cloudera®

Developer Training for Spark and Hadoop I

cloudera®

Introduction to Impala and Hive

Chapter 5

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ **Introduction to Impala and Hive** ▪ Working with Tables in Impala ▪ Data Formats ▪ Data File Partitioning	**Importing and Modeling Structured Data**
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	Distributed Data Processing with Spark
▪ Conclusion	Course Conclusion

Introduction to Impala and Hive

In this chapter you will learn

- **What Hive is**
- **What Impala is**
- **How Impala and Hive Compare**
- **How to query data using Impala and Hive**
- **How Hive and Impala differ from a relational database**
- **Ways in which organizations use Hive and Impala**

Chapter Topics

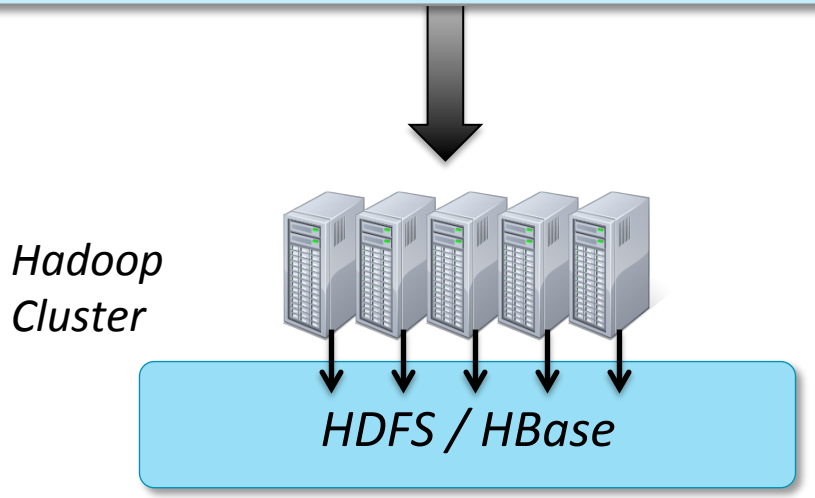
Introduction to Impala and Hive | **Importing and Modeling Structured Data**

- **Introduction to Impala and Hive**
- Why Use Impala and Hive?
- Querying Data With Impala and Hive
- Comparing Hive and Impala to Traditional Databases

Introduction to Impala and Hive (1)

- **Impala and Hive are both tools that provide SQL querying of data stored in HDFS / HBase**

```
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;
```



Introduction to Impala and Hive (2)

- **Apache Hive is a high-level abstraction on top of MapReduce**
  - Uses HiveQL
  - Generates MapReduce or Spark* jobs that run on the Hadoop cluster
  - Originally developed at Facebook around 2007
    - Now an open-source Apache project

- **Cloudera Impala is a high-performance dedicated SQL engine**
  - Uses Impala SQL
  - Inspired by Google's Dremel project
  - Query latency measured in milliseconds
  - Developed at Cloudera in 2012
    - Open-source with an Apache license

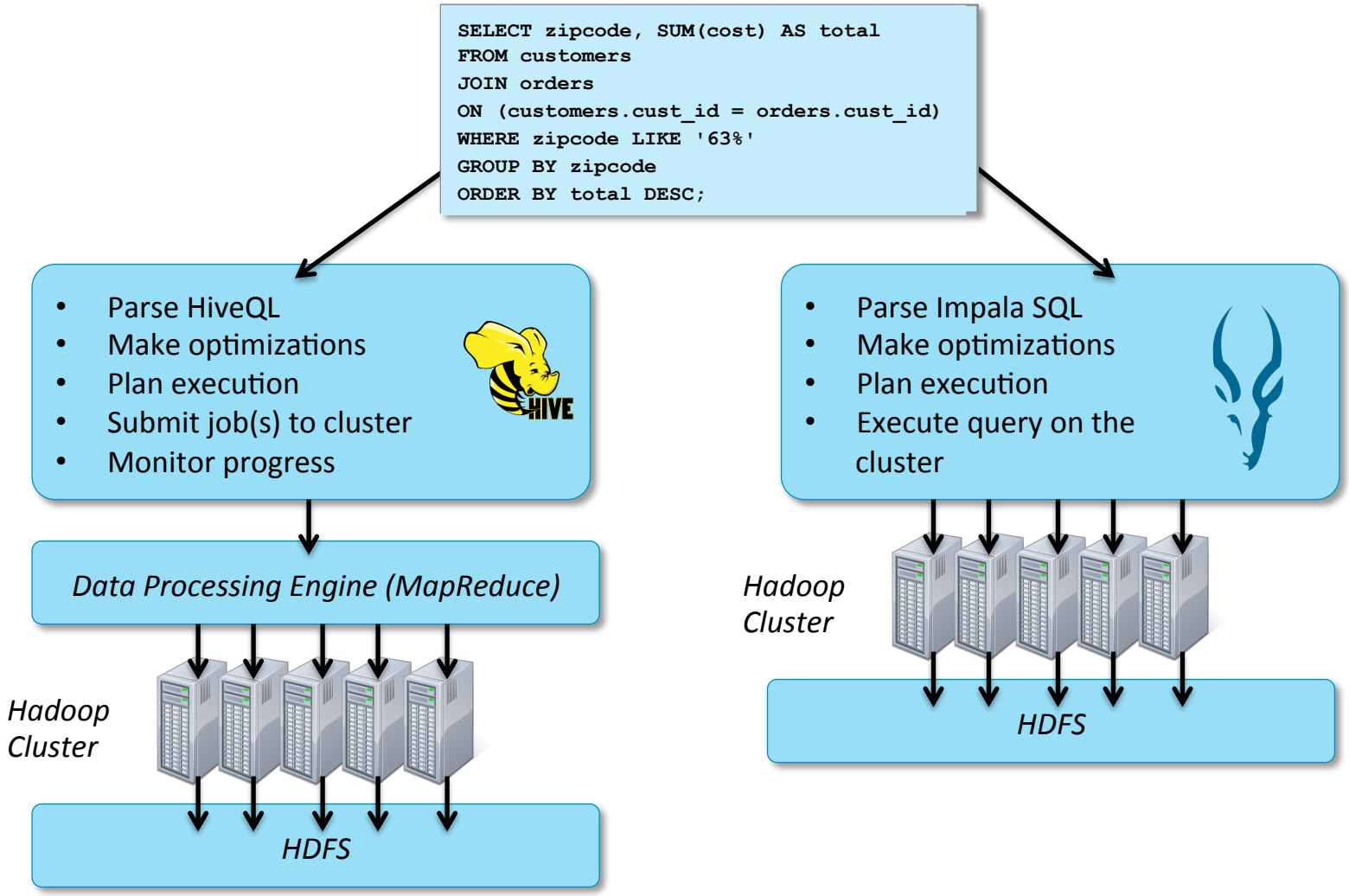
* Hive-on-Spark is currently in beta testing

What's the Difference?

- **Hive has more features**
  - E.g. Complex data types (arrays, maps) and full support for windowing analytics
  - Highly extensible
  - Commonly used for batch processing

- **Impala is much faster**
  - Specialized SQL engine offers 5x to 50x better performance
  - Ideal for interactive queries and data analysis
  - More features being added over time

High-Level Overview



Chapter Topics

Introduction to Impala and Hive | **Importing and Modeling Structured Data**

- Introduction to Impala and Hive
- **Why Use Impala and Hive?**
- Querying Data With Impala and Hive
- Comparing Hive to Traditional Databases
- Conclusion

Why Use Hive and Impala?

- **Brings large-scale data analysis to a broader audience**
  - No software development experience required
  - Leverage existing knowledge of SQL
- **More productive than writing MapReduce or Spark directly**
  - Five lines of HiveQL/Impala SQL might be equivalent to 200 lines or more of Java
- **Offers interoperability with other systems**
  - Extensible through Java and external scripts
  - Many business intelligence (BI) tools support Hive and/or Impala

Use Case: Log File Analytics

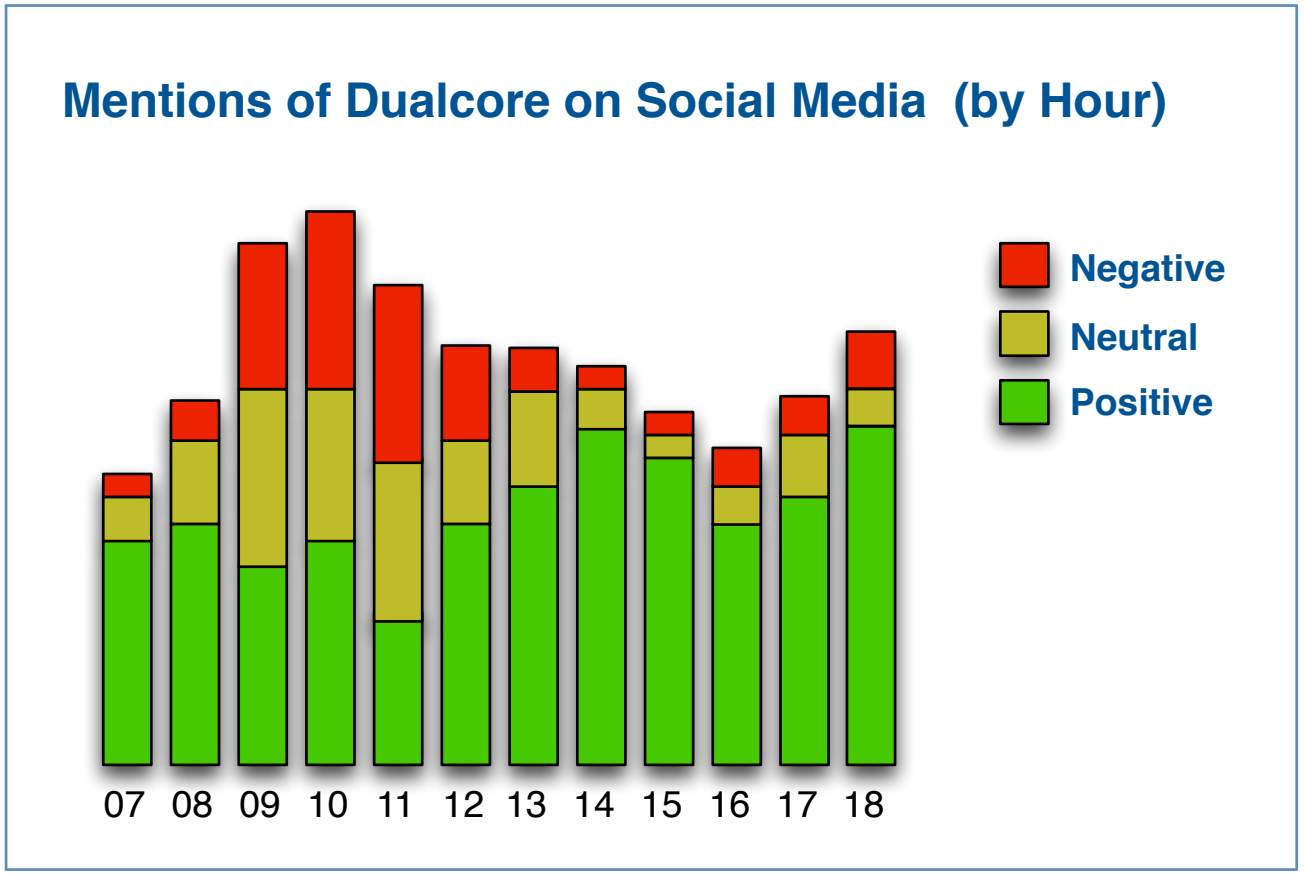
- **Server log files are an important source of data**
- **Hive and Impala allow you to treat a directory of log files like a table**
  - Allows SQL-like queries against raw data

Dualcore Inc. Public Web Site (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%

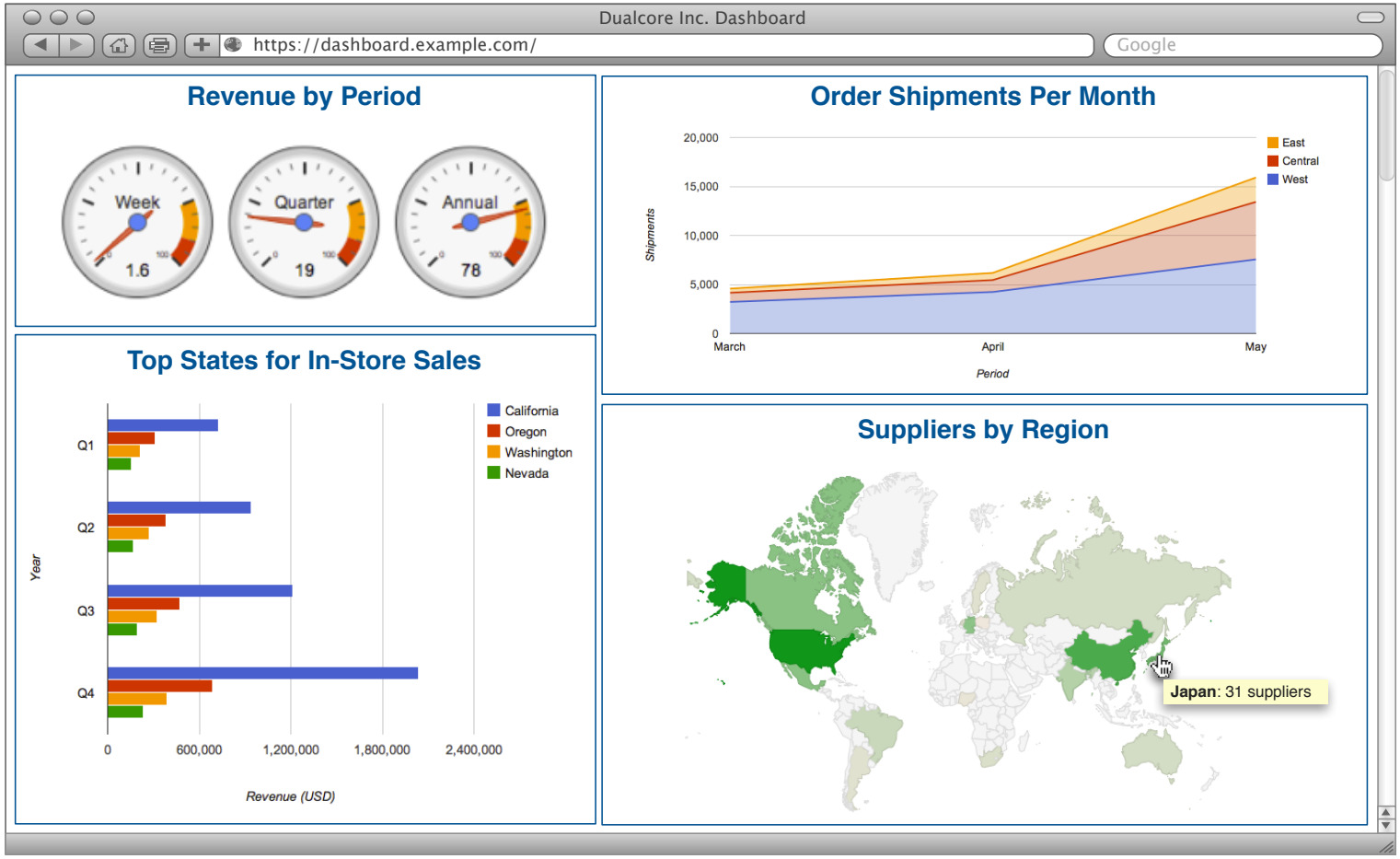
Use Case: Sentiment Analysis

- **Many organizations use Hive or Impala to analyze social media coverage**



Use Case: Business Intelligence

- **Many leading business intelligence tools support Hive and Impala**



Chapter Topics

Introduction to Impala and Hive | **Importing and Modeling Structured Data**

- Introduction to Impala and Hive
- Why Use Impala and Hive?
- **Querying Data With Hive and Impala**
- Comparing Hive to Traditional Databases
- Conclusion

Interacting with Hive and Impala

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

Starting the Impala Shell

- **You can execute statements in the Impala shell**
  - This interactive tool is similar to the shell in MySQL
- **Execute the `impala-shell` command to start the shell**
  - Some log messages truncated to better fit the slide

```
$ impala-shell
Connected to localhost.localdomain:21000
Server version: impalad version 2.1.0-cdh5 (…)
Welcome to the Impala shell.
[localhost.localdomain:21000] >
```

- **Use `-i hostname:port` option to connect to a different server**

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

Using the Impala Shell

- **Enter semicolon-terminated statements at the prompt**
  - Hit [Enter] to execute a query or command
  - Use the `quit` command to exit the shell
- **Use `impala-shell --help` for a full list of 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     |
…
| Falgoust   | Jennifer   |
+------------+------------+
Returned 50 row(s) in 0.17s

>
```

Note: shell prompt abbreviated as **>**

Interacting with the Operating System

- **Use `shell` to execute system commands from within Impala shell**

```
> shell date;
Mon May 20 16:44:35 PDT 2013
```

- **No direct support for HDFS commands**
  - But could run `hdfs dfs` using `shell`

```
> shell hdfs dfs -mkdir /reports/sales/2013;
```

Running Impala Queries from the Command Line

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

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

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

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

- **Use `-o` to capture output to file**
  - Optionally specify delimiter

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

Starting Beeline (Hive's Shell)

- **You can execute HiveQL statements in the Beeline shell**
  - Interactive shell based on the SQLLine utility
  - Similar to the Impala shell
- **Start Beeline by specifying the URL for a Hive2 server**
  - Plus username and password if required

```
$ beeline -u jdbc:hive2://host:10000 \
-n username -p password

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

Executing Queries in Beeline

- **SQL commands are terminated with semi-colon (;)**
- **Similar to Impala shell**
  - Results formatting is slightly different

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

+-------------+-------------+
|    lname    |    fname    |
+-------------+-------------+
| Ham         | Marilyn     |
| Franks      | Gerard      |
| Preston     | Mason       |
...
| Falgoust    | Jennifer    |
+-------------+-------------+
50 rows selected (15.829 seconds)

1: url>
```

Using Beeline

- **Execute Beeline commands with '!'**
  - No terminator character
- **Some commands**
  - **`!connect`** ***`url`*** – connect to a different Hive2 server
  - **`!exit`** – exit the shell
  - **`!help`** – show the full list of commands
  - **`!verbose`** – show added details of queries

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

Executing Hive Queries from the Command Line

- **You can also 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'
```

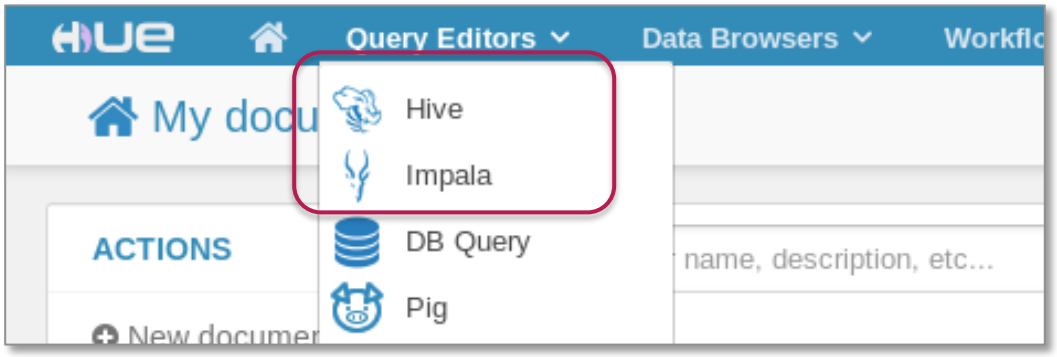
- **Use the `--silent` option to suppress informational messages**
  - Can also be used with `-e` or `-f` options

```
$ beeline -u ... --silent
```

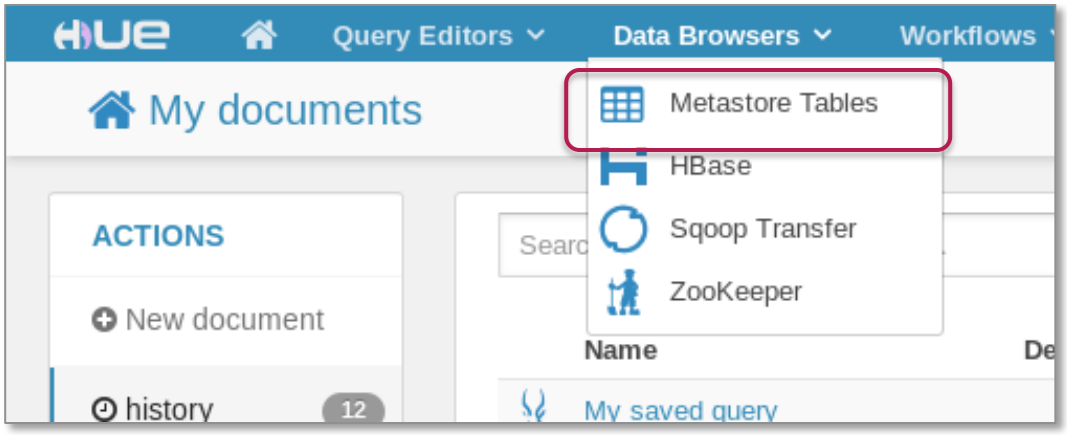
Using Hue with Hive and Impala

You can use Hue to…

Query data with Hive or Impala

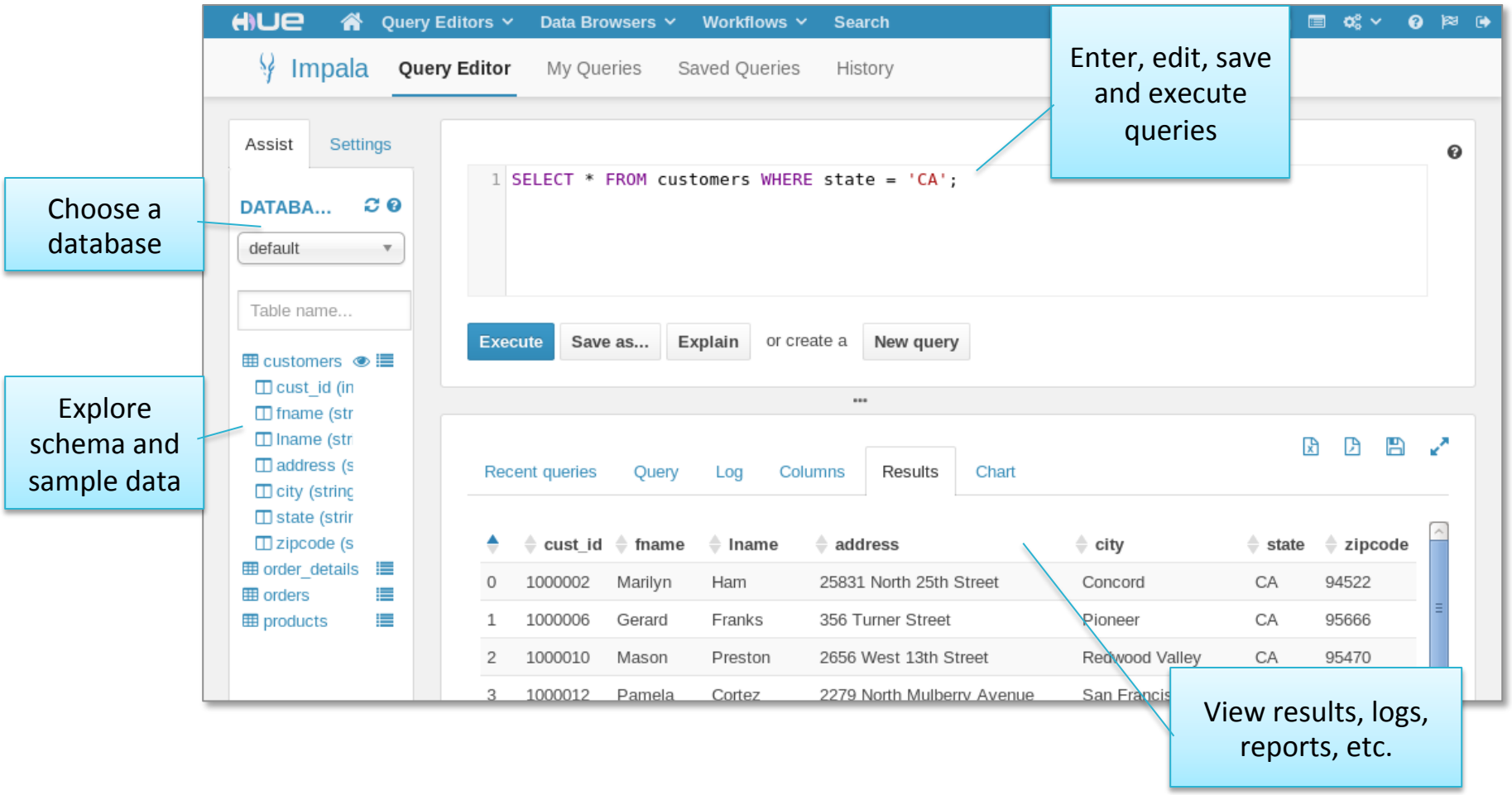


View and manage the Metastore



The Hue Query Editor

- **The Impala and Hive Query editors are nearly identical**



Chapter Topics

Introduction to Impala and Hive | **Importing and Modeling Structured Data**

- Introduction to Impala and Hive
- Why Use Impala and Hive?
- Querying Data With Impala and Hive
- **Comparing Hive and Impala to Traditional Databases**
- Conclusion

Your Cluster is Not a Database Server

- **Client-server database management systems have many strengths**
  - Very fast response time
  - Support for transactions
  - Allow modification of existing records
  - Can serve thousands of simultaneous clients
- **Your Hadoop cluster is not an RDBMS**
  - Hive generates processing engine jobs (MapReduce) from HiveQL queries
    - Limitations of HDFS and MapReduce still apply
  - Impala is faster but not intended for the throughput speed required for an OLTP database
  - No transaction support

Comparing Hive and Impala To A Relational Database

	Relational Database	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

Chapter Topics

Introduction to Impala and Hive | **Importing and Modeling Structured Data**

- Introduction to Impala and Hive
- Why Use Impala and Hive?
- Querying Data With Impala and Hive
- Comparing Hive and Impala to Traditional Databases
- **Conclusion**

Essential Points

- **Impala and Hive are tools for performing SQL queries on data in HDFS**
- **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 jobs on Hadoop based on HiveQL statements
- **Impala execute queries directly on the Hadoop cluster**
  - Uses a very fast specialized SQL engine, not MapReduce

Bibliography

The following offer more information on topics discussed in this chapter

- ***Cloudera Impala*** **(free O'Reilly book)**
  - `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`

cloudera®

Modeling and Managing Data with Impala and Hive

Chapter 6

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ **Modeling and Managing Data with Impala and Hive** ▪ Data Formats ▪ Data File Partitioning	**Importing and Modeling Structured Data**
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	Distributed Data Processing with Spark
▪ Conclusion	Course Conclusion

Modeling and Managing Data in Impala and Hive

In this chapter you will learn

- **How Impala and Hive use the Metastore**
- **How to use Impala SQL and HiveQL DDL to create tables**
- **How to create and manage tables using Hue or HCatalog**
- **How to load data into tables using Impala, Hive, or Sqoop**

Chapter Topics

Modeling and Managing Data With Impala and Hive | **Importing and Modeling Structured Data**

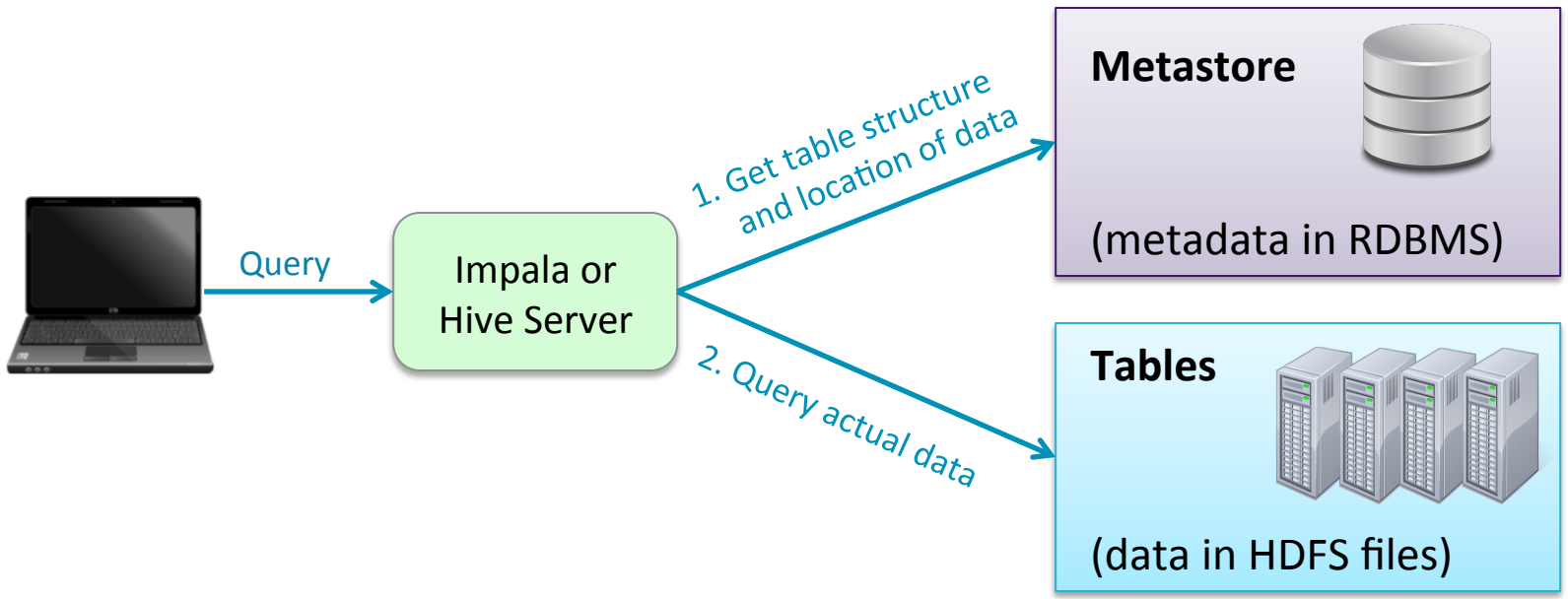
- **Data Storage Overview**
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

How Hive and Impala Load and Store Data (1)

- **Queries operate on tables, just like in an RDBMS**
  - A table is simply an HDFS directory containing one or more files
  - Default path: **`/user/hive/warehouse/<table_name>`**
  - Supports many formats for data storage and retrieval
- **What is the structure and location of tables?**
  - These are specified when tables are created
  - 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

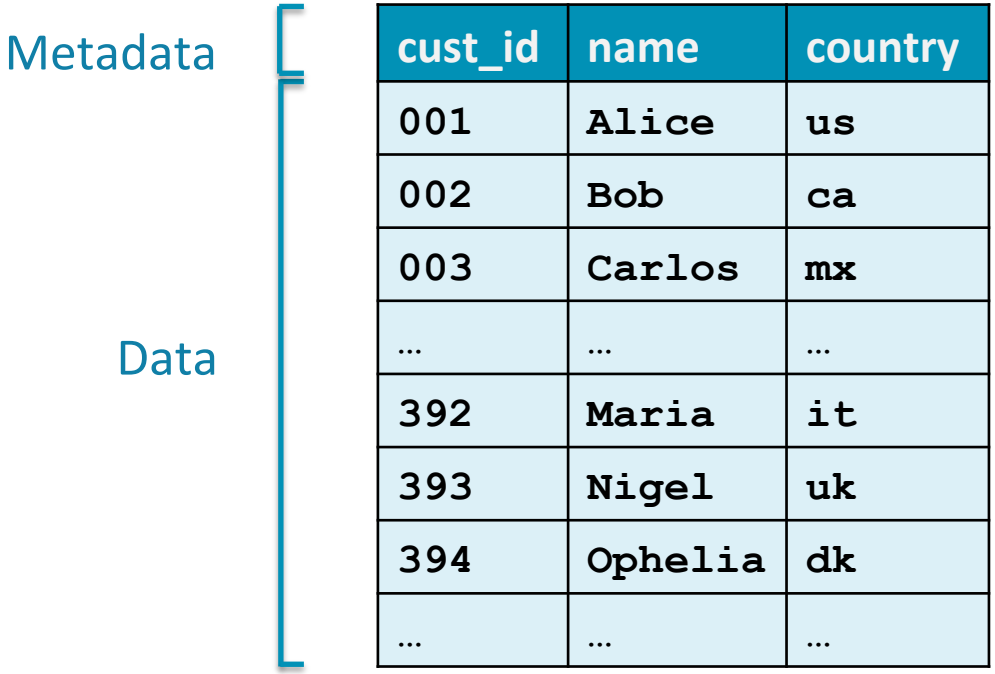
How Hive and Impala Load and Store Data (2)

- **Hive and Impala use the Metastore to determine data format and location**
  - The query itself operates on data stored in HDFS



Data and Metadata

- ***Data* refers to the information you store and process**
  - Billing records, sensor readings, and server logs are examples of data
- ***Metadata* describes important aspects of that data**
  - Field name and order are examples of metadata

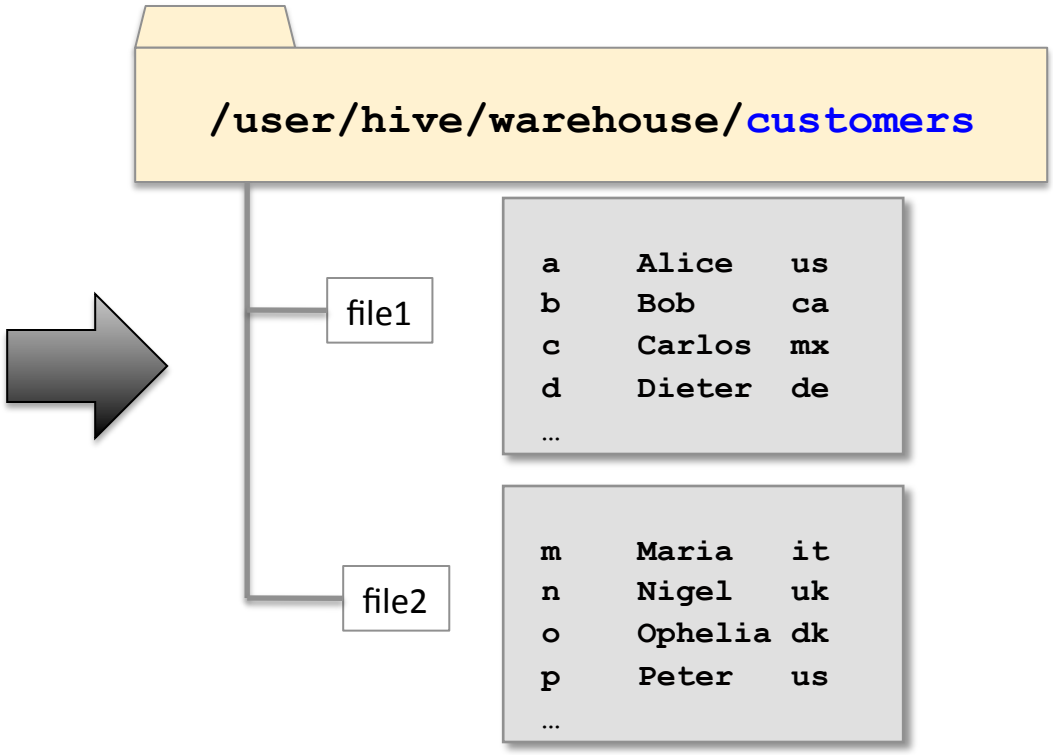


The Data Warehouse Directory

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

customers table

cust_id	name	country
001	Alice	us
002	Bob	ca
003	Carlos	mx
…	…	…
392	Maria	it
393	Nigel	uk
394	Ophelia	dk
…	…	…



Chapter Topics

Modeling and Managing Data With Impala and Hive | **Importing and Modeling Structured Data**

- Data Storage Overview
- **Creating Databases and Tables**
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

Defining Databases and Tables

- **Databases and tables are created and managed using the DDL (Data Definition Language) of HiveQL or Impala SQL**
  - Very similar to standard SQL DDL
  - Some minor differences between Hive and Impala DDL will be noted

Creating a Database

- **Hive and Impala databases are simply namespaces**
  - Helps to organize your tables
- **To create a new database**

```
CREATE DATABASE loudacre;
```

1. Adds the database definition to the metastore
2. Creates a storage directory in HDFS

   e.g. `/user/hive/warehouse/loudacre.db`

- **To conditionally create a new database**
  - Avoids error in case database already exists (useful for scripting)

```
CREATE DATABASE IF NOT EXISTS loudacre;
```

Removing a Database

- **Removing a database is similar to creating it**
  - Just replace **CREATE** with **DROP**

```
DROP DATABASE loudacre;
```

```
DROP DATABASE IF EXISTS loudacre;
```

- **These commands will fail if the database contains tables**
  - In Hive: Add the **CASCADE** keyword to force removal
  - Caution: this command might remove data in HDFS!


```
DROP DATABASE loudacre CASCADE;
```

Data Types

- **Each column is assigned a specific data type**
  - These are specified when the table is created
  - `NULL` values are returned for non-conforming data in HDFS
- **Here are some common data types**

Name	Description	Example Value
`STRING`	Character data (of any length)	**`Alice`**
`BOOLEAN`	**`TRUE`** or **`FALSE`**	**`TRUE`**
`TIMESTAMP`	Instant in time	**`2014-03-14 17:01:29`**
`INT`	Range: same as Java **`int`**	**`84127213`**
`BIGINT`	Range: same as Java **`long`**	**`76132929365142153l7`**
`FLOAT`	Range: same as Java **`float`**	**`3.14159`**
`DOUBLE`	Range: same as Java **`double`**	**`3.141592653589793238 5`**

Hive (not Impala) also supports a few complex types such as maps and arrays

Creating a Table (1)

- **Basic syntax for creating a table:**

```
CREATE TABLE 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 tablename (colname DATATYPE, ...)
  ROW FORMAT DELIMITED
  FIELD
  STORE
```

Specify a name for the table, and list the column names and datatypes (see later)

Creating a Table (3)

```
CREATE TABLE tablename (colname DATATYPE, ...)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY char
  STORE
```

This line states that fields in each file in the table's directory are delimited by some character. The default delimiter is Control-A, but you may specify an alternate delimiter...

Creating a Table (4)

```
CREATE TABLE 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 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.
Other formats will be discussed later in the course.

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,100000,2013-06-21 15:52:03
```

Creating Tables Based On Existing Schema

- **Use `LIKE` to create a new table based on an existing table definition**

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

- **Column definitions and names are derived from the existing table**
  - New table will contain no data

Creating Tables Based On Existing Data

- **Create a table based on a `SELECT` statement**
  - Often know as 'Create Table As Select' (CTAS)

```
CREATE TABLE ny_customers 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
- **New table will contain the selected data**

Controlling Table Data Location

- **By default, table data is stored in the warehouse directory**
- **This is not always ideal**
  - Data might be shared by several users
- **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 '/loudacre/jobs';
```

Externally Managed Tables

- **CAUTION: Dropping a table removes its data in HDFS**
  - Tables are "managed" or "internal" by default
- **Using `EXTERNAL` when creating the table avoids this behavior**
  - Dropping an *external* 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 '/loudacre/ad_data';
```

Exploring Tables (1)

- The `SHOW TABLES` command lists all tables in the current database

```
SHOW TABLES;
+---------------+
|   tab_name    |
+---------------+
| accounts      |
| employees     |
| job           |
| vendors       |
+---------------+
```

- The `DESCRIBE` command lists the fields in the specified table

```
DESCRIBE jobs;
+--------+-----------+---------+
| name   | type      | comment |
+--------+-----------+---------+
| id     | int       |         |
| title  | string    |         |
| salary | int       |         |
| posted | timestamp |         |
+--------+-----------+---------+
```

Exploring Tables (2)

- **DESCRIBE FORMATTED also shows table properties**

```
DESCRIBE FORMATTED jobs;
+------------------+----------------------------------------+---------+
| name             | type                                   | comment |
+------------------+----------------------------------------+---------+
| # col_name       | data_type                              | comment |
| id               | int                                    | NULL    |
| title            | string                                 | NULL    |
| salary           | int                                    | NULL    |
| posted           | timestamp                              | NULL    |
|                  | NULL                                   | NULL    |
| # Detailed Table | NULL                                   | NULL    |
| Information
| Database:        | default                                | NULL    |
| Owner:           | training                               | NULL    |
| CreateTime:      | Wed Jun 17 09:41:23 PDT 2015           | NULL    |
| LastAccessTime:  | UNKNOWN                                | NULL    |
| Protect Mode:    | None                                   | NULL    |
| Retention:       | 0                                      | NULL    |
| Location:        | hdfs://localhost:8020/loudacre/jobs    | NULL    |
| Table Type:      | MANAGED_TABLE                          | NULL    |
…
```

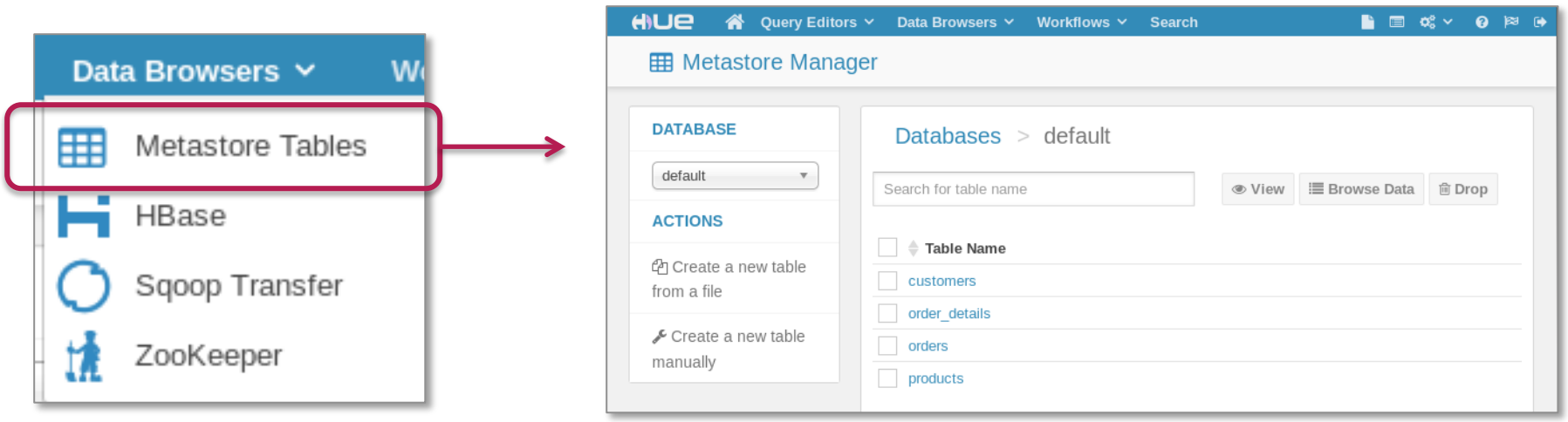
Exploring Tables (3)

- **SHOW CREATE TABLE displays the SQL command to create the table**

```
SHOW CREATE TABLE jobs;
+----------------------------------------------------+
| CREATE TABLE default.jobs                          |
|   id INT,                                          |
|   title STRING,                                    |
|   salary INT,                                      |
|   posted TIMESTAMP                                 |
| )                                                  |
| ROW FORMAT DELIMITED FIELDS TERMINATED BY ','      |
…
```

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
    - Not all features are supported yet



Chapter Topics

Modeling and Managing Data With Impala and Hive | **Importing and Modeling Structured Data**

- Data Storage Overview
- Creating Databases and Tables
- **Loading Data into Tables**
- HCatalog
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

Data Validation

- **Impala and Hive 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 \
  /tmp/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 '/tmp/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 '/tmp/sales.txt'
  OVERWRITE INTO TABLE sales;
```

Appending Selected Records to a Table

- **Another way to populate a table is through a query**
  - Use **`INSERT INTO`** to add results to an *existing* Hive table

```
INSERT INTO TABLE accounts_copy
  SELECT * FROM accounts;
```

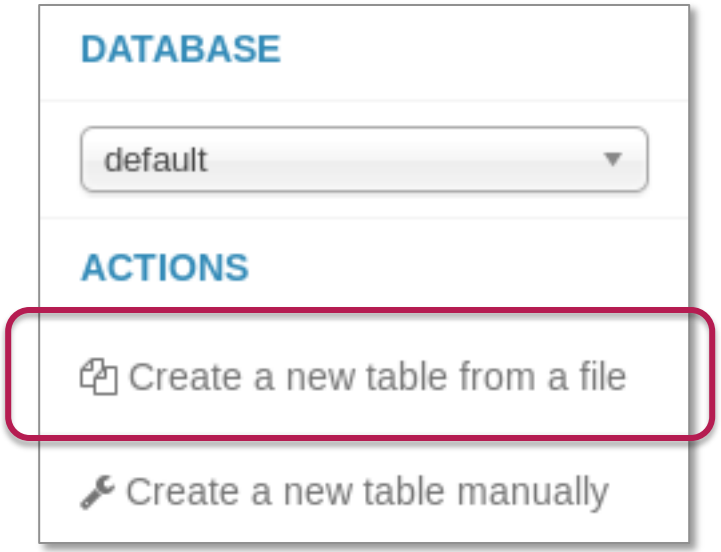
  - Specify a **`WHERE`** clause to control which records are appended

```
INSERT INTO TABLE loyal_customers
  SELECT * FROM accounts
  WHERE YEAR(acct_create_dt) = 2008
    AND acct_close_dt IS NULL;
```

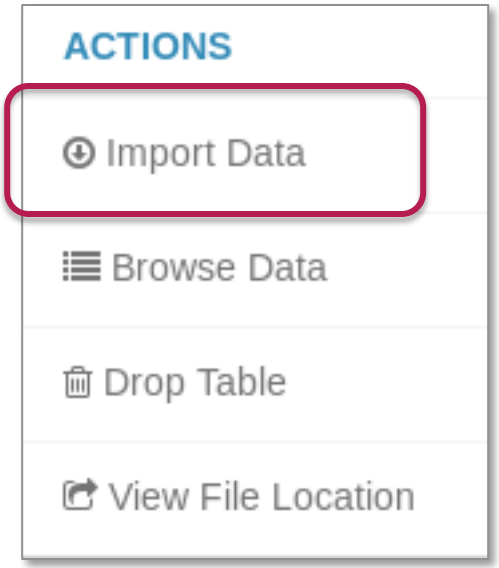
Loading Data Using the Metastore Manager

- **The Metastore Manager provides two ways to load data into a table**

Table creation wizard



Import data wizard



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 Hive metastore
  - Imports data from the RDBMS to the table's directory in HDFS

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

  - Note that **`--hive-import`** creates a table accessible in both Hive and Impala

Chapter Topics

Modeling and Managing Data With Impala and Hive | **Importing and Modeling Structured Data**

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- **HCatalog**
- Impala Metadata Caching
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

The Metastore and HCatalog

- **HCatalog is a Hive sub-project that provides access to the Metastore**
  - Accessible via command line and REST API
  - Allows you to define tables using HiveQL DDL syntax
  - Access those tables from Hive, Impala, MapReduce, Pig, and other tools
  - Included with CDH 4.2 and later

Creating Tables in HCatalog

- **HCatalog uses Hive's DDL (data definition language) syntax**
  - You can specify a single command using the `-e` option

```
$ hcat -e "CREATE TABLE vendors \
      (id INT, company STRING, email STRING) \
      ROW FORMAT DELIMITED FIELDS TERMINATED BY ',' \
      LOCATION '/dualcore/vendors'"
```

- **Tip: save longer commands to a text file and use the `-f` option**
  - If the file has more than one command, separate each with a semicolon

```
$ hcat -f createtable.txt
```

Displaying Metadata in HCatalog

- **The `SHOW TABLES` command also shows tables created directly in Hive**

```
$ hcat -e 'SHOW TABLES'
employees
vendors
```

- **The `DESCRIBE` command lists the fields in a specified table**
  - Use `DESCRIBE FORMATTED` instead to see detailed information

```
$ hcat -e 'DESCRIBE vendors'
id          int
company     string
email       string
```

Removing a Table in HCatalog

- **The `DROP TABLE` command has the same behavior as it does in Hive and Impala**
  - Caution: this will remove the data as well as the metadata (unless table is **`EXTERNAL`**)!

```
$ hcat -e 'DROP TABLE vendors'
```

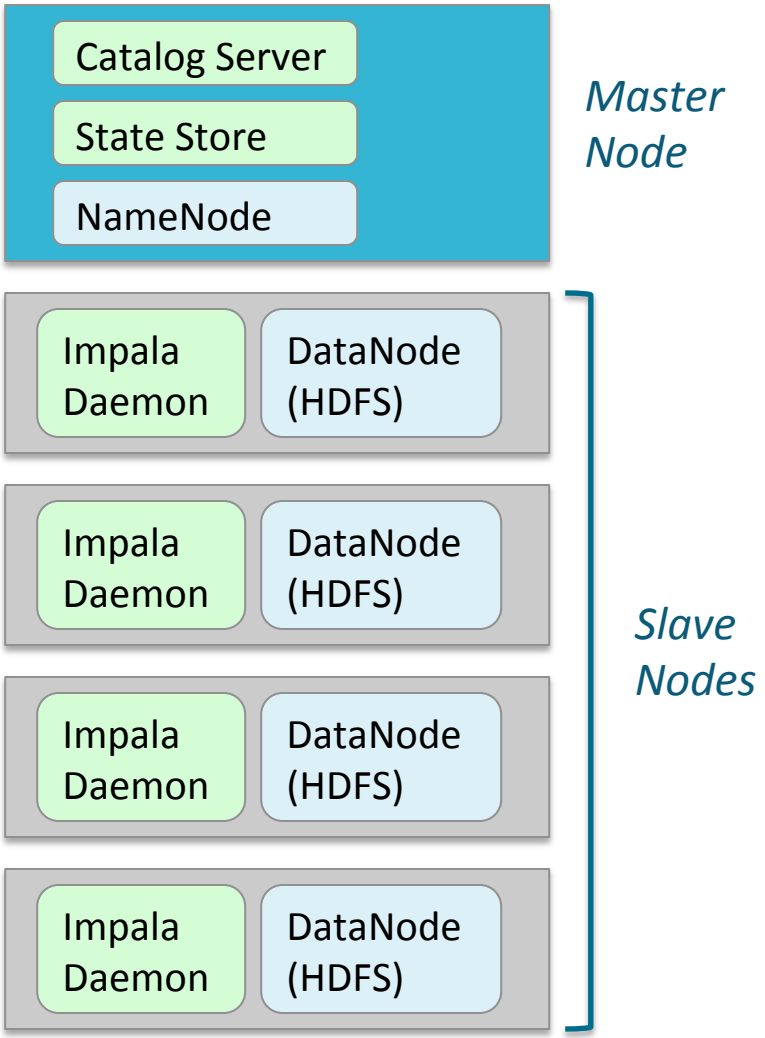
Chapter Topics

Modeling and Managing Data With Impala and Hive | **Importing and Modeling Structured Data**

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- **Impala Metadata Caching**
- Conclusion
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

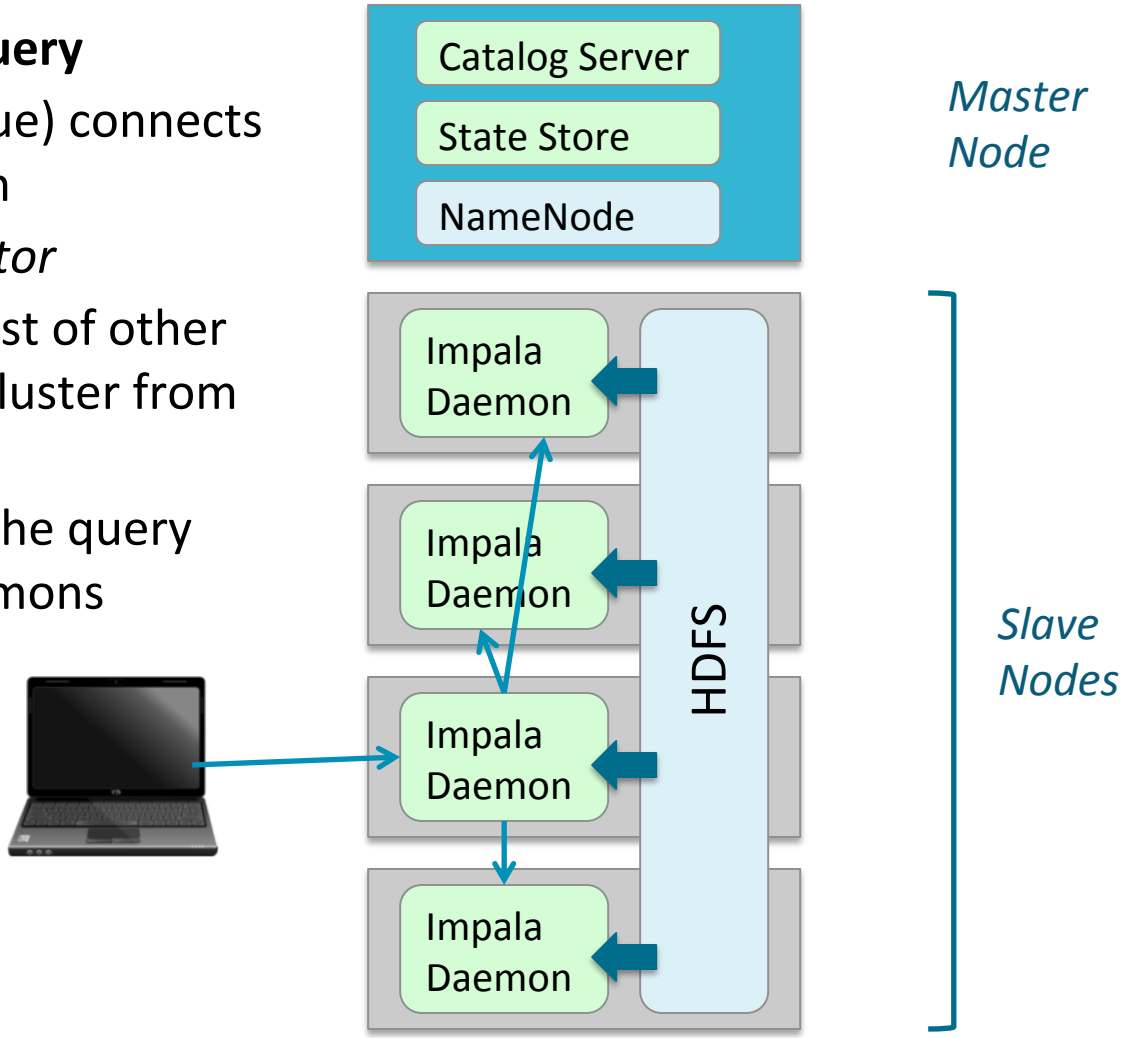
Impala in the Cluster

- **Each slave node in the cluster runs an Impala daemon**
  - Co-located with the HDFS slave daemon (DataNode)
- **Two other daemons running on master nodes support query execution**
  - The **State Store** daemon
    - Provides lookup service for Impala daemons
    - Periodically checks status of Impala daemons
  - The **Catalog** daemon
    - Relays metadata changes to all the Impala daemons in a cluster



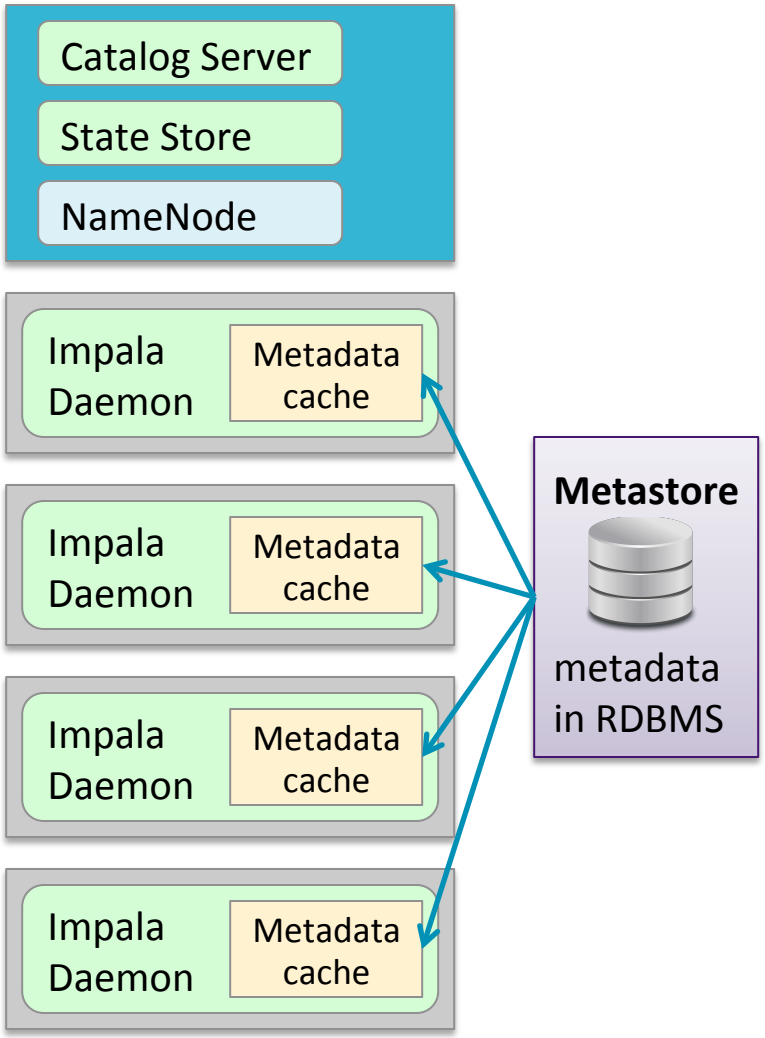
How Impala Executes a Query

- **Impala daemon plans the query**
  - Client (impala-shell or Hue) connects to a local impala daemon
    - This is the *coordinator*
  - Coordinator requests a list of other Impala daemons in the cluster from the State Store
  - Coordinator distributes the query across other Impala daemons
  - Streams results to client



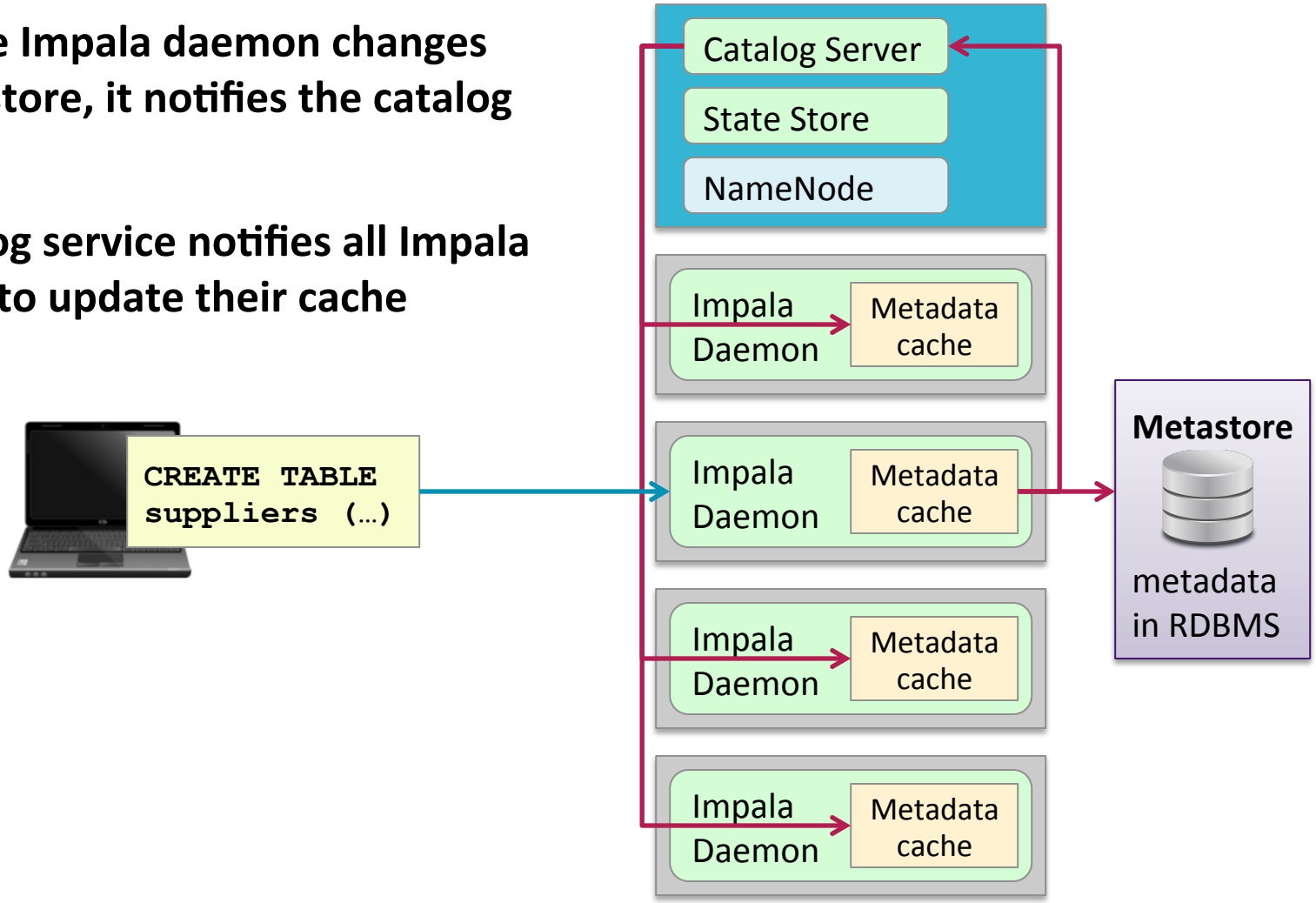
Metadata Caching (1)

- **Impala daemons cache metadata**
  - The tables' schema definitions
  - The locations of tables' HDFS blocks
- **Metadata is cached from the Metastore at startup**



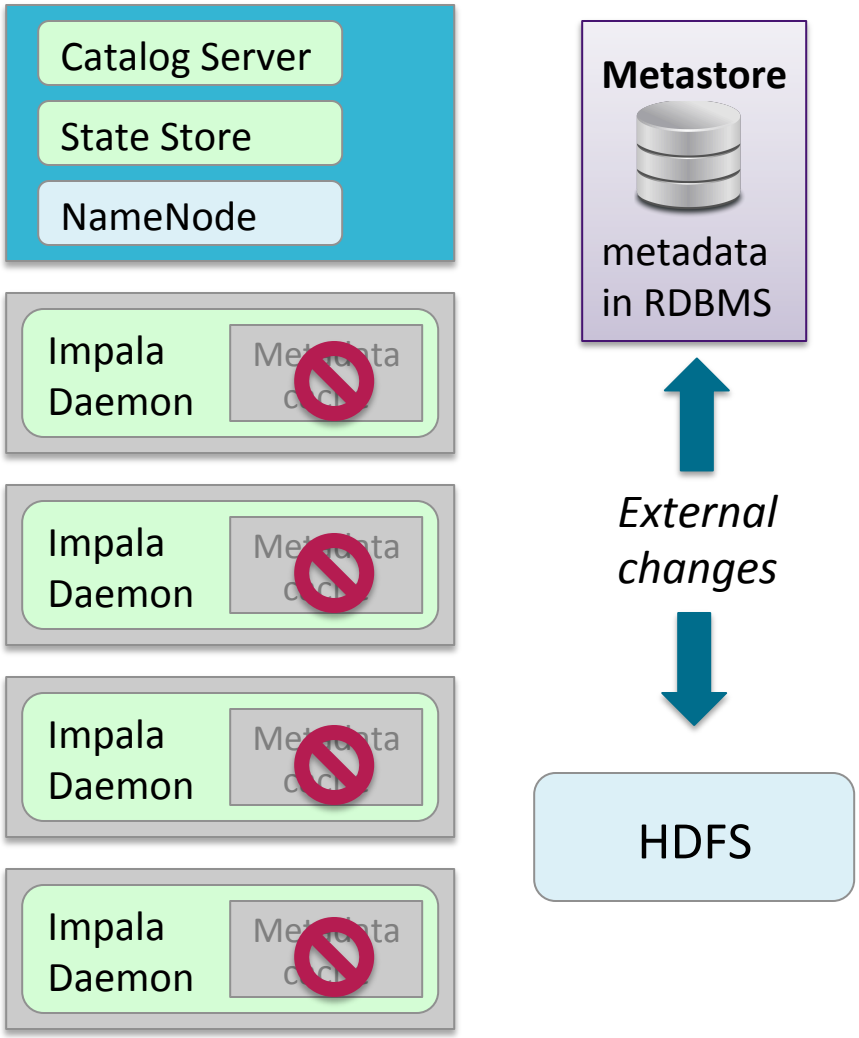
Metadata Caching (2)

- **When one Impala daemon changes the metastore, it notifies the catalog service**
- **The catalog service notifies all Impala daemons to update their cache**



External Changes and Metadata Caching

- **Metadata updates made *from outside of Impala* are not known to Impala, e.g.**
  - Changes via Hive, HCatalog or Hue Metadata Manager
  - Data added directly to directory in HDFS
- **Therefore the Impala metadata caches will be invalid**
- **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 cache as stale; metadata cache is reloaded as needed.
Table schema modified *or* New data added to a table	**`REFRESH <table>`**	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 <table>`**	Marks the metadata for a single table as stale. When the metadata is needed, all HDFS block locations are retrieved.

Chapter Topics

Modeling and Managing Data With Impala and Hive | **Importing and Modeling Structured Data**

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- **Conclusion**
- Hands-On Exercise: Create and Populate Tables in Impala or Hive

Essential Points

- **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 Metastore stores data *about* the data in an RDBMS**
  - E.g. Location, column names and types
- **Tables are created and managed using the Impala SQL or HiveQL Data Definition Language**
- **Impala caches metadata from the Metastore**
  - Invalidate or refresh the cache if tables are modified outside Impala
- **HCatalog provides access to the Metastore from tools outside Hive or Impala (e.g. Pig, MapReduce)**

Bibliography

The following offer more information on topics discussed in this chapter

- **Impala Concepts and Architecture**
  - http://tiny.cloudera.com/adcc12a
- **Impala SQL Language Reference**
  - http://tiny.cloudera.com/impalasql
- **Impala-related Articles on Cloudera's Blog**
  - http://tiny.cloudera.com/adcc12e
- **Apache Hive Web Site**
  - http://hive.apache.org/
- **HiveQL Language Manual**
  - http://tiny.cloudera.com/adcc10b

Chapter Topics

Modeling and Managing Data With Impala and Hive | **Importing and Modeling Structured Data**

- Data Storage Overview
- Creating Databases and Tables
- Loading Data into Tables
- HCatalog
- Impala Metadata Caching
- Conclusion
- **Hands-On Exercise: Create and Populate Tables in Impala or Hive**

Hands-On Exercise: Create and Populate Tables in Impala

- **In this exercise you will**
  - Create a table in Impala to model and view existing data
  - Use Sqoop to create a new table automatically from data imported from MySQL
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®

Data Formats

Chapter 7

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ **Data Formats** ▪ Data File Partitioning	**Importing and Modeling Structured Data**
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	Distributed Data Processing with Spark
▪ Conclusion	Course Conclusion

Data Formats

In this chapter you will learn

- **How to select the best data format for your needs**
- **How various Hadoop tools support different data formats**
- **How to define Avro schemas**
- **How Avro schemas can evolve to accommodate changing requirements**
- **How to extract data and metadata from an Avro data file**

Chapter Topics

Data Formats | **Importing and Modeling Structured Data**

- **File Formats**
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- Using Parquet with Impala, Hive and Sqoop
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

File Storage Formats

- **In previous chapters you saw that alternate file formats were available**
  - E.g., in Hive and Impala

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

  - E.g., in Sqoop

```
--as-format
```

- **What formats are available?**
- **Which should you choose and why?**

Hadoop File Formats: Text Files

- **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, since everything is a string**
  - 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**

Hadoop File Formats: Sequence Files

- **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**

Hadoop File Formats: Avro Data Files

- **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**
  - Can read and write from many languages
  - 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
- ***More detail in coming slides***

Columnar Formats

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

id	name	city	occupation	income	phone
1	Alice	Palo Alto	Accountant	85000	650-555-9748
2	Bob	Sunnyvale	Accountant	81500	650-555-8865
3	Bob	Palo Alto	Dentist	196000	650-555-7185
4	Bob	Palo Alto	Manager	87000	650-555-2518
5	Carol	Palo Alto	Manager	79000	650-555-3951
6	David	Sunnyvale	Mechanic	62000	650-555-4754

Organization of data in traditional row-based formats

id	name	city	occupation	income	phone
1	Alice	Palo Alto	Accountant	85000	650-555-9748
2	Bob	Sunnyvale	Accountant	81500	650-555-8865
3	Bob	Palo Alto	Dentist	196000	650-555-7185
4	Bob	Palo Alto	Manager	87000	650-555-2518
5	Carol	Palo Alto	Manager	79000	650-555-3951
6	David	Sunnyvale	Mechanic	62000	650-555-4754

Organization of data in columnar formats

Hadoop File Formats: Parquet Files

- **Parquet is a columnar format developed by Cloudera and Twitter**
  - Supported in Spark, MapReduce, Hive, Pig, Impala, Crunch, and others
  - Schema metadata 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

- Selecting a File Format
- Hadoop Tool Support for File Formats
- **Avro Schemas**
- Using Avro with Hive and Sqoop
- Avro Schema Evolution
- Compression
- Conclusion
- Hands-On Exercise: Working with Avro Schemas

Data Serialization

- **To understand Avro, you must first understand *serialization***
  - A way of representing data in memory as a series of bytes
  - Allows you to save data to disk or send it across the network
  - *Deserialization* allows you to read that data back into memory
- **For example, how do you serialize the number 108125150?**
  - 4 bytes when stored as a Java `int`
  - 9 bytes when stored as a Java `String`
- **Many programming languages and libraries support serialization**
  - Such as `Serializable` in Java or `pickle` in Python
- **Backwards compatibility and cross-language support can be challenging**
  - Avro was developed to address these challenges

What is Apache Avro?

- **Avro Data File Format is just one part of the Avro project**
  - But it is the part this course focuses on
- **Avro is an efficient data serialization framework**
  - Top-level Apache project created by Doug Cutting (creator of Hadoop)
  - Widely supported throughout Hadoop and its ecosystem
- **Offers compatibility without sacrificing performance**
  - Data is serialized according to a *schema* you define
  - Read/write data in Java, C, C++, C#, Python, PHP, and other languages
  - Serializes data using a highly-optimized binary encoding
  - Specifies rules for *evolving* your schema over time
- **Avro also supports Remote Procedure Calls (RPC)**
  - Can be used for building custom network protocols
  - Flume uses this for internal communication

Supported Types in Avro Schemas (Simple)

- **A simple type holds exactly one value**

Name	Description	Java Equivalent
`null`	An absence of a value	`null`
`boolean`	A binary value	`boolean`
`int`	32-bit signed integer	`int`
`long`	64-bit signed integer	`long`
`float`	Single-precision floating point value	`float`
`double`	Double-precision floating point value	`double`
`bytes`	Sequence of 8-bit unsigned bytes	`java.nio.ByteBuffer`
`string`	Sequence of Unicode characters	`java.lang.CharSequence`

Supported Types in Avro Schemas (Complex)

- **Avro also supports complex types**

Name	Description
`record`	A user-defined type composed of one or more named fields
`enum`	A specified set of values
`array`	Zero or more values of the same type
`map`	Set of key-value pairs; key is string while value is of specified type
`union`	Exactly one value matching a specified set of types
`fixed`	A fixed number of 8-bit unsigned bytes

- **The `record` type is the most important**
  - Main use of other types is to define a record's fields

Basic Schema Example

- **Excerpt from a SQL `CREATE TABLE` statement**

```
CREATE TABLE employees
    (id INT, name STRING, title STRING, bonus INT)
```

- **Equivalent Avro schema**

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"Employee",
 "fields":[
     {"name":"id", "type":"int"},
     {"name":"name", "type":"string"},
     {"name":"title", "type":"string"},
     {"name":"bonus", "type":"int"}]
}
```

Specifying Default Values in the Schema

- **Avro also supports setting a default value in the schema**
  - Used when no value was explicitly set for a field
  - Similar to SQL

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"Invoice",
 "fields":[
    {"name":"id", "type":"int"},
    {"name":"taxcode", "type":"int", "default":"39"},
    {"name":"lang", "type":"string", "default":"EN_US"}]
}
```

The **taxcode** and **lang** fields have default values

Avro Schemas and Null Values

- **Avro checks for null values when serializing the data**
- **Null values are only allowed *when explicitly specified* in the schema**

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"Employee",
 "fields":[
    {"name":"id", "type":"int"},
    {"name":"name", "type":"string"},
    {"name":"title", "type":["null","string"]},
    {"name":"bonus", "type":["null","int"]}]
}
```

The **title** and **bonus** fields allow null values

Schema Example with Complex Types

- **The following example shows a record with an enum and a string array**

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"CustomerServiceTicket",
 "fields":[
    {"name":"id", "type":"int"},
    {"name":"agent", "type":"string"},
    {"name":"category", "type":{
          "name":"CSCategory", "type":"enum",
          "symbols":["Order","Shipping","Device"]}
    },
    {"name":"tags", "type":{
          "type":"array", "items":"string"}
    }]
}
```

The **category** field has three enumerated possible values

tags is an array of strings

Documenting Your Schema

- **It's a good practice to document any ambiguities in a schema**
  - All types (including record) support an optional `doc` attribute

```
{"namespace":"com.loudacre.data",
 "type":"record",
 "name":"WebProduct",
 "doc":"Item currently sold in Loudacre's online store",
 "fields":[
    {"name":"id", "type":"int", "doc":"Product SKU"},
    {"name":"shipwt", "type":"int",
          "doc": "Shipping weight, in pounds"},
    {"name":"price", "type":"int",
          "doc":"Retail price, in cents (US)"}]
}
```

Avro Container Format

- **Avro also defines a container file format for storing Avro records**
  - Also known as "Avro data file format"
  - Similar to Hadoop `SequenceFile` format
  - Cross-language support for reading and writing data
- **Supports compressing *blocks* (groups) of records**
  - It is "splittable" for efficient processing in Hadoop
- **This format is self-describing**
  - Each file contains a copy of the schema used to write its data
  - All records in a file must use the same schema

Inspecting Avro Data Files with Avro Tools

- **Avro data files are an efficient way to store data**
  - However, the binary format makes debugging difficult
- **Each Avro release contains an Avro Tools JAR file**
  - Allows you to read the schema or data for an Avro file
  - Included with CDH 5 and later
  - Available for download from the Avro Web site or Maven repository

```
$ java -jar avro-tools*.jar tojson mydatafile.avro
{"name":"Alice","salary": 56500,"city":"Anaheim"}
{"name":"Bob","salary": 51400,"city":"Bellevue"}

$ java -jar avro-tools*.jar getschema mydatafile.avro
{
  "type" : "record",
  "name" : "DeviceData",
  "namespace" : "com.loudacre.data",   ...rest of schema follows
```

Chapter Topics

- Selecting a File Format
- Hadoop Tool Support for File Formats
- Avro Schemas
- Using Avro with Impala, Hive and Sqoop
- **Avro Schema Evolution**
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Schema Evolution

- **The structure of your data will change over time**
  - Fields may be added, removed, changed, or renamed
  - In SQL, these are handled with **`ALTER TABLE`** statements
- **These changes can break compatibility with many formats**
  - Objects serialized in **`SequenceFile`**s become unreadable
- **Data written to Avro data files is always readable**
  - The schema used to write the data is embedded in the file itself
  - However, an application reading data might expect the *new* structure
- **Avro has a unique approach to maintaining forward compatibility**
  - A reader can use a different schema than the writer

Schema Evolution: A Practical Example (1)

- **Imagine that we have written millions of records with this schema**

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name", "type": "string"},
     {"name": "faxNumber", "type": "string"}
]}
```

Schema Evolution: A Practical Example (2)

- **We would like to modernize this based on the schema below**
  - Rename `id` field to `customerId` and change type from `int` to `long`
  - Remove `faxNumber` field
  - Add `prefLang` field
  - Add `email` field

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
     {"name": "customerId", "type": "long"},
     {"name": "name", "type": "string"},
     {"name": "prefLang", "type": "string”},
     {"name": "email", "type": "string"}
]}
```

Schema Evolution: A Practical Example (3)

- **We could use the new schema to write new data**
  - Applications that use the new schema could read the new data
- **Unfortunately, new applications wouldn't be able to read the *old* data**
  - We must make a few schema changes to improve compatibility

Schema Evolution: A Practical Example (4)

- **If you rename a field, you must specify an alias for the old name(s)**
  - Here, we map the old `id` field to the new `customerId` field

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
    {"name":"customerId", "type":"long",
         "aliases":["id"]},
    {"name":"name", "type":"string"},
    {"name":"prefLang", "type":"string"},
    {"name":"email", "type":"string"}
]}
```

Schema Evolution: A Practical Example (5)

- **Newly-added fields will lack values for records previously written**
  - You must specify a default value

```
{"namespace": "com.loudacre.data",
 "type": "record",
 "name": "CustomerContact",
 "fields": [
    {"name":"customerId", "type":"long",
         "aliases":["id"]},
    {"name":"name", "type":"string"},
    {"name":"prefLang", "type":"string",
         "default":"en_US"},
    {"name":"email",
         "type":["null","string"],"default":null}
]}
```

Default value for **prefLang** is **en_US**

email is nullable so **null** can be the default

Schema Evolution: Compatible Changes

- **The following changes will not affect existing readers**
  - Adding, changing, or removing a `doc` attribute
  - Changing a field's default value
  - Adding a new field with a default value
  - Removing a field that specified a default value
  - Promoting a field to a wider type (e.g., `int` to `long`)
  - Adding aliases for a field

Schema Evolution: Incompatible Changes

- **The following are some changes that might break compatibility**
  - Changing the record's `name` or `namespace` attributes
  - Adding a new field without a default value
  - Removing a symbol from an `enum`
  - Removing a type from a `union`
  - Modifying a field's type to one that could result in truncation
- **To handle these incompatibilities**
  1. Read your old data (using the original schema)
  2. Modify data as needed in your application
  3. Write the new data (using the new schema)
- **Existing readers/writers may need to be updated to use new schema**

Chapter Topics

- Selecting a File Format
- Hadoop Tool Support for File Formats
- Avro Schemas
- Avro Schema Evolution
- **Using Avro with Impala, Hive and Sqoop**
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Using Avro with Sqoop

- **Sqoop supports importing data as Avro, or exporting data from existing Avro data files**
- **`sqoop import` saves the schema JSON file in local directory**

```
$ sqoop import \
 --connect jdbc:mysql://localhost/loudacre \
 --username training --password training \
 --table accounts \
 --target-dir /loudacre/accounts_avro \
--as-avrodatafile
```

Using Avro with Impala and Hive (1)

- **Hive and Impala support Avro**
  - Hive supports all Avro types
  - Impala does not support complex types
    - **`enum`**, **`array`**, etc.

Using Avro with Impala and Hive (2)

- **Table creation can include schema inline or in a separate file**

```
CREATE TABLE order_details_avro
  STORED AS AVRO
  TBLPROPERTIES ('avro.schema.url'=
    'hdfs://localhost/loudacre/accounts_schema.json');
```

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

Chapter Topics

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- **Using Parquet with Impala, Hive and Sqoop**
- Compression
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Using Parquet with Sqoop

- **Sqoop supports importing data as Parquet, or exporting data from existing Parquet data files**
  - Use the **`--as-parquetfile`** option

```
$ sqoop import \
 --connect jdbc:mysql://localhost/loudacre \
 --username training --password training \
 --table accounts \
 --target-dir /loudacre/accounts_parquet \
 --as-parquetfile
```

Using Parquet with Hive and Impala (1)

- **Create a new table stored in Parquet format**

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

* **STORED AS PARQUET** supported in Impala, and in Hive 0.13 and later

Using Parquet with Hive and Impala (2)

- **In Impala, use `LIKE PARQUET` to use column metadata from an existing Parquet data file**
- **Example: Create a new table to access existing Parquet format data**

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

Chapter Topics

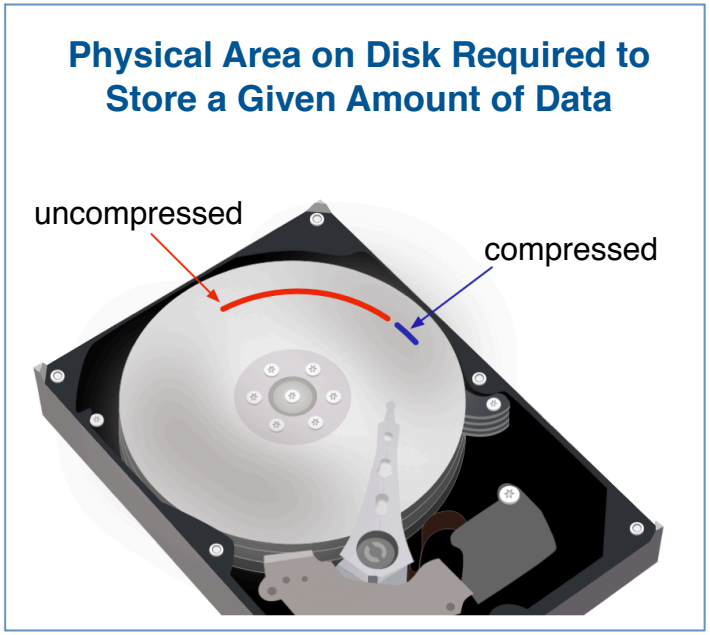
- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- Using Parquet with Impala, Hive and Sqoop
- **Compression**
- Conclusion
- Hands-On Exercise: Select a Format for a Data File

Performance Considerations

- **You have just learned how different file formats affect performance**
- **Another factor can significantly affect performance: data compression**

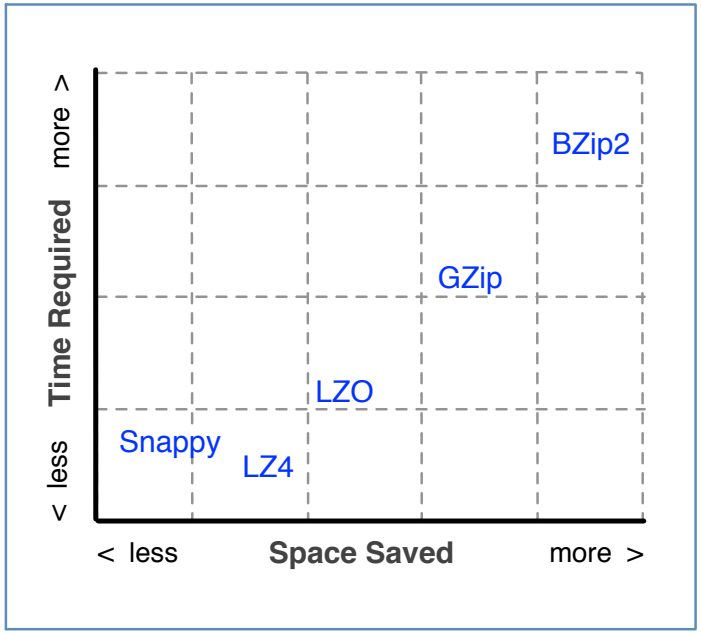
Data Compression

- **Each file format may also support compression**
  - This reduces amount of disk space required to store data
- **Compression is a tradeoff between CPU time and bandwidth/storage space**
  - Aggressive algorithms take a long time, but save more space
  - Less aggressive algorithms save less space but are much faster
- **Can significantly improve performance**
  - Many Hadoop jobs are I/O-bound
  - Using compression allows you to handle more data per I/O operation
  - Compression can also improve the performance of network transfers



Compression Codecs

- **The implementation of a compression algorithm is known as a *codec***
  - Short for compressor/decompressor
- **Many codecs are commonly used with Hadoop**
  - Each has different performance characteristics
  - Not all Hadoop tools are compatible with all codecs
- **Overall, BZip2 saves the most space**
  - But LZ4 and Snappy are much faster
  - Impala supports Snappy but not LZ4
- **For "hot" data, speed matters most**
  - Better to compress by 40% in one second than by 80% in 10 seconds



Using Compression With Sqoop

- **Sqoop – use `--compression-codec` flag**
  - Example

```
--compression-codec
org.apache.hadoop.io.compress.SnappyCodec
```

Using Compression With Impala and Hive

- **Not all file format/compression combinations are supported**
  - Properties and syntax varies
  - See the documentation for a full list of supported formats and codecs for Impala and Hive
  - Caution: Impala queries data in memory – both compressed and uncompressed data are stored in memory
- **Impala example**

```
> CREATE TABLE mytable_parquet LIKE mytable_text
    STORED AS PARQUET;
> set PARQUET_COMPRESSION_CODEC=snappy;
> INSERT INTO mytable_parquet
    SELECT * FROM mytable_text;
```

Chapter Topics

- Selecting a File Format
- Hadoop Tool Support for File Formats
- Avro Schemas
- Using Avro with Hive and Sqoop
- Avro Schema Evolution
- Compression
- **Conclusion**
- Hands-On Exercise: Select a Format for a Data File

Essential Points (1)

- **Hadoop and its ecosystem support many file formats**
  - May ingest data in one format, but convert to another as needed
- **Selecting the format for your data set involves several considerations**
  - Ingest pattern
  - Tool compatibility
  - Expected lifetime
  - Storage and performance requirements

Essential Points (2)

- **Choose from the three main Hadoop file format options**
  - Text – Good for testing and interoperability
  - Avro – Best for general purpose performance and evolving schemas
  - Parquet – Best performance for column-oriented access patterns
- **Avro is a serialization framework that includes a data file format**
  - Compact binary encodings provide good performance
  - Supports schema evolution for long-term storage
- **Compression saves disk storage space and IO times at the cost of CPU time**
  - Hadoop tools support a number of different codecs

Bibliography

The following offer more information on topics discussed in this chapter

- **Avro Getting Started Guide (Java)**
  - `http://tiny.cloudera.com/adcc03a`
- **Avro Specification**
  - `http://tiny.cloudera.com/adcc03b`
- **Parquet**
  - `https://parquet.apache.org`
- **Announcing Parquet 1.0: Columnar Storage for Hadoop**
  - `http://tiny.cloudera.com/adcc03c`

Chapter Topics

- Selecting a File Format
- Avro Schemas
- Avro Schema Evolution
- Using Avro with Impala, Hive and Sqoop
- Using Parquet with Impala, Hive and Sqoop
- Compression
- Conclusion
- **Hands-On Exercise: Select a Format for a Data File**

Hands-On Exercise: Select a Format for a Data File

- **In this exercise you will**
  - Use Sqoop to import the accounts table in Avro format
  - Define an Impala table to access the Avro accounts data
  - Bonus: Save an existing plain text Impala table as Parquet
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®

Data File Partitioning

Chapter 8

Course Chapters

▪ Introduction	Course Introduction
▪ Introduction to Hadoop and the Hadoop Ecosystem ▪ Hadoop Architecture and HDFS	Introduction to Hadoop
▪ Importing Relational Data with Apache Sqoop ▪ Introduction to Impala and Hive ▪ Modeling and Managing Data with Impala and Hive ▪ Data Formats ▪ **Data File Partitioning**	**Importing and Modeling Structured Data**
▪ Capturing Data with Apache Flume	Ingesting Streaming Data
▪ Spark Basics ▪ Working with RDDs in Spark ▪ Aggregating Data with Pair RDDs ▪ Writing and Deploying Spark Applications ▪ Parallel Processing in Spark ▪ Spark RDD Persistence ▪ Common Patterns in Spark Data Processing ▪ Spark SQL and DataFrames	Distributed Data Processing with Spark
▪ Conclusion	Course Conclusion

Data File Partitioning

In this chapter you will learn

- **How to improve query performance with data file partitioning**
- **How to create and populate partitioned tables in Impala and Hive**

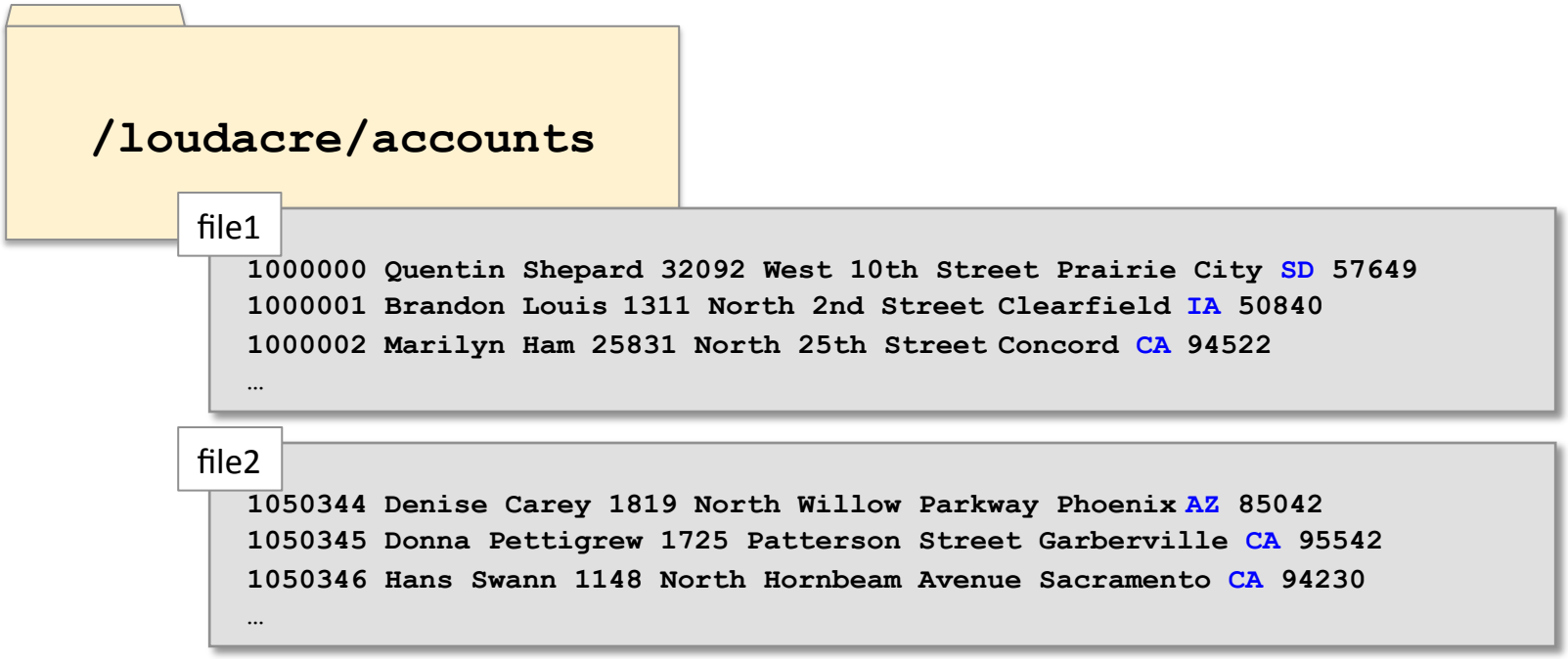
Chapter Topics

Data File Partitioning | **Importing and Modeling Structured Data**

- **Partitioning Overview**
- Partitioning in Impala and Hive
- Conclusion
- Hands-On Exercise: Partition Data in Impala or Hive

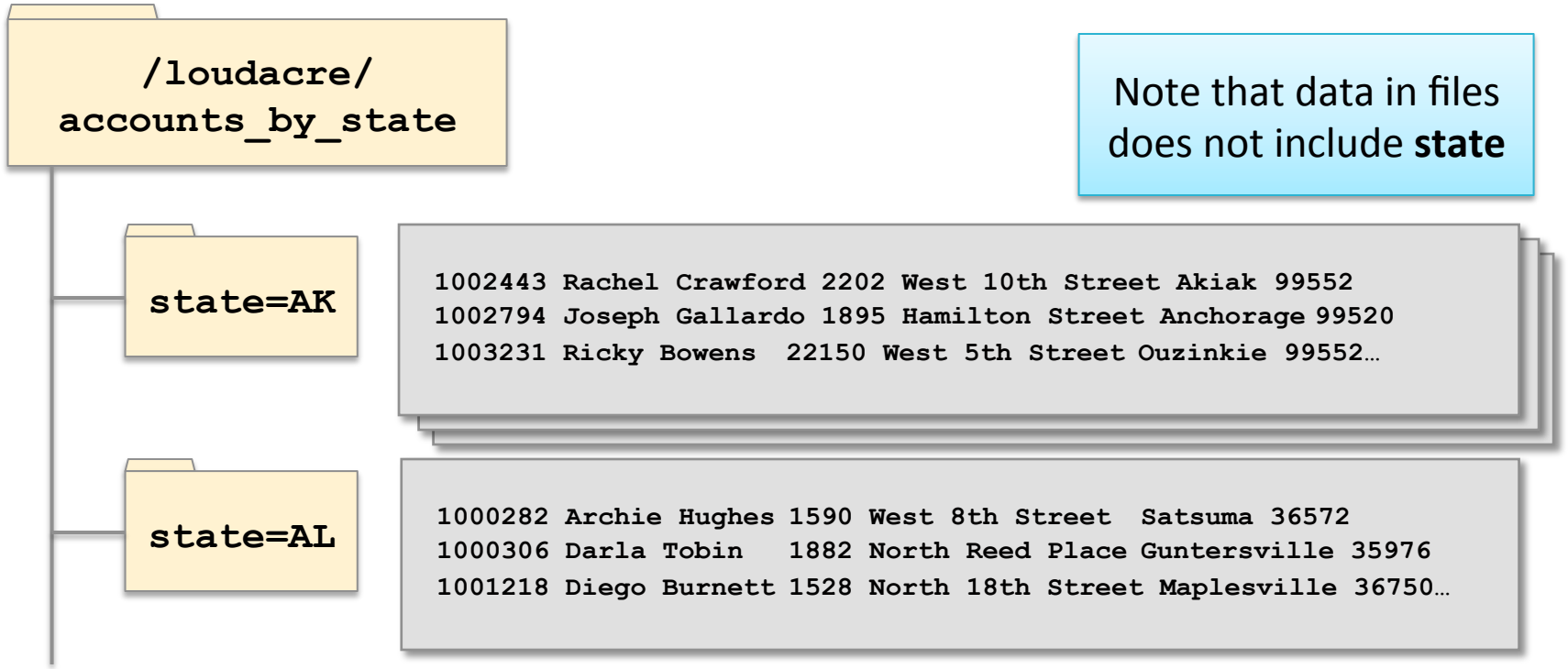
Data Storage Partitioning (1)

- **By default, all files in a data set are stored in a single HDFS directory**
  - *All* files in the directory are read during analysis or processing
  - "Full table scan"



Data Storage Partitioning (2)

- ***Partitioning* subdivides the data**
  - Analysis can be done on only the relevant subset of data
  - Potentially much faster!
- **Hadoop partitions using subdirectories**



Hadoop Partitioning

- **Partitioning is involved at two phases**
  - Storage – putting the data into correct partition (subdirectory)
  - Retrieval – getting the data out of the correct partition based on the query or analysis being done
- **Hadoop with built-in support for partitioning**
  - Hive and Impala (covered in next section)
  - Sqoop – When using the **`--hive-import`** option you can specify flags **`--hive-partition-key`** and **`--hive-partition-value`**
- **Other tools can be used to store partitioned data**
  - Spark and MapReduce
  - Flume (at ingestion)

Chapter Topics

- Partitioning Overview
- **Partitioning in Impala and Hive**
- Conclusion
- Hands-On Exercise: Partition Data in Impala or Hive

Example: Impala/Hive Partitioning Accounts By State (1)

- **Example: `accounts` is a non-partitioned table**

```
CREATE EXTERNAL TABLE accounts(
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    state STRING,
    zipcode STRING)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LOCATION '/loudacre/accounts';
```

Example: Impala/Hive Partitioning Accounts By State (2)

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

```
SELECT fname, lname
  FROM accounts
  WHERE state='NY';
```

- **By default, all queries have to scan *all* files in the directory**
- **Use partitioning to store data in separate files by state**
  - State-based queries scan only the relevant files

Example: Impala/Hive Partitioning Accounts By State (3)

- **Create a partitioned table using PARTITIONED BY**

```
CREATE EXTERNAL TABLE accounts_by_state(
    cust_id INT,
    fname STRING,
    lname STRING,
    address STRING,
    city STRING,
    state STRING,
    zipcode STRING)
  PARTITIONED BY (state STRING)
  ROW FORMAT DELIMITED
  FIELDS TERMINATED BY ','
  LOCATION '/loudacre/accounts_by_state';
```

Partition Columns

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

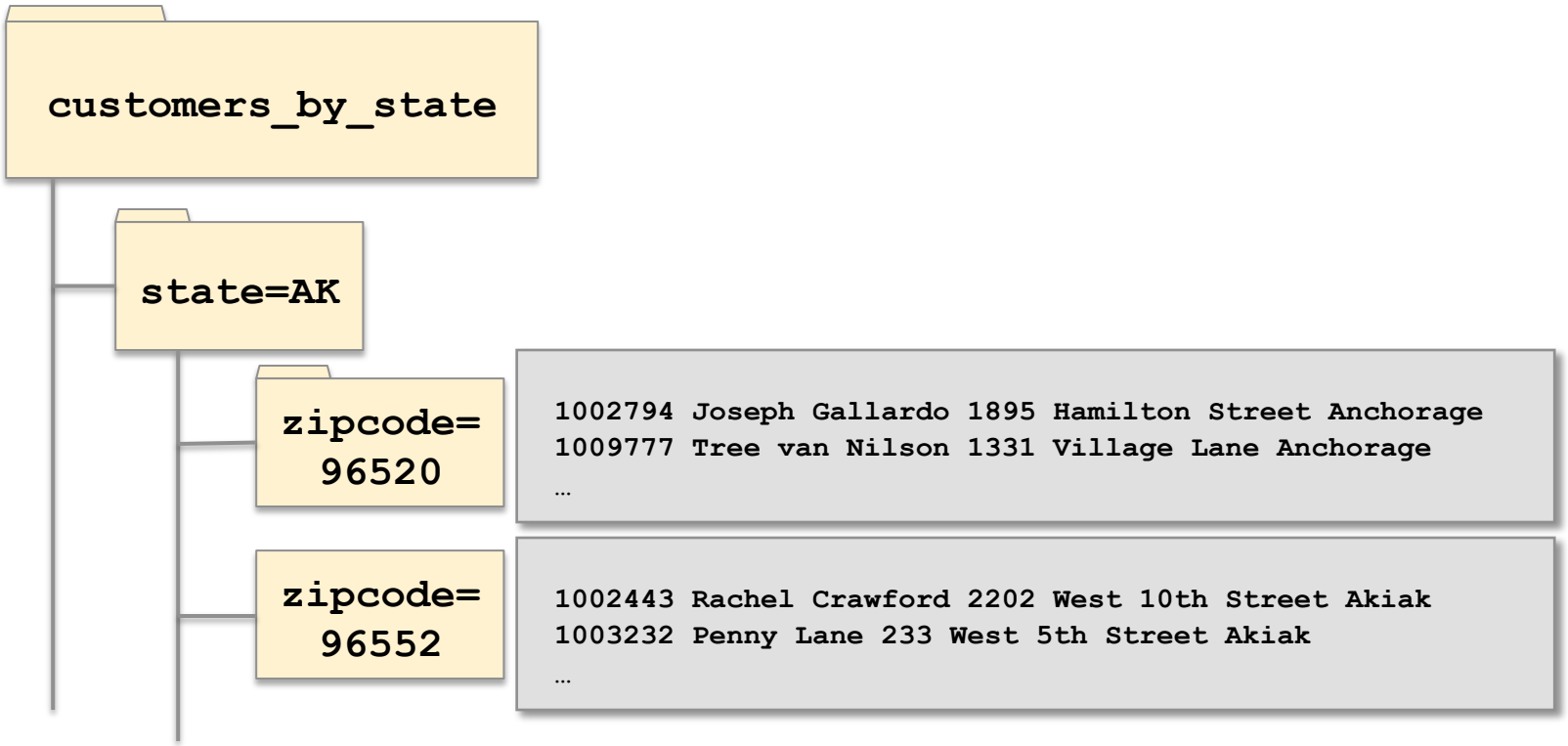
```
DESCRIBE accounts_by_state;
+---------+--------+---------+
| name    | type   | comment |
+---------+--------+---------+
| cust_id | int    |         |
| fname   | string |         |
| lname   | string |         |
| address | string |         |
| city    | string |         |
| zipcode | string |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

A partition column is a "virtual column"; data is not stored in the file

Nested Partitions

- **You can also created nested partitions**

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



Loading Data Into a Partitioned Table

- **Dynamic partitioning**
  - Impala/Hive add new partitions automatically as needed at load time
  - Data is stored into the correct partition (subdirectory) based on column value
- **Static partitioning**
  - You define new partitions using `ADD PARTITION`
  - When loading data, you specify which partition to store data in

Dynamic Partitioning

- **We can create new partitions dynamically from existing data**

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

- **Partitions are automatically created based on the value of the *last* column**
  - If the partition does not already exist, it will be created
  - If the partition does exist, it will be overwritten

Static Partitioning Example: Partition Calls by Day (1)

- **Loudacre's customer service phone system generates logs detailing calls received**
  - Analysts use this data to summarize previous days' calls
  - For example:

```
SELECT event_type, COUNT(event_type)
  FROM call_log
  WHERE call_date = '2014-10-01'
  GROUP BY event_type;
```

Static Partitioning Example: Partition Calls by Day (2)

- **Logs are generated daily, e.g.**

call-20141001.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-20141002.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 (3)

- **In the previous example, existing data was partitioned dynamically based on a column value**
- **This time we use static partitioning**
  - Because the data files do not include the partitioning data

Static Partitioning Example: Partition Calls by Day (4)

- **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 ',';
```

Loading Data Into Static Partitions (1)

- **With static partitioning, you create new partitions as needed**
- **e.g. For each new day of call log data, add a partition:**

```
ALTER TABLE call_logs
  ADD PARTITION (call_date='2014-10-02');
```

- **This command**
  1. Adds the partition to the table's metadata
  2. Creates subdirectory `/user/hive/warehouse/call_logs/call_date=2014-10-02`

Loading Data Into Static Partitions (2)

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

```
LOAD DATA INPATH '/mystaging/call-20141002.log'
  INTO TABLE call_logs
  PARTITION(call_date='2014-10-02');
```

  - This command moves the HDFS file **`call-20141002.log`** to the partition subdirectory

- **To overwrite all data in a partition**

```
LOAD DATA INPATH '/mystaging/call-20141002.log'
  INTO TABLE call_logs OVERWRITE
  PARTITION(call_date='2014-10-02');
```

Hive Only: Shortcut for Loading Data Into Partitions

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


```
LOAD DATA INPATH '/mystaging/call-20141002.log'
  INTO TABLE call_logs
  PARTITION(call_date='2014-10-02');
```

- **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=2014-10-02`** if it doesn't exist
  3. Moves the HDFS file **`call-20141002.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='2013-06-05')
   LOCATION '/loudacre/call_logs/call_date=2013-06-05';
```

```
ALTER TABLE call_logs
   DROP PARTITION (call_date='2013-06-06');
```

Creating Partitions from Existing Partition Directories in HDFS

- **Partition directories in HDFS can be created and populated outside Hive or Impala**
  - For example, by a Spark or MapReduce application
- **In Hive, use the `MSCK REPAIR TABLE` command to create (or recreate) partitions for an existing table**


```
MSCK REPAIR TABLE call_logs;
```

When To Use Partitioning

- **Use partitioning for tables when**
  - Reading the entire data set takes too long
  - Queries almost always filter on the partition columns
  - There are a reasonable number of different values for partition columns
  - The data generation or ETL process segments 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**
  - Don't partition 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)

- **In older versions of Hive, dynamic partitioning is not enabled by default**
  - Enable it by setting these two properties

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

- **Note: Hive variables set in Beeline are for the current session only**
  - Your system administrator can configure settings 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 File Partitioning | **Importing and Modeling Structured Data**

- Partitioning Overview
- Partitioning in Impala and Hive
- **Conclusion**
- Hands-On Exercise: Partition Data in Impala or Hive

Essential Points

- **Partitioning splits table storage by column values for improved query performance**
- **Partitions are HDFS directories**
  - Names follow the format ***`column=value`***
- **Partitions can be defined and loaded dynamically or statically**
- **Only partition on columns with a reasonable number of possible values**

Bibliography

The following offer more information on topics discussed in this chapter

- **Impala documentation on partitioning**
  - `http://tiny.cloudera.com/impalapart`
- **Improving Query Performance Using Partitioning in Apache Hive (Cloudera Engineering Blog)**
  - `http://tiny.cloudera.com/partblog`

Chapter Topics

- Partitioning Overview
- Partitioning in Impala and Hive
- Conclusion
- **Hands-On Exercise: Partition Data in Impala or Hive**

Hands-On Exercise: Partition Data in Impala or Hive

- **In this exercise you will**
  - Create a table for accounts that is partitioned by area code
- **Please refer to the Hands-On Exercise Manual for instructions**