Table of Contents

Chapter 1 - MapReduce Overview

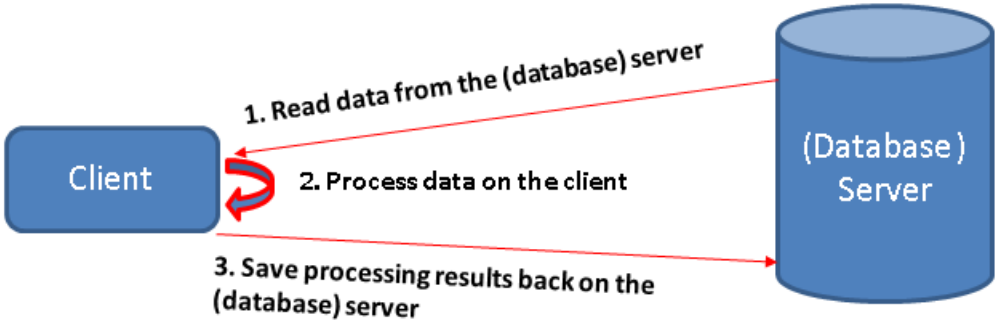
Objectives

In this chapter, participants will learn about:

- MapReduce Programming Model
- Main MapReduce design principles

1.1 The Client – Server Processing Pattern

- Good for small-to-medium data set sizes
- Fetching 1TB worth of data might take longer than 1 hour



- We need a new approach … Distributed computing!

1.2 Distributed Computing Challenges

- Process (Software) / Hardware failures
- Data consistency
- System (High) Availability
- Distributed processing optimizations

1.3 MapReduce Defined

- There are different definitions of what MapReduce (single word) is:
  - a programming model for distributed processing
  - parallel processing framework
  - a computational paradigm
  - batch query processor
- MapReduce design is influenced by functional programming languages that have the **map** and **reduce** functions
- MapReduce shields developers from technical details:
  - Networking
  - File I/O
  - Synchronization

1.4 Google's MapReduce

- MapReduce was first introduced to the world by Google engineers in a white paper back in 2004
- Google applied for and was granted US Patent 7,650,331 on MapReduce called "System and method for efficient large-scale data processing"
- The patent lists *Jeffrey Dean and Sanjay Ghemawat* as its inventors
- The value proposition of the MapReduce framework is its scalability and fault-tolerance achieved by its architecture

1.5 MapReduce Phases

- MapReduce works by breaking the data processing job into two phases:
  - The *map* phase and the *reduce* phase executed sequentially
  - The data processing flow is scheduled and coordinated by the MapReduce Master process; the map and reduce tasks are executed on worker nodes

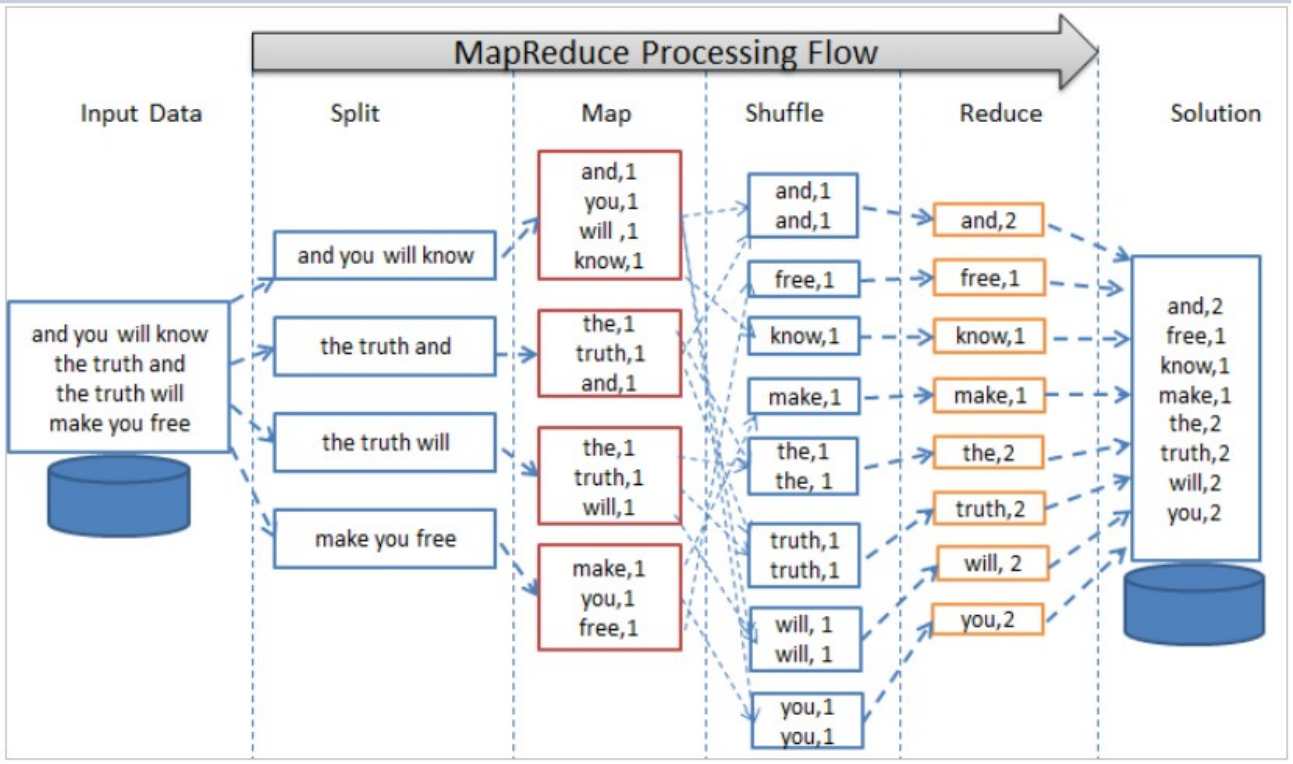
1.6 The Map Phase

- The *map* phase is backed up by a Map() procedure that receives a chunk of original data
  - Data splitting is performed by a data split utility operation that is part of the framework
- The output of the map operation is piped (emitted) as input for the reduce operation
- **Note**: While the *map* phase is normally a part of an overall MapReduce job, it can also be used independently in cases where the *reduce* phase is not required, e.g. backing up or deleting files, data migration, etc.

1.7 The Reduce Phase

- The *reduce* phase is backed up by a Reduce() procedure that performs some sort of an aggregation operation by key (e.g. counting, finding the max element in the data set, etc.) on data received from the Map() procedure
- There is also an additional phase that sits between the *map* and *reduce* phases, called "Shuffle/Sort", which processes the output of the map phase (a stream of key-value pairs) by consolidating values by key, performing inter-node data transfer, sorting data by key and sending such arranged data to the same reducer
  - The shuffle step is a utility operation transparently performed by the framework
  - Basically, the map stage transforms original input items to key-value pairs, the shuffle stage merges values with the same key, and the reduce stage processes all the items with the same key

1.8 MapReduce Word Count Job



Note: Usually, there is only one reducer process (which is the default value in Hadoop)

1.9 MapReduce Shared-Nothing Architecture

- MapReduce is designed around the *shared-nothing* architecture which leads to computational efficiencies in distributed computing environments
- Shared-nothing means "independent of others", with no synch points, shared memory, or disk
  - A mapper process is independent from other mapper processes, and so are reducer processes
- This architecture allows the map and reduce operations to be executed in parallel (in their respective phases)
- MapReduce programming model is linearly scalable

1.10 Similarity with SQL Aggregation Operations

- It may help compare MapReduce with aggregation operations used in SQL
- In SQL, aggregation is achieved by using COUNT(), AVG(), MIN(), and other such functions (which act as some sort of reducers) with the GROUP BY clause, e.g.

```
SELECT MONTH, SUM(SALES) FROM YEAR_END_REPORT GROUP BY MONTH
```

- MapReduce offers a more fine-grained programmatic control over data processing in multi-node computing environments using parallel programming algorithms

1.11 Example of Map & Reduce Operations using JavaScript

- JavaScript supports some elements of functional programming in the form of the *map()* and *reduce()* functions
- The JavaScript example here helps illustrate the overall idea behind MapReduce
  - **Note**: The actual implementation of MapReduce involves dealing with key/value pairs, which is not considered in this example.
- **Problem**: Find out the sum of elements of the *[1,2,3]* array after all its elements have been increased by 10%
  - **Note**: An alternative solution is to find the sum of the array elements before the increase and then apply the 10% increase to the total

1.12 Example of Map & Reduce Operations using JavaScript

- **Solution**:
  - The Map phase (apply the function of a 10% value increase to each element of the input array of [1,2,3]):

```
[1,2,3].map(function(x){return x  +  x/10});
Result: [1.1, 2.2, 3.3]
```

  - The Reduce phase (use the result array of the Map operation as input and sum up all its elements):

```
[1.1, 2.2, 3.3].reduce(function(x,y){return x + y});
Result: 6.6
```

- **Note**: While the *map()* and *reduce()* functions here can also be used independently from each other; MapReduce is always a single operation

1.13 Problems Suitable for Solving with MapReduce

- The following is the criteria that you can use to see if the problem at hand can be efficiently solved by MapReduce:
  - The problem can be split into smaller problems with no shared state that can be solved in parallel
    - Those smaller problems are independent from one another and do not require interactions
  - The problem can be decomposed into the *map* and *reduce* operations
    - The map operation: execute the same operation on all data
    - The reduce operation: execute the same operation on each group of data produced by the map operation
  - Basically, you should see the generic "divide and conquer" pattern

1.14 Typical MapReduce Jobs

- Counting tokens (words, URLs, etc.)
- Finding aggregate values in the target data set, e.g. the average value
- Processing geographical data
  - Google Maps uses MapReduce to find the nearest feature, like coffee shop, museum, etc., to a given address
- etc.

1.15 Fault-tolerance of MapReduce

- MapReduce operations have a degree of fault-tolerance and built-in recoverability from certain types of run-time failures which is leveraged in production-ready systems (e.g. Hadoop)
- MapReduce enjoys these qualities of service due to its architecture based on process parallelism
- Failed units of work are rescheduled and resubmitted for execution should some mappers or reducers fail (provided the source data is still available)
- **Note:** High Performance (e.g. Grid) Computing systems that use the Message Passing Interface communication for check-points are more difficult to program for failure recovery

1.16 Distributed Computing Economics

- MapReduce splits the workload into units of work (independent computation tasks) and intelligently distributes them across available worker nodes for parallel processing
- To improve system performance, MapReduce tries to start a computation task on the node where the data to be worked on is stored
  - This helps avoid unnecessary network operations by bringing the computation to the data
- "Data locality" (collocation of data with the compute node) underlies the principles of Distributed Computing Economics
- Another term to refer to this model is "*Intelligent job scheduling*"

1.17 MapReduce Systems

- Many systems leverage MapReduce programming model:
  - Hadoop has a built-in MapReduce engine for executing both regular and streaming MapReduce jobs
  - Amazon WS offers their clients Elastic MapReduce (EMR) Service (running on a Hadoop cluster)

1.18 Summary

- The MapReduce programming model was influenced by functional programming languages
- MapReduce (one word) has the map and reduce components working together in a distributed computational environment
- Production MapReduce systems offer a number of qualities of service, such as fault-tolerance

Chapter 2 - Hadoop Overview

Objectives

In this chapter, participants will learn about:

- Apache Hadoop and its core components
- Hadoop's main design considerations
- Hadoop distributions

2.1 Apache Hadoop

- Apache Hadoop is a distributed fault-tolerant computing platform
  - Written in Java
- Designed as a massively parallel processing (MPP) system based on a distributed master-slave architecture for both data storage and distributed data processing
- Hadoop allows for distributed processing of large data sets across clusters of computers using simple programming models, e.g. MapReduce
- It is linearly (usually horizontally) scalable
  - Just add more servers, when needed
- Hadoop uses machines with internal hard drives as it helps ensure data locality for computing jobs
  - A data processing job is attempted to run on a machine which hard drive stores the data to be processed by the job

2.2 Apache Hadoop Logo

2.3 Typical Hadoop Applications

- Log and/or clickstream analysis
- Web crawling results processing
- Marketing analytics
- Machine learning and data mining
- Data archiving (e.g. for regulatory compliance, etc.)
- See http://wiki.apache.org/hadoop/PoweredBy for the list of educational and production uses of Hadoop

2.4 Hadoop Clusters

- First versions of Hadoop were only able to handle 20 machines; newer versions are capable to run Hadoop clusters comprising thousands of nodes
- Hadoop clusters run on moderately high-end commodity hardware ($2-5K per machine)
- Hadoop clusters can be used as a data hub, data warehouse or a business analytics platform

2.5 Hadoop Design Principles

- Hadoop's design was influenced by ideas published in Google File System (GFS) and MapReduce white papers
- Hadoop's core component, Hadoop Distributed File System (HDFS) is the counterpart of GFS
- Hadoop uses functionally equivalent to Google's MapReduce data processing system also called MapReduce (the term coined by Google's engineers)
- One of the main principle of Hadoop's architecture is "design for failure"
  - To deliver high-availability quality of service, Hadoop detects and handles failures at the application layer (rather than relying on hardware)

2.6 Hadoop Versions

- Over its history, Hadoop has undergone significant changes in parts of the platform
- Usually, two major versions of Hadoop are mentioned:
  - The old Hadoop, referred to as Hadoop 1 (you may not see it in production today)
  - The modern Hadoop, referred to as Hadoop 2, or the second-generation Hadoop
- The main change introduced in Hadoop 2 was the cluster resource management layer called YARN (Yet Another Resource Negotiator)

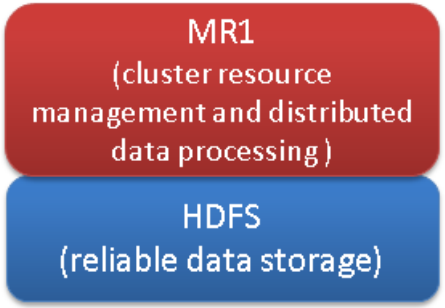
2.7 Hadoop's Main Components

- The Hadoop project is made up of the following main components:
  - **Common (a.k.a. Core)**
    - Contains Hadoop infrastructure elements (interfaces with HDFS, system libraries, RPC connectors, Hadoop admin scripts, Job scheduling and monitoring, Web UI, etc.)
  - **Hadoop Distributed File System (HDFS)**
    - Hadoop's persistence component designed to run on clusters of commodity hardware built around the "*load* once and *read* many times" concept
  - **YARN (used to be MapReduce in Hadoop 1)**
    - Cluster resource management system which provisions cluster-wide execution environment for a number of data processing systems and engines, including MapReduce

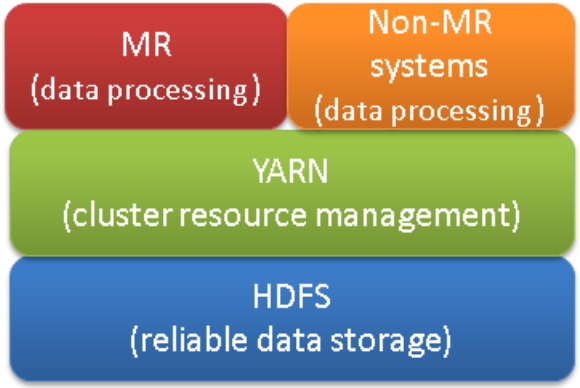
2.8 Hadoop Simple Definition

- In a nutshell, Hadoop is a distributed computing framework that consists of:
  - Reliable distributed data storage (provided via HDFS)
  - Distributed data processing system (provided by YARN)

2.9 Side-by-Side Comparison: Hadoop 1 and Hadoop 2



Hadoop 1



Hadoop 2

2.10 Hadoop-based Systems for Data Analysis

- Hadoop (via HDFS) can host the following systems for data analysis:
  - The **MapReduce** (MR) engine (the major data analytics component of the Hadoop project)
  - **Apache Pig**
  - **HBase** database
    - By-passes MR (it is a Non-MR system)
  - **Apache Hive** data warehouse system
  - **Apache Mahout** machine learning system
  - **Spark**
    - By-passes MR
  - etc.

2.11 Other Hadoop Ecosystem Projects

- **Ambari:** Admin Web UI for provisioning, managing, and monitoring Hadoop resources
  - Used by Hortonworks Data Platform (HDP)
- **Hue:** Web Admin Integrated UI for Hadoop applications
- **HCatalog**: Metadata service of SQL schemas with a REST interface
- **Sqoop**: Import/export utility used for data extraction and loading between Hadoop and RDBMSes
- **Flume**: Streaming data collection and HDFS ingestion framework
- **Storm:** Real-time processing of streams of data
- **ZooKeeper**: Coordination, synchronization and naming registry service for distributed applications
- **Oozie**: Workflow management of interdependent jobs
- **Solr**: A search engine with a REST-like API
- …. and more ...

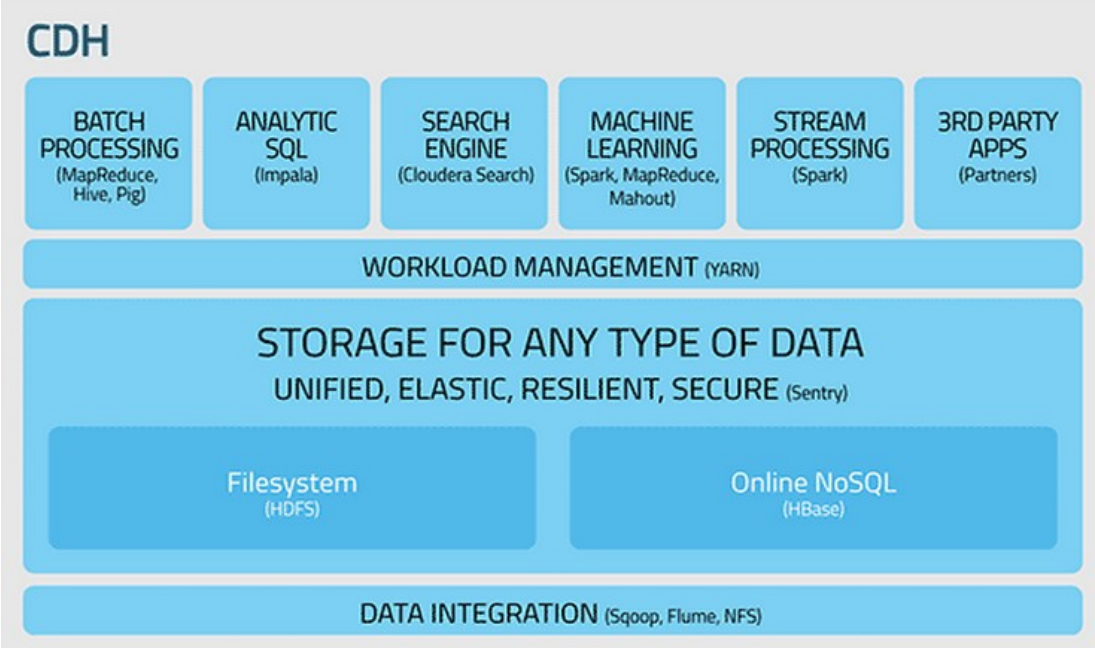
2.12 Hadoop Caveats

- Hadoop is a batch-oriented processing system
- Many Hadoop-centric business analytics systems have high processing latency which comes from their dependencies on the MapReduce sub-system
  - Querying even small data sets (under a gigabyte in size) using the MapReduce sub-system may take up to a minute to run to completion
  - This is in sharp contrast with querying speed of relational databases such as MySQL, Oracle, DB2, etc. where functionally similar work can be done several orders of magnitude faster by applying indexes and other techniques
- Systems that bypass MapReduce when building queries have near real-time querying times (e.g. HBase and Cloudera Impala)

2.13 Hadoop Distributions

- **Cloudera** (http://www.cloudera.com)
  - Open Source Hadoop Platform which consists of Apache Hadoop and additional key open source projects
  - You have an option of an Express (free and unsupported) edition and Enterprise (paid supported) edition
- **Hortonworks** (http://hortonworks.com)
  - Open Source Hadoop Platform with additional open source projects
  - Runs on Linux, the Microsoft Windows and Azure cloud (through partnership with Microsoft)
- **MapR** (https://www.mapr.com)
  - Offers a distribution of Hadoop with proprietary(and more efficient) components
  - MapR was selected by Amazon Web Services (AWS) for their Elastic Map Reduce (EMR) service (a premium option)
  - MapR is Google's technology partner

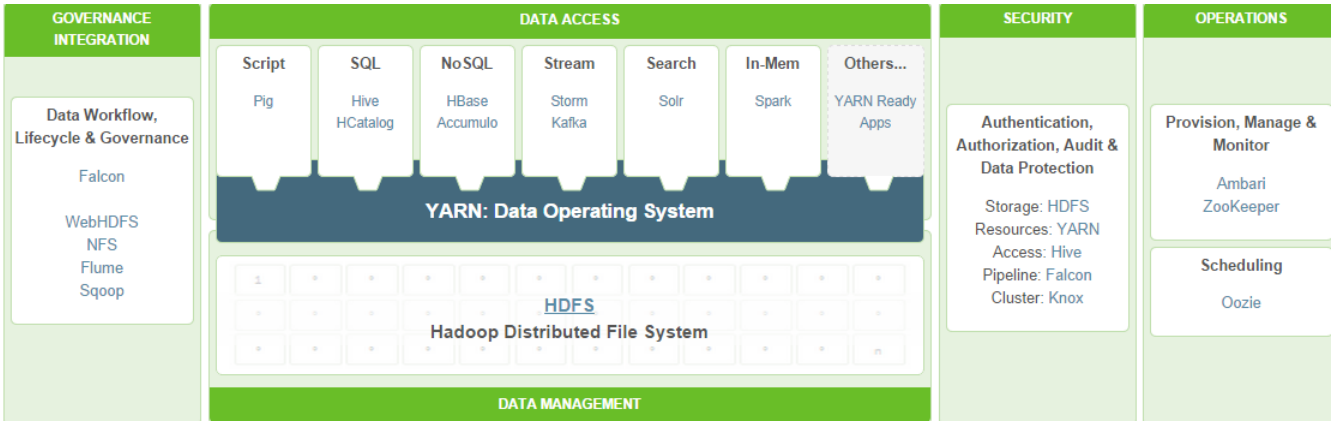
2.14 Cloudera Distribution of Hadoop (CDH)



2.15 Cloudera Distributions

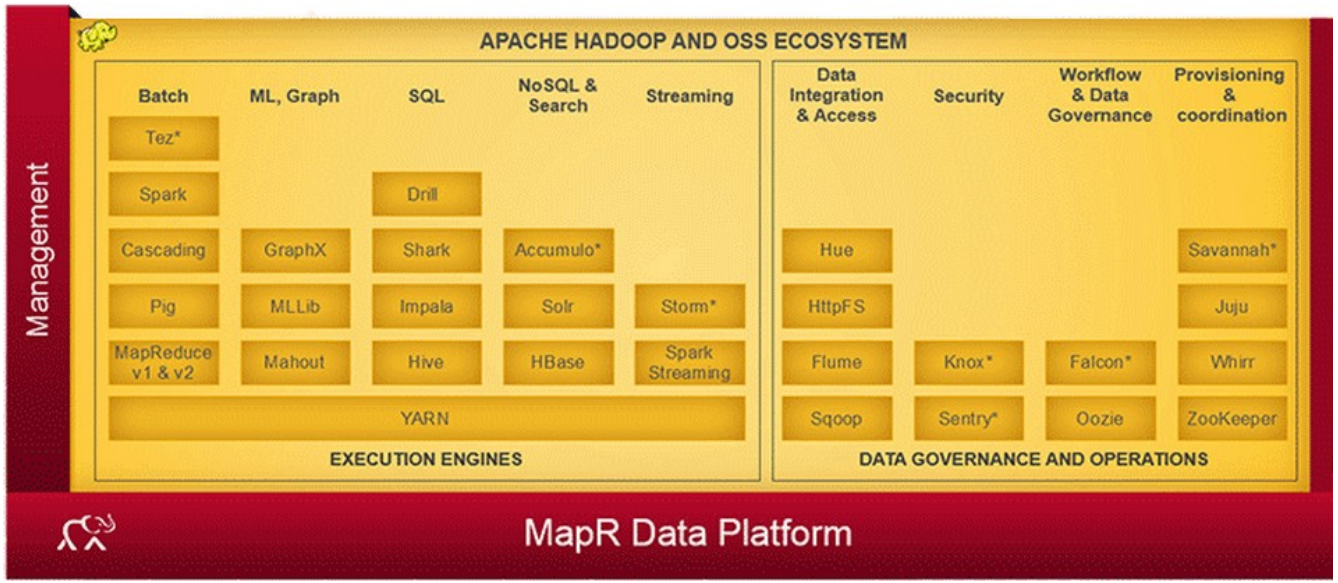
- **Cloudera Express** (sort of a community edition; free download)
- **Cloudera Enterprise** Editions (requires annular subscription)
  - Basic
  - Flex
  - Data Hub
- Cloudera Enterprise adds the following features on top of those in the Express Edition:
  - AD Kerberos Integration
  - SNMP Support
  - LDAP Integration
  - Automated Disaster Recovery
  - Operational Reports
  - and many more …
- Both are shipped with the Cloudera Manager proprietary Admin Web UI for resource provisioning, managing and monitoring

2.16 Hortonworks Data Platform (HDP)



- HDP uses Apache Ambari Admin Web UI

2.17 MapR


APACHE HADOOP AND OSS ECOSYSTEM
Management
Batch
ML, Graph
SQL
NoSQL & Search
Streaming
Data Integration & Access
Security
Workflow & Data Governance
Provisioning & coordination
Tez*
Spark
Drill
Cascading
GraphX
Shark
Accumulo*
Hue
Savannah*
Pig
MLLib
Impala
Solr
Storm*
HttpFS
Juju
MapReduce v1 & v2
Mahout
Hive
HBase
Spark Streaming
Flume
Knox*
Falcon*
Whirr
YARN
Sqoop
Sentry*
Oozie
ZooKeeper
EXECUTION ENGINES
DATA GOVERNANCE AND OPERATIONS
MapR Data Platform

2.18 Summary

- Apache Hadoop is a distributed fault-tolerant computing platform written in Java used for processing large data sets across clusters of computers
- Hadoop clusters may comprise thousands of nodes
- One of the main principle of Hadoop's architecture is "design for failure"

Chapter 3 - Hadoop Distributed File System Overview

Objectives

In this chapter, participants will learn about:

- Hadoop Distributed File System (HDFS)
- Ways to access HDFS

3.1 Hadoop Distributed File System (HDFS)

- The Hadoop Distributed File System (HDFS) is a distributed, scalable, fault-tolerant, and portable file system written in Java
- Its design was influenced by the Google File System (GFS) paper published in 2003 (*https://research.google.com/archive/gfs.html*)
- The architecture of HDFS is based on the master/slave design pattern
- An HDFS cluster consists of:
  - The NameNode server (the Master) holding directory information about file fragments (called blocks) persisted on data nodes
  - The DataNode daemon running on data nodes
    - File blocks are stored on their internal hard drives

3.2 HDFS Considerations

- HDFS is designed for efficient implementation of the following data processing pattern:
  - Write once (normally, just load the data set on the file system)
  - Read many times
- Supports large data reads (and not random I/O); processing of data-intensive jobs can be done in parallel
- HDFS functionality (and that of Hadoop) is geared towards batch-oriented rather than real-time scenarios
- HDFS does not have a built-in (on-heap) cache
  - Caching off-Java process-heap is possible – it is done at the local OS level
- Worker nodes host both DataNode (HDFS) and NodeManager (YARN) daemons to support data locality physics

3.3 HDFS High Availability

- Until Hadoop 2 released in May 2012, the NameNode server had been a Single Point of Failure (SPoF)
  - Hadoop included (and still does) a server called a Secondary NameNode, which does not provide the failover capability for the NameNode
- The HDFS HA (always “on”) feature for a Hadoop cluster, when enabled, addresses the SPoF issue by providing the option of running two NameNodes in an Active/Passive configuration
  - These two NameNodes are referred to as Active and (hot) Standby servers
  - This HA arrangement allows a fast failover from a failed Active NameNode or because of an administrator-initiated failover to the Standby node
- DataNodes announce of their availability by sending heartbeats to NameNode every 3 seconds
- While HA is an attractive feature, you may also want to consider ways to enhance resiliency of the NameNode by using RAID storage, frequent back-ups, dual power supplies, etc.

3.4 Storing Raw Data in HDFS

- In HDFS, you store any type of data in its raw format – without applying a specific data schema that would help you structure your data
- As a result, HDFS is more agile in ingesting your data comparing with the mandatory requirement in RDBMSes to have a rigid schema upfront before data import
- You can apply a schema when you start processing your data
  - In other words, your application that reads data from HDFS can apply the (desired) schema on read operation
  - The choice of schema is dictated by your application needs when transforming / analyzing data
  - This schema arrangement is referred to as the "*schema-on-demand*", or "*schema-on-read*"

3.5 HDFS Security

- HDFS implements Unix-like file permissions
  - Files have an owner, a group, and read / write permissions for local system users
- Authentication is configured at Hadoop's level by turning on the service level authentication
  - By default, Hadoop runs in non-secure mode with no authentication required: it basically helps honest developers avoid making bad mistakes; but it won't stop hackers from wreaking havoc with the system
  - When Hadoop is configured to run in secure mode, then each user and service are required to be authenticated by Kerberos
- Service level authorization, when turned on, checks user / service permissions for accessing a given service

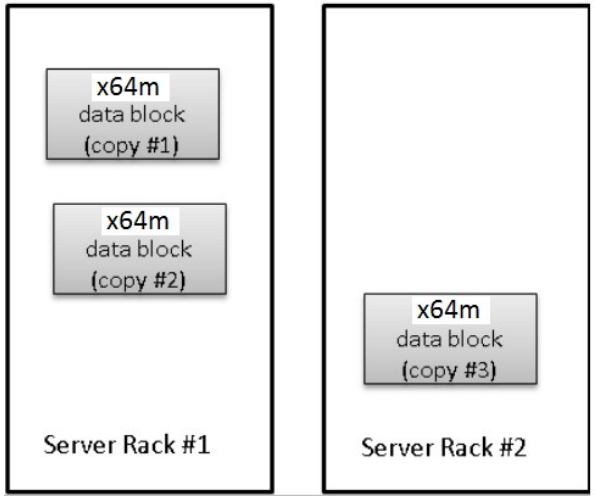
3.6 HDFS Rack-awareness

- HDFS design is "rack-aware" to minimize network latency
  - **Note**: Storage Area Network (SAN) or similar storage technologies on slaves are not recommended for performance considerations
- Rack-awareness (or, more accurately, rack switch awareness) means taking into account a machine's physical location (rack) while scheduling tasks and allocating storage
- Basically, HDFS is aware of the fact that network latencies between machines sitting in the same server rack is much smaller than those between machines in different racks
  - This knowledge helps ResourceManager allocate containers (and, by extension, computing tasks) closest to the data
- Rack-awareness adds to maintaining system efficiency in case of hardware failure (except for the failure of a rack switch!)

3.7 Data Blocks

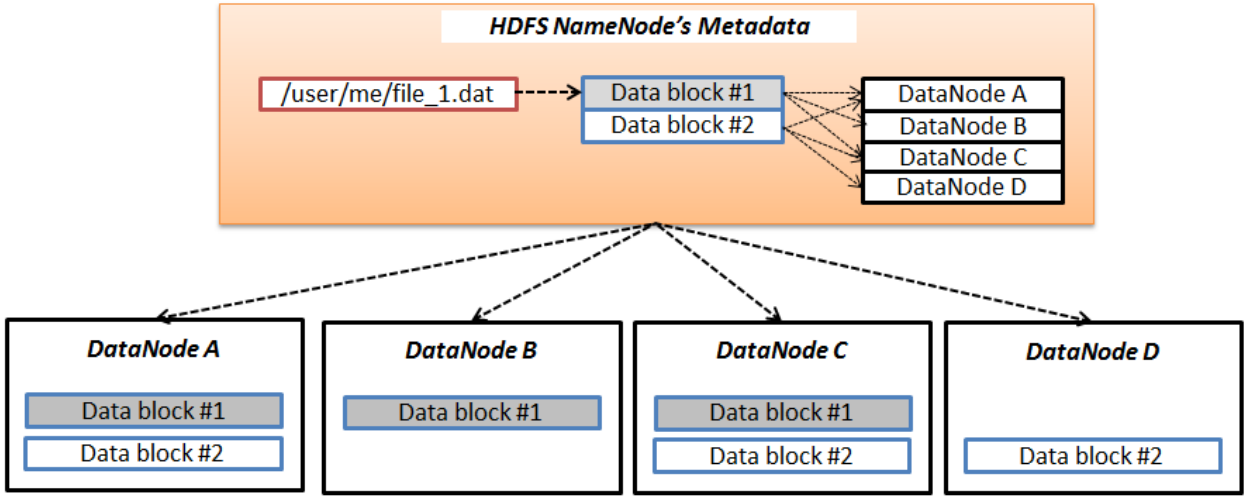
- A data file in HDFS is split into blocks (a typical block size used by HDFS is 64 / 128 MB)
  - For large files, bigger block sizes will help reduce the amount of metadata stored in the NameNode (metadata server)
- Blocks are files named *blk_<system resource stamp>* sitting on Data Nodes' local file system in a designated directory
- Data reliability is achieved by replicating data blocks across multiple machines
  - By default, data blocks get replicated to three nodes (3 replicas of the same data block are created): two on the same server rack, and one on a different rack for redundancy
- To aid in enforcing file integrity, DataNode daemons periodically send block reports to the NameNode
- When the NameNode is notified on a corrupted block (e.g. its checksum is found corrupt, or a DataNode is marked as down), it re-replicates the block(s) elsewhere

3.8 Data Block Replication Example



Example of a three-way (default) replication of a single data block for redundancy and achieving high data availability (the block size is usually a multiple of 64M)

3.9 HDFS NameNode Directory Diagram



3.10 File Metadata Records (Conceptual View)

File Name	Block ID	Block Seq#	Node Location	Checksum
Myfile.txt	blk_78787871	1	[dn1,dn4,dn6]	121212121212
Myfile.txt	blk_78787873	3	[dn2,dn3dn5]	323232323232
Myfile.txt	blk_78787872	2	[dn5,dn1,dn3]	454545454545
Myfile.txt	blk_78787874	4	[dn1,dn2,dn6]	767676767676

Note: Every record also contains Unix-like file permissions (not shown above for size)

3.11 NameNode Meta Information Size

- Every block record takes about 150 - 200 bytes
- For example, a 1GB file will require 8 blocks, if the block size = 128M
  - So, roughly, you will need up to about 1,600 bytes for storage

3.12 HDFS Balancing

- HDFS tries to intelligently place data blocks throughout the cluster
- Placement balancing maximizes the performance of data processing tasks as it eliminates hot spots

3.13 Accessing HDFS

- HDFS is not a Portable Operating System Interface (POSIX) compliant file system
- It is modeled after traditional hierarchical file systems containing directories and files
- File-system commands (copy, move, delete, etc.) against HDFS can be performed in a number of ways:
  - Through the HDFS Command-Line Interface (CLI) which supports Unix-like commands: *cat*, *chown*, *ls*, *mkdir, mv*, etc.
  - Using Java API for HDFS
  - Via the C-language wrapper around the Java API
  - Using regular HTTP browser for file-system and file content viewing
    - This is made possible through Web server (Jetty) embedded in the NameNode and DataNodes
    - Using WebHDFS API
- When a command is received by HDFS, it gets translated into appropriate command(s) executed against the native file system

3.14 Examples of HDFS Commands

- The common way to invoke the HDFS CLI:
  - *hadoop fs {HDFS commands}*
- Copying a file from the local file system over to HDFS (with the same name):
  - *hadoop fs -put <filename_on_local_sysetm>*
- Copying the file from HDFS to the local file system (with the same name):
  - *hadoop fs -get <filename_on_HDFS>*
- Recursively listing files in a directory:
  - *hadoop fs -ls -R <directory_on_HDFS>*
- Creating a directory under the current user's home directory (e.g. */user/userid/*) on HDFS
  - *hadoop fs -mkdir REPORT*
- For more examples on accessing HDFS from the command line, visit *http://hortonworks.com/hadoop-tutorial/using-commandline-manage-files-hdfs/*

3.15 Other Supported File Systems

- HDFS is not the only file system supported by Hadoop
- As of 2011, Hadoop provides support for other types of file systems, including:
  - FTP File system via links to remotely accessible FTP servers
  - Amazon Cloud S3 (Simple Storage Service) file system
    - Rack-awareness in this file system is not supported
  - Windows Azure Storage Blobs (WASB) file system. WASB is mounted on top of HDFS and provides data access API for Hadoop applications

3.16 WebHDFS

- WebHDFS is a REST API for accessing all of the HDFS file system interfaces.
- Via WebHDFS you can use such common tools as curl and wget to interface with HDFS
- WebHDFS is embedded in HDFS' NameNode and DataNodes
- You can use Kerberos (SPNEGO) and Hadoop delegation tokens for client authentication

3.17 Examples of WebHDFS Calls

- GET (read) a file named /user/test/mydata.dat:

```
http://host:port/webhdfs/v1/user/test/mydata.dat?op=OPEN
```

- PUT (create) a folder named */user/test/myfolder*

```
http://host:port/webhdfs/v1/user/test/myfolder?op=MKDIRS
```

- POST a file to append to an existing file:

```
http://host:port/webhdfs/v1/user/test/mydata.dat?op=APPEND
```

3.18 HDFS Daemon Web UI Ports

NameNode	50070
DataNode	50075
Secondary NameNode	50090

3.19 Viewing Replica Factor and Block Size in NameNode Web UI

- You can browse HDFS directories and files by navigating to http://<NameNodeServer>:500070/

Permission	Owner	Group	Size	Replication	Block Size	Name
-rw-r--r--	cloudera	cloudera	613 B	1	128 MB	stopExtraService.sh

3.20 HDFS Write Operation

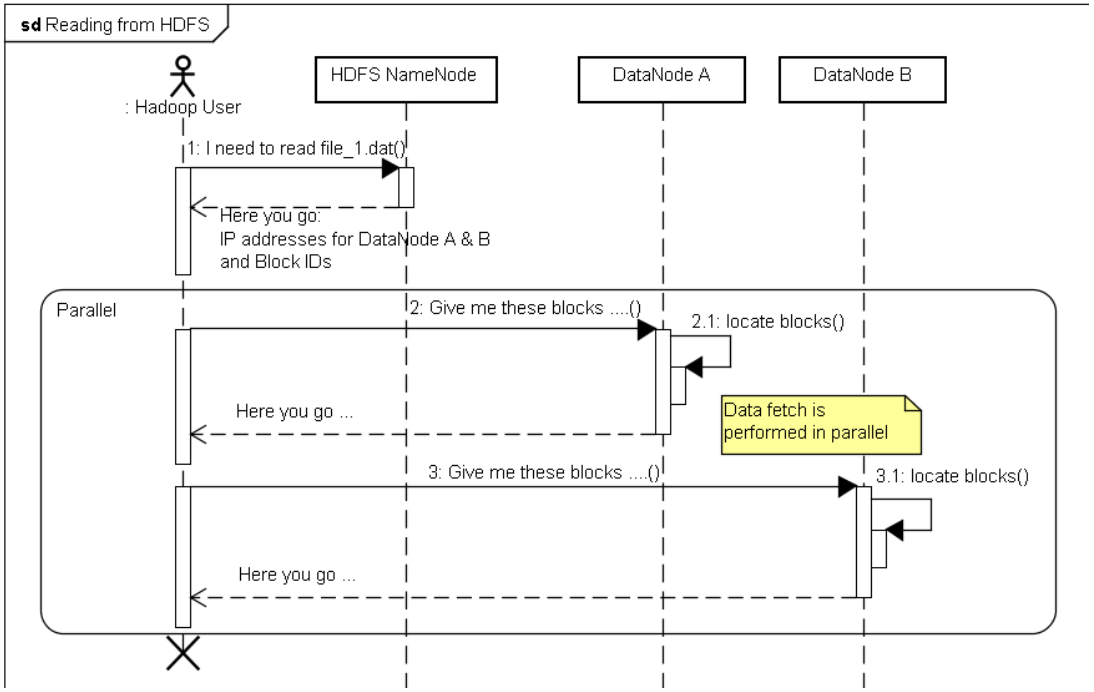
- The Client first contacts the NameNode
- For each data block in the file, the NameNode generates and registers meta-data about the file and allocates suitable DataNodes to keep the file's replicas
  - All blocks are written by the client consecutively
- The list of DataNodes (3 by default) allocated for storing the block replicas is sent back to the client
- The Client contacts the first DataNode in the list (which becomes the "Primary" DataNode)
- The "Primary" DataNode acts as the coordinator for block replication writes; it makes a local copy of related data and engages other DataNodes in a peer-to-peer data sharing communication for data replication
- Each DataNode sends acknowledgments of receiving their data back to the "Primary" DataNode
- The client gets an acknowledgment from the "Primary" DataNode as a confirmation of success of the whole HDFS write operation for the block
- The client notifies the NameNode on completion of the operation
- **Note:** When the client writes to the DataNodes, it also calculates and provides the checksum of the data block, which helps with data integrity

- The checksum is calculated and compared by the client on the block read
- To prevent “bit rot”, DataNodes also verify block checksums every three weeks (default) after the block was created

3.21 HDFS Read Operation

- Whenever a Hadoop client needs to read a file on HDFS, it first contacts the NameNode
- The NameNode locates block ids associated with the file as well as IP address of the DataNodes storing those blocks
- The NameNode returns the related information to the client
- The client contacts the related DataNodes and supplies the block ids for DataNodes to locate the blocks on their local HDFS storage
- The DataNodes serve the blocks of data back to the client
- Client verifies the checksum for every block

3.22 Read Operation Sequence Diagram



3.23 Communication inside HDFS

- HDFS was originally designed for supporting batch oriented processing rather than interactive and real time use cases
- This affected the design of communication protocols inside Hadoop and HDFS
- Communication inside Hadoop is done via the Remote Procedure Call (RPC) application protocol layered on top of TCP/IP

Summary

- The Hadoop Distributed File System (HDFS) is a distributed, scalable, fault-tolerant and portable file system written in Java
- HDFS architecture is based on the master/slave design pattern
- File-system commands against HDFS can be performed in a number of ways:
  - Through the **fs** command-line interface (which supports Unix-like commands: cat, chown, ls, mkdir, mv, etc.)
  - Using Java API for HDFS
  - Via the C-language wrapper around the Java API

Chapter 4 - Hive

Objectives

This chapter introduces the Apache Hive™ data warehouse framework where participants will learn about:

- Main Hive features
- Hive Architecture
- HiveQL Basics

4.1 What is Hive?

- Hive is a Hadoop-based data warehouse framework/infrastructure written in Java
- Works with large datasets stored in Hadoop's HDFS and compatible file systems such as Amazon S3 file system
- Originally developed by Facebook to cope with the growing volumes of data in their data warehouse that they were not able to solve using traditional technologies

4.2 Apache Hive Logo

4.3 Hive's Value Proposition

- Hive shields non-programmers (business analysts, etc.) from programming MapReduce jobs directly with its own SQL-like query language called HiveQL
  - Internally, HiveQL queries are converted into actual MapReduce jobs submitted to Hadoop for execution against files stored on HDFS
  - In YARN, in addition to MapReduce, Hive also supports Apache Tez and Apache Spark execution engines (MR is used by Hive by default)
- HiveQL is extendable via new types, user functions and scripts
- Custom map and reduce modules can be plugged in HiveQL for fine-tuned processing logic

4.4 Who uses Hive?

- Facebook
- Netflix
- Amazon Web Services (AWS)
  - AWS maintains a software fork of Apache Hive included in Amazon Elastic MapReduce (EMR)
- … and many others ...

4.5 What Hive Does Not Have

- Declarative referential integrity
  - No primary keys, foreign keys, and other such constraints
- Support for transactions (it is not an OLTP database)

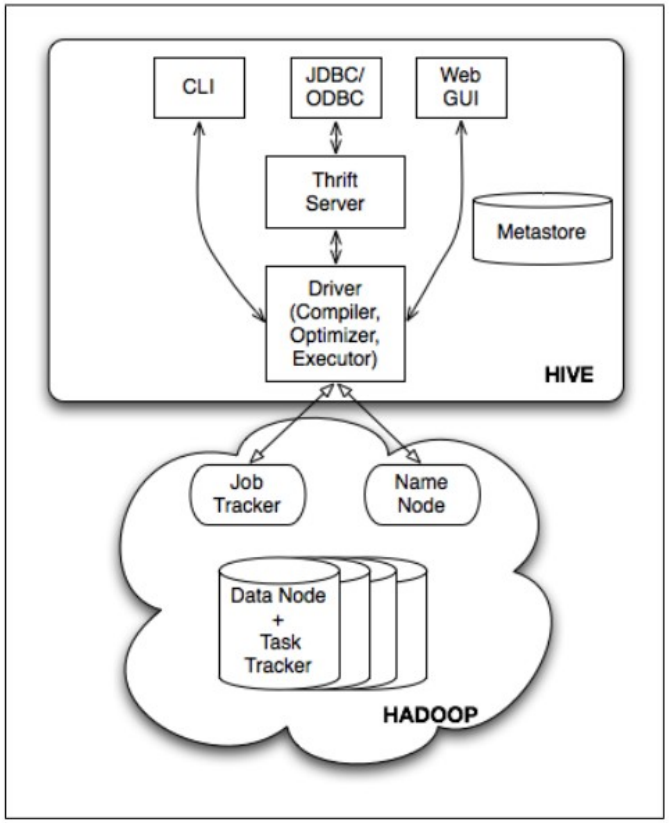
4.6 Hive's Main Sub-Systems

- Hive consists of three main sub-systems:
  - Metadata store (metastore) for schema information (table column data types, related files location, etc.) is stored in an embedded or external database
    - Another Hive's sub-system called HCatalog sits on top of the metastore and provides access to the metastore by other Hadoop-centric products, such as Pig and MapReduce jobs
    - HCatalog provides RESTful interface to the metastore through its web server called WebHCat
  - Serialization / deserialization framework for reading and writing data
  - Query processor that translates HiveQL statements into MapReduce instructions

4.7 Hive Features

- Support for four file formats: TEXTFILE, SEQUENCEFILE, RCFILE and ORC
  - As of version 0.13, Hive supports Apache Parquet columnar storage format
- Bitmap indexing as of 0.10 with other index types planned for future releases
- Metadata storage in an internal (embedded) or external RDBMS
- Ability to work on data compressed with gzip, bzip2 and other algorithm
- Support for User-Defined Functions
- SQL-like query language (HiveQL)
- Support for JDBC, ODBC and Thrift clients
- Hive lends itself to integration with business intelligence and visualization tools such as Microstrategy, Tableau, Revolutions Analytics, etc.
  - In many cases, integration is done using Hive's high performance ODBC driver which supports all SQL-92 interfaces

4.8 The "Classic" Hive Architecture

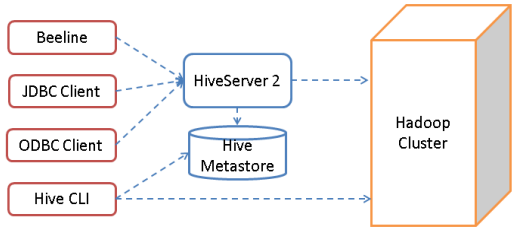


Hive simplified architecture

(Source: http://www.vldb.org/pvldb/2/vldb09-938.pdf)

4.9 The New Hive Architecture

- In Hive 0.11, Hive architecture was redesigned as a client / server system to enable various remote clients to execute queries against the Hive Server, referred to as HiveServer2 (you may also see references to the new design as Hive 2, HS2, etc.)
- The new architecture provides better support for multi-client concurrency and authentication
- In addition to the original Hive CLI, Hive 2 also supports the new CLI, called Beeline, and JDBC and ODBC clients



4.10 HiveQL

- SQL-like query language developed to hide complexities of manually coding MapReduce jobs
- HiveQL does not fully implement SQL-92 standard
- It has Hive-specific syntactic extensions
- Hive is suited for batch processing over large sets of *immutable* data (e.g. web logs)
  - UPDATE or DELETE operations are not supported; but INSERT INTO is acceptable
- HiveQL does not support transactions
  - Full ACID transactional support is planned for future releases
- Hive often works more efficiently on denormalized data (see *Notes*)
- Generally, HiveQL statements are case-insensitive

4.11 Where are the Hive Tables Located?

- Location of the tables that you create in Hive is specified in the *hive.metastore.warehouse.dir* property of the *hive-site.xml* configuration file:

```
<property>
    <name>hive.metastore.warehouse.dir</name>
    <value>/user/hive/warehouse</value>
  </property>
```

  - The value of the property points to the directory on HDFS
  - For the above value, you can verify the existence of the *warehouse* directory by running the following command:

```
hadoop fs -ls /user/hive/warehouse
```

4.12 Hive Command-line Interface (CLI)

- Hive offers command-line interface (CLI) through its *hive* utility that can be used in two modes of operations:
  - An interactive mode, where the user enters HiveQL-based queries manually
  - In unattended (batch) mode, where the *hive* utility takes commands on the command-line as arguments

4.13 The Beeline Command Shell

- HiveServer2 comes with its own CLI called **Beeline**
- Beeline is a JDBC client that is based on the SQLLine CLI (http://sqlline.sourceforge.net/)
- The Beeline shell supports two operational modes: embedded and remote
  - In embedded mode, Beeline runs an embedded Hive Server
  - In remote mode, Beeline connects to a separate HiveServer2 process
    - This mode supports concurrent clients connections
    - Client authentication is possible with LDAP and Kerberos
    - SSL encryption of connections is supported
- You can run Hive CLI commands from Beeline
- The Hive CLI will eventually be deprecated in favor of Beeline

4.14 Summary

- Hive is a data warehouse framework/infrastructure written in Java that runs on top of Hadoop
- Hive offers users an SQL-like query language called HiveQL which shields users from complexities of programming MapReduce jobs directly
- Hive also offers a CLI through its *hive* utility that can be used in two modes of operations:
  - Interactive, and
  - Unattended (batch)
- Beeline, the CLI for Hive 2, provides support for Hive CLI commands

Chapter 5 - Hive Command-line Interface

Objectives

This chapter introduces the Hive command-line interface that allows users to execute HiveQL scripts in two modes:

- Interactive mode
- Unattended (batch) mode

5.1 Hive Command-line Interface (CLI)

- Hive CLI is based on the command-line *hive* shell utility which can be used to execute HiveQL commands in either interactive or unattended (batch) mode
  - In interactive mode, the user enters HiveQL-based queries manually, submitting commands sequentially
  - In unattended (batch) mode, the *hive s*hell takes command parameters on the command-line
- Hive CLI also supports variable substitution that helps create dynamic scripts

5.2 The Hive Interactive Shell

- You start the interactive shell by running the *hive* command
  - To suppress information messages printed while you are in the shell, start the shell with the *-S* (silent) flag: *hive -S*
- After successful initialization, you will get the *hive>* command prompt
- To end the shell session, enter *quit;* or *exit;* at the command prompt
  - **Note**: Don't forget a semi-colon (;) at the end of each command
- Command syntax of the Hive shell was influenced by MySQL command-line, e.g.

```
SHOW tables; DESCRIBE tblFoo;
```

5.3 Running Host OS Commands from the Hive Shell

- The Hive shell supports execution of host OS (operation system) commands
- OS commands must be prefixed with '!' and terminated with ';'
  - For example, to print the current working directory, issue the following command:

```
hive>!pwd;
```

5.4 Interfacing with HDFS from the Hive Shell

- Use the *dfs* command while in the shell to get access to the HDFS API
- For example,
  - To get the listing of files in the user home directory in HDFS, issue this command:

    ```
    hive> dfs -ls;
    ```
  - To remove a file from HDFS skipping HDFS's Trash bin:

    ```
    hive> dfs -rm -skipTrash myDS;
    ```
  - **Note**: The *dfs* command returns a reference to the HDFS command-line interface so to get help on *dfs* commands, run the following command from the host OS terminal (not from the Hive shell):

    ```
    hadoop fs -help
    ```

5.5 The Hive in Unattended Mode

- In additional to the interactive shell interface, Hive supports invocation of commands in unattended mode
- Commands to execute on command-line are preceded by the '*-e*' flag
- This mode is suitable for executing short commands that can be issued directly against Hive, e.g.

```
$ hive -e 'SHOW TABLES;'
```

- You can submit more than one command, just use the ';' command separator, e.g.:

```
hive -e 'SHOW TABLES; DESCRIBE foo;'
```

- To suppress information messages, use the -S (silent) flag
- To get the CLI help, run the *hive -H* command

5.6 The Hive CLI Integration with the OS Shell

- The Hive CLI can be easily integrated with the underlying OS shell from which the *hive* utility is launched
- For example:

```
$ hive -S -e "select * FROM mytable LIMIT 3" > /tmp/f.dat
```

  - The output of the above command is redirected into the */tmp/f.dat* file on the file system
- More sophisticated command chains can be built using Unix command pipelines

5.7 Executing HiveQL Scripts

- Hive can take scripts that contain one or more HiveQL commands for execution in non-interactive (a.k.a. unattended or batch) mode, e.g.:

```
$ hive -f myscript.hql
```

- The HiveQL scripts may have any extensions; by convention, the *.hql* extension is used
- To execute a script file from inside the Hive shell, use the **SOURCE** command

```
hive> SOURCE /path/to/file/hive_queries.hql;
```

5.8 Comments in Hive Scripts

- When you execute a file script using the command line, e.g. *hive -f myscript.hql,* the script may have comments which are denoted as '--' (two dashes) in front of the comment line, e.g.

```
-- Monthly report data generator
---------------------------------
```

- This options is available in Hive since version 0.8.0
- **Note**: Comments don't work inside the interactive Hive shell

5.9 Variables and Properties in Hive CLI

- Using command-line, you can define custom variables (a.k.a. properties) that you can later use in your Hive scripts using the *${varname}* syntax
  - Hive replaces references to variables with their values and then submits the query
- This facility aids in creating dynamic scripts

5.10 Setting Properties in CLI

- For setting properties when executing commands in unattended mode, you have two functionally equivalent options:

```
--define key1=value1
--hivevar key1=value1
```

  - The above assignments add the *key1=value1* key-value pair in the *hivevar* namespace (more on this later) for the duration of the script execution session
  - **Note:** These options are supported in Hive since version 0.8.0

5.11 Example of Setting Properties in CLI

- The command below shows how to set a property (variable) on command-line

```
hive -S -e  'DESCRIBE ${tblName};' --define tblName=foo
```

  - The above command will execute the command in silent mode (-S)
  - We define the variable *tblName* as having the value of *foo*
  - The Hive shell will substitute the *${tblName}* parameter with its value (*foo*) and then execute the command as '*DESCRIBE foo;'* which will show the structure of the table *foo*

5.12 Hive Namespaces

- Hive namespaces help group variables into buckets of functionally similar properties
- Hive supports four namespaces (access types are: ***R*** for read-only and ***W*** for Write-enabled)

Namespace	Access Type	Description
hivevar	R/W	User-defined variables (properties)
hiveconf	R/W	Hive configuration properties
system	R/W	Java system properties
env	R	Shell environment variables

5.13 Using the SET Command

- In the Hive shell, you read / write (set) variables using the SET command
- Except for the user-defined *hivevar* and *hiveconf* namespaces, you need to prefix the variable with the namespace it belongs to
- For example:
  - To read the OS user's home directory (which is a property of the *env* namespace):

```
hive> SET env:HOME;
```

  - Using the SET command without the argument, will print all available variables in all the namespaces: *env, hivevar, hiveconf, and system*

```
hive> SET;
```

  - **Note**: To get extensive information on the Hadoop configuration, use the *SET -v;* command

5.14 Setting Properties in the Shell

- To set the value of a writable (settable) property in the interactive shell, use the following command:

```
SET [namespace:]myVar=myVarValue;
```

  - The above command is also used to create the variable if it does not exist
- **Note:** Inside the interactive Hive shell, the *hivevar* and *hiveconf* namespaces can be omitted
- You can verify the new property's value by running the following command:

```
SET [namespace:]myVar;
```

- **Note:** You can reset the configuration to the default values by using the *reset* command (since Hive version 0.10)

5.15 Setting Properties for the New Shell Session

- One of the useful features of the property facility is the ability to set properties before invoking the interactive shell

```
$ hive --define var1=val1 --define var2=val2
```

  - The above command will set two session-side properties *var1* and *var2* that can be used in the launched Hive interactive shell session

```
hive> set var1;
  var1=value1
hive> set var2;
  var2=value2
```

5.16 Setting Alternative Hive Execution Engines

- By default, Hive uses the MR engine
- You can configure to run Hive on alternative execution engines:
  - Apache Tez:

    ```
    set hive.execution.engine=tez;
    ```

  - Apache Spark:

    ```
    set hive.execution.engine=spark;
    ```

- The default value for this configuration is set as follows:

  ```
  set hive.execution.engine=mr;
  ```

5.17 The Beeline Shell

- In the latest versions of Hadoop distros, like Cloudera's CDH and Hortonworks' HDP, you are given a choice to use the Beeline shell instead of the standard Hive shell offered through the *hive* command tool
- You can run most Hive commands from Beeline; Beeline also supports most of the command switches used by the *hive* command tool, like **-e** and **-f** for query execution
- Transitioning from the Hive shell to Beeline shell, when required, is quite straightforward:
  - Migration steps from the Hive CLI to Beeline are discussed here: *http://blog.cloudera.com/blog/2014/02/migrating-from-hive-cli-to-beeline-a-primer/*

5.18 Connecting to the Hive Server in Beeline

- Beeline supports embedded and remote modes of operation when connecting to the target server
- The target server URL you want to connect to from the Beeline client is identified as a parameter to the **-u** command flag
- In both modes, Beeline requires the JDBC transport protocol prefix for HiveServer2 -- **jdbc:hive2://** -- present in the target server URL
  - In embedded mode, the connection URL is just the JDBC transport protocol prefix for HiveServer2:
    - **beeline -u jdbc:hive2://**
  - In remote mode, you need to specify the target server's name / IP address, port, and valid client credentials, e.g.:
    - **beeline -u jdbc:hive2://<IP>:<P>/<DB>;user=x;password=y**

5.19 Beeline Command Switches

- Beeline supports most of the command switches of the *hive* utility
- On top of that, Beeline adds a significant number of new command switches that help you get a better control over how query results are presented and formatted
- The complete list of Beeline command switches is shown in the slide's notes section for your reference

5.20 Beeline Internal Commands

- When you start a Beeline shell session and connect to the target server, you get access to a suite of commands that you can execute from inside Beeline
- The internal Beeline commands are prefixed with a bang '!'
- You can get a list of those commands by typing in *help*
- The complete list of the Beeline internal commands is given in the slide's notes for your reference

5.21 Summary

- The Hive CLI is built around the Hive shell utility which allows users to run commands in two modes:
  - Interactive mode, where users enter HiveQL-based queries manually inside the shell
  - Unattended (batch) mode, where the Hive shell is invoked from the host OS command-line and executes commands passed on as arguments
- Beeline supports embedded and remote modes of operation when connecting to the target server and it offers developers a wide range of command switches and internal commands

Chapter 6 - Hive Data Definition Language

Objectives

This chapter provides an extended overview Hive's Data Definition Language (DDL). Participants will learn about

- Creating and Dropping Hive Databases
- Creating and Dropping Tables in Hive
- Supported Data Type Categories
- Table Partitioning
- The EXTERNAL Keyword
- Hive Views

6.1 Hive Data Definition Language

- The following commands are supported by the Hive Data Definition Language (DDL):

```
Create/Drop/Alter Database
Create/Drop/Truncate Table
Alter Table/Partition/Column
Create/Drop/Alter View
Create/Drop Index
Show
Describe
```

- We will review some of the more useful DDL commands, for a complete coverage of Hive DDL, visit https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL

6.2 Creating Databases in Hive

- When you start working with Hive, you are provided with the *default* database which is sufficient in most cases
- To create a new database in Hive, issue the following command:

```
CREATE (DATABASE|SCHEMA) [IF NOT EXISTS] database_name
  [COMMENT database_comment]
  [LOCATION hdfs_path]
```

  - **Note**: The SCHEMA and DATABASE terms are used interchangeably
  - COMMENT helps document the database purpose

6.3 Using Databases

- To show available databases on the system, issue this command in the Hive shell:

```
hive> SHOW DATABASES;
```

- The *USE* command sets the current working database, e.g.:

```
hive> USE myHiveDB;
```

- To drop a database, issue this command:

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

6.4 Creating Tables in Hive

- To create a table, use the following (simplified) table creation statement:

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] [db_name.]table_name
  [(col_name data_type [COMMENT col_comment], ...)]
  [COMMENT table_comment]
  [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
  [ROW FORMAT row_format]
        DELIMITED [FIELDS TERMINATED BY char]
        [COLLECTION ITEMS TERMINATED BY char]
        [MAP KEYS TERMINATED BY char]
  [STORED AS file_format]
  [LOCATION hdfs_path]
```

  - **Note**: The *CREATE TABLE* statement supports inserting string comments for documenting purposes

- Before creating a new table, you can check the list of existing tables by issuing the following command:

```
SHOW TABLES;
```

- Hive stores the table data in a subdirectory of the directory defined by *hive.metastore.warehouse.dir* property of the *hive-site.xml* configuration file (the default value is */user/hive/warehouse*)

6.5 Supported Data Type Categories

- The following data type categories are supported in Hive DDL:
  - Primitive types
  - Array type (holds elements of the same data type)
  - Map type (holds maps of data types as key-values pairs)
  - Struct (holds a C-like structure of grouped elements)
  - Union (C-like union of types)

6.6 Common Numeric Types

- The following numeric types are supported:

```
TINYINT     1 byte signed integer
SMALLINT    2 byte signed integer
INT         4 byte signed integer
BIGINT            8 byte signed integer
BOOLEAN     { true | false }
FLOAT       4-byte single precision floating point number
DOUBLE            8-byte double precision floating point number
DECIMAL
```

```
Introduced in Hive 0.11.0 with a precision of 38 digits
Hive 0.13.0 introduced user definable precision and scale
```

6.7 String and Date / Time Types

- STRING
  - String literals can be used with either single quotes (') or double quotes ("). C-style escaping within the strings (e.g. '\t', etc.) is supported
- VARCHAR (as of Hive 0.12.0)
  - Created with a length specifier (between 1 and 65355)
- CHAR (as of Hive 0.13.0)
  - Similar to Varchar but fixed-length (1 to 255); values shorter than the specified length value are padded with spaces
- TIMESTAMP (as of Hive 0.8.0)
  - Traditional Unix timestamp (seconds from 1970-01-01 00:00:01 UTC to 2038-01-19 03:14:07 UTC) with optional nanosecond precision
  - Timestamp text should be in the YYYY-MM-DD HH:MM:SS[.fffffffff] format
- DATE (as of Hive 0.12.0)
  - In the YYYY-MM-DD format without the time of day part
- For more information on supported types, see https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Types

6.8 Miscellaneous Types

- BOOLEAN
- BINARY (as of Hive 0.8.0)

6.9 Example of the CREATE TABLE Statement

```
CREATE TABLE gamesAnalysisTable (
  uname        STRING        COMMENT  'User name',
  udetails     STRUCT<email:STRING, city:STRING, zip:INT> COMMENT  'User info',
  score        FLOAT         COMMENT  'User max game score',
  commands     ARRAY<INT> COMMENT 'User issued command ids',
  decisions    MAP<STRING, INT> COMMENT 'User gaming decisions')
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
    COLLECTION ITEMS TERMINATED BY '$'
    MAP KEYS TERMINATED BY '#' ;
```

- Hive uses Java generics' syntax (<DATA_TYPE>) for specifying the type of data held in ARRAYs and MAPs

6.10 Working with Complex Types

- The *COLLECTION ITEMS TERMINATED BY char* clause defines the delimiter character, e.g. '$'
- The collection items can be struct members, array elements, and map key-value pairs
- For example, with a '$' used as the delimiting char:

1. *udetails STRUCT<email:STRING, city:STRING, zip:INT>* will require that the underlying rows will have its elements grouped as follows:

```
rocketman@r0ktm.us$New York$10001
joedoe@someserver.com$Los Angeles$90001
```

- ◇ You access struct elements as follows:

```
SELECT udetails.city, udetails.email from  gamesAnalysisTable;
```

2. *commands ARRAY<INT>* will require elements of the array be packed as follows:

```
101$102$103....
```

- ◇ To read the first element (101) of the *commands* array in all rows:

```
SELECT commands[0] FROM gamesAnalysisTable;
```

6.11 Working with Complex Types

- *MAP KEYS TERMINATED BY '#'* would require you separate keys from values with a '#'
- The *COLLECTION ITEMS TERMINATED BY char* clause controls the key-value pair separator
- The DELIMITED FIELDS TERMINATED BY *char* clause controls the field separator
- The `decisions MAP<STRING, INT>` definition would require the input file elements representing the map be grouped as follows:

```
Go up#177$Stay put#2000$Exit#0
```

  - In order to read values keyed, for example, by "*Stay put*" from the *decisions* map:

```
SELECT decisions["Stay put"] FROM gamesAnalysisTable;
```

6.12 Table Partitioning

- Table partitioning significantly improves query performance by having Hive store table data in physically isolated directories and files
- Partitioning is done on partitioning pseudo-column(s) that you specify in the PARTITIONED BY clause

```
PARTITIONED BY (make STRING, year SMALLINT);
```

- Partitioning columns usually used in the WHERE clause of your HiveQL queries, e.g.:

```
SELECT * FROM CARS WHERE make='Ford' AND year=2014;
```

- The partitioning columns must not be repeated in the table definition itself
  - If they do, you will get this error: *"FAILED: Error in semantic analysis: Column repeated in partitioning columns"*

6.13 Table Partitioning

- Table partitioning changes Hive table directory structure on the data storage
- The table directory will have new sub-directories that reflect the sequence of columns in the PARTITON BY clause as well as discrete values in the partitioning columns, e.g.:

```
CREATE TABLE CAR (
. . .
PARTITIONED BY (make STRING);
```

clause will create a bunch of .../CAR/make=<CAR_MAKE> directories, e.g.

```
.../CAR/make='Acura'
.../CAR/make='Ford'
.../CAR/make='Nissan'
…
```

- **Note:** The specific *make=<CAR_MAKE>* directory now must have all the data that belongs to the particular *CAR_MAKE*;
- **Note:** The *make* column is not repeated in the table definition
- Partitioning makes data storage more efficient

6.14 Table Partitioning on Multiple Columns

- Partitioning a table on more than one column will create a hierarchy of sub-directories that follows the sequence of columns in the *PARTITION BY* clause

```
CREATE TABLE CAR (
. . .
PARTITIONED BY (make STRING, year SMALLINT);
```

clause will create a bunch of

```
  .../CAR/make=<CAR_MAKE_VALUE>/year=<YEAR>
```

sub-directories of the CAR table, e.g.

```
.../CAR/make='Acura'/year=2012/
.../CAR/make='Acura'/year=2013/
.../CAR/make='Acura'/year=2014/
.../CAR/make='Ford'/year=2012/
.../CAR/make='Ford'/year=2013/
.../CAR/make='Ford'/year=2014/
.../CAR/make='Nissan'/year=2012/
… … …
```

6.15 Viewing Table Partitions

- The following command lists the partitions defined on a table:

```
SHOW PARTITIONS <table_name>
```

- You can narrow down a list of returned partitions defined on a table, e.g:

```
SHOW PARTITIONS car PARTITION(make='Ford');
  make=Ford/year=2012
  make=Ford/year=2013
  make=Ford/year=2014
```

6.16 Row Format

- Declared by the [ROW FORMAT row_format] DDL clause
- If the ROW FORMAT section not specified, the default ROW FORMAT DELIMITED … format is used by Hive
- The following most common formats are supported:
  - DELIMITED FIELDS TERMINATED BY <*char*>, e.g.

    ```
    ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
    ```
  - SERDE serde_name [WITH SERDEPROPERTIES (property_name=property_value, property_name=property_value, …)]

    (more on this in a moment)

6.17 Data Serializers / Deserializers

- Hive supports reading / writing data using specific data serializers / deserializers (The *SERDE …* DDL fragment)
- Reading / writing of data using SerDes occurs as follows

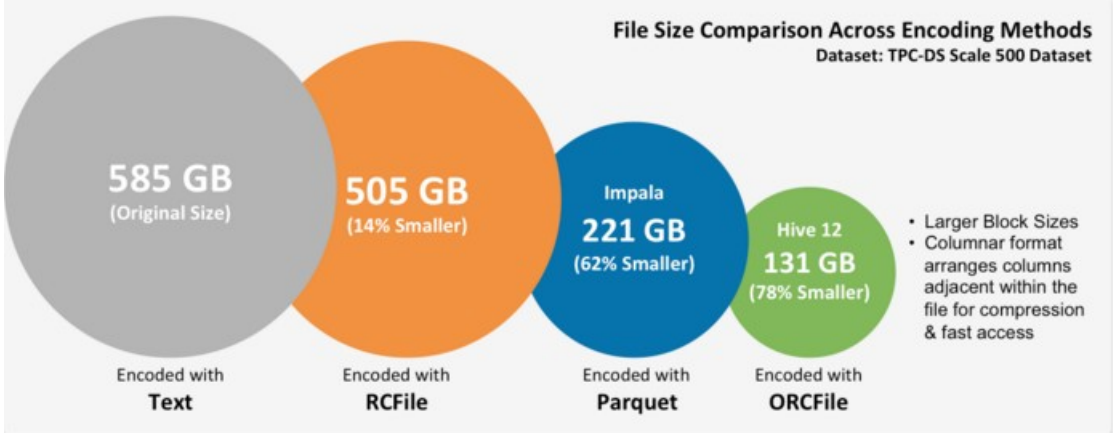
```
HDFS files -->  Deserializer --> Row object in Hive

Row object in Hive --> Serializer --> HDFS files
```

- Hive has the following built-in SerDes protocols
  - Avro
  - ORC
  - RegEx
  - Thrift
- There exist external SerDes libraries, e.g. for handling JSON files
- Users can write custom SerDes for their own data formats

6.18 File Format Storage

- Declared with the [STORED AS *file_format*] DDL caluse
- The following *file_format* options are supported:
  - *SEQUENCEFILE* is used if data in the file needs to be compressed
  - *TEXTFILE*
  - *RCFILE* (Only available as of Hive 0.6)
  - *ORC* (Only available as of Hive 0.11)
  - *PARQUET* (used without ROW FORMAT SERDE) for the Parquet columnar storage format (as of Hive 0.13)
  - *AVRO* (as of Hive 0.14)
- For more details, see https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL

6.19 File Compression



- Source: http://hortonworks.com/blog/orcfile-in-hdp-2-better-compression-better-performance/

6.20 More on File Formats

- The *STORED BY* clause is used to support non-native artifacts, e.g. HBase tables
- *STORED AS INPUTFORMAT … OUTPUTFORMAT* allows users to specify the file format managing Java class names, e.g.

```
org.apache.hadoop.hive.contrib.fileformat.base64.Base64TextInputFormat
com.hadoop.mapred.DeprecatedLzoTextInputFormat
org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
```

6.21 The ORC Data Format

- The ORC (Optimized Row Columnar) file format provides a highly efficient way to store Hive data
- ORC has the following advantages:
  - Reduced load on the NameNode
  - Support for an extended range of types: datetime, decimal, and the complex types (struct, list, map, and union)
  - Seeking for a given row in the data set
  - (Optional) data compression
  - Efficient and flexible metadata storage using Google's Protocol Buffers

6.22 Converting Text to ORC Data Format

- Let's say you have a text-based Hive table **tblTEXT**
- Create a table that has the same schema as *tblTEXT*, but has the STORED AS ORC qualifier:

```
CREATE TABLE tblORC (
// Same schema as the text table ...
) STORED AS ORC;
```

- Insert data into the ORC table from the TEXT table:

```
INSERT INTO TABLE tblORC SELECT * FROM tblTEXT;
```

  - The Text-to-ORC conversion will happen automatically

6.23 The EXTERNAL DDL Parameter

- The EXTERNAL parameter allows users to create a table in location different from that specified by the *hive.metastore.warehouse.dir* property in the *hive-site.xml* configuration file
- The directory for the table is specified by the LOCATION DDL parameter of the CREATE EXTERNAL TABLE statement
- This option gives an advantage of re-using data already generated in that location and which has data structure that corresponds to the field data types declared by the CREATE statement
  - **Note**: When you drop a table created with the EXTERNAL parameter, data in the table is not deleted in HDFS (since Hive does not own the data)
- Tables created without the EXTERNAL parameter are referred to as Hive-managed table

6.24 Example of Using EXTERNAL

- If you have an HDFS folder named */user/me/myexternal_folder/* which contains a text file with two tab-delimited columns, you can use the following CREATE **EXTERNAL** TABLE … statement against this location:

```
CREATE EXTERNAL TABLE tblExternal (col1 STRING, col2 STRING)
COMMENT 'Creating a table at a specific location'
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/user/me/myexternal_folder/';
```

- Hive will automatically detect the presence of the file in the folder specified by the *LOCATION* parameter and try to covert underlying values into the table column types
  - Where Hive fails to convert data (e.g. from a STRING ('abc') into a FLOAT), the operation will still be successfully completed with non-converted values registered as *NULL*

6.25 Creating an Empty Table

- In some cases, you may need to have a table with the structure similar to that of an already existing table
- Use the following command to copy the table definition:

```
CREATE TABLE T2 LIKE T1;
```

  - In this case, the T2 table will be created with the same table definition used in creating the T1 table
  - No records will be copied and T2 will be empty

6.26 Dropping a Table

- Use the following command to drop a table:

```
DROP TABLE [IF EXISTS] table_name
```

- The DROP TABLE statement removes table's metadata as well as the data in the table
  - **Note 1**: The data is moved to the *.Trash/Current* directory in the user's home HDFS directory (if trashing is configured); the table metadata in the metastore can no longer be recovered
  - **Note 2**: When dropping a table created with the EXTERNAL parameter, the original data file will not be deleted from the file system

6.27 Table / Partition(s) Truncation

- Table / Partition(s) truncation removes all rows from a table or partition(s)
- The command has the following syntax:

```
TRUNCATE TABLE table_name [PARTITION partition_spec];
```

- User can specify partial *partition_spec* for truncating specific partitions

6.28 Alter Table/Partition/Column

- These commands allow user to have a fine-grained control over the table structure, e.g.
  - Changing table name:

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

  - Adding a partition:

```
ALTER TABLE table_name ADD PARTITION (partCol = 'pc_name') location
'path_to_data_location';
```

  - Dropping a partition:

```
ALTER TABLE table_name DROP [IF EXISTS] PARTITION partition_spec,
PARTITION partition_spec,...
```

  - Changing column:

```
ALTER TABLE test_name CHANGE orig_col_name new_col_name new_type;
```

  - This command will allow users to change a column's name, and (optionally) its data type

6.29 Views

- Hive views are virtual tables with no associated physical storage
- Views are built as a logical construct on top of physical tables
- You can run regular queries against views:

```
SELECT name FROM myView WHERE phone LIKE '416%'
```

- A view's schema is immutable and is defined when the view is created; subsequent changes to underlying tables (e.g. renaming or adding a column) will not be propagated to the view's schema.
  - Any subsequent invalid references to the underlying changed parameters will raise an exception

6.30 Create View Statement

- A view is created with the following statement:

```
CREATE VIEW [IF NOT EXISTS] view_name [(column_name [COMMENT
column_comment], ...) ]
[COMMENT view_comment]
      AS SELECT …
```

- For example:

```
CREATE VIEW vNames (firstName, lastName)
            AS SELECT fn,ln FROM User;
```

- **Note**: If no column names are supplied (they are optional) in the view definition, the names of the view's columns will be derived automatically from the defining AS SELECT expression. Column names may be redefined in the view definition

6.31 Why Use Views?

- Views are useful when you need:
  - Restrict viewable data based on some conditions (limiting columns and rows for security and other considerations)
  - Wrap up complex queries

6.32 Restricting Amount of Viewable Data

- Views help restrict the amount of viewable data by the following techniques:
  - Providing a view of the subset of columns in the source table
  - Providing a subset of rows matching the WHERE clause
  - Using a combination of the above techniques
- Using these techniques, you can hide sensitive information by not declaring them in the view

6.33 Examples of Restricting Amount of Viewable Data

- You can restrict the viewing of the client *income* (which may be regarded as confidential information) in the *client* table defined as follows:

```
CREATE TABLE client  (
     fn STRING, ln STRING, income FLOAT) . . . ;
```

  - by omitting the *income* column in the view based on the *client* table:

```
CREATE VIEW publicClientView AS
    SELECT fn, ln FROM client;
```

- Views can also be used to limit the number of returned records by specifying the WHERE matching condition in the AS SELECT clause:

```
CREATE VIEW shortView AS
      SELECT * FROM bigTable WHERE <limiting conditions>;
```

6.34 Creating and Dropping Indexes

- Hive supports column indexing to speed up data querying
  - Hive indexing capabilities are limited (and not reviewed in detail here)
- Indexes are created with the CREATE INDEX … statement
- The indexed data for a table is stored in another table defined in the CREATE INDEX … statement
- You can drop a created index by using the DROP INDEX … statement
  - DROP INDEX drops the index and deletes the index table
- For more information, visit https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL

6.35 Describing Data

- Hive offers the DESCRIBE statement that can be applied to a number of objects: databases, tables, partitions, views, and columns
- The DESCRIBE statement shows metadata associated with the target object
- The general syntax of the statement:

```
DESCRIBE some_object;
```

  - For example:

```
DESCRIBE myTable;
```

6.36 Summary

- Hive offers an extensive Data Definition Language (DDL) that addresses the most practical user needs, such as:
  - Creating and dropping Hive databases
  - Creating and dropping tables
  - Performing table partitioning for faster data querying
  - Creating views to minimize the amount of viewable data either for performance of security considerations

Chapter 7 - Hive Data Manipulation Language

Objectives

In this chapter, participants will learn about Hive's Data Manipulation Language (DML) and its two primary ways of data loading:

- Using the LOAD DATA statement
- Using the INSERT statement

7.1 Hive Data Manipulation Language (DML)

- The Hive Data Manipulation Language (DML) deals with loading data in a table or partition
- There are two primary ways to load data in Hive:
  - Using the LOAD statement
  - Using the INSERT statement

7.2 Using the LOAD DATA statement

- The LOAD DATA statement performs the bulk load operation and has the following syntax:

```
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO TABLE
tablename [PARTITION (partcol1=val1, partcol2=val2 ...)]
```

- If the keyword LOCAL is specified, then the contents of the file specified by the *filepath* parameter is loaded (copied over from the OS file system) into the target file system
  - The *filepath* parameter can be a relative or an absolute path to the file
- If the keyword LOCAL is *not* specified, then Hive will apply some file location algorithm and move the file specified by the *filepath* parameter into Hive
- If the OVERWRITE keyword is present, then the contents of the target table (or partition) will be overwritten by the file referred to by *filepath*
- If the OVERWRITE keyword is *not* present, the source file's contents is added to the target table

7.3 Example of Loading Data into a Hive Table

```
hive>  LOAD DATA
       LOCAL INPATH './data/source.dat'
       OVERWRITE INTO TABLE tblTarget;
```

7.4 Loading Data with the INSERT Statement

- The INESRT statement allows inserting data from SELECT-based queries
- The INSERT statement has two variants:
  - `INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement;`
  - `INSERT INTO TABLE tablename1 [PARTITION (partcol1=val1, partcol2=val2 ...)] select_statement1 FROM from_statement;` *(available as of version 0.8)*

7.5 Appending and Replacing Data with the INSERT Statement

- The *INSERT OVERWRITE* variant will overwrite any existing data in the table or partition
- The *INSERT INTO* variant will append to the table or partition
- Data inserts with the INSERT statement can be done to a table or a partition

7.6 Examples of Using the INSERT Statement

```
INSERT INTO TABLE Q1 SELECT sales FROM JanSales;
INSERT INTO TABLE Q1 SELECT sales FROM FebSales;
INSERT INTO TABLE Q1 SELECT sales FROM MarSales;
```

- The above three INSERT statements will append data from three different tables to the Q1 table

7.7 Multi Table Inserts

- Multi Table Inserts is an optimization technique that helps minimize the number of data scans on the source table
  - The source data is scanned only once and it is then re-used as input for building different queries which produce result sets for multiple INSERT statements

7.8 Multi Table Inserts Syntax

- Multiple Table Inserts work for both data appending and data overwriting

```
FROM from_statement
INSERT OVERWRITE TABLE tablename1 [PARTITION (partcol1=val1,
partcol2=val2 ...)] select_statement1
[INSERT INTO TABLE tablename2 [PARTITION ...] select_statement2] ...;
```

  - **Note**: The Multi Table statement starts with the *FROM* keyword and ends with the ';'
- The *FROM from_statement* clause causes a full scan of the source data
  - The *from_statement* can be the name of a Hive table or a JOIN statement
  - The scanned result returned by the *FROM from_statement clause* can now be re-used for multiple table inserts

7.9 Multi Table Inserts Example

```
FROM sourceTable s
  INSERT OVERWRITE TABLE destTable1 SELECT s.col1 WHERE s.col1 != 999
  INSERT INTO TABLE destTable2 SELECT s.* WHERE s.col2 < 0
```

- **Note**: There is no *FROM* clause in the *INSERT* statements that follow the *FROM* clause

7.10 Summary

- Hive supports data loading into tables and partitions using two statements:
  - LOAD DATA
  - INSERT
- Multi Table inserts is an optimization technique that requires only one data source scan which can be subsequently re-used for query-based data insertion into multiple tables

Chapter 8 - Hive Select Statement

Objectives

In this chapter, participants will learn about

- Hive Query Language (HiveQL)
- SELECT and related statements
- HiveQL built-in functions

8.1 HiveQL

- For data query, Hive uses an SQL-like language called HiveQL
- While many of the language constructs resemble SQL-92, HiveQL does not claim full standard compliance
- HiveQL offers Hive-specific extensions that help leverage Hive internal architecture
- Hive offers only basic support for indexes, so, in most cases, a full table scan is performed when a query is run against a table

8.2 The SELECT Statement Syntax

- The center-piece of HiveQL is the SELECT statement which has the following syntax:

```
SELECT [ALL | DISTINCT] select_expr1, select_expr2, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[LIMIT number]
```

8.3 The WHERE Clause

- Hive closely models the SQL's WHERE clause syntax
- As of Hive 0.13, some types of subqueries are supported in the WHERE clause
- The WHERE clause supports a number of Hive operators and user-defined functions that evaluate to a boolean result

8.4 Examples of the WHERE Statement

- For example:

```
SELECT … FROM … WHERE X <= Y AND X != Z;
SELECT … FROM … WHERE round(X) = Y;
```

  - Using Hive's *round()* built-in function

```
SELECT … FROM … WHERE X IS NULL;
SELECT … FROM … WHERE X IS NOT NULL;
SELECT … FROM … WHERE X LIKE 'A%'
```

  - Finding all rows where values in the X column start with 'A'

```
SELECT … FROM … WHERE X LIKE '%Z'
```

  - Finding all rows where values in the X column end with 'Z'

8.5 Partition-based Queries

- Generally, HiveQL's SELECT query performs a full table scan (a highly inefficient process) when finding the matching rows
- Hive offers a query optimization technique bases on table partitioning
- Partitions are column-based and are created using the PARTITIONED BY clause of the CREATE TABLE statement
- SELECT statements that take advantage of existing table partitions help reduce amount of data to be scanned

8.6 Example of an Efficient Use Of Partitions in SELECT Statement

- Provided that the *stats* table is partitioned by the *date* field, the following select statement will speed up queries by limiting the amount of data to be scanned (it is assumed that there are data partitioning directories named *2014-01-01, 2014-01-02, …, 2014-02-01, 2014-02-02, ... 2014-12-31*):

```
SELECT *
FROM stats
WHERE date > '2014-05-01' AND date <= '2014-09-01'
```

8.7 Create Table As Select Operation

- You can create a table and populate it with records from the source table in a single HiveQL command:

```
CREATE TABLE foo_Copy AS SELECT * FROM foo_Source;
```

8.8 Supported Numeric Operators

- HiveQL supports the standard set of arithmetic operations on numeric types in the SELECT statement

Operator	Description
A + B	Addition of A and B (e.g. SELECT principal + interest FROM loans;)
A - B	Subtraction of B from A
A * B	Multiplication of A and B
A / B	Division of A with B
A % B	The remainder of dividing A with B (modulo operation)
A & B	Bitwise AND of A and B
A | B	Bitwise OR of A and B
A ^ B	Bitwise XOR of A and B
~A	Bitwise NOT of A

- Arithmetic operators take any numeric type
- No type coercion (casting) is performed if both operands are of the same numeric type
  - Otherwise, the value of the smaller of the two types is promoted to the wider type (with more allocated bytes) of the other operand

8.9 Built-in Mathematical Functions

- HiveQL supports the usual set of mathematical functions in its SELECT statement
  - *round(), floor(), ceil(), exp(), log(), sqrt(), sin(), cos(), etc.*
- Most mathematical functions return a DOUBLE or a NULL if a NULL parameter is passed to the function
- For a complete list of mathematical functions, visit https://cwiki.apache.org/confluence/display/Hive/LanguageManual

8.10 Built-in Aggregate Functions

Function	Description
AVG(col)	Return the average of the values in column col
AVG(DISTINCT col)	Return the average of the distinct values in column col
COUNT (*)	Return the total number of retrieved rows, including rows containing NULL values
SUM(col)	Return the sum of the values in column col
SUM(DISTINCT col)	Return the sum of the distinct values in column col
MAX(col)	Return the maximum value in column col
MIN(col)	Return the minimum value in column col

- **Note:** All functions return their results as a DOUBLE except for COUNT() which returns a BIGINT

8.11 Built-in Statistical Functions

Function	Description
CORR (col1, col2)	Return the correlation of two sets of numbers
COVAR_POP (col1, col2)	Return the covariance of a set of numbers
STDDEV_POP(col)	Return the standard deviation of a set of numbers

- **Note:** All statistical functions return result as a DOUBLE
- For finding the mean of a population, use the AVG() function

8.12 Other Useful Built-in Functions

Function	Return Type	Description
INSTR(str, substr)	INT	Return the index of str where substr is found
LENGTH(s)	INT	Return the length of the string
REPEAT(str, n)	STRING	Repeat str n times
SPACE(n)	STRING	Returns n spaces
YEAR(timestamp)	INT	Return the year part as an INT of a timestamp string, e.g., year("2014-01-31 00:00:00") returns 2014
MONTH(timestamp)	INT	Return the month part as an INT of a timestamp string, e.g., month("2014-01-31 00:00:00") returns 1
DAY(timestamp)	INT	Return the day part as an INT of a timestamp string, e.g., day("2014-01-31 00:00:00") returns 31
TO_DATE(timestamp)	STRING	Return the date part of a timestamp string, e.g., to_date("2014-01-31 00:00:00") returns '2014-01-31'
col IN(val1, val2, …)	BOOLEAN	Return true if col equals one of the values in the list (val1, val2, ...), false otherwise

8.13 The GROUP BY Clause

- The GROUP BY statement is normally used in conjunction with the aggregate functions to group the result-set by one or more columns
- The SELECT statement performs an aggregation over each group

```
SELECT sales_month, SUM(sales) FROM sales_2014
    WHERE  city = 'Toronto' AND price >= '1000'
    GROUP BY sales_month;
```

8.14 The HAVING Clause

- The HAVING clause was added to HiveQL in ver. 0.7 because the WHERE keyword cannot be used with aggregate functions (as per the SQL spec)
- The following SELECT statement will list all months' total sales in Toronto where the total sales per month were in excess of 1 million dollars

```
SELECT sales_month, SUM(sales) FROM sales
    WHERE  city = 'Toronto'
    GROUP BY sales_month
    HAVING SUM(sales)> 1000000;
```

8.15 The LIMIT Clause

- The LIMIT clause sets the number of rows to be returned
- **Note**: The rows that will be returned are randomly chosen. Repeating this command may yield different results (different rows may be returned)

```
SELECT * FROM myTable LIMIT 3
```

8.16 The ORDER BY Clause

- The ORDER BY syntax in Hive QL is similar to that of ORDER BY in SQL language
- The ORDER BY supports ascending and descending ordering represented by the ASC (default) and DESC keywords respectively, e.g.

```
SELECT * FROM myTable ORDER BY col1 DESC;
```

8.17 The JOIN Clause

- HiveQL supports the SQL-like JOIN clause which is used to combine rows from two or more tables on values from a common column
- Only equality joins are allowed where the equal condition is used (`t1.col1 = t2.col2`)
- Examples:

```
SELECT A.* FROM A JOIN B ON (A.col1 = B.col2)
```

- More than 2 tables can be joined in the same query, e.g.:

```
SELECT A.col1, B.col2, C.col3
FROM A JOIN B ON (A.id1 = B.id2)
JOIN C ON (C.id3 = B.id4)
```

8.18 The CASE … Clause

- HiveQL supports the if-like CASE … WHEN … THEN … ELSE combined statement which has the following syntax:

```
SELECT col1, … ,
     CASE
      WHEN <condition A> THEN <label 1>
      [WHEN <condition B> THEN <label 2>, WHEN …]
      ELSE <label N>
     END AS <pseudo_column_name>
FROM table_name;
```

8.19 Example of CASE … Clause

```
hive> SELECT location, dateTime,
    > CASE
    > WHEN speedMPH > 40   AND speedMPH < 73 THEN 'Light'
    > WHEN speedMPH => 73  AND speedMPH < 113 THEN 'Moderate'
    > WHEN speedMPH => 113 AND speed < 158 THEN 'Considerable'
    > WHEN speedMPH => 158 THEN 'Severe'
    > ELSE 'Unknown' END AS category FROM Tornadoes;
```

8.20 Summary

- HiveQL offers SQL-like SELECT statement with the related WHERE, GROUP BY, and HAVING clauses
- HiveQL supports the common set of
  - Numeric Operators (+ / - , %, etc.)
  - Built-in Aggregate Functions (AVG(), SUM(), COUNT(), etc.)
  - Assorted functions (INSTR(), LENGTH(), YEAR(), etc.)
- In addition, HiveQL has a number of built-in statistical functions (CORR(), STDDEV_POP(), etc.)
- The CASE … WHEN … THEN … ELSE combined statement is also supported

Chapter 9 - Apache Sqoop

Objectives

In this chapter, participants will learn about

- Apache Sqoop, a tool for data transfer between relational databases and Hadoop
- How to control the performance of import and export processes
- Sqoop's supported file formats

9.1 What is Sqoop?

- Sqoop is a command-line tool for efficient two-way data transfer between relational databases and the Hadoop Distributed File System (HDFS)
- Hadoop-based systems, such as Hive and HBase can leverage Sqoop's data load facilities
- Sqoop is written in Java
- It was successfully graduated from the Apache Incubator in March of 2012
- Sqoop comes bundled with a number of connectors to popular databases and data warehousing systems
  - If needed, developer can use Sqoop's API for development of new system connectors

9.2 Apache Sqoop Logo

9.3 Sqoop Import / Export

- In Sqoop's terminology,
  - *import* refers to extracting data from a relational database and loading it into HDFS
  - *export* is the reverse operation, where data is extracted from HDFS and inserted into a relational database
- From the relational databases' point of view, the above operations are the opposite:
  - Sqoop's import operation is seen as the database export operation
  - The export operation (extraction of from HDFS into the target relational database) is seen as the import operation
- Sqoop uses MapReduce to import and export the data that offers parallel task execution and fault tolerance (failed tasks are transparently restarted)

9.4 Sqoop Help

- To list all available commands in Sqoop, run the *scoop help* command which produces the following (shortened version) output:

```
$ sqoop help
. . .
  eval                Evaluate a SQL statement and display the results
  export              Export an HDFS directory to a database table
  import              Import a table from a database to HDFS
  import-all-tables   Import tables from a database to HDFS
  list-databases      List available databases on a server
  list-tables         List available tables in a database
  version             Display version information
```

9.5 Examples of Using Sqoop Commands

- In examples below, the *jdbc:mysql://localhost/[DB_name]* string is a sample JDBC (Java Database Connectivity) connect string to MySQL relational database
- ***sqoop list-databases*** lists the available database schemas, e.g.

```
$ sqoop list-databases --connect jdbc:mysql://localhost
```

- ***sqoop list-tables*** lists the tables in the specified database schema, e.g.

```
$ sqoop list-tables  --connect jdbc:mysql://localhost/myDB
```

- ***sqoop import*** imports a table from a database to HDFS, e.g.

```
sqoop import --connect jdbc:mysql://localhost/myDB --table t1
```

- ***sqoop eval*** is Sqoop's primitive SQL execution shell which you can use as follows:

```
$ sqoop eval  --connect jdbc:mysql://localhost/myDB -e 'SELECT * FROM
myTable;'
```

9.6 Data Import Example

- The following import command will create the target file in the default text format (the *--as-textfile* argument is omitted) with comma-separated fields

```
$ sqoop import --connect … --table myTable --target-dir /user/me/myTable --fields-terminated-by ','
```

9.7 Fine-tuning Data Import

- By default, Sqoop uses the *SELECT * FROM **myTable*** command when importing data with the *sqoop import …. --table **myTable*** command
- You can limit the number of rows to be imported by providing the WHERE clause with the *--where* import control argument:

```
--where <where clause>, e.g.  --where "city= 'Toronto'"
```

- Similarly, you can control the choice of columns that you may want to include in the data import; for that use the *--columns* import control argument:

```
--columns <col1 [,col2,col…]>
```

9.8 Controlling the Number of Import Processes

- Sqoop imports data in parallel from most relational databases
- By default, it uses four concurrent import processes (Hadoop map tasks)
- You can explicitly specify the number of the import parallel processes by specifying the value to the ***-m*** argument of the import command, e.g.
  - -m 8
- **Note**: Do not set this number too high (definitely, it should not be greater than the number of map tasks available within your Hadoop cluster), a value between 8 and 16 may be optimal

9.9 Data Splitting

- When running parallel import processes, Sqoop tries to horizontally split source table data into chunks of the same size in order to evenly distribute workload across import processes
- By default, Sqoop tries to use the source table's primary key column's values to identify split points
- If the values in the primary key column are not uniformly distributed, Sqoop may create uneven workload across the processes; to avoid this, you should explicitly choose a more suitable "splitting" column and using it as a value to the *--split-by* argument

9.10 Helping Sqoop Out

- You can help Sqoop perform better by letting it leverage Hadoop's built-in processing parallelism (on top of the *-m* configuration parameter)
- You can do so by specifying the following attributes of the import statement:
  - The ***$CONDITIONS*** token (which is just a Sqoop's placeholder in the WHERE clause; Sqoop will replace it with a unique condition expression per process to help partition the data to fetch)
  - ***--target-dir*** followed by HDFS destination directory (normally, it has the same name as the table which data is being imported)
  - ***--split-by*** followed by the column name that will be used to split the data for parallel load

9.11 Example of Executing Sqoop Load in Parallel

- The data import (load) optimized command syntax looks as follows:

```
$ sqoop import \
  --query 'SELECT * FROM T WHERE $CONDITIONS' \
  --split-by T.id --target-dir /user/me/T   -m 8
```

- **Note**: The above free-form query is enclosed in single quotes, if you want to use double quotes instead (""), then you will need to escape the Sqoop's optimization token: **** $CONDITIONS, e.g.

```
...  --query "SELECT * FROM T WHERE \$CONDITIONS" ...
```

9.12 A Word of Caution: Avoid Complex Free-Form Queries

- You can combine your WHERE clause with the Sqoop's *$CONDITIONS* placeholder, e.g.:

```
… 'SELECT * FROM T WHERE col1=1024 AND $CONDITIONS' …
```

- **Note**: Avoid using complex queries such as queries that have sub-queries and using the ***OR*** condition as it may lead to unexpected results; the ***AND*** condition is fine

9.13 Using Direct Export from Databases

- By default, Sqoop uses cross-platform JDBC (Java Database Connectivity) statements to import data
- Many databases also support native import/export tools which perform better than JDBC
  - For example, MySQL provides the *mysqldump* tool for exporting (dumping) data from MySQL
- You can override the default JDBC-based import by supplying the *--direct* import command argument (Scoop will use the appropriate database data export tool transparently)

9.14 Example of Using Direct Export from MySQL

- The following command will use MySQL's default export utility *mysqldump* (you don't need to specify its name as Sqoop is intelligent enough to know how to use it)

```
$ sqoop import --connect jdbc:mysql://<IP>/mydb \
              --table T  --direct
```

9.15 More on Direct Mode Import

- By default, data from the source table will go to a new HDFS directory
  - If the destination directory already exists, Sqoop will abort the import operation
- Use the *--append* command argument to append the data
- In direct mode, you can specify arguments that are passed to the export utility directly; use the *-- argument* for that:

```
--direct -- --character-set-server=utf8
```

9.16 Changing Data Types

- By default, Sqoop transparently maps most SQL types to appropriate Java or Hive counterparts
- You have an ability to override the default mapping using the following parameters:
  - *--map-column-java <mappings>*
  - *--map-column-hive <mappings>*
- The <*mappings*> is a comma separated list of mappings in the form of *name_of_column=target_type [,name_of_column2=target_type2]*

9.17 Example of Default Types Overriding

- Changing the type of *col1* (e.g. it may be *varchar*) to Java's *Integer* type:

```
$ sqoop import ... --map-column-java col1=Integer
```

- Sqoop will throw an exception in case of failed mapping operation

9.18 File Formats

- Sqoop supports three imported file formats of the file created in HDFS after the import operation:
  - CSV (Character-Separated Values) text file format (default format, using comma as field separator)
  - SequenceFile binary format
  - Avro binary format
- You specify the import file format by passing the appropriate arguments:
  - *--as-textfile* argument (can be omitted as this is the default)
  - *--as-sequencefile*
  - *--as-avrodatafile*

9.19 Binary vs Text

- The CSV file format is used in cases where further data manipulation with other tools, such as Hive, is expected
- Processing files in the SequenceFile or Avro binary formats is more efficient than processing text files but requires custom MapReduce programs
  - The binary data is stored in the source databases in columns of the binary type, e.g. VARBINARY

9.20 Data Export from HDFS

- Sqoop supports data export from HDFS into relational databases
- It uses the *export* tool to exports a set of files
- The target table must already exist in the database
- The definition of the table must match the values in the source file(s)

9.21 Export Tool Common Arguments

- The export command has the following syntax:
  - *sqoop export arguments*
- The most common *export* arguments are:
- **--connect <jdbc-uri>** Specify JDBC connect string
- **--driver <class-name>** Manually specify JDBC driver class to use
- **--help** Print usage instructions
- **--password-file** Path to file containing the password
- **-P** Read password from console
- **--password <password>** Set authentication password
- **--username <username>** Set authentication username

9.22 Data Export Control Arguments

- Many data export control arguments match those used in the import operation
- The most common export control arguments are:
- **`--direct`** `Use direct export fast path`
- **`--export-dir <dir>`** `HDFS source path for the export`
- **`-m`** `Set the number of parallel export map tasks`
- **`--table <table-name>`** `Table to populate`
- **`--call <stored-proc-name>`** `Stored Procedure to call`
- **`--batch`** `Use batch mode for underlying statement execution`

9.23 Data Export Example

- The following sqoop export command will export data from the */user/me/export/* HDFS folder and insert it into a table named *foo* created in a MySQL database named *DBNAME*:
- **Note:** The target table must already exist in the database and its structure should match the data to be inserted

```
$ sqoop export --connect jdbc:mysql://server/DBNAME \
      --table foo  \
      --export-dir /user/me/export/
```

- Sqoop performs a set of INSERT INTO JDBC operations to populate the target table
- If any INSERT transaction fails (e.g. table constraints are violated), then the export will fails

9.24 Using a Staging Table

- During the data export process some of the parallel jobs may fail
- As a result, the target table may end up to be only partially populated
- In some scenarios, you may also end up with duplicate records
- Sqoop helps users avoid data export problems by offering the *--staging-table* option which designates a temp table from where data is inserted in the target table in a single database transaction
  - This temp table must have the structure identical to that of the target table
- **Note**: The staging temp table must exist (and be empty) prior to running the export job

9.25 INSERT and UPDATE Statements

- Sqoop uses a set of JDBC-based INSERT or UPDATE statements to make changes to the content of the target table
- In default mode, Sqoop generates the necessary INSERT statements to inject new rows into the target table
- In update mode, Sqoop generates UPDATE statements that update existing records in the target table

9.26 INSERT Operations

- By default, Sqoop export operation injects new rows into the target table
- Sqoop will generate a JDBC-based INSERT statement for each row from the source file
- In most cases, a new and empty table is used to receive the exported rows
- The export operation will fail if an exception is thrown in the underlying JDBC system (e.g. primary key constraint is violated, or there is type mismatch, etc.)

9.27 UPDATE Operations

- By specifying the *--update-key* argument to the export command, you can instruct Sqoop to modify existing rows in the target table
- In this case, row attributes are made part of an UPDATE statement that attempts to modify an existing row (if any found)
- If an UPDATE statement modifies no rows (there are no rows matching the WHERE clause), the export process continues without an error

9.28 Example of the Update Operation

- If you have an existing table with data having the following definition:

```
CREATE TABLE T (
    id INT NOT NULL PRIMARY KEY,
    tweet VARCHAR(140));
```

- and the source data file in HDFS with the following content:

```
1000,whatever ...
1001,whenever ...
```

- Then the following export command

```
$ sqoop export --table T --update-key id --export-dir
/path/to/datafile --connect …
```

- Will have Sqoop generate the following UPDATE statements

```
UPDATE T SET tweet='whatever ...' WHERE id=1000;
UPDATE T SET tweet='whenever ...' WHERE id=1001;
```

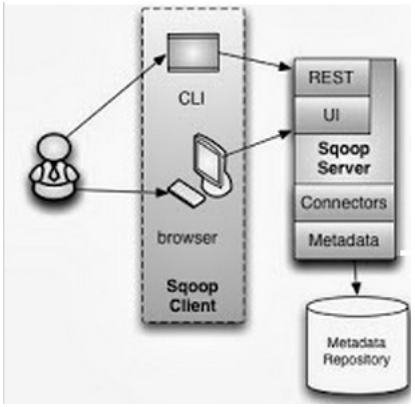
9.29 Failed Exports

- Exports may fail for any of these reasons:
  - Lost connectivity to the target database
  - Capacity issues (insufficient size quotas for file system partitions, etc.)
  - Violation of a database constraints (e.g. inserting a duplicate value in the column with a unique index constraint)
  - Corrupt data in the source data file in HDFS

9.30 Sqoop2

- Currently, in order to run Sqoop, all the Sqoop drivers and connectors must be installed on the client machine
- The next iteration of Sqoop, Sqoop2, is re-architected using a service based model where the Sqoop server is set-up remotely from the clients
- The new design helps with security and concurrent multi-user support
- Sqoop2 supports the Sqoop's CLI that is connected to the server via a RESTful end-point
- The project is still under development and not recommended for production use

9.31 Sqoop2 Architecture



Adapted from https://blogs.apache.org/sqoop/entry/apache_sqoop_highlights_of_sqoop

9.32 Summary

- Sqoop is a command-line tool for efficient two-way data transfer between relational databases and HDFS
- You can control the performance of import and export processes by allocating the number of parallel processes
- Data splitting by specifying the column to split the data by is a useful technique to help with Sqoop's performance
- Sqoop supports three imported file formats: two binary and one text
- For more information, visit http://sqoop.apache.org/docs/1.4.4/SqoopUserGuide.html

Chapter 10 - Apache HBase

Objectives

In this chapter, participants will learn about

- HBase distributed database system and its features
- How HBase compares with relational databases and Cassandra
- Client interfaces supported by HBase
- HBase table design
- HBase Interactive Shell

10.1 What is HBase?

- HBase (http://hbase.apache.org/) is a column-oriented, sorted map database management system written in Java that runs on top of HDFS
- Designed for hosting large tables of sizes in the petabyte (10^{15}) range, consisting of billions (10^{9}) of rows and millions (10^{6}) of columns
- Main use cases are an efficient handling of Big Data stored in the form of sparse data sets
- HBase supports fast random reads and writes in real time meeting low-latency requirements of online systems
- Integrates with MapReduce
- Support direct data querying using Java and other programming languages

10.2 HBase Design

- HBase is modeled after Google's Bigtable system
- "Just as Bigtable leverages the distributed data storage provided by the Google File System, Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS." [http://hbase.apache.org/]
- HBase's architecture is based on the master/slave design pattern
- HBase system has a Master node that manages the cluster and Region Servers (slaves) to store portions of the database (data is horizontally partitioned by rows)

10.3 HBase Features

- Linear scalability with automatic and configurable sharding (partitioning) of tables
- Strictly consistent reads and writes
  - HBase is not an "eventually consistent" datastore as most NoSQL Systems (see comparison with Apache Cassandra a bit later) - it is a strongly consistent data store
  - Strong (high) data consistency is ensured by routing all write operations for a key through the RegionServer made responsible for that key (it simplifies the system design and avoids last-write-wins scenarios)
    - If that RegionServer is lost, the entire key range owned by that RegionServer is made unavailable pending data recovery
  - High Availability properties with automatic fail-over between Region Servers are reviewed a bit later ...
- HBase achieves operational efficiencies through data compression, in-memory data processing and some other features originally published in Google's Bigtable paper in 2006
- Front cache for real-time queries

10.4 HBase High Availability

- An area of active design and development work
  - From the very beginning, HBase has been leveraging HDFS-based data replication
  - HBase supports its own data partitioning across nodes in the cluster
  - The two above features combined give a three 9s (99.9%) data availability
- The next iteration in delivering HBase HA is called *HBase Read HA*, where a failed read from one (Primary) RegionServer can be failed over to another (Secondary) RegionServer
  - Number of secondary RegionServers is configurable via the *region replication factor* which is usually set to 2
  - This is an automatic recovery process during which HBase loses write availability and data for reads may be stale (as only the Primary RegionServer accept writes and keeps the latest data)
  - With a *region replication factor* of 2, *HBase Read HA* delivers four 9s (99.99%) data availability
- For more information, see http://hortonworks.com/blog/apache-hbase-high-availability-next-level/

10.5 The Write-Ahead Log (WAL) and MemStore

- When HBase writes data to a table (the *write path*), it takes extra effort to ensure data durability by writing data into two paces:
  - The Write-Ahead Log (WAL), a.k.a. the HLog (one per server), and
  - The MemStore (a Java heap-based write buffer, one per column family)
- The write operation to the table is considered complete when writes to both places (WAL and MemStore) succeed

10.6 HBase vs RDBS

- HBase is not a relational database system and it is not positioned as a direct replacement for relational databases
  - HBase is a "data store" rather than a "database"
- HBase does not support a declarative query language like SQL
  - Data access is performed imperatively through the client-side API
  - It lacks column types (byte arrays are used instead), triggers, and other common features of a RDBMS
- Traditional relational databases are not inherently distributed and don't scale out easily
  - HBase is a distributed persistence store by design that uses the storage, memory and CPU resources of nodes in a “cluster” scaling out as the data size increases
  - HBase leverages HDFS for data durability (data is transparently replicated, 3-way by default)
- HBase does not support the full set of ACID properties

10.7 HBase vs Apache Cassandra

- Apache Cassandra is an open source distributed NoSQL database management system similar to HBase
- HBase leverage the Hadoop infrastructure (HDFS, Zookeeper, etc.); Cassandra has its own infrastructure (different Ops skill sets are required)
- HBase is a strongly consistent NoSQL system (through explicit row lock), Cassandra is eventually consistent
  - Cassandra does supports a tunable level consistency for writes and reads from "writes never fail" to "block for all replicas to be readable"
- Cassandra has its own SQL-like declarative Query Language
- Cassandra can have more than one Write Master
- HBase is more optimized for reads, Cassandra is optimized for writes
- HBase supports cell value versioning, Cassandra does not
- In Cassandra cluster, all nodes are equal (peer-to-peer); HBase has Master Node and Region Servers (master/slave) with distributed coordination provided by the Zookeeper system
- Hive can access HBase tables directly, Cassandra is catching up

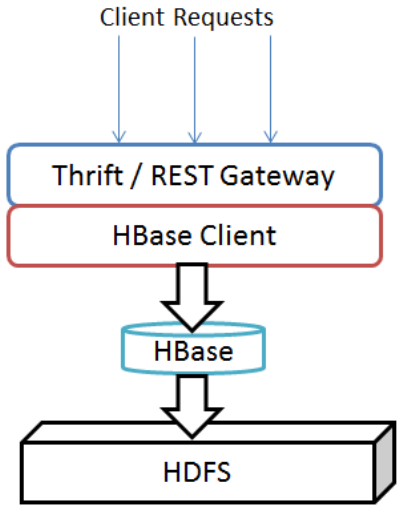
10.8 Not Good Use Cases for HBase

- Relatively small number of rows (under a few million) where a RDBMS is a better choice
  - The size threshold would be in the region of hundreds of millions or billions of records
- Situations where you need to have the features of a common RDMS: ACID properties, column types, triggers, etc.
- Limited number of physical nodes, e.g. 5 : limitation imposed by HDFS that requires the default 3-way replication and a dedicated NameNode machine

10.9 Interfacing with HBase

- HBase client applications are mostly written in Java using HBase Java client API
  - **Note**: There is no JDBC driver for HBase, you have to operate on HBase Java classes
- The HBase Shell
- In addition to Java API, HBase supports the Avro serialization, the Thrift and RESTful API (Stargate) gateways

10.10 HBase Thrift And REST Gateway



10.11 HBase Table Design

- HBase uses its own *hFile* storage format
- HBase is a quasi-schemaless system
  - The system designer must define the table schema and column families upfront
- An HBase system consists of a set of tables made of rows and columns that hold objects
  - A column holds an entity's attribute
- The first column must be allocated for storing a unique row identifier defined as the Primary Key (this unique identifier is called the *rowkey* or *rowid*)
- Table cells (the intersection of rows and columns) are versioned (by default, three versions are kept) with a timestamp assigned by HBase, or programmatically by the client, written alongside the value
- A cell’s content is stored as an uninterpreted array of bytes
- Table rows are sorted by the table’s *rowkey*
  - *So,* the *rowkey* can also be viewed as the table's index
- All table accesses are via the table's *rowkey*
- Row updates are atomic

10.12 Column Families

- Following the Google's Bigtable design, HBase allows for grouping entity's attributes (columns) into column families
- All column family members have a common prefix
  - For example, the column family *engine* may include these two columns (a column in HBase is referred to as the "*column qualifier*"): *engine:hp and engine:cyl*
- Column families must be specified up front as part of the table schema definition
- New column family members can be added to predefined column families at run-time
  - For example, a new column *engine:weight* can be added to the existing engine column family
  - This feature of HBase helps accommodate agile business requirements
- Column families help with entity storage and retrieval optimization

10.13 A Cell's Value Versioning

- Changes of a cell's value are versioned by a timestamp
- By default, HBase stores the last three versions of a cell's value
  - This is parameter is configurable per column family
- You have access to the version history and can fetch element from previous versions by supplying the version's timestamp

10.14 Timestamps

- By default, HBase uses the current time in milliseconds since Unix epoch time (00:00:00 UTC on 1 January 1970)
- The current time is captured on the RegionServer that performs an operation that changes the cell value's version
- For the timestamp facility to work properly, system clocks on all machines in the HBase cluster must be in sync

10.15 Accessing Cells

- HBase uses a cell coordinate (cell path) to locate a cell in a table
- A cell coordinate is made up as follows:
  - The table name, followed by the primary key (*rowkey*), followed by the column family, followed by the column qualifier (column family member) and, optionally, the timestamp, or
  - *table_name/rowkey/column/[timestamp]*

10.16 HBase Table Design Digest

- Rows are sorted by the primary key (*rowkey*)
- Cell values are accessed using the "*table_name/rowkey/column/(and optionally) timestamp"* positioning coordinates
- Columns can be added to existing column families at run-time
- Cell values are typeless and are always treated as byte arrays (leaving the type casting exercise to client applications)

10.17 Table Horizontal Partitioning with Regions

- HBase tables are automatically partitioned horizontally (by rows) when a configurable size threshold is passed
- A table partition (containing a chunk of table rows) is referred to as a region
- A table's initial storage consists of a single region with new regions added as the number of rows crosses a configurable size threshold
- Regions get distributed over an HBase cluster

10.18 HBase Compaction

- HBase provides optimal read performance by having only one file per column family which minimizes the number of disk seek reads
- Using some internal heuristics (regulated by compaction policies), HBase finds the good candidates (hFiles) and combines them into one contiguous file on disk
- This "merging" process is called compaction
- This is a "work-in-progress" and the compaction policies are subject to change

10.19 Loading Data in HBase

- The recommended way to load data from HDFS into HBase is with a bulk loader
- The bulk loading process prepares and loads data in the *hFiles* storage format (HBase's own file format) directly into the RegionServers
  - The process bypasses the standard write path involving the WAL and MemStore which makes it very efficient
- HBase comes with its own MapReduce-based bulk loader called *importtsv* that can read CSV-delimited files stored in HDFS and import them into HBase
- After this, you can use HBase against the imported data

10.20 Column Families Notes

- In most of situations, you should have only one column family in your table
- More than three column families would degrade performance of your system
- If you have more than one column family (two or three), try to limit data access patterns to one column family at a time
  - That will minimize the number of disk seek reads and make an efficient use of the MemStore (a Java heap-based write buffer, allocated one per column family)

10.21 Rowkey Notes

- Usually, the row key ids are are sorted lexicographically to improve the scan (sequential access) of the table
- This arrangement can result in *hotspotting* where a single physical machine takes all the user traffic for reads/writes while other machines in the cluster that host other data regions may sit idle
- If you foresee random access to your data, you can use the following techniques / tricks to ensure a fair distribution of key values across multiple regions on writes (and subsequent reads):
  - **Salting**: randomly (kind of round-robin way of) adding a specially designed prefix to the key
  - **Hashing**: delegating the random key allocation to a one-way hash function
  - **Reversing** the fixed-width key: making the least significant digit (e.g. **1** in 987654321) appear first
  - **Note:** These techniques require deep understanding of effective management of data regions in HBase and the topic is beyond the scope of this module

10.22 HBase Shell

- HBase comes with the JRuby-based shell which acts as a command-line interface to the underlying HBase client Java API
- You start the shell with this command:

  `hbase shell`

- You can run it in either interactive or unattended (batch) mode
  - HBase scripts are JRuby files (that interface with Java HBase classes) that you execute as follows:

```
${HBASE_HOME}/bin/hbase org.jruby.Main <script.rb> args
```

10.23 HBase Shell Command Groups

- HBase shell commands are organized into groups some of which are shown in the table below

Command Group Name	Commands in the Group
general	status, table_help, version, whoami
ddl	alter, alter_async, alter_status, create, describe, disable, disable_all, drop, drop_all, enable, enable_all, exists, get_table, is_disabled, is_enabled, list, show_filters
dml	count, delete, deleteall, get, get_counter, incr, put, scan, truncate
security	grant, revoke, user_permission

Some HBase Shell Command Groups

- You can get help on each group by running this command: *help <group name>*, e.g. *help 'general'*
- **Note**: All names in HBase shell must be used in quotes

10.24 Creating and Populating a Table in HBase Shell

```
create 'myTable', 'myColFam'
```

- The above command creates the *myTable* table with the *myColFam* column family

```
put 'myTable', 'row1', 'myColFam:col1', 'Some value'
put 'myTable', 'row2', 'myColFam:col2', 'Some other value'
```

- The above commands insert *'Some value'* in a cell located at *myTable/row1/myColFam:col1* and *'Some other value'* in a cell located at *myTable/row2/myColFam:col2* respectively

- **Note**: You can optionally put your own timestamp when inserting values with the *put* command:

```
put '<tableName>', 'rowid', 'colid', 'cell value', <your timestamp>
```

- Since HBase is a schemaless storage system, you never need to predefine the column qualifiers upfront or assign them types - just provide the name at write time

10.25 Getting a Cell's Value

- The following command will retrieve a row identified with the *row1* rowkey from the *myTable* table

```
get 'myTable', 'row1'

   COLUMN       CELL
   myColFam:col1    timestamp=<...>, value=Some value
```

- If the row identified with the rowkey does not exist, you will get back an empty response

10.26 Counting Rows in an HBase Table

- The *count* command (the *dml* group) returns the number of rows in a table

```
count '<tablename>', CACHE => 1024
```

  - **Note**: **=>** is the assignment operator (**=**)
- The above count command fetches 1024 rows at a time to speed up row counting

10.27 Summary

- In this chapter we covered a number of topics related to HBase
  - HBase is designed for hosting tables of sizes in the petabyte (10^{15}) range
  - It offers linear scalability with automatic and configurable partitioning of tables with strictly consistent reads and writes
  - HBase is a schema-free database with data stored as byte arrays
  - Table rows are identified and accessed by their rowkey ids which act as a primary key
  - HBase offers a rich Java client API; access from other languages is also supported via the Thrift or RESTful gateways
  - HBase also offers an interactive shell as a command-line interface

Chapter 11 - Apache HBase Java API

Objectives

In this chapter, participants will learn about some of the elements of Java API for HBase:

- The ResultScanner Interface
- The Scan Class
- The KeyValue Class
- The Result Class
- The Cell Interface
- The Bytes Utility Class

11.1 HBase Java Client

- HBase is written in Java which is also its primary application client
- An HBase Java client is a regular Java app with the *main()* method
- You use HBase-specific classes to read the HBase configuration and set up HBase operations (put, get, scan, etc.)

11.2 HBase Scanners

- HBase scanners work like Java iterators that orderly go over all qualifying rows in a table
  - You may think of scanners as RDBMS cursors
- scanners' functionality is offered through the *org.apache.hadoop.hbase.client.ResultScanner* Interface
- The *ResultScanner* works in tandem with the *org.apache.hadoop.hbase.client.Scan* Java class that helps limit the number of rows to scan

```
Scan scan = new Scan();// or other constructors
ResultScanner scanner = table.getScanner(scan);
```

- After that you can scan through the table:

```
for (Result scannerResult: scanner) {
// get the next row (may be inefficient, more on this later)
  System.out.println("Scaned: " + scannerResult);
}
```

11.3 Using ResultScanner Efficiently

- When you obtain a reference to an instance of *ResultScanner*, you have the choice of its two *next()* methods to iterate over the data cursor making RPC calls to the server to fetch next batch of data:

```
Result next() - Fetch the next row's worth of values
Result[] next(int nbRows) - Fetch nbRows's worth of rows
```

- A call to *next()* results in a network round-trip to the RegionServer holding the data; a *next(int)* call grabs the specified number of rows on the server, cutting on the number of round-trips required to scan the whole data set

11.4 The Scan Class

- The *org.apache.hadoop.hbase.client.Scan* class has constructors that help specify row ranges to return for scanning thereby acting as a row filter:
  - **Scan()** - configures a Scan operation across all rows in the table
  - **Scan(byte[] startRow)** - configures a Scan operation starting at the specified row
  - **Scan(byte[] startRow, byte[] stopRow)** - creates a Scan for the range of rows specified
  - **Scan(byte[] startRow, Filter filter)** - creates a Scan with a starting row and a column filter directly applied within the RegionServer

11.5 The KeyValue Class

- The *org.apache.hadoop.hbase.KeyValue* class represents the Key/Value fundamental type in HBase stored in a cell
- The *KeyValue* class has a wide range of constructors to build instances of the class with specific values for column family, qualifiers, timestamps, etc.
- The class offers a rich API for interrogating the target cell's properties, e.g.
  - **getTimestamp()** - get the latest cell's value
  - **getRow()** - get the row that comprises this cell
  - **getValue()** - get cell's value

11.6 The Result Class

- An instance of the *org.apache.hadoop.hbase.client.Result* class represents a single row result of a Get or Scan (via *ReturnScanner*) operation
- You get access to a specific column in the current row wrapped up in *Result* as follows:

```
public List<KeyValue> getColumn(byte[] family, byte[] qualifier)
```

  - The list of *KeyValues* holds the available versions of the cell's value which are sorted with the first element in the list referencing the most recent version of the cell

11.7 Getting Versions of Cell Values Example

```
// get the list of versions of the ColFam:c1 column in the current row
List<KeyValue> versions =
 result.getColumn(Bytes.toBytes("ColFam"), Bytes.toBytes("c1"));

// Current cell value's version
byte [] b = versions.get(0).getValue();
String currVersion = Bytes.toString(b);

// Previous cell value's version
b = versions.get(1).getValue();
String currVersion = Bytes.toString(b);
```

11.8 The Cell Interface

- The *org.apache.hadoop.hbase.Cell* is a new interface in HBase that can be obtained from an instance of the *org.apache.hadoop.hbase.client.Result* class during a table scan
- *Cell* provides access to the HBase's cell returned during HBase querying for the following cell attributes:

```
row
column family
column qualifier
timestamp
value
```

  - and others

11.9 HBase Java Client Example

```
// Programmatically create a table
    Configuration hConfig = HBaseConfiguration.create();
    HTable table = new HTable(hConfig, "MyTableName");
    byte [] r1 = Bytes.toBytes("row1"); // the rowkey

// Object for holding data to be inserted into the table
    Put p1 = new Put(r1);
    byte [] cf = Bytes.toBytes("columnFamily");

// building  /r1/columnFamily:col1/  = "Some Value" path
    p1.add(cf, Bytes.toBytes("col1"), Bytes.toBytes("Some Value"));

// Insert (put in) the data; You also update data with put
    table.put(p1);

// Read the data back with the Get object passed to the get method of table
     Get getValue = new Get(r1);
    Result result = table.get(getValue);
```

11.10 Scanning the Table Rows

```
// Reading the whole table, row-by-row in a cursor-like manner
    Scan scan = new Scan(); // No-arg constructor - read all rows

    ResultScanner scanner = table.getScanner(scan);
    try {
      for (Result scannerResult: scanner) {
        System.out.println("Scan: " + scannerResult);
      }
    } finally {
      scanner.close(); // Don't forget to close the scanner!!!
    }
```

11.11 Dropping a Table

```
//  Configuration hConfig = HBaseConfiguration.create();
//  HTable table = new HTable(hConfig, "MyTableName");

    HBaseAdmin admin = new HBaseAdmin(hConfig);

// The table must be disabled before it can be droppped
    admin.disableTable(tablename);
    admin.deleteTable(tablename);
```

11.12 The Bytes Utility Class

- HBase's Java API comes with the *org.apache.hadoop.hbase.util.Bytes* utility class that offers utility methods for handling byte arrays, conversions to/from other types, comparisons, etc.
- Some of the utility methods are:
  - Type conversion (used to get the value from a byte[]): *toFloat(), toDouble(), toInt(), toLong(), toString(), toShort()*
  - *Put* convenience methods (used to put a value as a byte[]): *putLong(), putInt(), putFloat(), putDouble(), putShort()*

11.13 Summary

- In this chapter we reviewed some of the elements of HBase's Java API:
  - The ResultScanner Interface
  - The Scan Class
  - The KeyValue Class
  - The Result Class
  - The Cell Interface
  - The Bytes Utility Class

Chapter 12 - Introduction to Apache Spark

Objectives

In this chapter, participants will learn about:

- The Apache Spark processing platform
- Features that make Spark faster than Hadoop MapReduce

12.1 What is Apache Spark

- Apache Spark (Spark) is a general-purpose processing system for large-scale data
- Spark is effective for data processing of up to 100s of terabytes on a cluster of machines
- Runs on both Unix-like systems and Windows
- Main players behind Spark:
  - Apache Software, Databricks (founded by the creators of Spark), UC Berkeley AMPLab, Cloudera, Yahoo, MapR, etc.
- The Spark Overview page: *http://spark.apache.org/docs/latest/*

12.2 A Short History of Spark

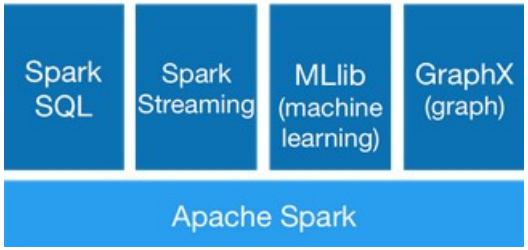
- Started as a research project at AMPLab at UC Berkeley in 2009
- First open sourced in 2010 under a BSD license; in 2013, Spark was donated to the Apache Software Foundation (ASF) and switched its license to Apache 2.0
- Currently (2015), Apache Spark is one of the most active projects in the Big Data open source space

12.3 Where to Get Spark?

- Apache Spark web site: http://spark.apache.org/
- Main Hadoop distributors: Cloudera (CDH), Hortonworks (HDP), MapR
  - Spark is usually added through forming partnerships with Databricks
- You can download and use Hadoop-free Spark bundles

12.4 The Spark Platform

- Spark includes specialized libraries for
  - SQL-type data processing (Spark SQL)
  - Machine learning tasks (MLlib)
  - Graph-parallel computation (GraphX)
  - Near-real time processing of streaming data (Spark Streaming)
- You can combine these libraries in the same Spark application



12.5 Spark Logo

12.6 Common Spark Use Cases

- Extract/Transform/Load (ETL) jobs
- Data Analysis
  - Text mining
  - Predictive analytics
  - User sentiment analysis
  - Risk assessment
  - Graph data analysis

12.7 Languages Supported by Spark

- The Spark platform provides API for applications written in Scala, Python, Java, and R
  - With the Spark SQL library, you can also create applications using SQL interface
  - The R API was added with Spark 1.4 and originally was marked as Experimental with limited support for Spark API
- Spark does not impose any limitations on supported run-times
  - For example, Python apps can call into existing Python C libraries
- Some additional libraries may be required to be installed
  - For example, Python API may require you to install the NumPy package
- Spark programs can query in-memory data repeatedly which makes Spark making an excellent platform for running machine learning algorithms

12.8 Running Spark on a Cluster

- Spark design is based on the master /slave architecture with a single master and a cluster of slave (worker) nodes
- Spark comes with its own (basic) cluster management system allowing it to be deployed as a stand-alone product
  - Supported run-times must be installed on every machine in the cluster
- For external cluster management requirements, Spark can be integrated with:
  - Hadoop YARN
  - The Apache Mesos cluster manager (also developed by UC Berkeley)
  - Amazon EC2 (via scripts)

12.9 The Driver Process

- A running Spark application process is called the "Driver"
- The Driver creates the **SparkContext** object referencing the Master node and acting as the entry point to Spark API
- The Driver process communicate with the Master for any required resources
- The Spark's Driver process is conceptually similar to YARN'a Application Master process

12.10 Spark Applications

- Spark applications are programs written in Spark-supported languages (currently, Python, Scala, Java, and R)
- Developing and running Spark applications require developers to attend to the following details:
  - Compiling applications and building the JAR (Java ARchive) file (this step only applies to Scala and Java -- not to Python or R)
  - Creating the *SparkContext* object
  - Submitting the application JAR file (or Python or R program) for execution

12.11 Spark Shell

- The Spark shell is a command-line interactive development environment which simplifies development work
  - The *SparkContext* object is configured and made available to the developer automatically when the shell starts
  - The shell executes every command as it is entered
  - The Spark shell only supports Python and Scala (Java is not supported yet)
    - The Python Spark Shell is launched by the **pyspark** command
    - The Scala Spark Shell is launched by the **spark-shell** command
    - SparkR shell was added recently to support development of R applications
    - SparkR Shell has only been thoroughly tested to work with Spark standalone so far and not all Hadoop distros available, and therefore is not covered here
- The shell is also referred to as **REPL** (Read/Eval/Print Loop)

12.12 The spark-submit Tool

- The **spark-submit** command-line tool is the standard way of submitting Spark applications for execution
- Submitting Scala and Java Spark application JAR file:

```
spark-submit <config options> --class <Your class with the main
method> <YourJarFile.jar> <The list of program arguments>
```

- Submitting Python Spark applications (regular Python programs):

```
spark-submit <config options>  <Your Python program> <The list of
program arguments>
```

12.13 The spark-submit Tool Configuration

- The main *spark-submit* configuration option is the **--master** parameter which specifies the cluster management master (if any) and may have the following values:
  - **local[*]** - your app will run locally using all CPU cores (the default value)
  - **local[n]** - your app will run locally using **n** threads (may be sub-optimal)
  - **<master URL>** - pointing to the cluster management master URL, e.g.
    - spark://<master IP>:7077 - a Spark native cluster
    - mesos://<master IP>:7077 - a Mesos cluster
    - yarn - a YARN cluster
- For example:

```
spark-submit --class was.labs.spark.JavaOnSpark
          --master spark://master.com:7077 JavaOnSpark.jar
```

  - **Note:** You can similarly to the above configure *spark-shell* and *pyspark* shells

- You can also reserve (aggregate) memory for your application using the **--executor-memory** configuration parameter, e.g.

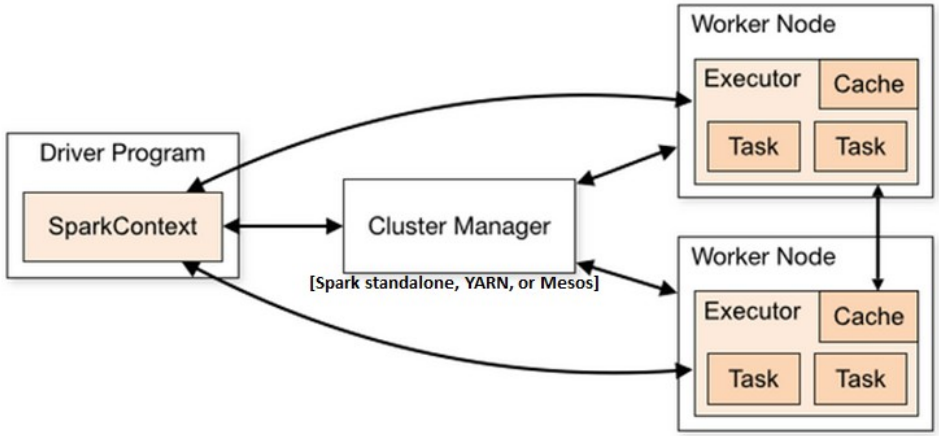
```
--executor-memory 32G
```

- For more details, visit *http://spark.apache.org/docs/latest/submitting-applications.html#master-urls*

12.14 The Executor and Worker Processes

- The Master allocates resources on the Worker nodes in the cluster and Driver's jobs/tasks are processed by Executor processes allocated on the Worker nodes
  - A Spark job consists of multiple tasks
  - The Executor processes correspond to YARN's Containers
  - An Executor process can run multiple tasks and has a built-in cache
  - Executor processes are started by Workers running on the same machine
- Basically, your Spark application consists of your Driver program and Executor processes running on the cluster

12.15 The Spark Application Architecture



12.16 Interfaces with Data Storage Systems

- Spark can be integrated with a wide variety of distributed storage systems:
  - Hadoop Distributed File System (HDFS)
  - Amazon S3
  - OpenStack Swift
  - etc.
- Spark understands Cassandra, HBase and any Hadoop's data formats
- For standalone Spark deployments, you can use NFS mounted at the same path on each node as a shared file system mechanism
- **Note**: Spark also supports a pseudo-distributed local mode running on a single machine where the local file system can be used instead of a distributed storage; this mode is only used for development or testing purposes.

12.17 Limitations of Hadoop's MapReduce

- Hadoop's MapReduce processing model is not always efficient
  - Generally, the complaint is about its inherent processing latencies
- Hadoop's MapReduce (MR) is designed as a fault-tolerant system heavily dependent on disk to persist intermediate results of MR jobs
  - This architecture is sometimes described as a disk-based system design
- The MR engine executes jobs arranged in a Directed Acyclic Graph (DAG) which prevents cost-efficient execution of applications that require multi-pass flow (such flows can benefit from passing temp results through shared memory of the same run-time resource)
  - Data processing in MR is acyclic and the processing flow is based on running a series of distinct jobs that can share data
  - In other words, Hadoop's MR is, by design, inefficient for multi-pass applications

12.18 Spark vs MapReduce

- Spark offers ~ 100X (in best for Spark multi-pass cases!) faster processing than Hadoop MR, or ~10X faster if intermediate results are committed to disk
  - Spark achieves these efficiencies by:
    - Re-using already loaded JVM (the heap is shared between related processes)
    - Data set caching
    - If a data set does not fit in memory, the data is spilled over to disk (with an unavoidable negative performance impact)

12.19 Spark as an Alternative to Apache Tez

- The Apache Tez project (https://tez.apache.org/) aims at boosting Hadoop's MR performance and optimizing resource utilization
  - Basically, Tez coalesces multiple MR jobs in a single Tez job which offers overall application performance improvement
- Projects are under way to make Apache Pig and Hive run on Spark as an alternative to Tez execution engine

12.20 The Resilient Distributed Dataset (RDD)

- A Resilient Distributed Dataset (RDD) is the fundamental abstraction in Spark representing data partitioned across machines in a Spark cluster
- RDDs are
  - **R**esilient against data loss (can be re-created / re-computed)
  - **D**istributed across the cluster (stored in workers' memory)
- Spark APIs are centered around RDD processing

12.21 Spark Streaming (Micro-batching)

- Spark supports near-real time (under 0.5 second latencies) streaming data processing
- Spark uses a micro-batch execution model based on discretized streams where incoming data is collected over a time window (e.g. 10 seconds)
- Micro-batching supports consistent exactly-once (no duplicates) semantics where Spark can recover all intermediate state in case of a data loss
- Spark Streaming model enables code re-use, where programs written for regular batch analytics can be (with little changes) used in micro-batching scenarios
  - **Note**: Duration of a micro-batch data chunk processing must be less than the data collection window

12.22 Spark SQL

- The Spark SQL module provides a mechanism for mixing SQL-based queries with Spark programs written in Scala, Python, R, and Java
- As of Spark version 1.3, the original term used to describe the Spark SQL RDD type, *SchemaRDD*, was renamed to *DataFrame*
- Spark has tight integration with Hive:
  - It can use Hive's metastore, SerDes codecs and User-defined functions
- Spark can run in server mode that offers connectivity to ODBC/JDBC clients (the feature normally used by BI tools to query Big Data fronted by Spark)

12.23 Example of Spark SQL

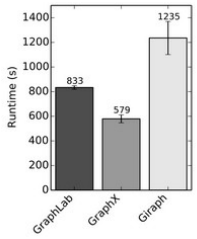
```
sqlCtx = new SQLContext(...)
results = sqlCtx.sql( "SELECT * FROM BigDataTable")
errorRecords=results.filter(r=>r.code.contains("error"))
```

12.24 Spark Machine Learning Library

- Spark Machine Learning Library (MLlib) runs on top of Spark and takes full advantage of Spark's distributed in-memory design
- MLlib developers claim 10X faster performance for similar applications created using Apache Mahout running on Hadoop via the MR engine
- The MLlib library implements a whole suite of statistical and machine learning algorithms (see Notes for details)

12.25 GraphX

- GraphX is Apache Spark's API for graphs and graph-parallel computation
- GraphX started initially as a research project at UC Berkeley AMPLab and Databricks, and was later donated to the Spark project
- It provides comparable performance to common graph processing systems, e.g. Giraph



End-to-end PageRank performance (20 iterations, 3.7B edges)

Source: https://spark.apache.org/graphx/

12.26 Spark vs R

- R is an interpreted single-threaded language
  - R runs on a single machine
    - You can collaborate with other users using RStudio Server
  - 64-bit architectures are supported
    - 2G+ RAM can be allocated for large data sets
- Spark can give you a choice of faster languages (Java, Scala)
  - Optimized for speed (local / external (Tachyon-based) caching, etc.)
  - Spark allows you to run Machine Learning (ML) apps on a cluster of machines
- R gives developers a richer choice and variations of supported statistical and ML algorithms
- R on Spark has very limited support for ML as yet [end 2016]

12.27 Summary

- Spark is a general-purpose processing system for large-scale data on a cluster of machines
- Used in a wide variety of applications:
  - Extract/Transform/Load (ETL) jobs
  - Data Analysis
- Spark applications are written in Scala, Python, Java, or R
- Spark offers between 10X to 100X faster processing times than Hadoop MapReduce
- A Resilient Distributed Dataset (RDD) is the main data unit in Spark which represents data partitioned across machines in the cluster
- Spark supports Streaming API, mixed SQL processing model, and a wide range of Machine Learning and Graph processing algorithms

Chapter 13 - The Spark Shell

Objectives

Key objectives of this chapter:

- The Spark Shell overview
- File operations in the Spark Shell
- Basic ETL operations in the Spark Shell

13.1 The Spark Shell

- The Spark Shell offers interactive command-line environments for Scala and Python users
  - SparkR Shell has only been thoroughly tested to work with Spark standalone so far and not all Hadoop distros available, and therefore is not covered here
- The Spark Shell is often referred to as REPL (Read/Eval/Print Loop)
- The Spark Shell session acts as the Driver process
- The Spark Shell supports only Scala and Python (Java is not supported yet)
  - The Python Spark Shell is launched by the **pyspark** command
  - The Scala Spark Shell is launched by the **spark-shell** command

13.2 The Spark Shell

- **Note**: The version of your Spark Shell may be different

Scala Spark Shell Welcome Screen

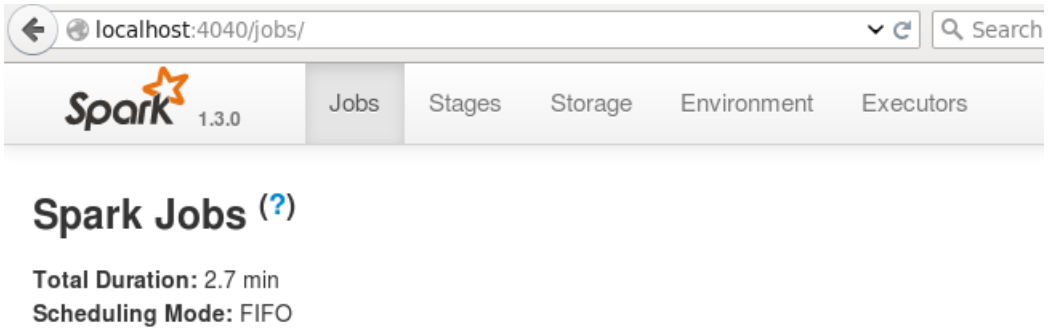
```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_67)
Type in expressions to have them evaluated.
Type :help for more information.
```

Python Spark Shell Welcome Screen

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.3.0
      /_/

Using Python version 2.6.6 (r266:84292, Feb 22 2013 00:00:18)
```

13.3 The Spark Shell UI

- When you start a Spark shell, it automatically launches an embedded Jetty web server which starts listening on port 4040



- Subsequently launched shells increment their embedded web servers' base port by one (4041, 4042, etc.)

13.4 Spark Shell Options

- You can get help on **spark-shell** and **pyspark** start-up options by invoking them with the **-h** flag:
  - **spark-shell -h**
  - **pyspark -h**
- **Note:** You can pass a source code fragment to be executed within the started Spark Shell using the **-i** flag
- You can pass parameters to Spark Shell using the Bash environment variables, e.g.
  - Set a variable (in a Bash shell session):
    - export **MYPARAM**=VALUE
  - Start a Spark Shell session
  - Read the value (e.g. inside a Scala Spark Shell session):
    - System.getenv("**MYPARAM**")

13.5 Getting Help

- In Scala Spark Shell, to get help on a supported command, type in the following command:

```
:help [command]
```

- the *:help* command prints the related help content and returns
- In Python Spark Shell, help is only available for Python objects (e.g. modules)

```
help(object)
```

- press **q** when done to return to prompt

13.6 The Spark Context (sc) and SQL Context (sqlContext)

- The Spark Context (sc) is the main Spark API object
- When you start Spark Shell, you are given an instance of a pre-built SC object aliased as **sc** that you can start using right away
  - Only one SC may be active per your Shell session
  - You also get a pre-built SQL Context object referenced as **sqlContext**

13.7 The Shell Spark Context

- The following *sc* properties and methods are available to developers in the Spark Shell version 1.3:

```
accumulable
accumulator
addJar
appName
asInstanceOf
binaryRecords
cancelAllJobs
clearCallSite
clearJars
defaultMinPartitions
defaultParallelism
files
getCheckpointDir
getExecutorMemoryStat
getLocalProperty
getPoolForName
getSchedulingMode
hadoopFile
initLocalProperties
isLocal
killExecutor
makeRDD
metricsSystem
newAPIHadoopRDD
accumulableCollection
addFile
addSparkListener
applicationId
binaryFiles
broadcast
cancelJobGroup
clearFiles
clearJobGroup
defaultMinSplits
emptyRDD
getAllPools
getConf
getExecutorStorageStatus
getPersistentRDDs
getRDDStorageInfo
hadoopConfiguration
hadoopRDD
isInstanceOf
jars
killExecutors
master
newAPIHadoopFile
objectFile
parallelize
runApproximateJob
sequenceFile
setCheckpointDir
setJobGroup
sparkUser
statusTracker
submitJob
textFile
union
wholeTextFiles
requestExecutors
runJob
setCallSite
setJobDescription
setLocalProperty
startTime
stop
tachyonFolderName
toString
version
```

13.8 Loading Files

- You load a text file (or files) from any of the supported file systems using the *textFile (path)* method on the Spark Context object:

```
sc.textFile(path)
```

  - the *textFile* method accepts comma-separated list of files, and a wildcard list of files
  - For local file system files you may omit the file system prefix (**file://**), for HDFS and S3 you need to specify the prefix and provide the full file path: **hdfs://** and **s3://**
  - Each line in the loaded file(s) will become a record in the resulting file-based RDD
- When the file is fully loaded, Spark will create an RDD in the memory of the clustered worker machines
- The *sc* object also exposes methods for loading other file types: *objectFile* (Java serialization object format)*, hadoopFile,* and *sequenceFile*

13.9 Saving Files

- The *saveAsTextFile(path)* method of an RDD reference allows you to write the elements of the dataset as a text file(s)
- You can save the file on HDFS or any other Spark-supported file system
- When saving the file, Spark calls the *toString* method of each data element to convert it to a line of text in the output file
- Methods for saving files in other output file formats:
  - *saveAsSequenceFile* (Java & Scala only)
  - *saveAsObjectFile* (Java & Scala only)

13.10 Basic Spark ETL Operations

- Examples of Spark basic operations below are shown as executed in the Scala Spark Shell (with #comments):

```
val f="file:///some/directory/somefile.dat"# a CSV file on local FS

val files = sc.textFile(f) #load file and create a text file RDD
files.count()#count and print the number of records in the file RDD

# print all records that contain the word "magic"
files.filter ( _.contains("magic")).foreach(println)

# build a new RDD containing a third (2) column from the parent RDD
val thirdColumn = files.map (_.split(';')(2))

files.toDebugString #list all RDDs (the lineage) generated so far

# Print pairs: "the 6th column values followed by 5th column values)
val ff = files.map(_.split(';')).foreach(w => println(w(5)+" "+w(4)))

# save the files RDD on HDFS
files.saveAsTextFile("hdfs://hostname:8020/user/me/tempfile.dat")
```

13.11 Summary

- Developers have an option to interactively explore and manipulate data using the Spark Shell for Python and for Scala
- The Spark Context object offers developers a rich API for data transformations suitable for data analytics and ETL jobs

Chapter 14 - Spark RDDs

Objectives

In this chapter, participants will learn about:

- The Resilient Distributed Dataset (RDD)
- RDD-related operations:
  - Actions and Transformations
- Pair RDDs

14.1 The Resilient Distributed Dataset (RDD)

- A Resilient Distributed Dataset (RDD) is the fundamental abstraction in Spark representing data partitioned across machines in the cluster
- RDDs are
  - **R**esilient against data lost (can be re-created)
  - **D**istributed across the cluster (stored in workers' memory)
- Spark APIs for Scala, Python and Java are centered around RDD processing
- For technical details on RDD internals, visit http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

14.2 Ways to Create an RDD

- RDDs can be created from:
  - Files coming from any of the supported file system
    - Local or shared file system, HDFS, HBase, S3, et. al.
  - From data already sitting in memory and available to the Driver application via SparkContext
    - This is the case of parallelized collections
  - As a result of intermediate data transformations applied to previous RDDs

14.3 Custom RDDs

- Users can implement custom RDDs (e.g. for reading data from a specific storage system)
- Language-specific RDD structures (e.g. an abstract class in Scala) represent immutable and partitionable collections of elements
- Operations on any RDD are done via a set of standard API methods (*cache, count, map, filter, persist*) which are implemented by users with the desired behavior
- Any internal state mutations in an RDD by a transformation operation are encapsulated in a new (child) RDD returned as a result of the transformation operation

14.4 Supported Data Types

- RDD can the following data types:
  - Primitives:
    - boolean, character, integer, etc.
  - Sequence types:
    - array, list, string, tuples, etc.
  - Java / Scala objects (instance of Serializable)
  - Key-Value pairs

14.5 RDD Operations

- There are two types of RDD operations:
  - **Actions**
    - Those operations return values, e.g. the count of records in an RDD
  - **Transformations**
    - Those operations return a new RDD as a derivative of the base (original) RDD, by applying some transformation function to every record in the base RDD, or filtering records as per some filter condition

14.6 RDDs are Immutable

- In line with the philosophy of functional programming, RDDs are immutable
  - When a transformation produces a new child RDD, that RDD, in turn, is also immutable
  - This RDD property makes Spark "memory hungry":
    - A new heap allocation is required for the new RDD (actually, for all its partitions) rather than in-place updates

14.7 Spark Actions

- An action is a method invoked on the target RDD that triggers a computation on that RDD and returns the result of the computation
- Some of the Spark actions:
  - **count**() - return the number of data elements in an RDD
  - **take**(**x**) - return first **x** data elements in an RDD
    - take (1) == first()
  - **collect**() - combine all the data elements in an RDD in an array
  - **reduce()** - return the result of an aggregation operation on data elements in an RDD
  - **saveAsTextFile(*path*)** - save as a text file RDD's content on any of the supported FS
- For more information on Spark actions, visit *http://spark.apache.org/docs/latest/programming-guide.html#actions*

14.8 RDD Transformations

- A transformation is an operation on the source RDD to produce a new (derivative) RDD, sometimes referred to as the *Child* RDD
  - The source (parent) RDD remains immutable
- Transformations are lazily (not immediately) evaluated
  - An evaluation, when it happens, results in a materialized RDD
  - Transformation evaluations are triggered by an action on that RDD, e.g. *count()*
  - All previously enqueued transformations, beginning from the root (base) transformation, are sequentially executed and the intermediate RDDs materialized

14.9 RDD Transformations

- Some of the Spark transformations invoked as methods on the target RDD(s):
  - **filter(func)** - return a new RDD by applying the predicate function
  - **map(func)** - return a new RDD by applying the input function
  - **union(*thatRDD*)** - merges *this* RDD with *thatRDD*
  - **join(*thatRDD*)** - join values for the same key in a list
  - **intersection(*otherDataset*)** - performs a set intersection operation
  - **distinct()** - return an RDD with duplicate data elements removed

14.10 Other RDD Operations

- **first()** - return the first data element in an RDD
- **foreach()** - apply a function to each data element in an RDD
- **top(n)** - return *n* largest elements in an RDD

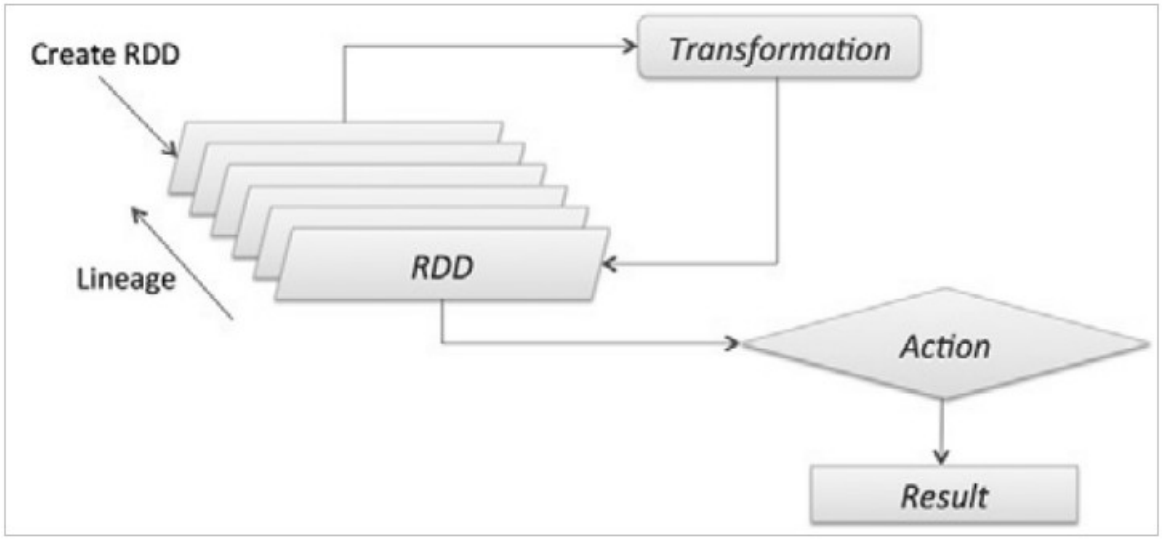
14.11 Chaining RDD Operations

- Transformations and actions can be chained together using the '*dot*' notation:
  - `rddRef.map(func1).filter(func2).count()`
- For more information on Spark transformations, visit *http://spark.apache.org/docs/latest/programming-guide.html#transformations*

14.12 RDD Lineage

- For every RDD, Spark maintains its lineage (a list of its ancestors - previous RDDs used in creating this current RDD)
- This feature helps with two things:
  - Lazy materialization (building) of RDDs when they are needed
  - Data resiliency (the **R** in RDD) against data loss -- lost data can be recreated from the base RDD by sequentially applying the required transformations (a list of applied transformations is maintained by the Spark Master)

14.13 The Big Picture



14.14 What May Go Wrong

- When an action operation (e.g. *count, collect, reduce, take*) against an RDD is encountered, all its lineage RDDs are sequentially executed and the intermediate RDDs are materialized
- Transformations are applied as a sequence of executable code that recursively allocates new stack frames for any new transformation operation
- With a long lineage you may get a *Stack Overflow* exception

14.15 Checkpointing RDDs

- Checkpointing of RDDs is a process of saving them to a supported storage (default is HDFS) and truncating their lineage graph
- Before you can use regular RDD checkpointing, you need to specify the checkpoint directory:

```
SparkContext.setCheckpointDir(directoryName)
```

  - *Local checkpointing*, the topic of one of the next slides, does not use this directory
- RDD checkpointing is happening on the Worker nodes
- Actual checkpointing is done by calling the *checkpoint()* method on the target RDD
- You also must checkpoint your RDDs before any action on those RDDs is executed

14.16 Local Checkpointing

- Spark also offers a light-weight checkpointing facility that uses Spark's caching layer instead of persistence store
  - In other words, the checkpoint directory set through *SparkContext's setCheckpointDir()* method is not used
- This is called local checkpointing and it is performed using the *localCheckpoint()* method of the RDD object
- Local checkpointing is used in use cases where it is important to periodically truncate RDD lineage graph(e.g. in Spark Streaming or GraphX)

14.17 Parallelized Collections

- The *SparkContext's* **parallelize** method is used to create a distributed RDD from a collection of elements that can be processed in parallel
- The original collection (e.g. an Array) of elements sits in memory of the Driver
- Example:

```
val aCollection = Array(1, 2, 3, 5, 8)# aCollection sits in the Driver's memory

val anRDD = sc.parallelize(aCollection) # anRDD gets partitioned and distributed
across the cluster
```

14.18 More on parallelize() Method

- The *parallelize* method takes an optional parameter for the number of partitions (splits) to break the input collection into, e.g.

```
sc.parallelize(aCollection, 3)
```

  - Spark will allocate one task per partition
  - If this parameter not provided, Spark will pick a value based on some cluster metrics
- The *parallelize()* operation works in tandem with the *collect()* operation, which re-assembles all processed partitioned RDD's data elements back in a collection in the Driver's memory
  - The machine running the Driver process must have enough memory to accommodate memory requirements of these operations

14.19 The Pair RDD

- The Pair RDD holds elements that are key-value pairs
- Each key-value pair is treated as a two-element tuple (record)
- Keys and values can be swapped (key ↔ value)
- Keys and values can be of any Spark-supported type

14.20 Where do I use Pair RDDs?

- Pair RDDs are used in
  - Programs based on the MapReduce processing model
  - Data sorting, grouping, etc.
- Usually, Pair RDDs are created using these functions:
  - **keyBy, map, flatMap**

14.21 Example of Creating a Pair RDD with Map

- Let's say the input text file's rows consist of two **tab**-delimited columns:

```
bankCardNumber1   netWorth1
bankCardNumber2   netWorth3
bankCardNumber2   netWorth3
                    ...
```

  - The following Scala code will produce a Pair RDD containing two-element tuples: *(bankCardNumberX,netWorthX)*
  - Note the () in (f(0),f(1)) - that's the making of the tuple

```
val clientsRDD = sc.textFile(inputFile);

val pairRDD = clientsRDD.map(_.split('\t')).
          map(f => (f(0),f(1)));
           // f(0) : bankCardNumber<X>
           // f(1) : netWorth<X>
```

14.22 Example of Creating a Pair RDD with keyBy

- Let's say the input file is a financial transactions log containing records in the following format (the file's fields are comma-delimited):

```
timeStamp, txnStatus, txnCode, bankCardNumber
...
```

  - The following Scala code will produce a Pair RDD containing two-element tuples: *(bankCardNumber,[the full log record])* designating the *bankCardNumber (the 3rd field)* as a key

```
val logRDD = sc.textFile(txnLogFile);

val pairRDD = logRDD.keyBy(row => row.split(',')(3));
                      // bankCardNumber =>[the full log record]
```

14.23 Miscellaneous Pair RDD Operations

- **countByKey** - return a map with the count of occurrences of each key in an RDD
- **groupByKey** - return the result of a grouping operation for each key in an RDD
- **join** - return a composite RDD with data elements from two RDDs joined by matching keys

14.24 RDD Caching

- As an operational efficiency technique, you can use RDD caching:

```
yourRDD.cache()
```

  - The target RDD gets retained in the Executor's memory (and gets prevented from being garbage collected)
- RDD caching improves performance of machine learning and iterative algorithms
- Cached data cannot be shared across Spark applications as each application gets executed in its own sandboxed environment
- **Note**: Caching is only a suggestion to Spark engine which may decide not to perform caching at all (e.g. due to memory constraints)
-
- You can cancel caching by calling the *unpersist()* method of the RDD API

14.25 RDD Persistence

- Spark offers developers a number of Storage Levels options for RDD persistence
- These Storage options are set as flags passed to the *persist()* method:
  - **MEMORY_ONLY** - this is the default value, equivalent to *cache()*
  - **MEMORY_AND_DISK** - a hybrid approach, spilling data to disk only when the *MEMORY_ONLY* option has been exhausted
  - **DISK_ONLY** - all RDD partitions are committed to disk
    - This may be a good option if lineage re-computation is more expensive than disk I/O
  - **OFF_HEAP** (Experimental) - store an RDD in serialized format in Tachyon; this option is attractive in environments with large heaps or multiple concurrent applications
- To stop RDD persistence and purge all its partitions from memory and disk, use *unpersist()*

14.26 The Tachyon Storage

- Tachyon is a clustered in-memory fault-tolerant file system developed at UC Berkeley AMPLab
- It supports the standard file system operations, such as *format, mv, rm, mkdir, cat, touch,* etc.
- It offers applications (e.g. multiple Executor processes) an efficient file sharing mechanism by creating a RAM disk
- In Spark, Tachyon is the default off-heap option for storing RDDs outside of the JVM's heap avoiding GC overheads and making RDDs resilient (Tachyon makes them tolerant to crashes of an application, e.g. an Executor)
- See http://ampcamp.berkeley.edu/big-data-mini-course/tachyon.html for more details

14.27 Summary

- The main data unit in Spark is a Resilient Distributed Dataset (RDD) which represents data partitioned across machines in the cluster
- RDDs support a variety of data types
- There are two main RDD-related operation types:
  - Actions
  - Transformations
- RDDs support in-memory caching and disk persistence
- Pair RDDs are a useful data structure that helps with such tasks as MapReduce programs, sorting, etc.

Chapter 15 - Parallel Data Processing with Spark

Objectives

In this chapter, participants will learn about:

- Options for running Apache Spark on a cluster
- Parallel Data Processing of Partitions
- Spark terminology

15.1 Running Spark on a Cluster

- The following options are available for running Spark applications on a cluster:
  - **Spark Stand-alone** - Spark's own cluster management system
    - Limited in terms of configuration options and scalability
  - External cluster management systems (the preferred option for large processing jobs):
    - **Hadoop's YARN**
    - **Mesos**
- Running Spark using a cluster management system aids in computing efficiency, fault-tolerance, and scalability of your data processing solutions
- For development and prototyping, you can run Spark on a single (local) machine (without distributed processing capabilities)
- In all scenarios there is a Driver program (your Spark application or a Spark Shell session) which creates a Spark Context pointing to the Spark Master

15.2 Spark Stand-alone Option

- Gives you:
  - One *Spark Master* daemon which distributes tasks across Spark Workers on a cluster
  - Multiple *Spark Worker* daemons running on Worker nodes which start and monitor *Executor* processes
  - *Executor* processes (each of which is running in a JVM or Python run-time)
    - Executors have high RAM requirements (>=16G) for efficient in-memory processing

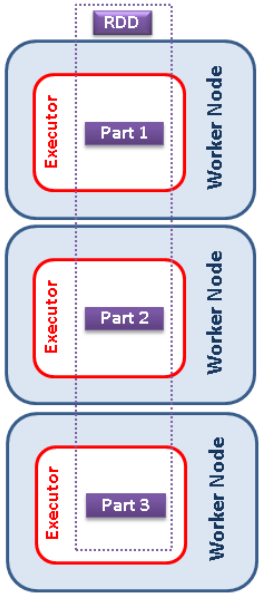
15.3 The High-Level Execution Flow in Stand-alone Spark Cluster

- The Spark Driver program contacts the Spark Master daemon
- The Master engages the Spark Worker daemons on Worker nodes
- Each engaged Spark Worker daemon launches an Executor process on the same node
- Tasks are sent to respective Executors for execution
  - Task allocation is aligned with the data locality principle (execution proximity to source data)
- Computed results are sent back to the Driver program

15.4 Data Partitioning

- Spark RDDs are automatically partitioned (split) across Worker nodes
  - Developers can configure the number of partitions to be created
- Every partition is processed in one Executor instance
- The default number of partitions in RDDs returned by transformations like *join, reduceByKey, and parallelize* when not set by user is configured by the *spark.default.parallelism* configuration parameter
- See Spark version-specific help page: *https://spark.apache.org/docs/<Spark Version>/configuration.html*

15.5 Data Partitioning Diagram



RDD partitioning across three Worker Nodes

15.6 Single Local File System RDD Partitioning

- By default, a single file is partitioned into two parts (splits)
- When loading, you can configure the number of partitions of your RDD, e.g.:

```
sc.textFile ("file:///some_dir/filename.dat", 5)
```

  - will split the source /*some_dir/filename.dat* file into 5 partitions
- The source file size, the number of machines in the cluster, and some other metrics are the factors in determining the optimal number of partitions

15.7 Multiple File RDD Partitioning

- When you load multiple files

  e.g. **`sc.textFile ("file:///some_dir/*.dat")`**
- Spark allocates each file (at least) one partition
- When you load file(s) from HDFS (*sc.textFile("hdfs://....")*), Spark allocates partitions per HDFS block size (64 / 128 MB)
- The data locality principle is observed:
  - The Executor process will be launched on the node where the HDFS data block resides
- When the processing of HDFS-based data files on a cluster is done, the computing results are returned to the Driver program by way of executing the *collect()* method on the resulting RDD that returns an array of the RDD elements

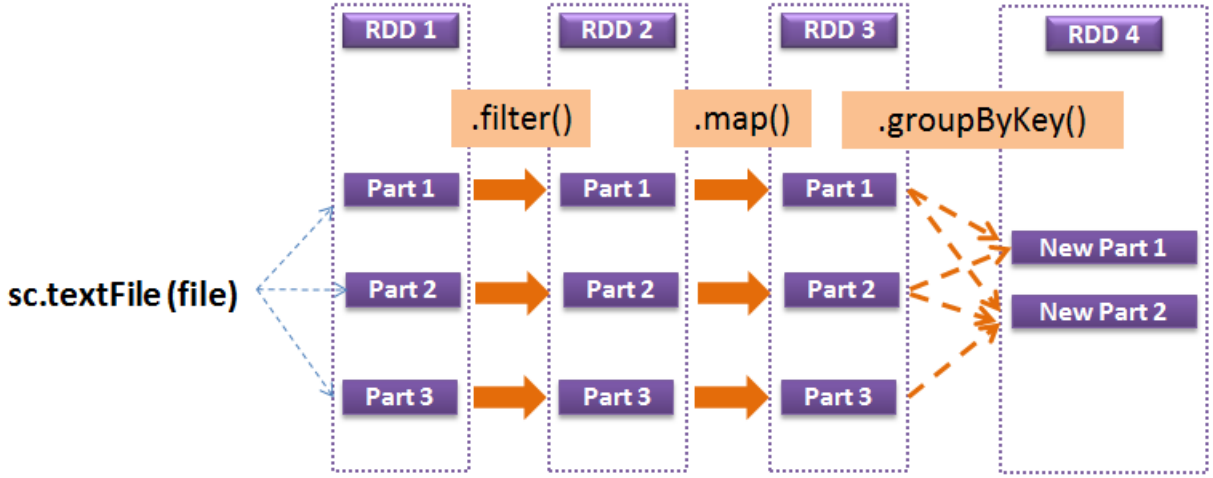
15.8 Special Cases for Small-sized Files

- If you have a large number of small-sized files (e.g. part-xxxxxx files on HDFS), you may want to consider loading and partitioning files using the *wholeTextFiles()* method of *SparkContext*
- The *wholeTextFiles()* method reads a directory of text files from the file system, or HDFS, or any other Hadoop-supported file system URI and creates a Key-Value RDD where the key is the file path and the value pointed to by that key is that file's content

15.9 Parallel Data Processing of Partitions

- RDD operations on each partition are done in isolation from and in parallel with operations in other partitions
- Some RDD operations preserve the partitioning boundaries:
  - *filter, map, flatMap*
- Other RDD operations require repartitioning -- creating a new partition, possibly in another Executor:
  - *groupBy, groupByKey, join, reduce, sort*
  - Repartitioning is a resource-expensive operation

15.10 Parallel Data Processing of Partitions



The *filter()* and *map()* RDD operations preserve partitioning; *groupByKey()* requires repartitioning

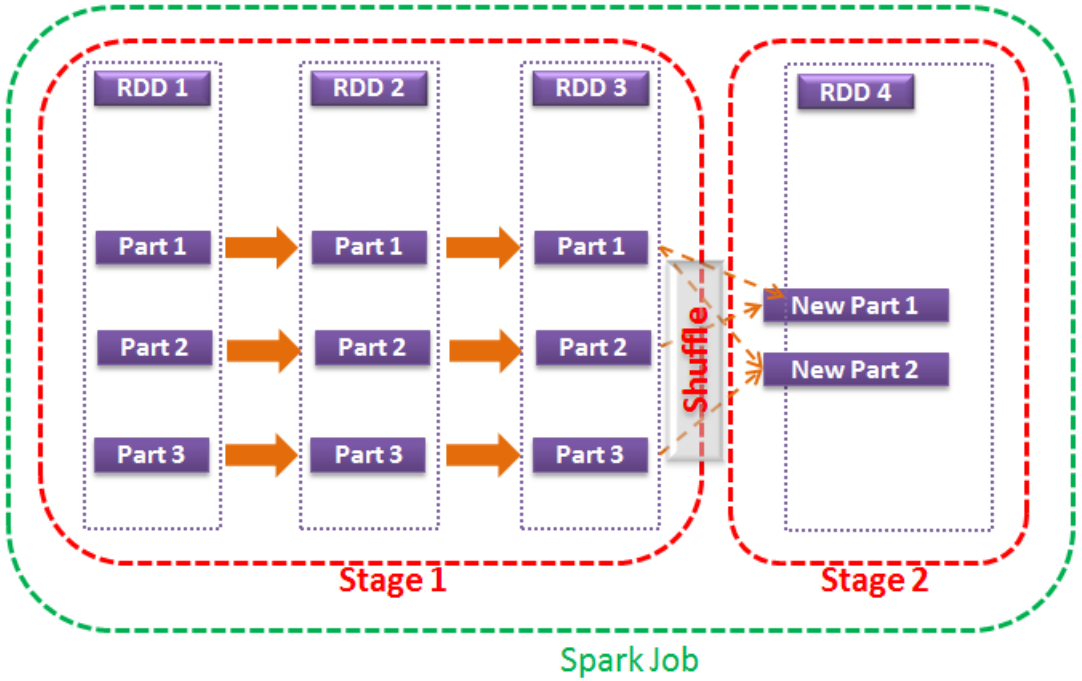
15.11 Spark Application, Jobs, and Tasks

- Spark documentation defines a *job* as a parallel computation of multiple *Task*s
- A *job* gets executed in response to a Spark action (e.g. *save*, or *collect*)
- Each Spark job may consist of multiple *Stages* (more on Stages a bit later ...)
- A Spark *Application* (defined as an instance of *SparkContext*) contains multiple jobs
- An Application may have multiple jobs running concurrently, given the jobs are submitted from separate threads
- RDD operations are performed in parallel as tasks - each task is processed by an Executor process
  - If you have three partitions which are to be processed by an operation -- Spark will allocate three tasks (equal to the number of partitions)
- An Executor process can run multiple tasks and has a built-in cache
- Tasks are scheduled by the Spark's Master daemon
- For more details, see http://spark.apache.org/docs/latest/job-scheduling.html

15.12 Stages and Shuffles

- Operations that run on the same data partition are grouped in a stage
  - This computing organization facilitates the creation of processing pipelines as tasks within a stage can effectively re-use allocated resources
- Repartitioning, also called *shuffles*, breaks processing flow in stages
- Shuffle steps are potentially expensive operations as they may involve any or all of these:
  - Data transfer over network (data needs to be un/marshaled using de/ser subsystem)
  - Data sorting
  - Disk I/O for storing intermediate results
- **Note**: The temporary storage directory is specified by the *spark.local.dir* configuration parameter when configuring the Spark context

15.13 The "Big Picture"



15.14 Summary

- Spark can run in the following cluster setup:
  - Using its own Stand-alone cluster management system
  - External cluster management systems:
    - Hadoop's YARN
    - Mesos
  - External cluster management systems are the preferred option for large processing jobs
- To efficiently process data on a cluster, Spark splits the input data into partitions that can be processed in parallel and in isolation of other tasks
- Data processing in a stage is considered highly efficient: such RDD operations as *filter, map, flatMap* preserve data partitions
- When your processing flow encounters such operations as *groupBy, groupByKey, join, reduce, sort*, the Spark engine performs a repartitioning operation and data reshuffling
  - New partitions are then processed in another stage
  - Repartitioning is a resource-expensive operation

Chapter 16 - Introduction to Spark SQL

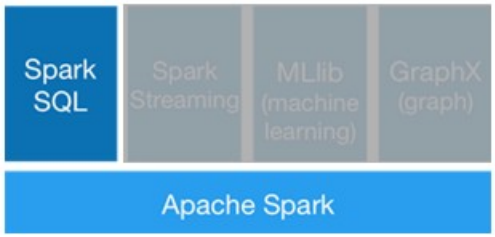
Objectives

In this chapter, participants will learn about:

- Spark SQL
- The SQL Context object
- The DataFrame object

16.1 What is Spark SQL?

- Spark SQL is a module that offers uniform programming API to work with structured data
- Spark SQL programs are executed in a distributed fashion by the SQL query engine
- The Spark SQL programming guide can be found here: http://spark.apache.org/docs/latest/sql-programming-guide.html



16.2 What is Spark SQL?

- Spark SQL queries can be mixed with Spark programs written in Scala, Python, and Java
- You also get a tight integration with Hive
- In its server mode, Spark SQL offers connectivity to ODBC/JDBC clients (the feature normally used by BI tools to query Big Data fronted by Spark)

16.3 Uniform Data Access with Spark SQL

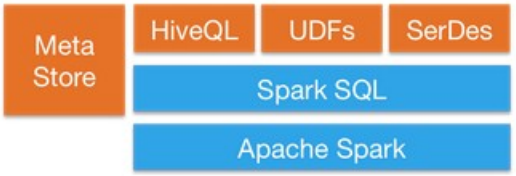
- Spark SQL provides uniform connectivity API to any data source, such as:
  - Avro
  - Parquet
  - ORC
  - JSON
  - Hive
  - JDBC/ODBC-enabled sources
- Joining data across the above data source is also supported

16.4 Hive Integration

- Spark SQL supports interfaces with Apache Hive for reading and writing data stored in Apache Hive
- Out of the box, the default Spark distribution bundle does not support Hive
  - You need to customize Spark's build script to include Hive and Thrift server binaries
  - The Hive assembly jar must also be present on all of the worker nodes in your Spark cluster

16.5 Hive Interface

- With Spark SQL, you can run unmodified Hive queries
- Through access to Hive's metastore, Spark SQL gives you full access to the existing Hive data, Hive queries, and Hive UDFs
- Spark SQL also supports Hive serialization and deserialization libraries (the SerDes subsystem)
- Configuration for Hive is read from the *hive-site.xml* configuration file that needs to be on the Spark's classpath



Source: http://spark.apache.org/sql/

16.6 Integration with BI Tools

- Spark SQL offers ODBC/JDBC connectivity for various Business Intelligence tools

Source: http://spark.apache.org/sql/

16.7 Spark SQL is No Longer Experimental Developer API!

- Up to version 1.3 Spark SQL was considered experimental technology with the "Alpha" label
- Spark 1.3+ versions promise binary compatibility with older releases (starting with 1.x)
  - Excluded are APIs that are explicitly marked as *DeveloperAPI* or *Experimental*

16.8 What is a DataFrame?

- A DataFrame in Spark SQL is a distributed collection of structured data
- Data in a DataFrame is organized into named columns
  - Conceptually close to a table in a relational database
  - Similar to a data frame in R and Python programming languages
- DataFrames can be constructed from Hive tables, structured data files, or Spark RDDs
- The DataFrame API is available in Scala, Java, Python, and R
- **Note:** As of Spark version 1.3, the original term used to describe the Spark SQL RDD type, SchemaRDD, was renamed to DataFrame

16.9 The SQLContext Object

- Similar to the *SparkContext*, the *SQLContext* object is the entry point for Spark SQL applications
- The SQLContext object is a wrapper around an instance of the SparkContext object
  - The *HiveContext* object extends SQLContext for integration with data stored in Hive
- It exposes rich API for data query, DataFrame management, DDL, and more ...

16.10 The SQLContext API

- SQLContext offers developers a set of APIs for:
  - Building SQL queries - the *sql()* method*;*
  - Persistent Catalog DDL;
  - DataFrame management;
  - Generic Data Source management - working with JSON, Parquet files, etc.;
  - Data caching / cache purging - *cacheTable(), clearCache()*, etc.

16.11 Changes Between Spark SQL 1.3 to 1.4

- As of 1.4.0, Spark SQL API for source management has changed to introduce the generic *DataFrameReader* object for reading and *DataFrameWriter* for writing data (both are annotated as *Experimental*) in various supported formats: JSON, JDBC, Parquet, and other file formats and protocols
- The new API is more fluid for both reading and writing data -- you need to use generic *SQLContext.read()* and *DataFrame.write()* methods
- Related *SQLContext* methods (e.g. *SQLContext.jsonFile*) were also deprecated (they were refactored out into methods of *DateFrameReader and DataFrameWriter)*

16.12 Example of Spark SQL (Scala Example)

```
// The '_' means '*' in Scala
import sqlContext.implicits._

// sc is an existing SparkContext instance
val sqlCtx = new org.apache.spark.sql.SQLContext(sc)

val dataFrame = sqlCtx.sql("SELECT * FROM table WHERE YEAR >= 1999")
// The sql method of the SQLContext object returns the result as a DataFrame

// You can mix-in functional programming elements
goodYears = dataFrame.filter (r=>r.code.contains("Good Year!"))
```

16.13 Example of Working with a JSON File

```
val dfSource = sqlCtx.read.json("/path/clients.json")
val clientDF = dfSource.select("timeStamp", "clientId", "status")

// Displays the content of the clientStateDF on stdout
clientDF.show()

// You can now save the JSON-based DataFrame as a Parquet file
clientDF.write.format("parquet").save("/path/clients.parquet")
```

16.14 Example of Working with a Parquet File

```
val parquetFileDF = sqlCtx.read.parquet("/path/file.parquet")

//You can register the DataFrame as a table --
// this gives you an ability to use SQL
parquetFileDF.registerTempTable("prqtTable")
val myRangeDF = sqlCntx.sql
("SELECT name FROM prqtTable WHERE range >= 100 AND range <= 200")

//Now unleash your FP talents!
myRangeDF.map(r => r(1) + ", " + r(0)).collect().foreach(println)
```

16.15 Using JDBC Sources

- Before you can use a JDBC-aware data source, you need to make sure you have the required JDBC jar on your Spark's classpath (defined by the *SPARK_CLASSPATH* environment variable)
- Be aware that some databases covert all schema names to uppercase
- A JDBC data source is easier to use from Java or Python than from Scala

16.16 JDBC Connection Example

- To connect to Postgres from a Java Spark program, you need to configure the JDBC connection as follows (the example is borrowed from Spark SQL documentation):

```
Map<String, String> options = new HashMap<String, String>();
options.put("url", "jdbc:postgresql:dbserver");
options.put("dbtable", "schema.tablename");

DataFrame jdbcDF = sqlContext.load("jdbc", options)
```

16.17 Performance & Scalability of Spark SQL

- Spark SQL leverages process optimization of the Spark computing platform
  - Spark SQL query engine includes a cost-based optimizer
- Scaling to hundreds and thousands of nodes with query-in-flight fault tolerance is supported
- Caching data in memory is the common practice to improve performance:
  - `sqlCtx.cacheTable("yourTableName")`

    `or`
  - `myDataFrame.cache()`

16.18 Summary

- Spark SQL enables you to mix SQL-based queries with Spark programs written in Scala, Python, and Java
- Spark SQL offers tight integration with Hive
- The main entry point in your Spark SQL programs is provided by the SQLContext object that offers you a rich set of APIs for:
  - Building SQL queries
  - Persistent Catalog DDL
  - DataFrame management
  - Generic Data Source management
  - Data caching / cache purging

Chapter 17 - Apache Kafka

Objectives

In this chapter, participants will learn about:

- Apache Kafka's
  - Architecture
  - Use cases
  - Message delivery guarantees

17.1 What is Apache Kafka

- Apache Kafka (Kafka) is a distributed, scalable, and fault-tolerant platform for handling real-time large data feeds, which offers
  - Near network processing speeds
  - Linear scalability
- Runs on Windows ® and *NIX platforms
- Efficient at processing real-time data streaming use cases
- Written in Scala and Java
- Originally developed by LinkedIn and subsequently donated to Apache Software Foundation in 2011
-

17.2 Kafka As a Messaging Platform

- In essence, Kafka is a distributed real-time commit (transaction) log service, functioning as a high throughput (10^5 messages/second) publish/subscribe messaging system
- Runs on a cluster
- Supports parallel data loads into Hadoop
- Supports real-time streaming data load scenarios when integrated with Apache Storm, Apache HBase, or Apache Spark

17.3 The Treatment of Messages

- Messages are persisted on the file system or cached
- The retention period is configurable
- A message can be retained on the store after it has been read (consumed)

17.4 Use Cases

- A website client's activity (page views, searches, etc.) tracking
  - The original use case
- Low-latency data transformation pipelines (ETL)
- Data feeds aggregation (change data capture, or CDC)
  - Logs
  - Monitoring distributed systems
- A Big Data ingest buffer for peak times
- Event sourcing (see the slide's notes for more info)
- Real-time, fault tolerant streaming applications
  - Can replace conventional enterprise messaging system like JMS
  - Kafka, as of version 0.10 .0.0 supports lightweight stream processing via the Kafka Streams library. Apache Storm and other OSS streaming libraries integration is supported as well
    - Many production use cases involve integration with Apache Hadoop, Apache Storm, Apache Spark Streaming, and other such like systems

17.5 Terminology

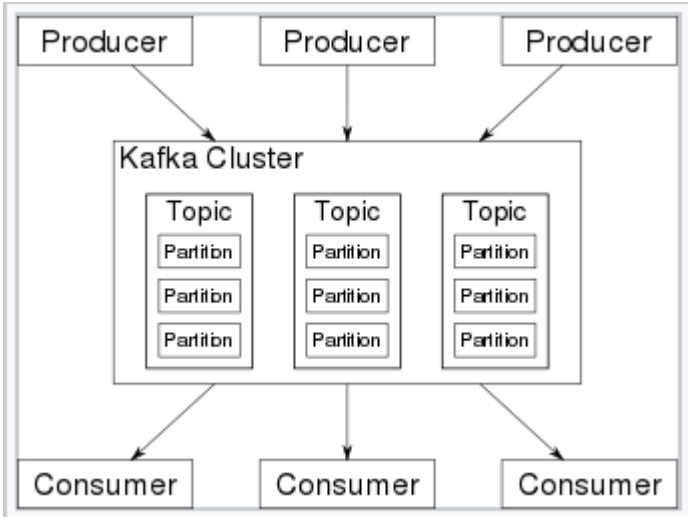
- Producer
  - Publishes a record to one or more topics
  - Sends the data directly to the lead broker (server)
- Consumer
  - Subscribes to one or more topics
  - Can be combined into a Consumer group acting as a clustered subscriber for scalability and fault tolerance
- Record (a.k.a. event, or message)
  - Essentially, a byte array
    - Formatted using JSON, Avro, etc.
  - Has a unique (within a partition) auto-incremented id (0,1,2, ...) called the 'offset', which acts as a key, a value, and a timestamp
  - Stored on partitions
  - Is immutable
- Topic (a.k.a. category, or feed name)
  - Essentially, a commit (transaction) log that is partitioned with one or more partitions

- A topic has a one-to-many relationship with consumers

17.6 Terminology

- Partition
  - Part of the commit log file (topic)
  - Replicated on a Kafka cluster for fault tolerance
- Broker
  - A service (server) that listens for client requests from Producers and Consumers
  - Manages partitions
  - Replicates messages
  - In a cluster, one broker service is dedicated to act as the controller, performing administrative tasks, like managing the states of partitions and replicas
- Connector
  - Helps with external system (e.g. RDBMSes) integration

17.7 The Architecture



Source: Apache Kafka Documentation

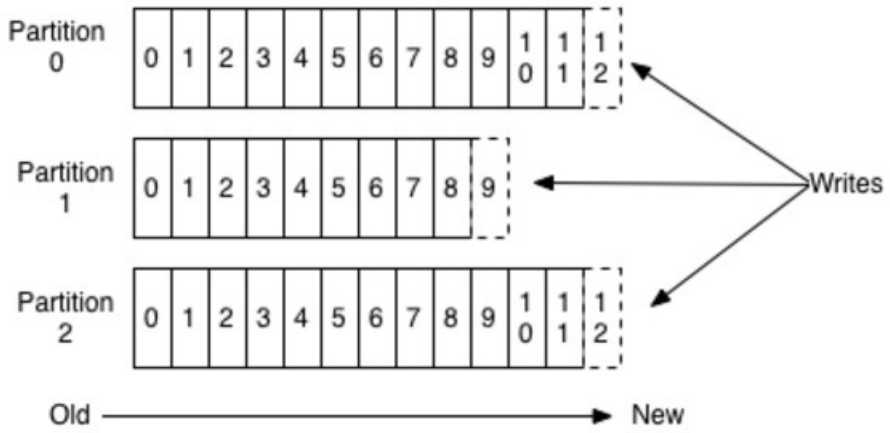
- Essentially, the Producers push data to the Brokers (the production side), the Consumers pull the data from the Brokers (the consumption side)
- Kafka uses Apache Zookeeper for state coordination
- Durable persistence is built around disk files (Kafka relies on the efficiencies of IO buffering / caching of modern OSes)

17.8 The APIs

- Kafka exposes its functionality through four APIs:
- The Producer API
  - Tells you how to publish streams of records
  - Performs append-only operations
- The Consumer API
  - Tells you how to subscribe to and read records from topic(s)
- The (Input) Streams API
- The Connector API
  - Links the topics to existing applications

17.9 The Anatomy of a Topic

- A topic is a multi-partition log
  - You can have as many partitions as you like
    - The actual number depends on the type of application
- Kafka provides strict record ordering within a single partition
  - Ordering across the partitions requires you to use record timestamps
- For efficiency, a producer batches (aggregates) data in memory before sending it to Kafka
  - The batching can be configured as either a fixed number of messages or a batching window (e.g. 128k or 50 ms)



Source: Apache Kafka Documentation

17.10 Partitions

- Partitions are distributed on a Kafka cluster
- Data is written to a partition randomly (Kafka provides round-robin load-balancing) unless a key is provided
- A partition acts as a unit of parallelism
  - More partitions allow for greater parallelism for message production and consumption at the expense of more files stored across broker services
- For fault-tolerance, each partition is replicated on one or more brokers (the replication factor configured at the topic's creation time)

17.11 The Read Operation

- Consumers subscribe to one or more topics and perform sequential or random reads
- To read a record, a consumer needs three pieces of information: the topic name, the partition id within the topic, and the offset of the record within the partition
  - A consumer can request a block of records
- The "fetch" request is processed by the Broker service leading the partition
- A consumer can go back and re-read a previously read record
  - The system retains the ids (record offsets) a consumer has read
  - This option supports stateful computations

17.12 The Leader and Followers

- The broker process managing the main partition replica acts as the "leader"; zero or more servers act as "followers"
- The leader handles all reads and writes
  - The followers passively replicate the leader's contents
- In case of the leader's failure, one of the followers will be automatically elected the "leader"
- Each broker hosting partitions can acts as a leader for some of its partitions and a follower for others
  - This arrangement ensures load balancing within the cluster

17.13 Guarantees

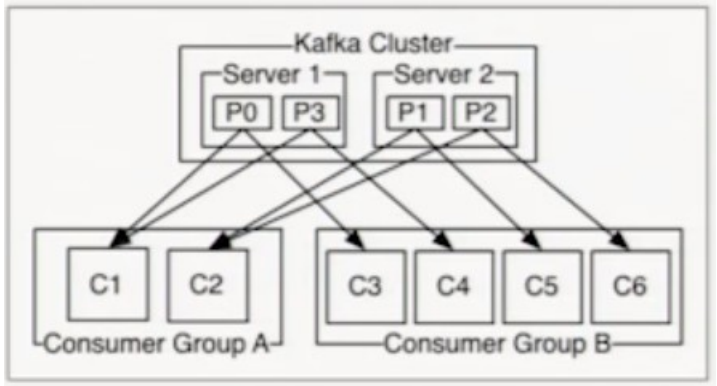
- Messages sent to a particular topic's partition by a single producer are inserted in the order they are sent (the message-ordering guarantee)
  - An earlier message (record) will have a lower offset (id) in the partition log
- A consumer sees records in the order they are stored in the log
- The configurable partition replication factor ensures the commit log's fault tolerance

17.14 Batch Compression

- For efficiency, Kafka uses compression of a batch of messages
- The batch of messages is written to the log in compressed form
- Consumers decompress the batch on the read
- Kafka supports GZIP, Snappy, and LZ4 compression protocols

17.15 A Consumer Group

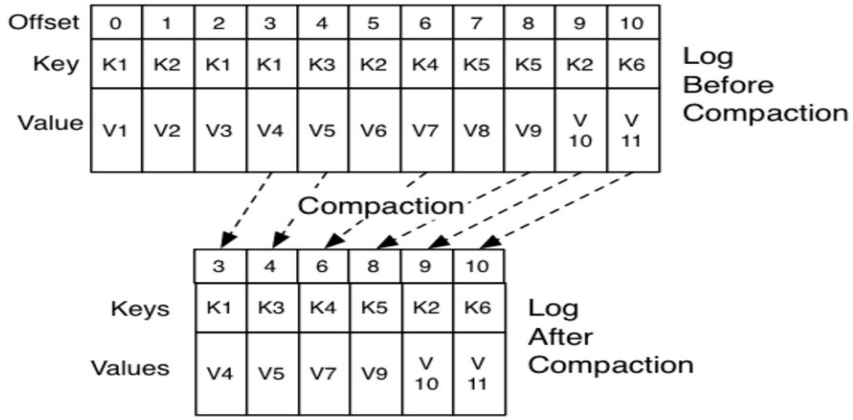
- A Consumer Group includes multiple consumers that act as a single entity for consuming events from a single topic
- Kafka uses the Consumer Group concept to divide processing over a collection of consumers in a scalable fashion
- Caveats:
  - You cannot have more consumers than you have partitions in a topic
  - You can have a single consumer from a Consumer Group reading messages from several partitions
  - No two or more different consumers from the same group can read from the same partition



Source: Apache Kafka Documentation

17.16 Log Compaction

- Log compaction is an optional feature that allows you to remove old records where more recent records with the same keys exist
  - Similar to the SQL's update operation
  - If you request a compacted record, you will get the latest record with the same key
- It gives you a per-record retention control, leading to smaller log sizes
- Message ordering is preserved



17.17 Summary

- Kafka is a distributed, scalable, and fault-tolerant platform for handling real-time large data feeds
- In essence, Kafka is a commit log file distributed across a cluster of machines
- Runs on Windows ® and *NIX platforms