to the script on standard input as tab-delimited fields
 - Script writes results to standard output as tab-delimited fields

STREAM Example in Python (1)

- This example will calculate a user's age given that user's birthdate
 - This calculation is done in a Python script named **agecalc.py**
- Here is the corresponding Pig Latin code
 - Backticks used to quote script name following the alias
 - Single quotes used for quoting script name within **SHIP**
 - The schema for the data produced by the script follows the **AS** keyword

```
DEFINE MYSCRIPT `agecalc.py` SHIP('agecalc.py');
users = LOAD 'data' AS (name:chararray, birthdate:chararray);

out = STREAM users THROUGH MYSCRIPT AS (name:chararray, age:int);

DUMP out;
```

STREAM Example in Python (2)

- Python code for `agecalc.py`

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

import sys
from datetime import datetime

for line in sys.stdin:
    line = line.strip()
    (name, birthdate) = line.split("\t")

    d1 = datetime.strptime(birthdate, '%Y-%m-%d')
    d2 = datetime.now()

    age = int((d2 - d1).days / 365)

    print "%s\t%i" % (name, age)
```

STREAM Example in Python (3)

- The Pig script again, and the data it reads and writes

```
DEFINE MYSCRIPT `agecalc.py` SHIP('agecalc.py') ;
users = LOAD 'data' AS (name:chararray, birthdate:chararray) ;

out = STREAM users THROUGH MYSCRIPT AS (name:chararray,
age:int) ;

DUMP out;
```

Input data

```
andy      1963-11-15
betty     1985-12-30
chuck     1979-02-23
debbie    1982-09-19
```



Output data

```
(andy,53)
(betty,30)
(chuck,37)
(debbie,34)
```

Chapter Topics

Extending Apache Pig

- Adding Flexibility with Parameters
- Macros and Imports
- User-Defined Functions
- Contributed Functions
- Load and Store Functions
- Using Other Languages to Process Data with Pig
- **Essential Points**
- Hands-On Exercise: Extending Pig with Streaming and UDFs

Essential Points

- **Pig supports several extension mechanisms**
- **Parameters and macros can help make your code more reusable**
 - And easier to maintain and share with others
- **Piggy Bank and DataFu are two sources of open source UDFs**
 - You can also write your own UDFs
- **HCatalog load and store functions allow Pig to access the metastore**
 - To read from and write to Hive/Impala tables
- **Pig's STREAM allows you use another language for processing**

Bibliography

The following offer more information on topics discussed in this chapter

- Documentation on parameter substitution in Pig
 - <http://tiny.cloudera.com/dacA1a>
- Documentation on macros in Pig
 - <http://tiny.cloudera.com/dacA1b>
- Documentation on user-defined functions in Pig
 - <http://tiny.cloudera.com/dacA1c>
- Documentation on Piggy Bank
 - <http://tiny.cloudera.com/dacA1d>
- Apache DataFu (incubating)
 - <http://tiny.cloudera.com/dacA1e>

Chapter Topics

Extending Apache Pig

- Adding Flexibility with Parameters
- Macros and Imports
- User-Defined Functions
- Contributed Functions
- Load and Store Functions
- Using Other Languages to Process Data with Pig
- Essential Points
- **Hands-On Exercise: Extending Pig with Streaming and UDFs**

Hands-On Exercise: Extending Pig with Streaming and UDFs

- In this Hands-On Exercise, you will process data with an external script and a user-defined function.
 - Please refer to the Hands-On Exercise Manual for instructions