Module -2

Introduction to Hadoop

2.1 Big Data Programming Model

A programming model is centralized computing of data in which the data is transferred from multiple distributed data sources to a central server. Analyzing, reporting, visualizing, business-intelligence tasks compute centrally. Data are inputs to the central server.

Another programming model is distributed computing that uses the databases at multiple computing nodes with data sharing between the nodes during computation. Distributed computing in this model requires the cooperation (sharing) between the DBs in a transparent manner. Transparent means that each user within the system may access all the data within all databases as if they were a single database. A second requirement is location independence. Analysis results should be independent of geographical locations. The access of one computing node to other nodes may fail due to a single link failure.

Distributed pieces of codes as well as the data at the computing nodes Transparency between data nodes at computing nodes do not fulfil for Big Data when distributed computing takes place using data sharing between local and remote. Following are the reasons for this:

- Distributed data storage systems do not use the concept of joins.

- Data need to be fault-tolerant and data stores should take into account the possibilities of network failure. When data need to be partitioned into data blocks and written at one set of nodes, then those blocks need replication at multiple nodes. This takes care of possibilities of network faults. When a network fault occurs, then replicated node makes the data available.

Big Data follows a theorem known as the CAP theorem. The CAP states that out of three properties (consistency, availability and partitions), two must at least be present for applications, services and processes.

i. Big Data Store Model

A model for Big Data store is as follows:

Data store in file system consisting of data blocks (physical division of data). The data blocks are distributed across multiple nodes. Data nodes are at the racks of a cluster. Racks are scalable.

A Rack has multiple data nodes (data servers), and each cluster is arranged in a number of racks.

Data Store model of files in data nodes in racks in the clusters Hadoop system uses the data store model in which storage is at clusters, racks, data nodes and data blocks. Data blocks replicate at the DataNodes such that a failure of link leads to access of the data block from the other nodes replicated at the same or other racks.

ii. Big Data Programming Model

Big Data programming model is that application in which application jobs and tasks (or sub-tasks) is scheduled on the same servers which store the data for processing.

2.2 Hadoop and its echo system

Hadoop is a computing environment in which input data stores, processes and stores the results. The environment consists of clusters which distribute at the cloud or set of servers. Each cluster consists of a string of data files constituting data blocks. The toy named Hadoop consisted of a stuffed elephant. The Hadoop system cluster stuffs files in data blocks. The complete system consists of a scalable distributed set of clusters.

Infrastructure consists of cloud for clusters. A cluster consists of sets of computers or PCs. The Hadoop platform provides a low cost Big Data platform, which is open source and uses cloud services. Tera Bytes of data processing takes just few minutes. Hadoop enables distributed processing of large datasets (above 10 million bytes) across clusters of computers using a programming model called MapReduce. The system characteristics are scalable, self-manageable, self-healing and distributed file system.

Scalable means can be scaled up (enhanced) by adding storage and processing units as per the requirements. Self-manageable means creation of storage and processing resources which are used, scheduled and reduced or increased with the help of the system itself. Self-healing means that in case of faults, they are taken care of by the system itself. Self-healing enables functioning and resources availability. Software detect and handle failures at the task level. Software enable the service or task execution even in case of communication or node failure.

2.1.1 Hadoop Core Components

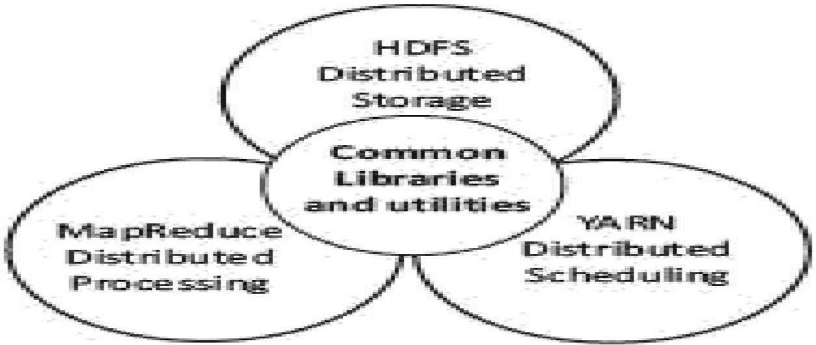


Figure 2.1 Core components of Hadoop

The Hadoop core components of the framework are:

Hadoop Common - The common module contains the libraries and utilities that are required by the other modules of Hadoop. For example, Hadoop common provides various components and interfaces for distributed file system and general input/output. This includes serialization, Java RPC (Remote Procedure Call) and file-based data structures.

Hadoop Distributed File System (HDFS) - A Java-based distributed file system which can store all kinds of data on the disks at the clusters.

MapReduce vl - Software programming model in Hadoop 1 using Mapper and Reducer. The vl processes large sets of data in parallel and in batches.

YARN - Software for managing resources for computing. The user application tasks or sub-tasks run in parallel at the Hadoop, uses scheduling and handles the requests for the resources in distributed running of the tasks.

MapReduce v2 - Hadoop 2 YARN-based system for parallel processing of large datasets and distributed processing of the application tasks.

2.2.2 Features of Hadoop

Hadoop features are as follows:

1. Fault-efficient scalable, flexible and modular design which uses simple and modular programming model. The system provides servers at high scalability. The system is scalable by adding new nodes to handle larger data. Hadoop proves very helpful in storing, managing, processing and analyzing Big Data.

2. Robust design of HDFS: Execution of Big Data applications continue even when an individual server or cluster fails. This is because of Hadoop provisions for backup (due to replications at least three times for each data block) and a data recovery mechanism. HDFS thus has high reliability.

3. **Store and process Big Data:** Processes Big Data of 3V characteristics.

4. Distributed clusters computing model with data locality: Processes Big Data at high speed as the application tasks and sub-tasks submit to the DataNodes. One can achieve more computing power by increasing the number of computing nodes. The processing splits across multiple DataNodes (servers), and thus fast processing and aggregated results.

5. Hardware fault-tolerant: A fault does not affect data and application processing. If a node goes down, the other nodes take care of the residue. This is due to multiple copies of all data blocks which replicate automatically. Default is three copies of data blocks.

6. Open-source framework: Open source access and cloud services enable large data store. Hadoop uses a cluster of multiple inexpensive servers or the cloud.

7. Java and Linux based: Hadoop uses Java interfaces. Hadoop base is Linux but has its own set of shell commands support.

2.2.3. Hadoop Eco system Components

The four layers in Figure 2.2 are as follows:

(i) Distributed storage layer

(ii) Resource-manager layer for job or application sub-tasks scheduling and execution

(iii) Processing-framework layer, consisting of Mapper and Reducer for the MapReduce process-flow.

(iv) APis at application support layer (applications such as Hive and Pig). The codes communicate and run using MapReduce or YARN at processing framework layer. Reducer output communicate to APis (Figure 2.2).

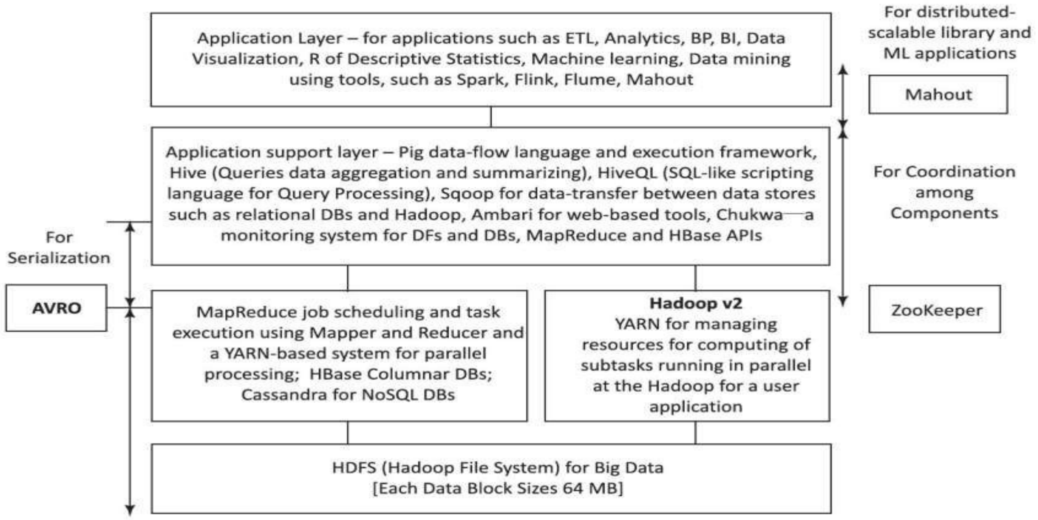

Application Layer – for applications such as ETL, Analytics, BP, BI, Data Visualization, R of Descriptive Statistics, Machine learning, Data mining using tools, such as Spark, Flink, Flume, Mahout
Application support layer – Pig data-flow language and execution framework, Hive (Queries data aggregation and summarizing), HiveQL (SQL-like scripting language for Query Processing), Sqoop for data-transfer between data stores such as relational DBs and Hadoop, Ambari for web-based tools, Chukwa—a monitoring system for DFs and DBs, MapReduce and HBase APIs
MapReduce job scheduling and task execution using Mapper and Reducer and a YARN-based system for parallel processing; HBase Columnar DBs; Cassandra for NoSQL DBs
Hadoop v2
YARN for managing resources for computing of subtasks running in parallel at the Hadoop for a user application
HDFS (Hadoop File System) for Big Data [Each Data Block Sizes 64 MB]
For Serialization
AVRO
For distributed-scalable library and ML applications
Mahout
For Coordination among Components
ZooKeeper

Figure 2.2 Hadoop main components and ecosystem components

AVRO enables data serialization between the layers. Zookeeper enables coordination among layer components.

The holistic view of Hadoop architecture provides an idea of implementation of Hadoop components of the ecosystem. Client hosts run applications using Hadoop ecosystem projects, such as Pig, Hive and Mahout.

2.3 HADOOP DISTRIBUTED FILE SYSTEM

HDFS is a core component of Hadoop. HDFS is designed to run on a cluster of computers and servers at cloud-based utility services.

HDFS stores Big Data which may range from GBs (1 GB= 230 B) to PBs (1 PB=

1015 B, nearly the 250 B). HDFS stores the data in a distributed manner in order to compute fast. The distributed data store in HDFS stores data in any format regardless of schema.

2.3.1 HDFS Storage

Hadoop data store concept implies storing the data at a number of dusters. Each cluster has a number of data stores, called racks. Each rack stores a number of DataNodes. Each DataNode has a large number of data blocks. The racks distribute across a cluster. The nodes have processing and storage capabilities. The nodes have the data in data blocks to run the application tasks. The data blocks replicate by default at least on three DataNodes in same or remote nodes.

Data at the stores enable running the distributed applications including analytics, data mining, OLAP using the clusters. A file, containing the data divides into data blocks. A data block default size is 64 MBs

Hadoop HDFS features are as follows

i. Create, append, delete, rename and attribute modification functions

ii. Content of individual file cannot be modified or replaced but appended with new data at the end of the file

iii. Write once but read many times during usages and processing

iv. Average file size can be more than 500 MB.

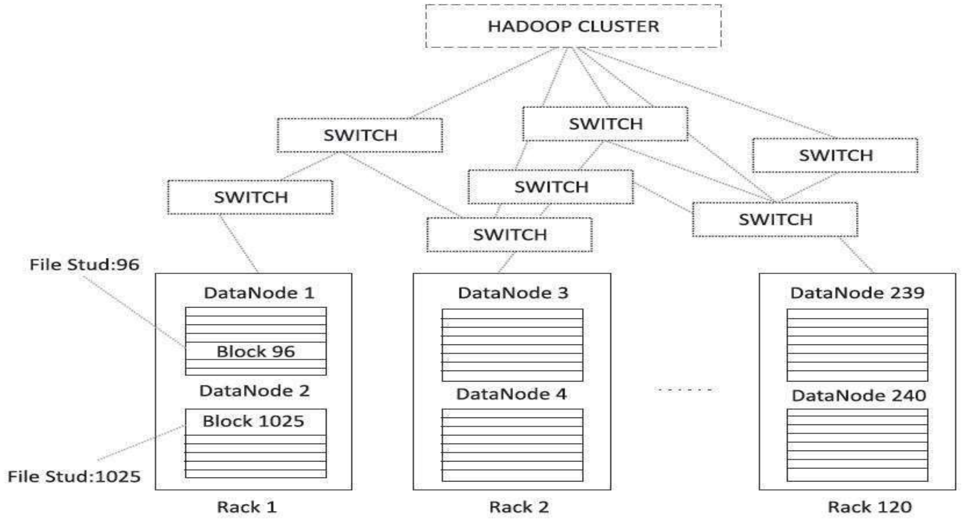


Figure 2.3 A Hadoop cluster example,

Consider a data storage for University students. Each student data, stuData which is in a file of size less than 64 MB (1 MB= 220 B). A data block stores the full file data for a student of stuData_idN, whereN = 1 to 500.

i. How the files of each student will be distributed at a Hadoop cluster? How many student data can be stored at one cluster? Assume that each rack has two DataNodes for processing each of 64 GB (1 GB= 2^{30} B) memory. Assume that cluster consists of 120 racks, and thus 240 DataNodes.

ii. What is the total memory capacity of the cluster in TB ((1 TB= 240 B) and DataNodes in each rack?

iii. Show the distributed blocks for students with ID= 96 and 1025. Assume default replication in the DataNodes = 3.

iv. What shall be the changes when a stuData file sizes 128 MB?

SOLUTION

i. Data block default size is 64 MB. Each students file size is less than 64MB. Therefore, for each student file one data block suffices. A data block is in a DataNode. Assume, for simplicity, each rack has two nodes each of memory capacity = 64 GB. Each node can thus store 64 GB/64MB = 1024 data blocks = 1024 student files. Each rack can thus store 2 x 64 GB/64MB = 2048 data blocks = 2048 student files. Each data block default replicates three times in the DataNodes. Therefore, the number of students whose data can be stored in the cluster = number of racks multiplied by number of files divided by 3 = 120 x 2048/3 = 81920. Therefore, the maximum number of 81920 stuData_IDN files can be distributed per cluster, with N = 1 to 81920.

ii. Total memory capacity of the cluster = 120 x 128 MB = 15360 GB = 15 TB. Total memory capacity of each DataNode in each rack= 1024 x 64 MB= 64 GB.

iii. Figure 2.3 shows a Hadoop cluster example, and the replication of data blocks in racks for two students of IDs 96 and 1025. Each stuData file stores at two data blocks, of capacity 64 MB each.

iv. Changes will be that each node will have half the number of data blocks.

2.3.1.1 Hadoop Physical organization

Figure 2.4 shows the client, master NameNode, primary and secondary MasterNodes and slave nodes in the Hadoop physical architecture. Clients as the users run the application with the help of Hadoop ecosystem projects. For example, Hive, Mahout and Pig are the ecosystem's projects. They are not required to be present at the Hadoop cluster.

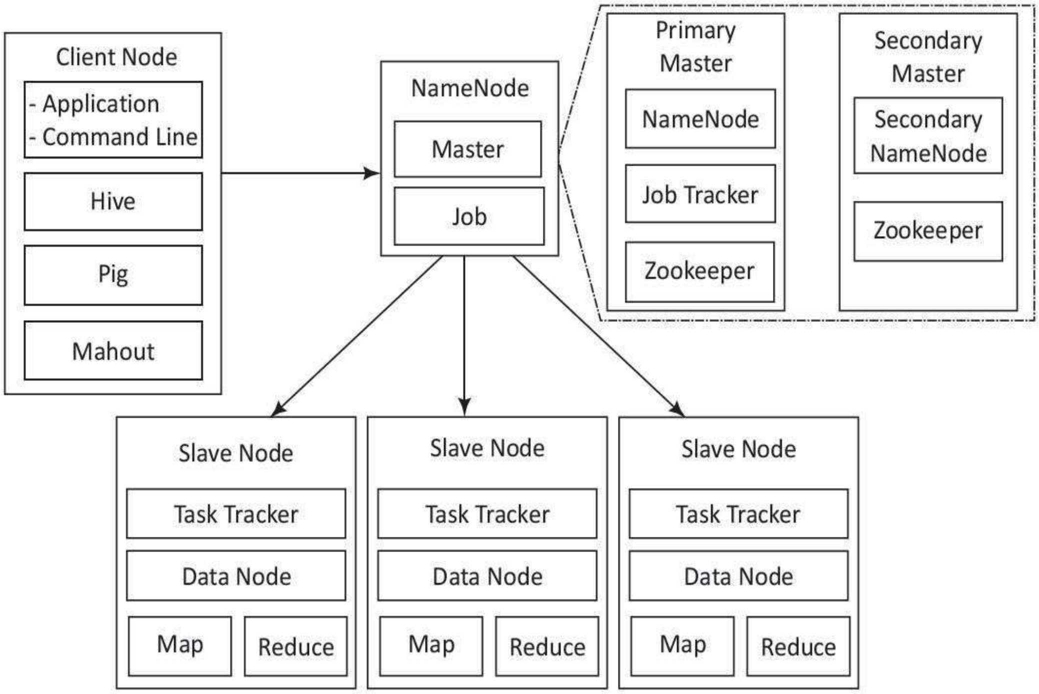


Figure 2.4 The client, master NameNode, MasterNodes and slave nodes

A single MasterNode provides HDFS, MapReduce and Hbase using threads in small to medium sized clusters. When the cluster size is large, multiple servers are used, such as to balance the load. The secondary NameNode provides NameNode management services and Zookeeper is used by HBase for metadata storage.

The MasterNode fundamentally plays the role of a coordinator. The MasterNode receives client connections, maintains the description of the global file system namespace, and the allocation of file blocks. It also monitors the state of the system in order to detect any failure. The Masters consists of three components NameNode, Secondary NameNode and JobTracker. The NameNode stores all the file system related information such as:

- The file section is stored in which part of the cluster
- Last access time for the files
- User permissions like which user has access to the file.

Secondary NameNode is an alternate for NameNode. Secondary node keeps a copy of

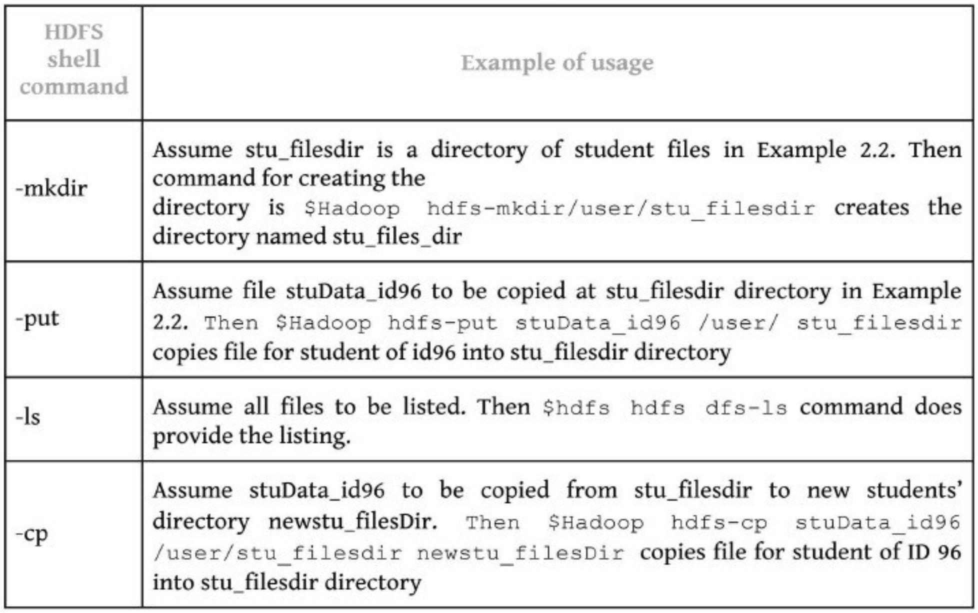
NameNode meta data. Thus, stored meta data can be rebuilt easily, in case ofNameNode failure. TheJobTracker coordinates the parallel processing of data.

2.3.1.1 Hadoop 2

- Single Name Node failure in Hadoop 1 is an operational limitation.
- Scaling up was restricted to scale beyond a few thousands of DataNodes and number of Clusters.
- Hadoop 2 provides the multiple NameNodes which enables higher resources availability

2.3.1.2 HDFS commands

Table 2.1 Examples of usages of commands



HDFS shell command	Example of usage
-mkdir	Assume stu_filesdir is a directory of student files in Example 2.2. Then command for creating the directory is `$Hadoop hdfs-mkdir/user/stu_filesdir` creates the directory named stu_files_dir
-put	Assume file stuData_id96 to be copied at stu_filesdir directory in Example 2.2. `Then $Hadoop hdfs-put stuData_id96 /user/ stu_filesdir` copies file for student of id96 into stu_filesdir directory
-ls	Assume all files to be listed. Then `$hdfs hdfs dfs-ls` command does provide the listing.
-cp	Assume stuData_id96 to be copied from stu_filesdir to new students' directory newstu_filesDir. `Then $Hadoop hdfs-cp stuData_id96 /user/stu_filesdir newstu_filesDir` copies file for student of ID 96 into stu_filesdir directory

2.4 MAPREDUCE FRAMEWORK AND PROGRAMMING MODEL

Mapper means software for doing the assigned task after organizing the data blocks imported using the keys. A key specifies in a command line of Mapper. The command maps the key to the data, which an application uses.

Reducer means software for reducing the mapped data by using the aggregation, query or user-specified function. The reducer provides a concise cohesive response for the application.

Aggregation function means the function that groups the values of multiple rows together to result a single value of more significant meaning or measurement. For example, function such as count, sum, maximum, minimum, deviation and standard deviation.

Querying function means a function that finds the desired values. For example, function for finding a best student of a class who has shown the best performance in examination.

MapReduce allows writing applications to process reliably the huge amounts of data, in parallel, on large clusters of servers. The cluster size does not limit as such to process in parallel. The parallel programs of MapReduce are useful for performing large scale data analysis using multiple machines in the cluster.

Features o fMapReduce framework are as follows:

- Provides automatic parallelization and distribution of computation based on several processors
- Processes data stored on distributed clusters of DataNodes and racks
- Allows processing large amount of data in parallel
- Provides scalability for usages of large number of servers
- Provides Map Reduce batch-oriented programming model in Hadoop version 1
- Provides additional processing modes in Hadoop 2 YARN-based system and enables required parallel processing. For example, for queries, graph databases, streaming data, messages, real-time OLAP and ad hoc analytics with Big Data 3V characteristics.

2.5 HADOOP YARN

- YARN is a resource a management platform. It manages the computer resources.
- YARN manages the schedules for running the sub tasks. Each sub tasks uses the resources in the allotted interval time.
- YARN separates the resources management and processing components.
- It stands for YET ANOTHER RESOURCE NEGOTIATOR , it manages and allocates resources for the application sub tasks and submit the resources for them in the Hadoop system.

Hadoop 2 Execution Model

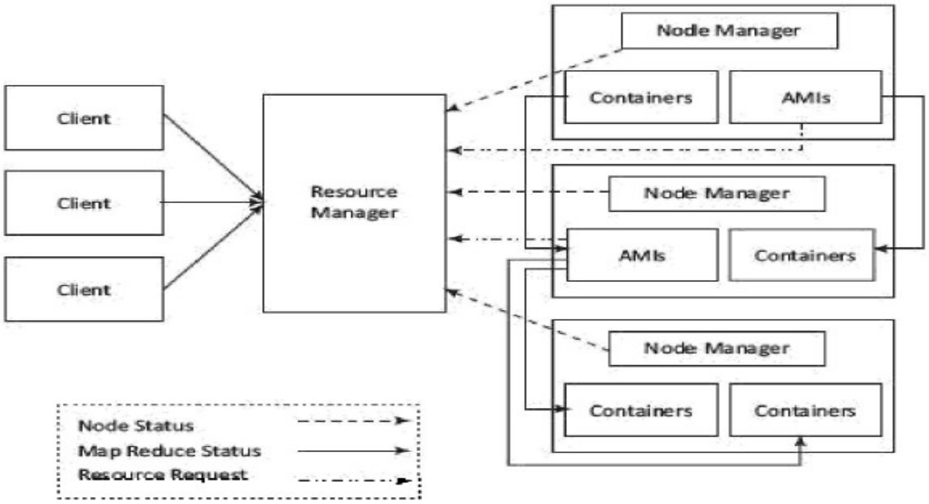


Figure 2.5 YARN based Execution Model

The figure shows the YARN components-Client, Resource Manager (RM), Node Manager (NM), Application Master (AM) and Containers.

Figure 2.5 also illustrates YARN components namely, Client, Resource Manager (RM), Node Manager (RM), Application Master (AM) and Containers.

List of actions of YARN resource allocation and scheduling functions is as follows:

A MasterNode has two components: (i) Job History Server and (ii) Resource Manager(RM).

A Client Node submits the request of an application to the RM. The RM is the master. One RM exists per cluster. The RM keeps information of all the slave NMs. Information is about the location (Rack Awareness) and the number of resources (data blocks and servers) they have. The RM also renders the Resource Scheduler service that decides how to assign the resources. It, therefore, performs resource management as well as scheduling.

Multiple NMs are at a cluster. An NM creates an AM instance (AMI) and starts up. The AMI initializes itself and registers with the RM. Multiple AMis can be created in an AM.

The AMI performs role of an Application Manager (ApplM), that estimates the resources requirement for running an application program or sub- task. The ApplMs send their requests for the necessary resources to the RM. Each NM includes several containers for uses by the subtasks of the application.

NM is a slave of the infrastructure. It signals whenever it initializes. All active NMs send the controlling signal periodically to the RM signaling their presence.

2.6 HADOOP ECOSYSTEM TOOLS

ZooKeeper-Coordination service	Provisions high-performance coordination service for distributed running of applications and tasks
Avro-Data serialization and transfer utility	Provisions data serialization during data transfer between application and processing layers
Oozie	Provides a way to package and bundles multiple coordinator and workflow jobs and manage the lifecycle of those jobs
Sqoop (SQL-to-Hadoop)-A data-transfer software	Provisions for data-transfer between data stores such as relational DBs and Hadoop
Flume - Large data transfer utility	Provisions for reliable data transfer and provides for recovery in case of failure. Transfers large amount of data in applications, such as related to social-media messages
Ambari-A web-based tool	Provisions, monitors, manages, and viewing of functioning of the cluster, MapReduce, Hive and Pig APIs
Chukwa-A data collection system	Provisions and manages data collection system for large and distributed systems
HBase-A structured data store using database	Provisions a scalable and structured database for large tables (Section 2.6.3)
Cassandra - A database	Provisions scalable and fault-tolerant database for multiple masters (Section 3.7)

Hive -A data warehouse system	Provisions data aggregation, data-summarization, data warehouse infrastructure, ad hoc (unstructured) querying and SQL-like scripting language for query processing using HiveQL (Sections 2.6.4, 4.4 and 4.5)
Pig-A high-level dataflow language	Provisions dataflow (DF) functionality and the execution framework for parallel computations
Mahout-A	Provisions scalable machine learning and library functions for data mining and analytics

module -1

capter -1

Hadoop Distributed file System Basics

Hadoop distributed file system was designed for Big data Processing. It is capable of supporting many users simultaneously. The design of HDFS is based on the design of the Google File System (GFS).

HDFS is designed for data streaming where Large amount of data are read from the disk in bulk. The HDFS size is typically 64MB or 128 MB.

HDFS components

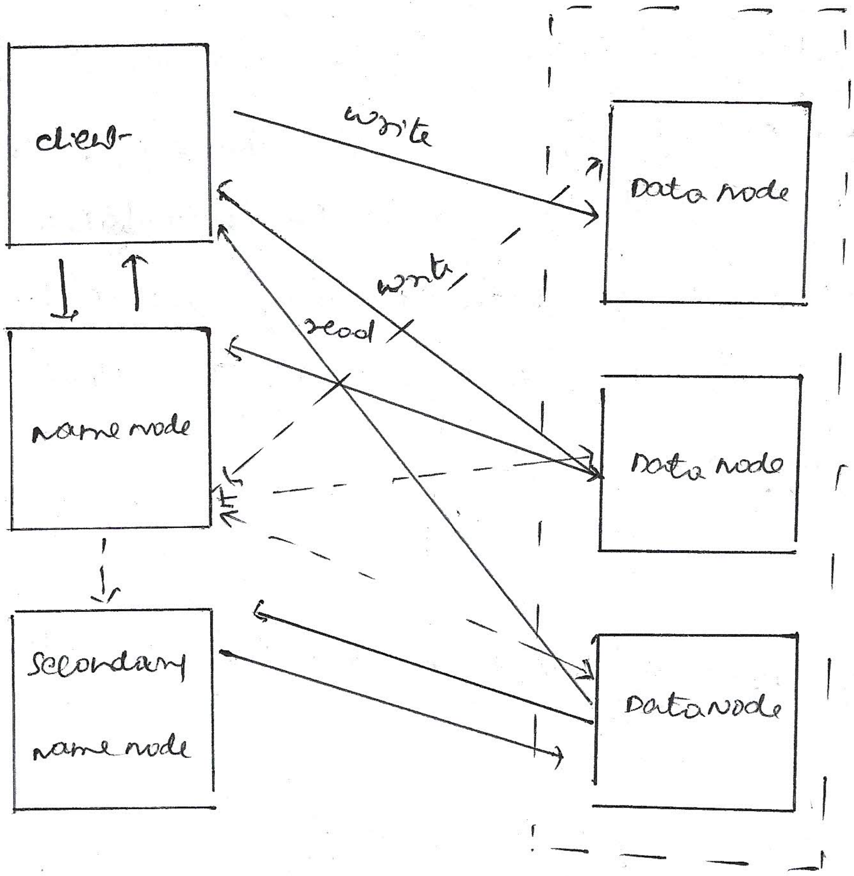
The design of HDFS is based on two types of nodes:

1) Name Node
2) Data Node

Name node manages all the metadata needed to store and retrieve actual data from the Data Node. no data is actually stored on the name node

master Name Node manages the file system namespace and regulate access to files by client. File System namespace operations such as opening, closing, and renaming files and directories are all managed by the namenode.

The slaves (Data Node) are responsible for serving read and write requests from the file system to the client. The name node manages block creation, deletion, and replication.



various System roles in an HDFS deployment.

Figure Shows client / name node / Data node interaction.

When a client write data, it first communicates with the Name node request to create a file. The Name node determines how many blocks are needed and provides client with the Data Node that will store the data.

The Name node will attempt to write replicas of the data blocks on nodes that are in other Separate block. If there is only one rack, then the replicated blocks are written to other server in the Same rack. After the Data node acknowledges that the file block replication is complete, the client closes the file and informs the name node that the operation is complete.

Reading data happens in a similiar fashion. The client request a file from the Name node, which returns the best Data Node from which to read the data. The client accen the data directly from the Data Node.

Once the meta data has been delivered to the client, the Namenode step back and lets the conversion b/w the client and the Data Node Proceed. While the data transfer is progressing, the name node also monitors the Data Node by listening for heart beats sent from DataNodes.

The purpose of the secondary name node is to perform periodic check point that evaluate the status of the name node.

Various roles of HDFS

1. HDFS uses a master/slave model designed for large file reading/streaming

2. The Name Node is a metadata server.

3. HDFS Provides a single Namespace that is managed by the namenode

4. HDFS Provides a single Namespace that is managed by the namenode.

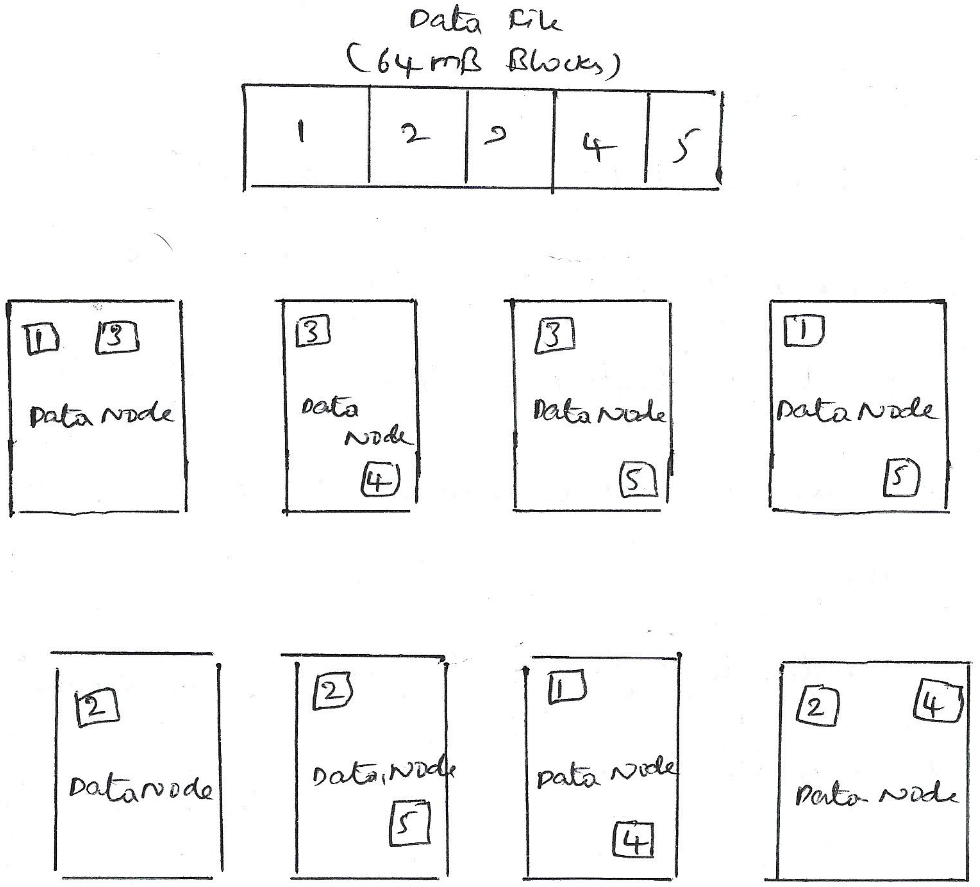
5. Data is redundantly stored on Data node. There is no data on the Name Node

6. Secondary Name node Performs check points of the Name Node file systems

HDFS Block Replication

when HDFS writes a file, it is replicated across the cluster. Hadoop cluster containing more than eight Data nodes, the replication value is usually set to 3. Hadoop cluster of eight or fewer DataNode but more than one Data Node, a replication factor is 2. For a single machine replication factor is 1.

HDFS default block size is 64 MB. In typical OS the block size is 4 KB or 8 KB. The HDFS default block size is not the minimum block size. If a 20 KB file is written to HDFS it will create a block that is approximately 20 KB only. If a file size is 80 MB, a 64 MB block and a 16 MB block will be created.



HDFS block replication example

Figure Provides an example of how a file is broken into blocks and replicated across the cluster. In this case, a replication factor of 3 ensures that any one Data Node can fail and the replicated blocks will be available on the other nodes— and then subseament re-replicated on other Data Node.

HDFS safe mode

When the Name node starts, it enters a read-only safe mode where blocks can't be replicated or deleted. Safe mode enables the Name Node to perform two important Process.

1. The previous file system state is reconstructed by loading the fs image file into memory and replying the edit log.

2. The mapping between blocks and data nodes is created by waiting for enough of the Data nodes to register so that at least one copy of the data is available. Not all the Data Nodes are required to register before HDFS exists from safe mode. The registration Process may continue for some time.

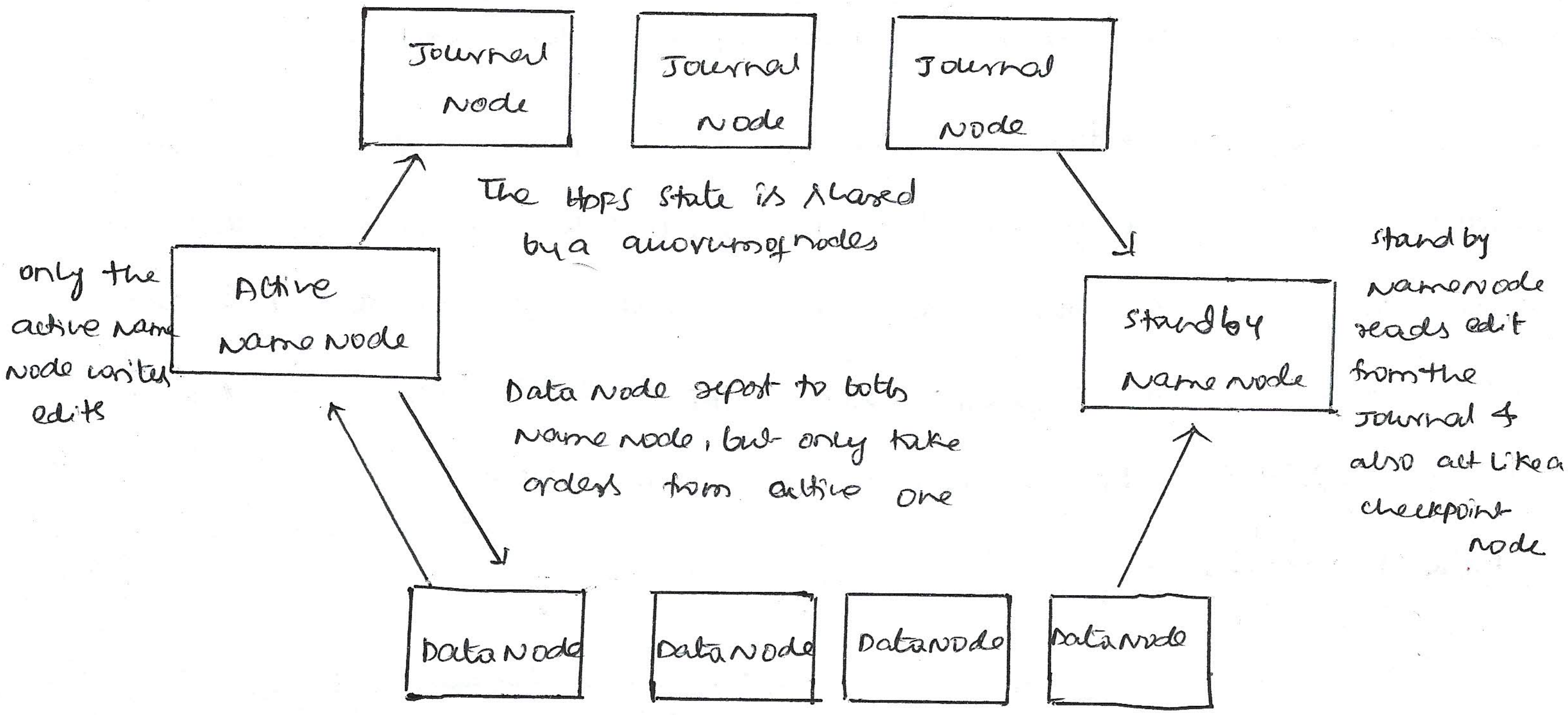
Rack Awareness

Rack awareness deals with data locality. Recall that one One of the main design goal of Hadoop MapReduce is to move the computation to the data. Assuming that most data center network don't ~~[illegible]~~ offer full bisection bandwidth, a typical hadoop cluster will exhibit three levels of data locality.

1. Data resides on the local machine (best)
2. Data resides on the same rack (better)
3. Data resides in a different rack (good).

Name node High Availabality

The name node was a single point of failure that could bring down the entire Hadoop cluster. Name node hardware often employed redundant power supplies and storage to guard against such problems, but it was still susceptible to other failure. The solution was to implement Name node High Availabality (HA) as a means to provide true failover service.


Journal Node
Journal Node
Journal Node
The HDFS state is shared by a quorum of nodes
only the active name node writes edits
Active Name Node
Standby Name node
Stand by Namenode reads edit from the Journal & also act like a checkpoint node
Data Node report to both Name node, but only take orders from active one
Data Node
Data Node
Data Node
Data Node

HDFS High Availabality design

An HA Hadoop cluster has two (or more) separate name node machines. Each machine is configured with exactly the same software one of the Name Node machines is in the Active state, and other is in the standby state. Like a single name node cluster, the Active name node is responsible for all client HDFS operations in the cluster. The standby name node maintains enough state to provide a fast failover.

To guarante the file system state is preserved, bothe the Active and Standby name node receive block reports from the Data Node. The Active node also sends all file system edit to a quoram of Journal nodes.

To prevent confusion between name node, the Journal nodes allow only one Name node to be a writer at a time. During ~~failover~~ failover, the Name node that is chosen to become active takes over the role of writing to the Journal nodes. A secondary namenode is not required in the HA configuration because the standby node also perform the task of the secondary Name node.

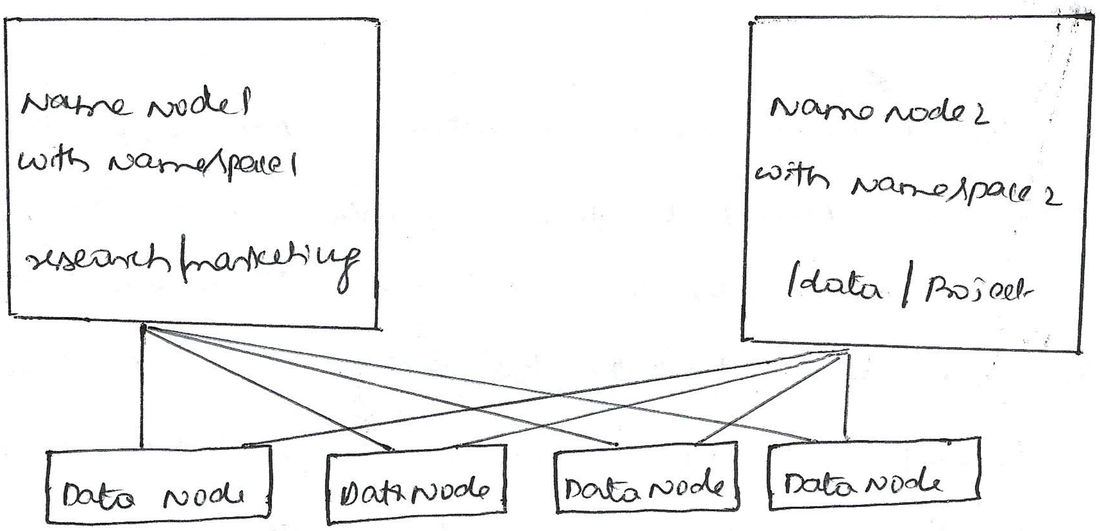
HDFS NameNode Federation

Another important feature of HDFS is namenode Federation. HDFS provided a single name space for the entire cluster managed by a single Name node. Thus resource of a single Name node determine the size of the name space. The key benefits are as follows

- Namespace scalability :- HDFS cluster storage scales horizontally without placing a burden on the Name node.

- Better performance :- Adding more name Node to the cluster scales the file systems read/write operations throughput by separating the total name space

- System isolation:- multiple Name Nodes enable different categories of applications to be distinguished, and user can be isolated to different name space.



HDFS Name Node federation

Figure illustrates how HDFS Name Node federation is accomplished. Name Node1 manages the /research and /marketing name spaces and Name node 2 manages the /data & /Project name spaces. The Name Node don't communicate with each other and the Data Nodes "just store data blocks" as directed by either Name Node.

HDFS checkpoints and Backup

Name Node Stores the metadata of the HDFS file system in a file called fsimage. File System modifications are written to an edit log file and at startup the Name Node merges the edits into a new fsimage. The secondary name Node or checkpoint Node periodically fetches edits from the name node, merge them and returns an updated fsimage to the Name node.

An HDFS Backup Node is simillar, but also maintain an up-to-date copy of the file system name space both in memory and on disk. A name node support one Backup Node at a time. No checkpoint nodes may be registered if a Backup node is in use.

HDFS Snapshots

HDFS snapshots are simillar to backups, but are created by administrators using the hdfs dfs -snapshot command. HDFS snapshots are read-only point-in-time copies of the file system. They offer following features.

1. Snap shots can be taken of a sub-tree of the file system or the entire file system.

2. Snap Shots can be used for data backup, protection against user errors, and disaster recovery.

3. Snap shots creation is instantaneous.

4. Blocks on the data node are not copied, because the snapshot files recorded the block list and the file size.

5. Snap shots donot adversely affect regular HDFS operations.

HDFS NFS Gateway

The HDFS NFS Gateway supports NFSV3 and enables HDFS to be mounted as part of the client's local file system. Users can browse the HDFS file system through their local file system that provide an NFSV3 client compatible OS. This feature offers users the following capabilities:

1. User can easily download/upload files from/to the HDFS file system to/from their local file system.

2. User can stream data directly to HDFS through the mount point. Appending to a file is supported, but random write capabality is not supported.

Module 2

1. Essential Hadoop Tools

In This Chapter:

- The Pig scripting tool is introduced as a way to quickly examine data both locally and on a Hadoop cluster.
- The Hive SQL-like query tool is explained using two examples.
- The Sqoop RDBMS tool is used to import and export data from MySQL to/from HDFS.
- The Flume streaming data transport utility is configured to capture weblog data into HDFS.
- The Oozie workflow manager is used to run basic and complex Hadoop workflows.
- The distributed HBase database is used to store and access data on a Hadoop cluster.

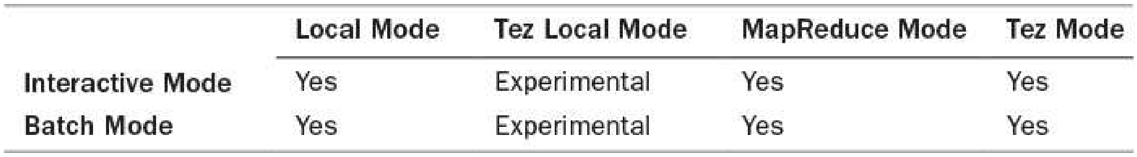
USING APACHE PIG

Apache Pig is a high-level language that enables programmers to write complex MapReduce transformations using a simple scripting language. Pig Latin (the actual language) defines a set of transformations on a data set such as aggregate, join, and sort.

Apache Pig has several usage modes.

- The first is a local mode in which all processing is done on the local machine.
- The non-local (cluster) modes are MapReduce and Tez. These modes execute the job on the cluster using either the MapReduce engine or the optimized Tez engine.

There are also interactive and batch modes available; they enable Pig applications to be developed locally in interactive modes, using small amounts of data, and then run at scale on the cluster in a production mode. The modes are summarized in Table 7.1.



	Local Mode	Tez Local Mode	MapReduce Mode	Tez Mode
Interactive Mode	Yes	Experimental	Yes	Yes
Batch Mode	Yes	Experimental	Yes	Yes

Table 7.1 **Apache Pig Usage Modes**

Pig Example Walk-Through

In this simple example, Pig is used The following example assumes the user is hdfs, but any valid user with access to HDFS can run the example.

To begin the example, copy the passwd file to a working directory for local Pig operation:

```
$ cp /etc/passwd .
```

Next, copy the data file into HDFS for Hadoop MapReduce operation:

```
$ hdfs dfs -put passwd passwd
```

You can confirm the file is in HDFS by entering the following command:

```
hdfs dfs -ls passwd
-rw-r--r--   2 hdfs hdfs       2526 2015-03-17 11:08 passwd
```

In the following example of local Pig operation, all processing is done on the local machine (Hadoop is not used). First, the interactive command line is started:

```
$ pig -x local
```

If Pig starts correctly, you will see a grunt> prompt. Next, enter the following commands to load the passwd file and then grab the user name and dump it to the terminal. Note that Pig commands must end with a semicolon (;).

```
grunt> A = load 'passwd' using PigStorage(':');
grunt> B = foreach A generate $0 as id;
grunt> dump B;
```

The processing will start and a list of user names will be printed to the screen. To exit the interactive session, enter the command quit.

```
$ grunt> quit
```

To use Hadoop MapReduce, start Pig as follows (or just enter pig):

```
$ pig -x mapreduce
```

The same sequence of commands can be entered at the grunt> prompt. You may wish to change the $0 argument to pull out other items in the passwd file. Also, because we are running this application under Hadoop, make sure the file is placed in HDFS.

If you are using the Hortonworks HDP distribution with tez installed, the tez engine can be used as follows:

```
$ pig -x tez
```

Pig can also be run from a script. This script, which is repeated here, is designed to do the same things as the interactive version:

```
/* id.pig */
A = load 'passwd' using PigStorage(':');  -- load the passwd file
B = foreach A generate $0 as id;  -- extract the user IDs
dump B;
store B into 'id.out'; -- write the results to a directory name id.out
```

Comments are delineated by /* */ and -- at the end of a line. First, ensure that the id.out directory is not in your local directory, and then start Pig with the script on the command line:

```
$ /bin/rm -r id.out/
$ pig -x local id.pig
```

If the script worked correctly, you should see at least one data file with the results and a zero-length file with the name _SUCCESS. To run the MapReduce version, use the same procedure; the only difference is that now all reading and writing takes place in HDFS.

```
$ hdfs dfs -rm -r id.out
$ pig id.pig
```

USING APACHE HIVE

Apache Hive is a data warehouse infrastructure built on top of Hadoop for providing data summarization, ad hoc queries, and the analysis of large data sets using a SQL-like language called HiveQL. Hive offers the following features:

- Tools to enable easy data extraction, transformation, and loading (ETL)
- A mechanism to impose structure on a variety of data formats

- Access to files stored either directly in HDFS or in other data storage systems such as HBase
- Query execution via MapReduce and Tez (optimized MapReduce)

Hive Example Walk-Through

To start Hive, simply enter the hive command. If Hive starts correctly, you should get a hive> prompt.

```
$ hive
(some messages may show up here)
hive>
```

As a simple test, create and drop a table. Note that Hive commands must end with a semicolon (;).

```
hive> CREATE TABLE pokes (foo INT, bar STRING);
OK
Time taken: 1.705 seconds
hive> SHOW TABLES;
OK
pokes
Time taken: 0.174 seconds, Fetched: 1 row(s)
hive> DROP TABLE pokes;
OK
Time taken: 4.038 seconds
```

A more detailed example can be developed using a web server log file to summarize message types. First, create a table using the following command:

```
hive> CREATE TABLE logs(t1 string, t2 string, t3 string, t4 string, t5 string, t6 string, t7 string) ROW
FORMAT DELIMITED FIELDS TERMINATED BY ' ';
OK
Time taken: 0.129 seconds
```

Next, load the data—in this case, from the sample.log file. Note that the file is found in the local directory and not in HDFS.

```
hive> LOAD DATA LOCAL INPATH 'sample.log' OVERWRITE INTO TABLE logs;
Loading data to table default.logs
Table default.logs stats: [numFiles=1, numRows=0, totalSize=99271, rawDataSize=0]
OK
Time taken: 0.953 seconds
```

Finally, apply the select step to the file. Note that this invokes a Hadoop MapReduce operation. The results appear at the end of the output (e.g., totals for the message types DEBUG, ERROR, and so on).

```
hive> SELECT t4 AS sev, COUNT(*) AS cnt FROM logs WHERE t4 LIKE '[%' GROUP BY t4;
Query ID = hdfs_20150327130000_d1e1a265-a5d7-4ed8-b785-2c6569791368
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1427397392757_0001, Tracking URL = http://norbert:8088/proxy/
application_1427397392757_0001/
Kill Command = /opt/hadoop-2.6.0/bin/hadoop job  -kill job_1427397392757_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2015-03-27 13:00:17,399 Stage-1 map = 0%,  reduce = 0%
2015-03-27 13:00:26,100 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 2.14 sec
2015-03-27 13:00:34,979 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 4.07 sec
MapReduce Total cumulative CPU time: 4 seconds 70 msec
Ended Job = job_1427397392757_0001
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 4.07 sec   HDFS Read: 106384
HDFS Write: 63 SUCCESS
Total MapReduce CPU Time Spent: 4 seconds 70 msec
OK
[DEBUG] 434
[ERROR] 3
[FATAL] 1
[INFO]  96
[TRACE] 816
[WARN]  4
Time taken: 32.624 seconds, Fetched: 6 row(s)
```

To exit Hive, simply type exit;

```
hive> exit;
```

A More Advanced Hive Example

In this example, 100,000 records will be transformed from userid, movieid, rating, unixtime to userid, movieid, rating, and weekday using Apache Hive and a Python program (i.e., the UNIX time notation will be transformed to the day of the week). The first step is to download and extract the data:

```
$ wget http://files.grouplens.org/datasets/movielens/ml-100k.zip
$ unzip ml-100k.zip
$ cd ml-100k
```

Before we use Hive, we will create a short Python program called weekday_mapper.py with following contents:

```
import sys
import datetime

for line in sys.stdin:
  line = line.strip()
  userid, movieid, rating, unixtime = line.split('\t')
  weekday = datetime.datetime.fromtimestamp(float(unixtime)).isoweekday()
  print '\t'.join([userid, movieid, rating, str(weekday)])LOAD DATA LOCAL INPATH './u.data'
OVERWRITE INTO TABLE u_data;
```

Next, start Hive and create the data table (u_data) by entering the following at the hive> prompt:

```
CREATE TABLE u_data (
  userid INT,
  movieid INT,
  rating INT,
  unixtime STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE;
```

Load the movie data into the table with the following command:

```
hive> LOAD DATA LOCAL INPATH './u.data' OVERWRITE INTO TABLE u_data;
```

The number of rows in the table can be reported by entering the following command:

```
hive > SELECT COUNT(*) FROM u_data;
```

This command will start a single MapReduce job and should finish with the following lines:

```
...
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 2.26 sec   HDFS Read: 1979380
HDFS Write: 7 SUCCESS
Total MapReduce CPU Time Spent: 2 seconds 260 msec
OK
100000
Time taken: 28.366 seconds, Fetched: 1 row(s)
```

Now that the table data are loaded, use the following command to make the new table (u_data_new):

```
hive> CREATE TABLE u_data_new (
  userid INT,
  movieid INT,
  rating INT,
  weekday INT)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t';
```

The next command adds the weekday_mapper.py to Hive resources:

```
hive> add FILE weekday_mapper.py;
```

Once weekday_mapper.py is successfully loaded, we can enter the transformation query:

```
hive> INSERT OVERWRITE TABLE u_data_new
SELECT
  TRANSFORM (userid, movieid, rating, unixtime)
  USING 'python weekday_mapper.py'
  AS (userid, movieid, rating, weekday)
FROM u_data;
```

If the transformation was successful, the following final portion of the output should be displayed:

```
...
Table default.u_data_new stats: [numFiles=1, numRows=100000, totalSize=1179173,
rawDataSize=1079173]
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1   Cumulative CPU: 3.44 sec   HDFS Read: 1979380 HDFS Write:
1179256 SUCCESS
Total MapReduce CPU Time Spent: 3 seconds 440 msec
OK
Time taken: 24.06 seconds
```

The final query will sort and group the reviews by weekday:

```
hive> SELECT weekday, COUNT(*) FROM u_data_new GROUP BY weekday;
```

Final output for the review counts by weekday should look like the following:

```
...
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 2.39 sec   HDFS Read: 1179386
HDFS Write: 56 SUCCESS
Total MapReduce CPU Time Spent: 2 seconds 390 msec
OK
1       13278
2       14816
3       15426
4       13774
5       17964
6       12318
7       12424
Time taken: 22.645 seconds, Fetched: 7 row(s)
```

As shown previously, you can remove the tables used in this example with the DROP TABLE command. In this case, we are also using the -e command-line option. Note that queries can be loaded from files using the -f option as well.

```
$ hive -e 'drop table u_data_new'
$ hive -e 'drop table u_data'
```

USING APACHE SQOOP TO ACQUIRE RELATIONAL DATA

Sqoop is a tool designed to transfer data between Hadoop and relational databases. You can use Sqoop to import data from a relational database management system (RDBMS) into the Hadoop Distributed File System (HDFS), transform the data in Hadoop, and then export the data back into an RDBMS.

Sqoop can be used with any Java Database Connectivity (JDBC)–compliant database and has been tested on Microsoft SQL Server, PostgresSQL, MySQL, and Oracle.

Apache Sqoop Import and Export Methods

Figure 7.1 describes the Sqoop data import (to HDFS) process. The data import is done in two steps. In the first step, shown in the figure, Sqoop examines the database to gather the necessary metadata for the data to be imported. The second step is a map-only (no reduce step) Hadoop job that Sqoop submits to the cluster. This job does the actual data transfer using the metadata captured in the previous step. Note that each node doing the import must have access to the database.

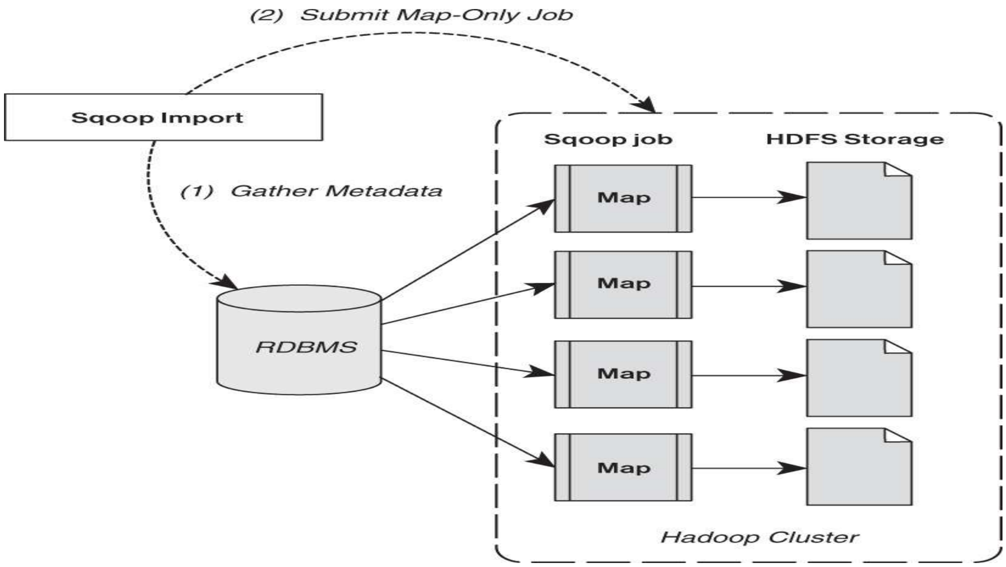


Figure 7.1 Two-step Apache Sqoop data import method (Adapted from Apache Sqoop Documentation)

The imported data are saved in an HDFS directory. Sqoop will use the database name for the directory, or the user can specify any alternative directory where the files should be populated. By default, these files contain comma-delimited fields, with new lines separating different records. You can easily override the format in which data are copied over by explicitly specifying the field separator and record terminator characters. Once placed in HDFS, the data are ready for processing.

Data export from the cluster works in a similar fashion. The export is done in two steps, as shown in Figure 7.2. As in the import process, the first step is to examine the database for metadata. The export step again uses a map-only Hadoop job to write the data to the database. Sqoop divides the input data set into splits, then uses individual map tasks to push the splits to the database. Again, this process assumes the map tasks have access to the database.

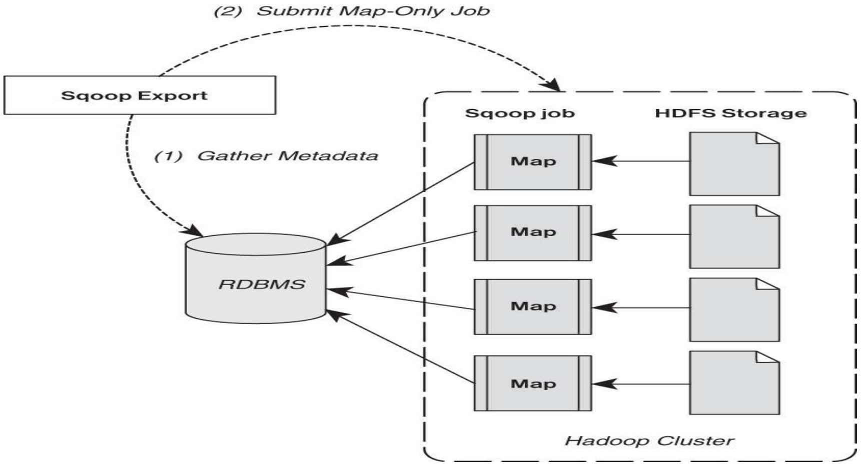
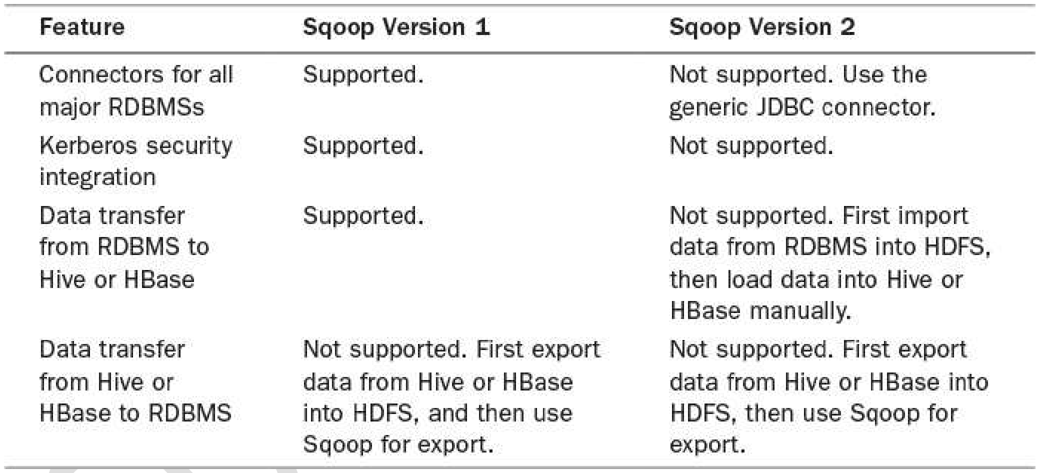


Figure 7.2 Two-step Sqoop data export method (Adapted from Apache Sqoop Documentation)

Apache Sqoop Version Changes

Sqoop Version 1 uses specialized connectors to access external systems. These connectors are often optimized for various RDBMSs or for systems that do not support JDBC. Connectors are plug-in components based on Sqoop's extension framework and can be added to any existing Sqoop installation. Once a connector is installed, Sqoop can use it to efficiently transfer data between Hadoop and the external store supported by the connector. By default, Sqoop version 1 includes connectors for popular databases such as MySQL, PostgreSQL, Oracle, SQL Server, and DB2. It also supports direct transfer to and from the RDBMS to HBase or Hive.

In contrast, to streamline the Sqoop input methods, Sqoop version 2 no longer supports specialized connectors or direct import into HBase or Hive. All imports and exports are done through the JDBC interface. Table 7.2 summarizes the changes from version 1 to version 2. Due to these changes, any new development should be done with Sqoop version 2.



Feature	Sqoop Version 1	Sqoop Version 2
Connectors for all major RDBMSs	Supported.	Not supported. Use the generic JDBC connector.
Kerberos security integration	Supported.	Not supported.
Data transfer from RDBMS to Hive or HBase	Supported.	Not supported. First import data from RDBMS into HDFS, then load data into Hive or HBase manually.
Data transfer from Hive or HBase to RDBMS	Not supported. First export data from Hive or HBase into HDFS, and then use Sqoop for export.	Not supported. First export data from Hive or HBase into HDFS, then use Sqoop for export.

Table 7.2 **Apache Sqoop Version Comparison**

Sqoop Example Walk-Through

The following simple example illustrates use of Sqoop

Step 1: Load Sample MySQL Database

```
$ wget http://downloads.mysql.com/docs/world_innodb.sql.gz
$ gunzip world_innodb.sql.gz
```

Next, log into MySQL (assumes you have privileges to create a database) and import the desired database by following these steps:

```
  $ mysql -u root -p
  mysql> CREATE DATABASE world;
  mysql> USE world;
  mysql> SOURCE world_innodb.sql;
  mysql> SHOW TABLES;
  +-----------------+
  | Tables_in_world |
  +-----------------+
  | City            |
  | Country         |
  | CountryLanguage |
  +-----------------+
  3 rows in set (0.01 sec)
```

The following MySQL command will let you see the table details.

Step 2: Add Sqoop User Permissions for the Local Machine and Cluster

In MySQL, add the following privileges for user sqoop to MySQL. Note that you must use both the local host name and the cluster subnet for Sqoop to work properly. Also, for the purposes of this example, the sqoop password is sqoop.

```
mysql> GRANT ALL PRIVILEGES ON world.* To 'sqoop'@'limulus' IDENTIFIED BY 'sqoop';
mysql> GRANT ALL PRIVILEGES ON world.* To 'sqoop'@'10.0.0.%' IDENTIFIED BY 'sqoop';
mysql> quit
```

Next, log in as sqoop to test the permissions:

```
$ mysql -u sqoop -p
  mysql> USE world;
  mysql> SHOW TABLES;
  +-----------------+
  | Tables_in_world |
  +-----------------+
  | City            |
  | Country         |
  | CountryLanguage |
  +-----------------+
  3 rows in set (0.01 sec)

  mysql> quit
```

Step 3: Import Data Using Sqoop

As a test, we can use Sqoop to list databases in MySQL. The results appear after the warnings at the end of the output. Note the use of local host name (limulus) in the JDBC statement.

```
$ sqoop list-databases --connect jdbc:mysql://limulus/world --username sqoop --password sqoop
  Warning: /usr/lib/sqoop/../accumulo does not exist! Accumulo imports will fail.
  Please set $ACCUMULO_HOME to the root of your Accumulo installation.
  14/08/18 14:38:55 INFO sqoop.Sqoop: Running Sqoop version: 1.4.4.2.1.2.1-471
  14/08/18 14:38:55 WARN tool.BaseSqoopTool: Setting your password on the
command-line is insecure. Consider using -P instead.
  14/08/18 14:38:55 INFO manager.MySQLManager: Preparing to use a MySQL streaming
resultset.
  information_schema
```

```
   test
   world
```

In a similar fashion, you can use Sqoop to connect to MySQL and list the tables in the world database:

```
  sqoop list-tables --connect jdbc:mysql://limulus/world --username sqoop --password sqoop
   ...
   14/08/18 14:39:43 INFO sqoop.Sqoop: Running Sqoop version: 1.4.4.2.1.2.1-471
   14/08/18 14:39:43 WARN tool.BaseSqoopTool: Setting your password on the
command-line is insecure. Consider using -P instead.
   14/08/18 14:39:43 INFO manager.MySQLManager: Preparing to use a MySQL streaming
resultset.
   City
   Country
   CountryLanguage
```

To import data, we need to make a directory in HDFS:

```
$ hdfs dfs -mkdir sqoop-mysql-import
```

The following command imports the Country table into HDFS. The option -table signifies the table to import, --target-dir is the directory created previously, and -m 1 tells Sqoop to use one map task to import the data.

```
$ sqoop import --connect jdbc:mysql://limulus/world  --username sqoop --password sqoop --table
Country  -m 1 --target-dir /user/hdfs/sqoop-mysql-import/country
   ...
   14/08/18 16:47:15 INFO mapreduce.ImportJobBase: Transferred 30.752 KB in
12.7348 seconds
   (2.4148 KB/sec)
   14/08/18 16:47:15 INFO mapreduce.ImportJobBase: Retrieved 239 records.
```

The import can be confirmed by examining HDFS:

```
$ hdfs dfs -ls sqoop-mysql-import/country
   Found 2 items
   -rw-r--r--   2 hdfs hdfs          0 2014-08-18 16:47 sqoop-mysql-import/
world/_SUCCESS
   -rw-r--r--   2 hdfs hdfs      31490 2014-08-18 16:47 sqoop-mysql-import/world/
part-m-00000
```

The file can be viewed using the hdfs dfs -cat command:

```
$ hdfs dfs -cat sqoop-mysql-import/country/part-m-00000
   ABW,Aruba,North America,Caribbean,193.0,null,103000,78.4,828.0,793.0,Aruba,
Nonmetropolitan
   Territory of The Netherlands,Beatrix,129,AW
   ...
   ZWE,Zimbabwe,Africa,Eastern Africa,390757.0,1980,11669000,37.8,5951.0,8670.0,
Zimbabwe,
   Republic,Robert G. Mugabe,4068,ZW
```

To make the Sqoop command more convenient, you can create an options file and use it on the command line. Such a file enables you to avoid having to rewrite the same options. For example, a file called world-options.txt with the following contents will include the import command, --connect, --username, and --password options:

```
import
--connect
jdbc:mysql://limulus/world
--username
sqoop
--password
sqoop
```

The same import command can be performed with the following shorter line:

```
$ sqoop  --options-file world-options.txt --table City  -m 1 --target-dir /user/hdfs/sqoop-mysql-import/city
```

It is also possible to include an SQL Query in the import step. For example, suppose we want just cities in Canada:

```
SELECT ID,Name from City WHERE CountryCode='CAN'
```

In such a case, we can include the --query option in the Sqoop import request. The --query option also needs a variable called $CONDITIONS, which will be explained next. In the following query example, a single mapper task is designated with the -m 1 option:

```
sqoop  --options-file world-options.txt -m 1 --target-dir /user/hdfs/sqoop-mysql-import/canada-city --
query "SELECT ID,Name from City WHERE CountryCode='CAN' AND \$CONDITIONS"
```

Inspecting the results confirms that only cities from Canada have been imported:

```
$ hdfs dfs -cat sqoop-mysql-import/canada-city/part-m-00000
1810,MontrÄal
1811,Calgary
1812,Toronto
...
1856,Sudbury
1857,Kelowna
1858,Barrie
```

Since there was only one mapper process, only one copy of the query needed to be run on the database. The results are also reported in a single file (part-m-0000).

Multiple mappers can be used to process the query if the --split-by option is used. The split-by option is used to parallelize the SQL query. Each parallel task runs a subset of the main query, with the results of each sub-query being partitioned by bounding conditions inferred by Sqoop. Your query must include the token $CONDITIONS that each Sqoop process will replace with a unique condition expression based on the --split-by option. Note that $CONDITIONS is not an environment variable. Although Sqoop will try to create balanced sub-queries based on the range of your primary key, it may be necessary to split on another column if your primary key is not uniformly distributed.

The following example illustrates the use of the --split-by option. First, remove the results of the previous query:

```
  $ hdfs dfs -rm -r -skipTrash  sqoop-mysql-import/canada-city
```

Next, run the query using four mappers (-m 4), where we split by the ID number (--split-by ID):

```
sqoop  --options-file world-options.txt -m 4 --target-dir /user/hdfs/sqoop-mysql-import/canada-city --
query "SELECT ID,Name from City WHERE CountryCode='CAN' AND \$CONDITIONS" --split-by
ID
```

If we look at the number of results files, we find four files corresponding to the four mappers we requested in the command:

```
$ hdfs dfs -ls  sqoop-mysql-import/canada-city
Found 5 items
-rw-r--r--   2 hdfs hdfs       0 2014-08-18 21:31 sqoop-mysql-import/
canada-city/_SUCCESS
-rw-r--r--   2 hdfs hdfs     175 2014-08-18 21:31 sqoop-mysql-import/canada-city/
part-m-00000
-rw-r--r--   2 hdfs hdfs     153 2014-08-18 21:31 sqoop-mysql-import/canada-city/
part-m-00001
-rw-r--r--   2 hdfs hdfs     186 2014-08-18 21:31 sqoop-mysql-import/canada-city/
part-m-00002
-rw-r--r--   2 hdfs hdfs     182 2014-08-18 21:31 sqoop-mysql-import/canada-city/
part-m-00003
```

Step 4: Export Data from HDFS to MySQL

Sqoop can also be used to export data from HDFS. The first step is to create tables for exported data. There are actually two tables needed for each exported table. The first table holds the exported data (CityExport), and the second is used for staging the exported data (CityExportStaging). Enter the following MySQL commands to create these tables:

```
  mysql> CREATE TABLE 'CityExport' (
         'ID' int(11) NOT NULL AUTO_INCREMENT,
         'Name' char(35) NOT NULL DEFAULT '',
         'CountryCode' char(3) NOT NULL DEFAULT '',
         'District' char(20) NOT NULL DEFAULT '',
         'Population' int(11) NOT NULL DEFAULT '0',
         PRIMARY KEY ('ID'));
  mysql> CREATE TABLE 'CityExportStaging' (
         'ID' int(11) NOT NULL AUTO_INCREMENT,
         'Name' char(35) NOT NULL DEFAULT '',
         'CountryCode' char(3) NOT NULL DEFAULT '',
         'District' char(20) NOT NULL DEFAULT '',
         'Population' int(11) NOT NULL DEFAULT '0',
         PRIMARY KEY ('ID'));
```

Next, create a cities-export-options.txt file similar to the world-options.txt created previously, but use the export command instead of the import command.

The following command will export the cities data we previously imported back into MySQL:

```
sqoop --options-file cities-export-options.txt --table CityExport  --staging-table CityExportStaging  --
clear-staging-table -m 4 --export-dir /user/hdfs/sqoop-mysql-import/city
```

Finally, to make sure everything worked correctly, check the table in MySQL to see if the cities are in the table:

```
$ mysql> select * from CityExport limit 10;
+----+----------------+-------------+---------------+------------+
| ID | Name           | CountryCode | District      | Population |
+----+----------------+-------------+---------------+------------+
|  1 | Kabul          | AFG         | Kabol         |    1780000 |
|  2 | Qandahar       | AFG         | Qandahar      |     237500 |
|  3 | Herat          | AFG         | Herat         |     186800 |
|  4 | Mazar-e-Sharif | AFG         | Balkh         |     127800 |
|  5 | Amsterdam      | NLD         | Noord-Holland |     731200 |
|  6 | Rotterdam      | NLD         | Zuid-Holland  |     593321 |
|  7 | Haag           | NLD         | Zuid-Holland  |     440900 |
|  8 | Utrecht        | NLD         | Utrecht       |     234323 |
|  9 | Eindhoven      | NLD         | Noord-Brabant |     201843 |
| 10 | Tilburg        | NLD         | Noord-Brabant |     193238 |
+----+----------------+-------------+---------------+------------+
10 rows in set (0.00 sec)
```

Some Handy Cleanup Commands

If you are not especially familiar with MySQL, the following commands may be helpful to clean up the examples. To remove the table in MySQL, enter the following command:

```
mysql> drop table 'CityExportStaging';
```

To remove the data in a table, enter this command:

```
mysql> delete from CityExportStaging;
```

To clean up imported files, enter this command:

```
$ hdfs dfs -rm -r  -skipTrash sqoop-mysql-import/{country,city, canada-city}
```

USING APACHE FLUME TO ACQUIRE DATA STREAMS

Apache Flume is an independent agent designed to collect, transport, and store data into HDFS. Often data transport involves a number of Flume agents that may traverse a series of machines and locations. Flume is often used for log files, social media-generated data, email messages, and just about any continuous data source. As shown in Figure 7.3, a Flume agent is composed of three components.

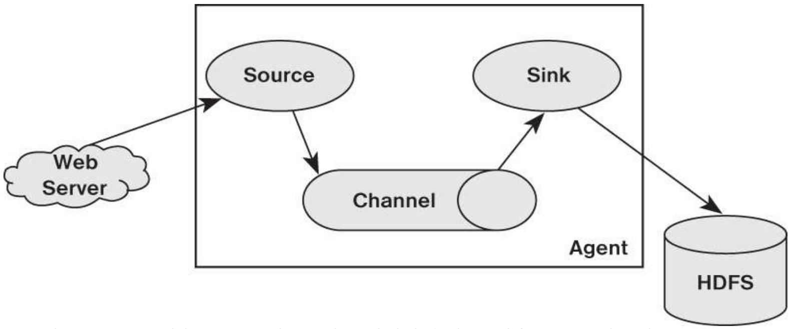


Figure 7.3 Flume agent with source, channel, and sink (Adapted from Apache Flume Documentation)

■ *Source.* The source component receives data and sends it to a channel. It can send the data to more than one channel. The input data can be from a real-time source (e.g., weblog) or another Flume agent.

■ *Channel.* A channel is a data queue that forwards the source data to the sink destination. It can be thought of as a buffer that manages input (source) and output (sink) flow rates.

■ *Sink.* The sink delivers data to destination such as HDFS, a local file, or another Flume agent.

A Flume agent must have all three of these components defined. A Flume agent can have several sources, channels, and sinks. Sources can write to multiple channels, but a sink can take data from only a single channel. Data written to a channel remain in the channel until a sink removes the data. By default, the data in a channel are kept in memory but may be optionally stored on disk to prevent data loss in the event of a network failure.

As shown in Figure 7.4, Sqoop agents may be placed in a pipeline, possibly to traverse several machines or domains. This configuration is normally used when data are collected on one machine (e.g., a web server) and sent to another machine that has access to HDFS.

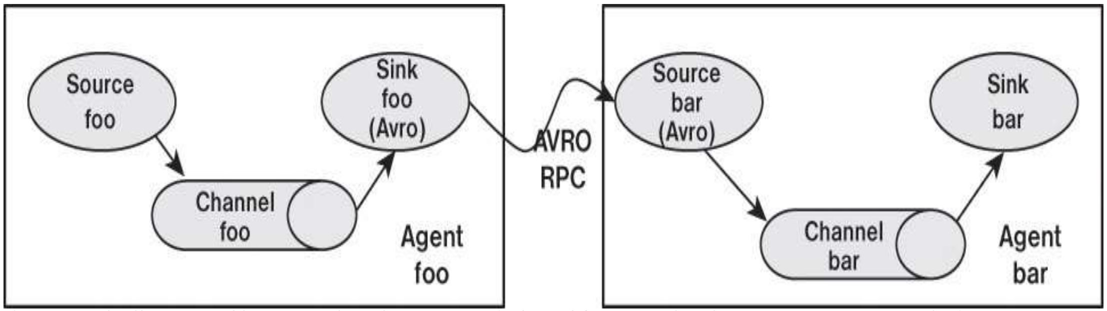


Figure 7.4 Pipeline created by connecting Flume agents (Adapted from Apache Flume Sqoop Documentation)

In a Flume pipeline, the sink from one agent is connected to the source of another. The data transfer format normally used by Flume, which is called Apache Avro, provides several useful features. First, Avro is a data serialization/deserialization system that uses a compact binary format. The schema is sent as part of the data exchange and is defined using JSON (JavaScript Object Notation). Avro also uses remote procedure calls (RPCs) to send data. That is, an Avro sink will contact an Avro source to send data.

Another useful Flume configuration is shown in Figure 7.5. In this configuration, Flume is used to consolidate several data sources before committing them to HDFS.

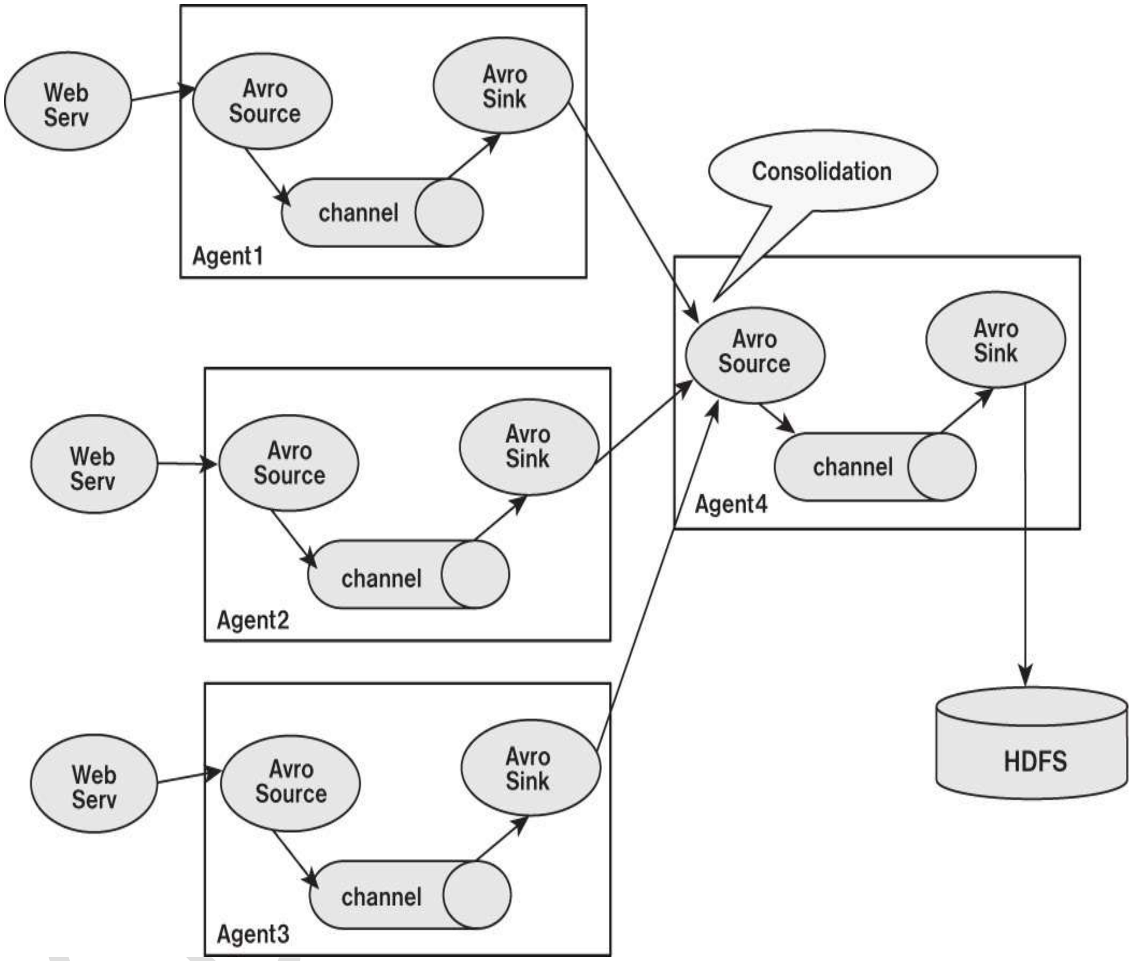


Figure 7.5 A Flume consolidation network (Adapted from Apache Flume Documentation)

There are many possible ways to construct Flume transport networks. In addition, other Flume features not described in depth here include plug-ins and interceptors that can enhance Flume pipelines.

Flume Example Walk-Through

Follow these steps to walk through a Flume example.

Step 1: Download and Install Apache Flume

Step 2: Simple Test

A simple test of Flume can be done on a single machine. To start the Flume agent, enter the flume-ng command shown here. This command uses the simple-example.conf file to configure the agent.

```
$ flume-ng agent --conf conf --conf-file simple-example.conf --name simple_agent -
Dflume.root.logger=INFO,console
```

In another terminal window, use telnet to contact the agent:

```
$ telnet localhost 44444
  Trying ::1...
  telnet: connect to address ::1: Connection refused
  Trying 127.0.0.1...
  Connected to localhost.
  Escape character is '^]'.
  testing  1 2 3
  OK
```

If Flume is working correctly, the window where the Flume agent was started will show the testing message entered in the telnet window:

Step 3: Weblog Example

In this example, a record from the weblogs from the local machine (Ambari output) will be placed into HDFS using Flume. This example is easily modified to use other weblogs from different machines. Two files are needed to configure Flume. (See the sidebar and Appendix A for file downloading instructions.)

- `web-server-target-agent.conf`—the target Flume agent that writes the data to HDFS
- `web-server-source-agent.conf`—the source Flume agent that captures the weblog data

The weblog is also mirrored on the local file system by the agent that writes to HDFS. To run the example, create the directory as root:

```
# mkdir /var/log/flume-hdfs
# chown hdfs:hadoop /var/log/flume-hdfs/
```

Next, as user hdfs, make a Flume data directory in HDFS:

```
$ hdfs dfs -mkdir /user/hdfs/flume-channel/
```

Now that you have created the data directories, you can start the Flume target agent (execute as user hdfs):

```
$ flume-ng agent -c conf -f web-server-target-agent.conf -n collector
```

This agent writes the data into HDFS and should be started before the source agent. (The source reads the weblogs.) This configuration enables automatic use of the Flume agent. The /etc/flume/conf/{flume.conf, flume-env.sh.template} files need to be configured for this purpose. For this example, the /etc/flume/conf/flume.conf file can be the same as the web-server-target.conf file (modified for your environment).

In this example, the source agent is started as root, which will start to feed the weblog data to the target agent. Alternatively, the source agent can be on another machine if desired.

```
# flume-ng agent -c conf -f web-server-source-agent.conf -n source_agent
```

To see if Flume is working correctly, check the local log by using the tail command. Also confirm that the flume-ng agents are not reporting any errors (the file name will vary).

```
$ tail -f /var/log/flume-hdfs/1430164482581-1
```

The contents of the local log under flume-hdfs should be identical to that written into HDFS. You can inspect this file by using the hdfs -tail command (the file name will vary). Note that while running Flume, the most recent file in HDFS may have the extension .tmp appended to it. The .tmpindicates that the file is still being written by Flume. The target agent can be configured to write the file (and start another .tmp file) by setting some or all of the rollCount, rollSize, rollInterval, idleTimeout, and batchSize options in the configuration file.

```
$ hdfs dfs -tail flume-channel/apache_access_combined/150427/FlumeData.1430164801381
```

Both files should contain the same data. For instance, the preceding example had the following data in both files:

```
10.0.0.1 - - [27/Apr/2015:16:04:21 -0400] "GET /ambarinagios/nagios/
nagios_alerts.php?q1=alerts&alert_type=all HTTP/1.1" 200 30801 "-" "Java/1.7.0_65"
10.0.0.1 - - [27/Apr/2015:16:04:25 -0400] "POST /cgi-bin/rrd.py HTTP/1.1" 200 784
"-" "Java/1.7.0_65"
10.0.0.1 - - [27/Apr/2015:16:04:25 -0400] "POST /cgi-bin/rrd.py HTTP/1.1" 200 508
"-" "Java/1.7.0_65"
```

MANAGE HADOOP WORKFLOWS WITH APACHE OOZIE

Oozie is a workflow director system designed to run and manage multiple related Apache Hadoop jobs. For instance, complete data input and analysis may require several discrete Hadoop jobs to be run as a workflow in which the output of one job serves as the input for a successive job. Oozie is designed to construct and manage these workflows. Oozie is not a substitute for the YARN scheduler. That is, YARN manages resources for individual Hadoop jobs, and Oozie provides a way to connect and control Hadoop jobs on the cluster.

Oozie workflow jobs are represented as directed acyclic graphs (DAGs) of actions. (DAGs are basically graphs that cannot have directed loops.) Three types of Oozie jobs are permitted:

- Workflow—a specified sequence of Hadoop jobs with outcome-based decision points and control dependency. Progress from one action to another cannot happen until the first action is complete.

- Coordinator—a scheduled workflow job that can run at various time intervals or when data become available.

- Bundle—a higher-level Oozie abstraction that will batch a set of coordinator jobs.

Oozie is integrated with the rest of the Hadoop stack, supporting several types of Hadoop jobs out of the box (e.g., Java MapReduce, Streaming MapReduce, Pig, Hive, and Sqoop) as well as system-specific jobs (e.g., Java programs and shell scripts). Oozie also provides a CLI and a web UI for monitoring jobs.

Figure 7.6 depicts a simple Oozie workflow. In this case, Oozie runs a basic MapReduce operation. If the application was successful, the job ends; if an error occurred, the job is killed.

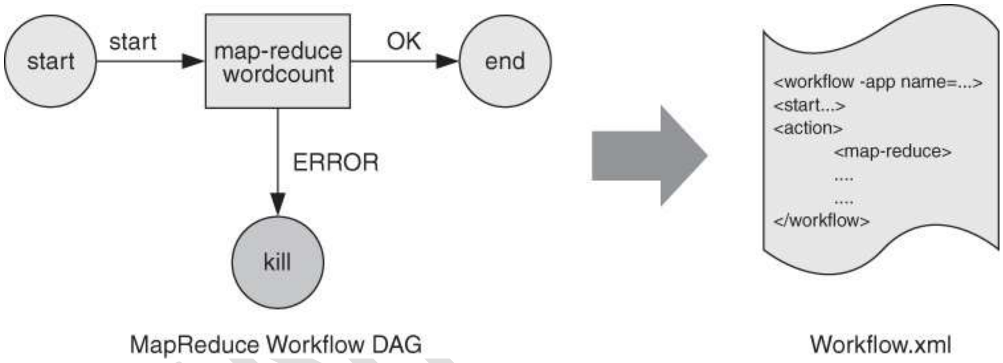


Figure 7.6 A simple Oozie DAG workflow (Adapted from Apache Oozie Documentation)

Oozie workflow definitions are written in hPDL (an XML Process Definition Language). Such workflows contain several types of nodes:

- **Control flow nodes** define the beginning and the end of a workflow. They include start, end, and optional fail nodes.

- **Action nodes** are where the actual processing tasks are defined. When an action node finishes, the remote systems notify Oozie and the next node in the workflow is executed. Action nodes can also include HDFS commands.

- **Fork/join nodes** enable parallel execution of tasks in the workflow. The fork node enables two or more tasks to run at the same time. A join node represents a rendezvous point that must wait until all forked tasks complete.
- **Control flow nodes** enable decisions to be made about the previous task. Control decisions are based on the results of the previous action (e.g., file size or file existence). Decision nodes are essentially switch-case statements that use JSP EL (Java Server Pages—Expression Language) that evaluate to either true or false.

Figure 7.7 depicts a more complex workflow that uses all of these node types.

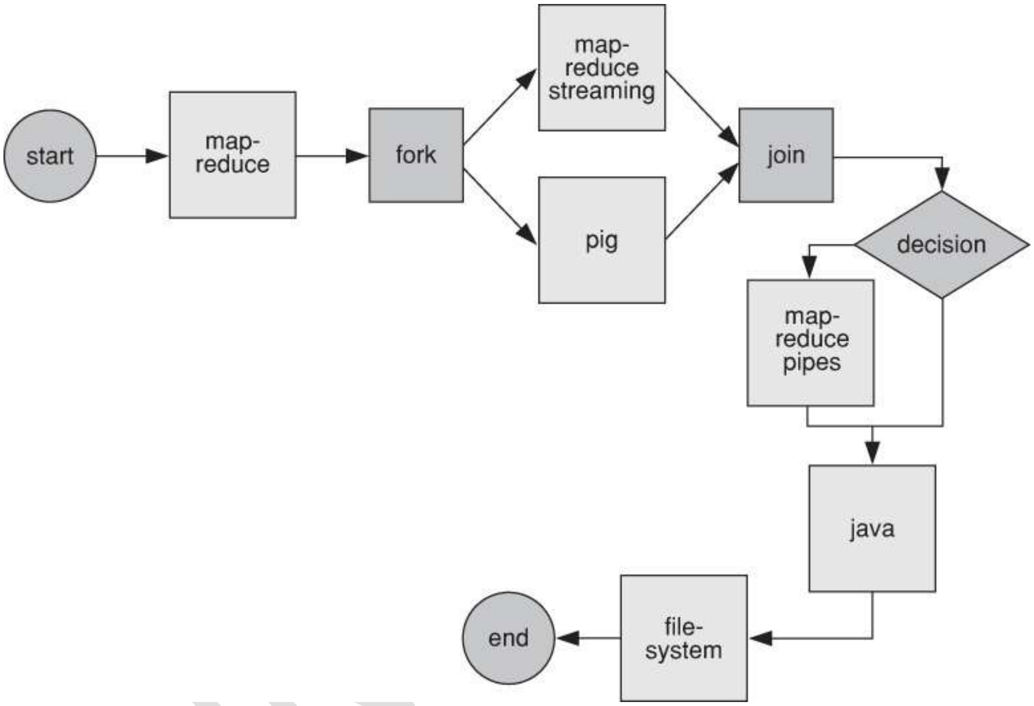


Figure 7.7 A more complex Oozie DAG workflow (Adapted from Apache Oozie Documentation)

Oozie Example Walk-Through

Step 1: Download Oozie Examples

The Oozie examples used in this section can be found on the book website (see Appendix A). They are also available as part of the oozie-client.noarch RPM in the Hortonworks HDP 2.x packages. For HDP 2.1, the following command can be used to extract the files into the working directory used for the demo:

```
$ tar xvzf /usr/share/doc/oozie-4.0.0.2.1.2.1/oozie-examples.tar.gz
```

For HDP 2.2, the following command will extract the files:

```
$ tar xvzf /usr/hdp/2.2.4.2-2/oozie/doc/oozie-examples.tar.gz
```

Once extracted, rename the examples directory to oozie-examples so that you will not confuse it with the other examples directories.

```
$ mv examples oozie-examples
```

The examples must also be placed in HDFS. Enter the following command to move the example files into HDFS:

```
$ hdfs dfs -put oozie-examples/ oozie-examples
```

The Oozie shared library must be installed in HDFS. If you are using the Ambari installation of HDP 2.x, this library is already found in HDFS: /user/oozie/share/lib.

Step 2: Run the Simple MapReduce Example

Move to the simple MapReduce example directory:

```
$ cd oozie-examples/apps/map-reduce/
```

This directory contains two files and a lib directory. The files are:

- The job.properties file defines parameters (e.g., path names, ports) for a job. This file may change per job.
- The workflow.xml file provides the actual workflow for the job. In this case, it is a simple MapReduce (pass/fail). This file usually stays the same between jobs.

The job.properties file included in the examples requires a few edits to work properly. Using a text editor, change the following lines by adding the host name of the NameNode and ResourceManager (indicated by jobTracker in the file).

As shown in Figure 7.6, this simple workflow runs an example MapReduce job and prints an error message if it fails.

To run the Oozie MapReduce example job from the oozie-examples/apps/map-reduce directory, enter the following line:

```
$ oozie job -run -oozie http://limulus:11000/oozie -config job.properties
```

When Oozie accepts the job, a job ID will be printed:

```
job: 0000001-150424174853048-oozie-oozi-W
```

You will need to change the “limulus” host name to match the name of the node running your Oozie server. The job ID can be used to track and control job progress.

To avoid having to provide the -oozie option with the Oozie URL every time you run the ooziecommand, set the OOZIE_URL environment variable as follows (using your Oozie server host name in place of “limulus”):

```
$ export OOZIE_URL="http://limulus:11000/oozie"
```

You can now run all subsequent Oozie commands without specifying the -oozie URL option. For instance, using the job ID, you can learn about a particular job's progress by issuing the following command:

```
$ oozie job -info 0000001-150424174853048-oozie-oozi-W
```

The resulting output (line length compressed) is shown in the following listing. Because this job is just a simple test, it may be complete by the time you issue the -info command. If it is not complete, its progress will be indicated in the listing.

```
Job ID : 0000001-150424174853048-oozie-oozi-W
------------------------------------------------------------------------------------
Workflow Name : map-reduce-wf
App Path      : hdfs://limulus:8020/user/hdfs/examples/apps/map-reduce
Status        : SUCCEEDED
Run           : 0
User          : hdfs
Group         : -
Created       : 2015-04-29 20:52 GMT
Started       : 2015-04-29 20:52 GMT
Last Modified : 2015-04-29 20:53 GMT
Ended         : 2015-04-29 20:53 GMT
CoordAction ID: -

Actions
------------------------------------------------------------------------------------
ID                                  Status   Ext ID    Ext Status  Err Code
------------------------------------------------------------------------------------
0000001-150424174853048-oozie
  -oozi-W@:start:                     OK       -          OK        -
------------------------------------------------------------------------------------
0000001-150424174853048-oozie
  -oozi-W@mr-node                          OK  job_1429912013449_0006 SUCCEEDED -
------------------------------------------------------------------------------------
0000001-150424174853048-oozie
  -oozi-W@end                         OK       -          OK        -
------------------------------------------------------------------------------------
```

The various steps shown in the output can be related directly to the workflow.xml mentioned previously. Note that the MapReduce job number is provided. This job will also be listed in the ResourceManager web user interface. The application output is located in HDFS under the oozie-examples/output-data/map-reduce directory.

Step 3: Run the Oozie Demo Application

A more sophisticated example can be found in the demo directory (oozie-examples/apps/demo). This workflow includes MapReduce, Pig, and file system tasks as well as fork, join, decision, action, start, stop, kill, and end nodes.

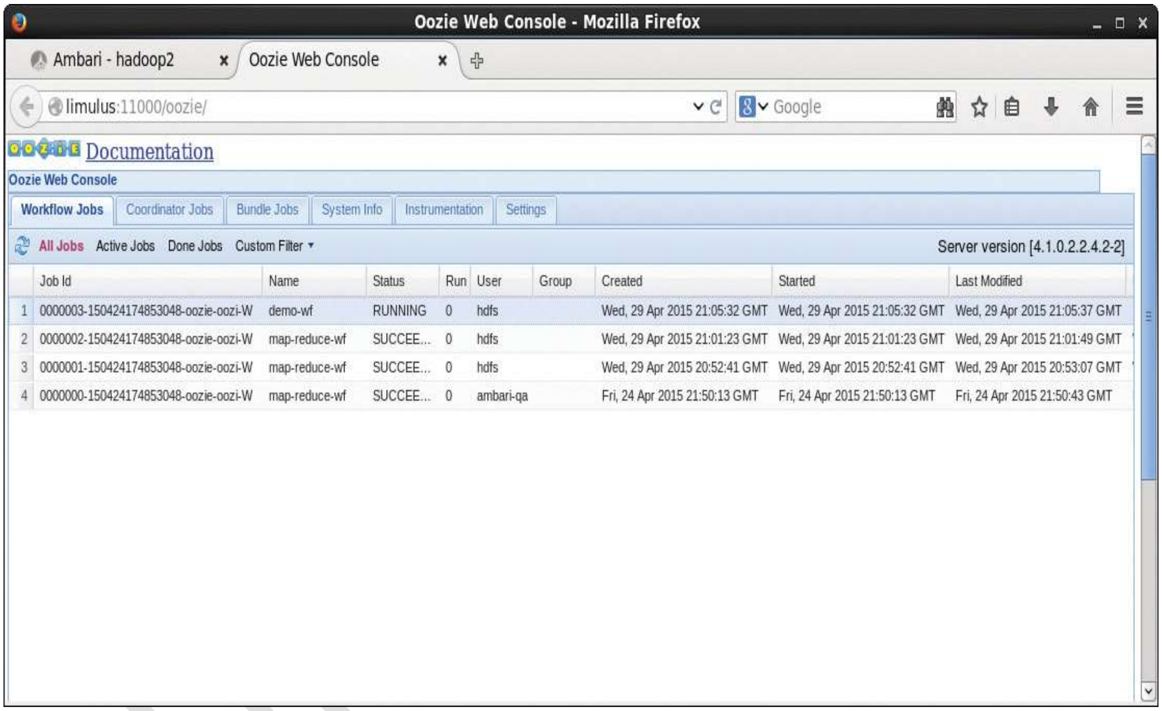
Move to the demo directory and edit the job.properties file as described previously. Entering the following command runs the workflow (assuming the OOZIE_URL environment variable has been set):

```
$ oozie job -run -config job.properties
```

You can track the job using either the Oozie command-line interface or the Oozie web console. To start the web console from within Ambari, click on the Oozie service, and then click on the Quick Links pull-down menu and select Oozie Web UI. Alternatively, you can start the Oozie web UI by connecting to the Oozie server directly. For example, the following command will bring up the Oozie UI (use your Oozie server host name in place of "limulus"):

```
$ firefox  http://limulus:11000/oozie/
```

Figure 7.8 shows the main Oozie console window.

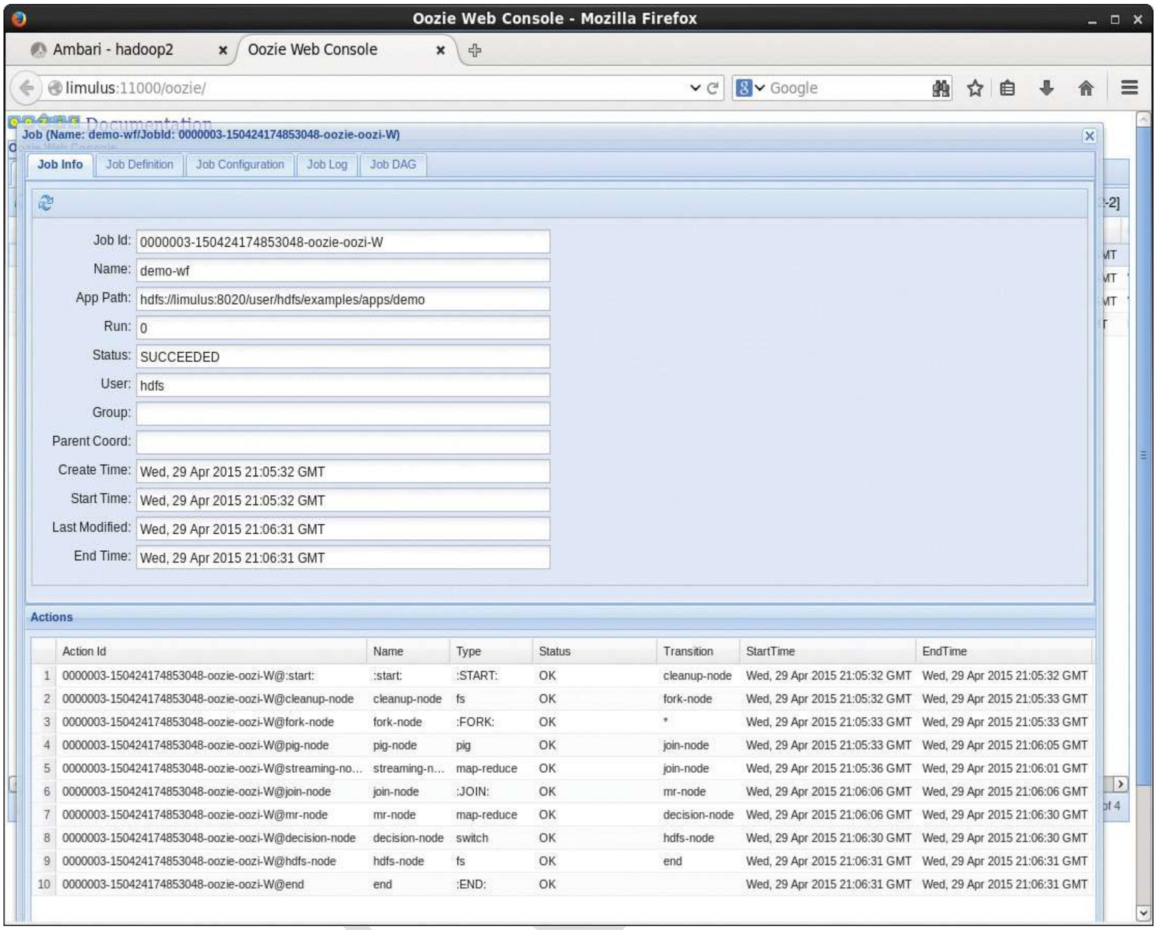


Server version [4.1.0.2.2.4.2-2]

	Job Id	Name	Status	Run	User	Group	Created	Started	Last Modified
1	0000003-150424174853048-oozie-oozi-W	demo-wf	RUNNING	0	hdfs		Wed, 29 Apr 2015 21:05:32 GMT	Wed, 29 Apr 2015 21:05:32 GMT	Wed, 29 Apr 2015 21:05:37 GMT
2	0000002-150424174853048-oozie-oozi-W	map-reduce-wf	SUCCEE...	0	hdfs		Wed, 29 Apr 2015 21:01:23 GMT	Wed, 29 Apr 2015 21:01:23 GMT	Wed, 29 Apr 2015 21:01:49 GMT
3	0000001-150424174853048-oozie-oozi-W	map-reduce-wf	SUCCEE...	0	hdfs		Wed, 29 Apr 2015 20:52:41 GMT	Wed, 29 Apr 2015 20:52:41 GMT	Wed, 29 Apr 2015 20:53:07 GMT
4	0000000-150424174853048-oozie-oozi-W	map-reduce-wf	SUCCEE...	0	ambari-qa		Fri, 24 Apr 2015 21:50:13 GMT	Fri, 24 Apr 2015 21:50:13 GMT	Fri, 24 Apr 2015 21:50:43 GMT

Figure 7.8 Oozie main console window

Workflow jobs are listed in tabular form, with the most recent job appearing first. If you click on a workflow, the Job Info window in Figure 7.9 will be displayed. The job progression results, similar to those printed by the Oozie command line, are shown in the Actions window at the bottom.



Job (Name: demo-wf/JobId: 0000003-150424174853048-oozie-oozi-W)

Job Id: 0000003-150424174853048-oozie-oozi-W
Name: demo-wf
App Path: hdfs://limulus:8020/user/hdfs/examples/apps/demo
Run: 0
Status: SUCCEEDED
User: hdfs
Group:
Parent Coord:
Create Time: Wed, 29 Apr 2015 21:05:32 GMT
Start Time: Wed, 29 Apr 2015 21:05:32 GMT
Last Modified: Wed, 29 Apr 2015 21:06:31 GMT
End Time: Wed, 29 Apr 2015 21:06:31 GMT

Actions

	Action Id	Name	Type	Status	Transition	StartTime	EndTime
1	0000003-150424174853048-oozie-oozi-W@:start:	:start:	:START:	OK	cleanup-node	Wed, 29 Apr 2015 21:05:32 GMT	Wed, 29 Apr 2015 21:05:32 GMT
2	0000003-150424174853048-oozie-oozi-W@cleanup-node	cleanup-node	fs	OK	fork-node	Wed, 29 Apr 2015 21:05:32 GMT	Wed, 29 Apr 2015 21:05:33 GMT
3	0000003-150424174853048-oozie-oozi-W@fork-node	fork-node	:FORK:	OK	*	Wed, 29 Apr 2015 21:05:33 GMT	Wed, 29 Apr 2015 21:05:33 GMT
4	0000003-150424174853048-oozie-oozi-W@pig-node	pig-node	pig	OK	join-node	Wed, 29 Apr 2015 21:05:33 GMT	Wed, 29 Apr 2015 21:06:05 GMT
5	0000003-150424174853048-oozie-oozi-W@streaming-no...	streaming-n...	map-reduce	OK	join-node	Wed, 29 Apr 2015 21:05:36 GMT	Wed, 29 Apr 2015 21:06:01 GMT
6	0000003-150424174853048-oozie-oozi-W@join-node	join-node	:JOIN:	OK	mr-node	Wed, 29 Apr 2015 21:06:06 GMT	Wed, 29 Apr 2015 21:06:06 GMT
7	0000003-150424174853048-oozie-oozi-W@mr-node	mr-node	map-reduce	OK	decision-node	Wed, 29 Apr 2015 21:06:06 GMT	Wed, 29 Apr 2015 21:06:30 GMT
8	0000003-150424174853048-oozie-oozi-W@decision-node	decision-node	switch	OK	hdfs-node	Wed, 29 Apr 2015 21:06:30 GMT	Wed, 29 Apr 2015 21:06:30 GMT
9	0000003-150424174853048-oozie-oozi-W@hdfs-node	hdfs-node	fs	OK	end	Wed, 29 Apr 2015 21:06:31 GMT	Wed, 29 Apr 2015 21:06:31 GMT
10	0000003-150424174853048-oozie-oozi-W@end	end	:END:	OK		Wed, 29 Apr 2015 21:06:31 GMT	Wed, 29 Apr 2015 21:06:31 GMT

Figure 7.9 Oozie workflow information window

Other aspects of the job can be examined by clicking the other tabs in the window. The last tab actually provides a graphical representation of the workflow DAG. If the job is not complete, it will highlight the steps that have been completed thus far. The DAG for the demo example is shown in Figure 7.10. The actual image was split to fit better on the page. As with the previous example, comparing this information to workflow.xml file can provide further insights into how Oozie operates.

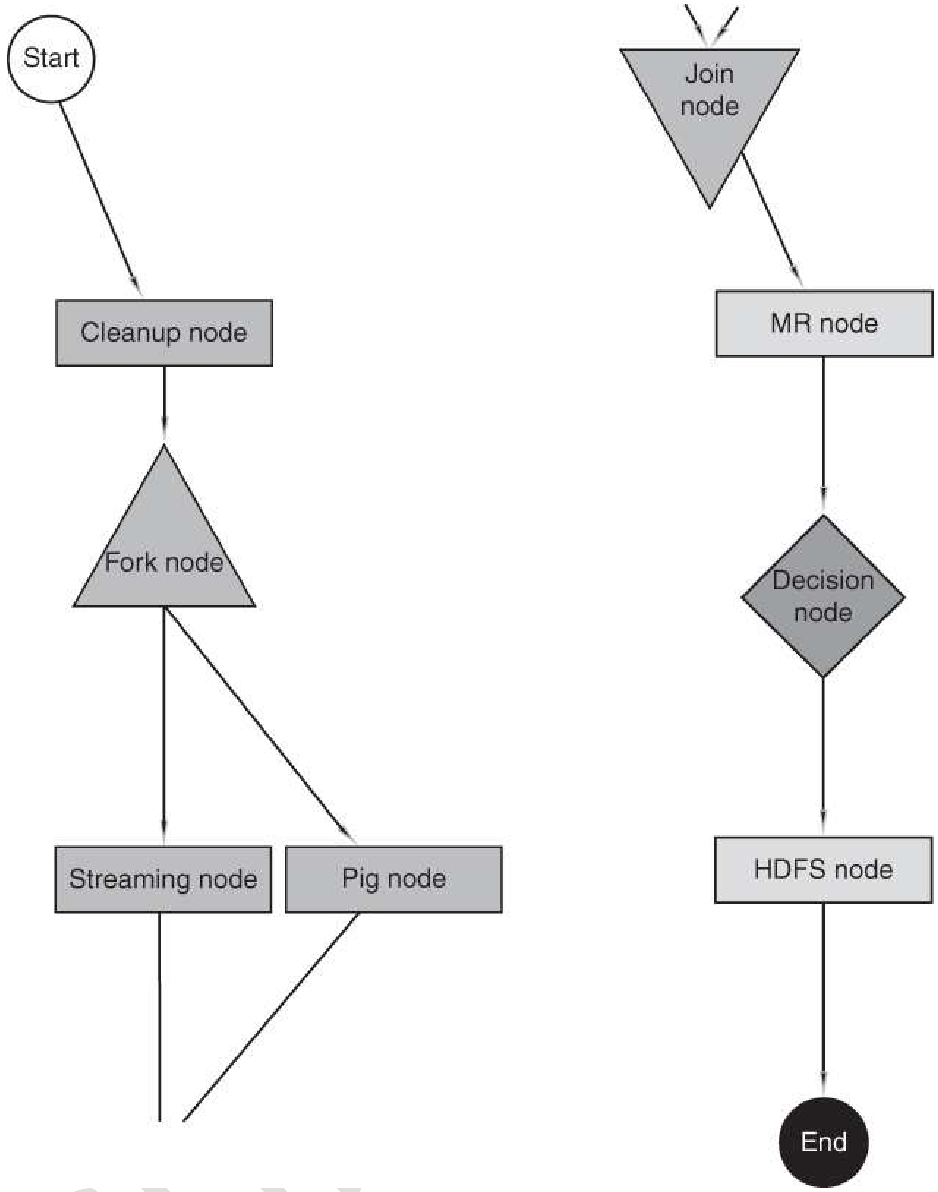


Figure 7.10 Oozie-generated workflow DAG for the demo example, as it appears on the screen

A Short Summary of Oozie Job Commands

The following summary lists some of the more commonly encountered Oozie commands. See the latest documentation at http://oozie.apache.org for more information. (Note that the examples here assume OOZIE_URL is defined.)

- Run a workflow job (returns _OOZIE_JOB_ID_):

```
$ oozie job -run -config JOB_PROPERITES
```

- Submit a workflow job (returns _OOZIE_JOB_ID_ but does not start):

```
$ oozie job -submit -config JOB_PROPERTIES
```

- Start a submitted job:

```
$ oozie job -start _OOZIE_JOB_ID_
```

- Check a job’s status:

```
$ oozie job -info _OOZIE_JOB_ID_
```

- Suspend a workflow:

```
$ oozie job -suspend _OOZIE_JOB_ID_
```

- Resume a workflow:

```
$ oozie job -resume _OOZIE_JOB_ID_
```

- Rerun a workflow:

```
$ oozie job -rerun _OOZIE_JOB_ID_ -config JOB_PROPERTIES
```

- Kill a job:

```
$ oozie job -kill _OOZIE_JOB_ID_
```

- View server logs:

```
$ oozie job  -logs _OOZIE_JOB_ID_
```

Full logs are available at /var/log/oozie on the Oozie server.

USING APACHE HBASE

Apache HBase is an open source, distributed, versioned, nonrelational database modeled after Google’s Bigtable. Like Bigtable, HBase leverages the distributed data storage provided by the underlying distributed file systems spread across commodity servers. Apache HBase provides Bigtable-like capabilities on top of Hadoop and HDFS. Some of the more important features include the following capabilities:

- Linear and modular scalability
- Strictly consistent reads and writes
- Automatic and configurable sharding of tables
- Automatic failover support between RegionServers
- Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables
- Easy-to-use Java API for client access

HBase Data Model Overview

A table in HBase is similar to other databases, having rows and columns. Columns in HBase are grouped into column families, all with the same prefix. For example, consider a table of daily stock prices. There may be a column family called “price” that has four members—price:open, price:close, price:low, and price:high. A column does not need to be a family. For instance, the stock table may have a column named “volume” indicating how many shares were traded. All column family members are stored together in the physical file system.

Specific HBase cell values are identified by a row key, column (column family and column), and version (timestamp). It is possible to have many versions of data within an HBase cell. A version is specified as a timestamp and is created each time data are written to a cell. Almost anything can serve as a row key, from strings to binary representations of longs to serialized data structures. Rows are lexicographically sorted with the lowest order appearing first in a table. The empty byte array denotes both the start and the end of a table’s namespace. All table accesses are via the table row key, which is considered its primary key.

HBase Example Walk-Through

HBase provides a shell for interactive use. To enter the shell, type the following as a user:

```
$ hbase shell
hbase(main):001:0>
```

To exit the shell, type **exit**.

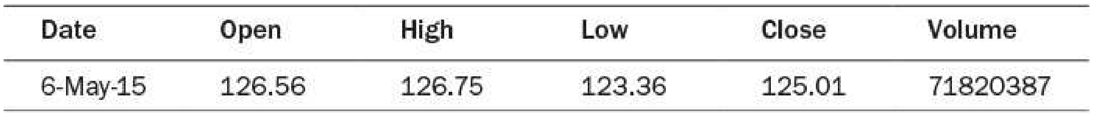
Various commands can be conveniently entered from the shell prompt. For instance, the status command provides the system status:

```
hbase(main):001:0> status
4 servers, 0 dead, 1.0000 average load
```

Additional arguments can be added to the status command, including 'simple', 'summary', or 'detailed'. The single quotes are needed for proper operation. For example, the following command will provide simple status information for the four HBase servers (actual server statistics have been removed for clarity):

```
hbase(main):002:0> status 'simple'
4 live servers
   n1:60020 1429912048329
      ...
   n2:60020 1429912040653
      ...
   limulus:60020 1429912041396
      ...
   n0:60020 1429912042885
      ...
0 dead servers
Aggregate load: 0, regions: 4
```

Other basic commands, such as `version` or `whoami`, can be entered directly at the hbase(main)prompt. In the example that follows, we will use a small set of daily stock price data for Apple computer. The data have the following form:



Date	Open	High	Low	Close	Volume
6-May-15	126.56	126.75	123.36	125.01	71820387

The data can be downloaded from Google using the following command. Note that other stock prices are available by changing the NASDAQ:AAPL argument to any other valid exchange and stock name (e.g., NYSE: IBM).

```
$ wget -O Apple-stock.csv
http://www.google.com/finance/historical?q=NASDAQ:AAPL\&authuser=0\&output=csv
```

The Apple stock price database is in comma-separated format (csv) and will be used to illustrate some basic operations in the HBase shell.

Create the Database

The next step is to create the database in HBase using the following command:

```
hbase(main):006:0> create 'apple', 'price' , 'volume'
0 row(s) in 0.8150 seconds
```

In this case, the table name is apple, and two columns are defined. The date will be used as the row key. The price column is a family of four values (open, close, low, high). The put command is used to add data to the database from within the shell. For instance, the preceding data can be entered by using the following commands:

```
put 'apple','6-May-15','price:open','126.56'
put 'apple','6-May-15','price:high','126.75'
put 'apple','6-May-15','price:low','123.36'
put 'apple','6-May-15','price:close','125.01'
put 'apple','6-May-15','volume','71820387'
```

The shell also keeps a history for the session, and previous commands can be retrieved and edited for resubmission.

Inspect the Database

The entire database can be listed using the scan command. Be careful when using this command with large databases. This example is for one row.

```
hbase(main):006:0> scan 'apple'
ROW            COLUMN+CELL
 6-May-15       column=price:close, timestamp=1430955128359, value=125.01
 6-May-15       column=price:high, timestamp=1430955126024, value=126.75
 6-May-15       column=price:low, timestamp=1430955126053, value=123.36
 6-May-15       column=price:open, timestamp=1430955125977, value=126.56
 6-May-15       column=volume:, timestamp=1430955141440, value=71820387
```

Get a Row

You can use the row key to access an individual row. In the stock price database, the date is the row key.

```
hbase(main):008:0> get 'apple', '6-May-15'
COLUMN                        CELL
 price:close              timestamp=1430955128359, value=125.01
 price:high               timestamp=1430955126024, value=126.75
 price:low                timestamp=1430955126053, value=123.36
 price:open                timestamp=1430955125977, value=126.56
 volume:                  timestamp=1430955141440, value=71820387
5 row(s) in 0.0130 seconds
```

Get Table Cells

A single cell can be accessed using the get command and the COLUMN option:

```
hbase(main):013:0> get 'apple', '5-May-15', {COLUMN => 'price:low'}

COLUMN                        CELL

 price:low                timestamp=1431020767444, value=125.78

1 row(s) in 0.0080 seconds
```

In a similar fashion, multiple columns can be accessed as follows:

```
hbase(main):012:0> get 'apple', '5-May-15', {COLUMN => ['price:low', 'price:high']}

COLUMN                        CELL

 price:high               timestamp=1431020767444, value=128.45

 price:low                timestamp=1431020767444, value=125.78

2 row(s) in 0.0070 seconds
```

Delete a Cell

A specific cell can be deleted using the following command:

```
hbase(main):009:0> delete 'apple', '6-May-15' , 'price:low'
```

If the row is inspected using get, the price:low cell is not listed.

```
hbase(main):010:0> get 'apple', '6-May-15'
COLUMN                        CELL
 price:close              timestamp=1430955128359, value=125.01
 price:high               timestamp=1430955126024, value=126.75
 price:open                timestamp=1430955125977, value=126.46
 volume:                  timestamp=1430955141440, value=71820387
4 row(s) in 0.0130 seconds
```

Delete a Row

You can delete an entire row by giving the deleteall command as follows:

```
hbase(main):009:0> deleteall 'apple', '6-May-15'
```

Remove a Table

To remove (drop) a table, you must first disable it. The following two commands remove the appletable from Hbase:

```
hbase(main):009:0> disable 'apple'
hbase(main):010:0> drop 'apple'
```

Scripting Input

Commands to the HBase shell can be placed in bash scripts for automated processing. For instance, the following can be placed in a bash script:

```
echo "put 'apple','6-May-15','price:open','126.56'" | hbase shell
```

The book software page includes a script (input_to_hbase.sh) that imports the Apple-stock.csv file into HBase using this method. It also removes the column titles in the first line. The script will load the entire file into HBase when you issue the following command:

```
$ input_to_hbase.sh Apple-stock.csv
```

While the script can be easily modified to accommodate other types of data, it is not recommended for production use because the upload is very inefficient and slow. Instead, this script is best used to experiment with small data files and different types of data.

Adding Data in Bulk

There are several ways to efficiently load bulk data into HBase. Covering all of these methods is beyond the scope of this chapter. Instead, we will focus on the ImportTsv utility, which loads data in tab-separated values (tsv) format into HBase. It has two distinct usage modes:

- Loading data from a tsv-format file in HDFS into HBase via the put command
- Preparing StoreFiles to be loaded via the completebulkload utility

The following example shows how to use ImportTsv for the first option, loading the tsv-format file using the put command.

The first step is to convert the Apple-stock.csv file to tsv format. The following script, which is included in the book software, will remove the first line and do the conversion. In doing so, it creates a file named Apple-stock.tsv.

```
$ convert-to-tsv.sh Apple-stock.csv
```

Next, the new file is copied to HDFS as follows:

```
$ hdfs dfs -put Apple-stock.tsv /tmp
```

Finally, ImportTsv is run using the following command line. Note the column designation in the -Dimporttsv.columns option. In the example, the HBASE_ROW_KEY is set as the first column—that is, the date for the data.

```
$ hbase org.apache.hadoop.hbase.mapreduce.ImportTsv -
Dimporttsv.columns=HBASE_ROW_KEY,price:open,price:high,price:low,price:close,volume apple
/tmp/Apple-stock.tsv
```

The ImportTsv command will use MapReduce to load the data into HBase. To verify that the command works, drop and re-create the apple database, as described previously, before running the import command.

8. Hadoop YARN Applications

In This Chapter:

- The YARN Distributed-Shell is introduced as a non-MapReduce application.
- The Hadoop YARN application and operation structure is explained.
- A summary of YARN application frameworks is provided.

YARN DISTRIBUTED-SHELL

The Hadoop YARN project includes the Distributed-Shell application, which is an example of a Hadoop non-MapReduce application built on top of YARN. Distributed-Shell is a simple mechanism for running shell commands and scripts in containers on multiple nodes in a Hadoop cluster. This application is not meant to be a production administration tool, but rather a demonstration of the non-MapReduce capability that can be implemented on top of YARN. There are multiple mature implementations of a distributed shell that administrators typically use to manage a cluster of machines.

In addition, Distributed-Shell can be used as a starting point for exploring and building Hadoop YARN applications. This chapter offers guidance on how the Distributed-Shell can be used to understand the operation of YARN applications.

USING THE YARN DISTRIBUTED-SHELL

For the purpose of the examples presented in the remainder of this chapter, we assume and assign the following installation path, based on Hortonworks HDP 2.2, the Distributed-Shell application:

```
$ export YARN_DS=/usr/hdp/current/hadoop-yarn-client/hadoop-yarn-applications-
distributedshell.jar
```

For the pseudo-distributed install using Apache Hadoop version 2.6.0, the following path will run the Distributed-Shell application (assuming $HADOOP_HOME is defined to reflect the location Hadoop):

```
$ export YARN_DS=$HADOOP_HOME/share/hadoop/yarn/hadoop-yarn-applications-
distributedshell-2.6.0.jar
```

If another distribution is used, search for the file hadoop-yarn-applications-distributedshell*.jar and set $YARN_DS based on its location. Distributed-Shell exposes various options that can be found by running the following command:

```
$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client -jar $YARN_DS -help
```

The output of this command follows; we will explore some of these options in the examples illustrated in this chapter.

```
usage: Client
 -appname <arg>              Application Name. Default value – DistributedShell
 -container_memory <arg>     Amount of memory in MB to be requested to run the shell command
 -container_vcores <arg>     Amount of virtual cores to be requested to run the shell command
 -create                     Flag to indicate whether to create the domain specified with -domain.
 -debug                      Dump out debug information
 -domain <arg>               ID of the timeline domain where the timeline entities will be put
 -help                       Print usage
 -jar <arg>                  Jar file containing the application master
 -log_properties <arg>       log4j.properties file
 -master_memory <arg>        Amount of memory in MB to be requested to run the application master
 -master_vcores <arg>        Amount of virtual cores to be requested to run the application master
 -modify_acls <arg>          Users and groups that allowed to modify the timeline entities in the given domain
 -timeout <arg>              Application timeout in milliseconds
 -view_acls <arg>            Users and groups that allowed to view the timeline entities in the  given domain
```

A Simple Example

The simplest use-case for the Distributed-Shell application is to run an arbitrary shell command in a container. We will demonstrate the use of the uptime command as an example. This command is run on the cluster using Distributed-Shell as follows:

```
$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client -jar $YARN_DS -shell_command
uptime
```

By default, Distributed-Shell spawns only one instance of a given shell command. When this command is run, you can see progress messages on the screen but nothing about the actual shell command. If the shell command succeeds, the following should appear at the end of the output:

```
15/05/27 14:48:53 INFO distributedshell.Client: Application completed successfully
```

If the shell command did not work for whatever reason, the following message will be displayed:

```
15/05/27 14:58:42 ERROR distributedshell.Client: Application failed to complete
successfully
```

The next step is to examine the output for the application. Distributed-Shell redirects the output of the individual shell commands run on the cluster nodes into the log files, which are found either on the individual nodes or aggregated onto HDFS, depending on whether log aggregation is enabled.

Assuming log aggregation is enabled, the results for each instance of the command can be found by using the yarn logs command. For the previous uptime example, the following command can be used to inspect the logs:

```
$ yarn logs -applicationId application_1432831236474_0001
```

The abbreviated output follows:

```
Container: container_1432831236474_0001_01_000001 on n0_45454
=================================================================
LogType:AppMaster.stderr
Log Upload Time:Thu May 28 12:41:58 -0400 2015
LogLength:3595
Log Contents:
15/05/28 12:41:52 INFO distributedshell.ApplicationMaster: Initializing
ApplicationMaster
[...]
Container: container_1432831236474_0001_01_000002 on n1_45454
=================================================================
LogType:stderr
Log Upload Time:Thu May 28 12:41:59 -0400 2015
LogLength:0
Log Contents:

LogType:stdout
Log Upload Time:Thu May 28 12:41:59 -0400 2015
LogLength:71
Log Contents:
 12:41:56 up 33 days, 19:28,  0 users,  load average: 0.08, 0.06, 0.01
```

Notice that there are two containers. The first container (`con..._000001`) is the ApplicationMaster for the job. The second container (con..._000002) is the actual shell script. The output for the uptime command is located in the second containers stdout after the Log Contents: label.

Using More Containers

Distributed-Shell can run commands to be executed on any number of containers by way of the -num_containers argument. For example, to see on which nodes the Distributed-Shell command was run, the following command can be used:

```
$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client -jar $YARN_DS -shell_command
hostname -num_containers 4
```

If we now examine the results for this job, there will be five containers in the log. The four command containers (2 through 5) will print the name of the node on which the container was run.

Distributed-Shell Examples with Shell Arguments

Arguments can be added to the shell command using the -shell_args option. For example, to do a ls -l in the directory from where the shell command was run, we can use the following commands:

```
$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client -jar $YARN_DS  -shell_command ls -
shell_args -l
```

The resulting output from the log file is as follows:

```
total 20
-rw-r--r-- 1 yarn hadoop   74 May 28 10:37 container_tokens
-rwx------ 1 yarn hadoop  643 May 28 10:37 default_container_executor_session.sh
-rwx------ 1 yarn hadoop  697 May 28 10:37 default_container_executor.sh
-rwx------ 1 yarn hadoop 1700 May 28 10:37 launch_container.sh
drwx--x--- 2 yarn hadoop 4096 May 28 10:37 tmp
```

As can be seen, the resulting files are new and not located anywhere in HDFS or the local file system. When we explore further by giving a pwd command for Distributed-Shell, the following directory is listed and created on the node that ran the shell command:

```
/hdfs2/hadoop/yarn/local/usercache/hdfs/appcache/application_1432831236474_0003/container_14328312
36474_0003_01_000002/
```

Searching for this directory will prove to be problematic because these transient files are used by YARN to run the Distributed-Shell application and are removed once the application finishes. You can preserve these files for a specific interval by adding the following lines to the yarn-site.xmlconfiguration file and restarting YARN:

```
<property>
   <name>yarn.nodemanager.delete.debug-delay-sec</name>
   <value>100000</value>
</property>
```

Choose a delay, in seconds, to preserve these files, and remember that all applications will create these files. If you are using Ambari, look on the YARN Configs tab under the Advanced yarn-site options, make the change and restart YARN. (See Chapter 9, "Managing Hadoop with Apache Ambari," for more information on Ambari administration.) These files will be retained on the individual nodes only for the duration of the specified delay.

When debugging or investigating YARN applications, these files—in particular, launch_container.sh—offer important information about YARN processes. Distributed-Shell can be used to see what this file contains. Using DistributedShell, the contents of the launch_container.sh file can be printed with the following command:

```
$ yarn org.apache.hadoop.yarn.applications.distributedshell.Client -jar $YARN_DS -shell_command cat
-shell_args launch_container.sh
```

This command prints the launch_container.sh file that is created and run by YARN. The contents of this file are shown in Listing 8.1. The file basically exports some important YARN variables and then, at the end, "execs" the command (cat launch_container.sh) directly and sends any output to logs.

Listing 8.1 **Distributed-Shell launch_container.sh File**

```
#!/bin/bash

export NM_HTTP_PORT="8042"
export LOCAL_DIRS="/opt/hadoop/yarn/local/usercache/hdfs/appcache/
application_1432816241597_0004,/hdfs1/hadoop/yarn/local/usercache/hdfs/appc
ache/
application_1432816241597_0004,/hdfs2/hadoop/yarn/local/usercache/hdfs/appc
ache/
application_1432816241597_0004"
export JAVA_HOME="/usr/lib/jvm/java-1.7.0-openjdk.x86_64"
export
NM_AUX_SERVICE_mapreduce_shuffle="AAA0+gAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA=
"
export HADOOP_YARN_HOME="/usr/hdp/current/hadoop-yarn-client"
export HADOOP_TOKEN_FILE_LOCATION="/hdfs2/hadoop/yarn/local/usercache/hdfs/
appcache/application_1432816241597_0004/container_1432816241597_0004_01_000
002/
container_tokens"
export NM_HOST="limulus"
export JVM_PID="$$"
export USER="hdfs"
export PWD="/hdfs2/hadoop/yarn/local/usercache/hdfs/appcache/
application_1432816241597_0004/container_1432816241597_0004_01_000002"
export CONTAINER_ID="container_1432816241597_0004_01_000002"
export NM_PORT="45454"
export HOME="/home/"
export LOGNAME="hdfs"
export HADOOP_CONF_DIR="/etc/hadoop/conf"
export MALLOC_ARENA_MAX="4"
export LOG_DIRS="/opt/hadoop/yarn/log/application_1432816241597_0004/
container_1432816241597_0004_01_000002,/hdfs1/hadoop/yarn/log/
application_1432816241597_0004/container_1432816241597_0004_01_000002,/hdfs
2/
hadoop/yarn/log/application_1432816241597_0004/
container_1432816241597_0004_01_000002"
exec /bin/bash -c "cat launch_container.sh
1>/hdfs2/hadoop/yarn/log/application_1432816241597_0004/
container_1432816241597_0004_01_000002/stdout 2>/hdfs2/hadoop/yarn/log/
application_1432816241597_0004/container_1432816241597_0004_01_000002/stder
r "
hadoop_shell_errorcode=$?
if [ $hadoop_shell_errorcode -ne 0 ]
then
  exit $hadoop_shell_errorcode
fi
```

There are more options for the Distributed-Shell that you can test. The real value of the Distributed-Shell application is its ability to demonstrate how applications are launched within the Hadoop YARN infrastructure. It is also a good starting point when you are creating YARN applications.

STRUCTURE OF YARN APPLICATIONS

The structure and operation of a YARN application are covered briefly in this section.

The central YARN ResourceManager runs as a scheduling daemon on a dedicated machine and acts as the central authority for allocating resources to the various competing applications in the cluster. The ResourceManager has a central and global view of all cluster resources and, therefore, can ensure fairness, capacity, and locality are shared across all users. Depending on the application demand, scheduling priorities, and resource availability, the ResourceManager dynamically allocates resource containers to applications to run on particular nodes. A container is a logical bundle of resources (e.g., memory, cores) bound to a particular cluster node. To enforce and track such assignments, the ResourceManager interacts with a special system daemon running on each node called the NodeManager. Communications between the ResourceManager and NodeManagers are heartbeat based for scalability. NodeManagers are responsible for local monitoring of resource availability, fault reporting, and container life-cycle management (e.g., starting and killing jobs). The ResourceManager depends on the NodeManagers for its “global view” of the cluster.

User applications are submitted to the ResourceManager via a public protocol and go through an admission control phase during which security credentials are validated and various operational and administrative checks are performed. Those applications that are accepted pass to the scheduler and are allowed to run. Once the scheduler has enough resources to satisfy the request, the application is moved from an accepted state to a running state. Aside from internal bookkeeping, this process involves allocating a container for the single ApplicationMaster and spawning it on a node in the cluster. Often called container 0, the ApplicationMaster does not have any additional resources at this point, but rather must request additional resources from the ResourceManager.

The ApplicationMaster is the "master" user job that manages all application life-cycle aspects, including dynamically increasing and decreasing resource consumption (i.e., containers), managing the flow of execution (e.g., in case of MapReduce jobs, running reducers against the output of maps), handling faults and computation skew, and performing other local optimizations. The ApplicationMaster is designed to run arbitrary user code that can be written in any programming language, as all communication with the ResourceManager and NodeManager is encoded using extensible network protocols

YARN makes few assumptions about the ApplicationMaster, although in practice it expects most jobs will use a higher-level programming framework. By delegating all these functions to ApplicationMasters, YARN's architecture gains a great deal of scalability, programming model flexibility, and improved user agility. For example, upgrading and testing a new MapReduce framework can be done independently of other running MapReduce frameworks.

Typically, an ApplicationMaster will need to harness the processing power of multiple servers to complete a job. To achieve this, the ApplicationMaster issues resource requests to the ResourceManager. The form of these requests includes specification of locality preferences (e.g., to accommodate HDFS use) and properties of the containers. The ResourceManager will attempt to satisfy the resource requests coming from each application according to availability and scheduling policies. When a resource is scheduled on behalf of an ApplicationMaster, the ResourceManager generates a lease for the resource, which is acquired by a subsequent ApplicationMaster heartbeat. The ApplicationMaster then works with the NodeManagers to start the resource. A token-based security mechanism guarantees its authenticity when the ApplicationMaster presents the container lease to the NodeManager. In a typical situation, running containers will communicate with the ApplicationMaster through an application-specific protocol to report status and health information and to receive framework-specific commands. In this way, YARN provides a basic infrastructure for monitoring and life-cycle management of containers, while each framework manages application-specific semantics independently. This design stands in sharp contrast to the original Hadoop version 1 design, in which scheduling was designed and integrated around managing only MapReduce tasks.

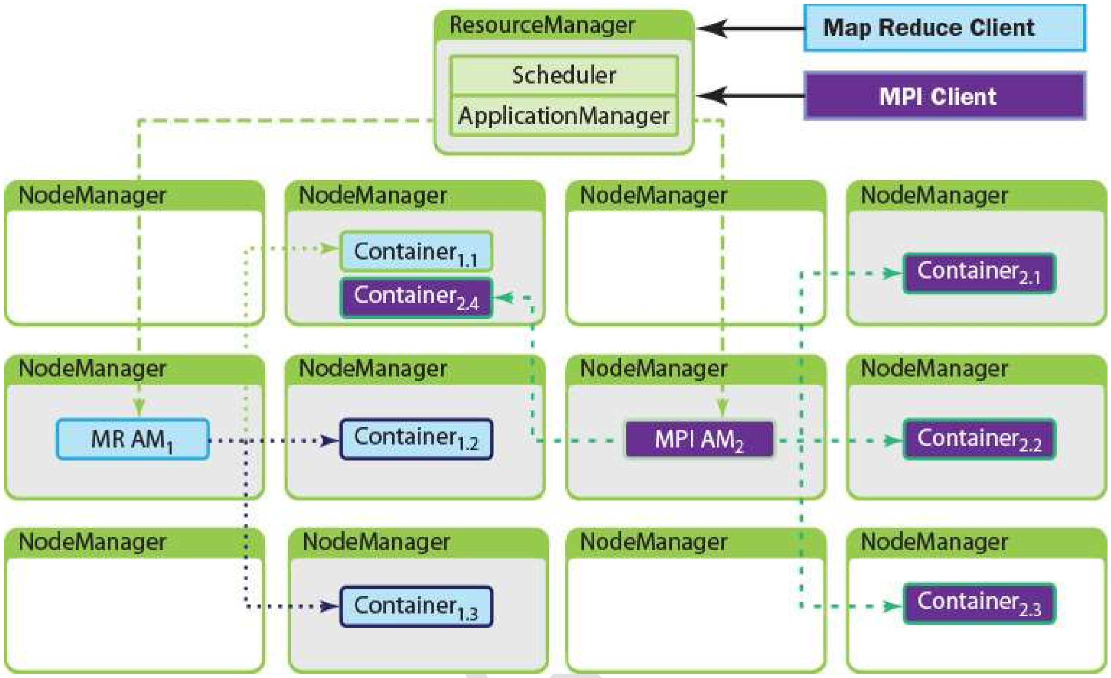


Figure 8.1 YARN architecture with two clients (MapReduce and MPI). The darker client (MPI AM$_2$) is running an MPI application, and the lighter client (MR AM$_1$) is running a MapReduce application. (From Arun C. Murthy, et al., *Apache Hadoop™ YARN*, copyright © 2014, p. 45. Reprinted and electronically reproduced by permission of Pearson Education, Inc., New York, NY.)

YARN APPLICATION FRAMEWORKS

One of the most exciting aspects of Hadoop version 2 is the capability to run all types of applications on a Hadoop cluster. In Hadoop version 1, the only processing model available to users is MapReduce. In Hadoop version 2, MapReduce is separated from the resource management layer of Hadoop and placed into its own application framework. Indeed, the growing number of YARN applications offers a high level and multifaceted interface to the Hadoop data lake.

YARN presents a resource management platform, which provides services such as scheduling, fault monitoring, data locality, and more to MapReduce and other frameworks. Figure 8.2 illustrates some of the various frameworks that will run under YARN. Note that the Hadoop version 1 applications (e.g., Pig and Hive) run under the MapReduce framework.

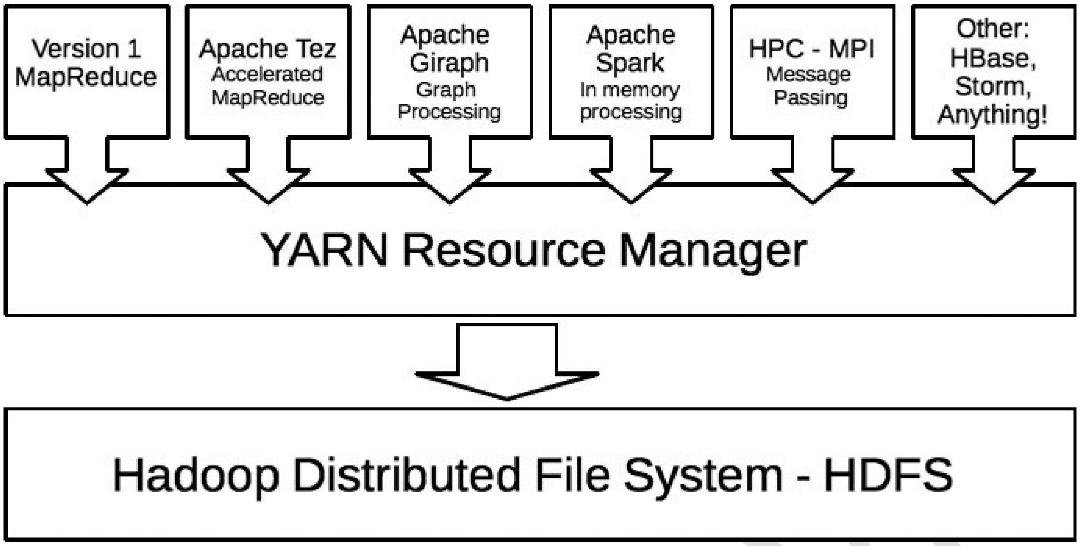


Figure 8.2 Example of the Hadoop version 2 ecosystem. Hadoop version 1 supports batch MapReduce applications only.

This section presents a brief survey of emerging open source YARN application frameworks that are being developed to run under YARN. As of this writing, many YARN frameworks are under active development and the framework landscape is expected to change rapidly. Commercial vendors are also taking advantage of the YARN platform. Consult the webpage for each individual framework for full details of its current stage of development and deployment.

Distributed-Shell

As described earlier in this chapter, Distributed-Shell is an example application included with the Hadoop core components that demonstrates how to write applications on top of YARN. It provides a simple method for running shell commands and scripts in containers in parallel on a Hadoop YARN cluster.

Hadoop MapReduce

MapReduce was the first YARN framework and drove many of YARN's requirements. It is integrated tightly with the rest of the Hadoop ecosystem projects, such as Apache Pig, Apache Hive, and Apache Oozie.

Apache Tez

One great example of a new YARN framework is Apache Tez. Many Hadoop jobs involve the execution of a complex directed acyclic graph (DAG) of tasks using separate MapReduce stages. Apache Tez generalizes this process and enables these tasks to be spread across stages so that they can be run as a single, all-encompassing job.

Tez can be used as a MapReduce replacement for projects such as Apache Hive and Apache Pig. No changes are needed to the Hive or Pig applications.

Apache Giraph

Apache Giraph is an iterative graph processing system built for high scalability. Facebook, Twitter, and LinkedIn use it to create social graphs of users. Giraph was originally written to run on standard Hadoop V1 using the MapReduce framework, but that approach proved inefficient and totally unnatural for various reasons. The native Giraph implementation under YARN provides the user with an iterative processing model that is not directly available with MapReduce. Support for YARN has been present in Giraph since its own version 1.0 release. In addition, using the flexibility of YARN, the Giraph developers plan on implementing their own web interface to monitor job progress

Hoya: HBase on YARN

The Hoya project creates dynamic and elastic Apache HBase clusters on top of YARN. A client application creates the persistent configuration files, sets up the HBase cluster XML files, and then asks YARN to create an ApplicationMaster. YARN copies all files listed in the client's application-launch request from HDFS into the local file system of the chosen server, and then executes the command to start the Hoya ApplicationMaster. Hoya also asks YARN for the number of containers matching the number of HBase region servers it needs.

Dryad on YARN

Similar to Apache Tez, Microsoft's Dryad provides a DAG as the abstraction of execution flow. This framework is ported to run natively on YARN and is fully compatible with its non-YARN version. The code is written completely in native C++ and C# for worker nodes and uses a thin layer of Java within the application.

Apache Spark

Spark was initially developed for applications in which keeping data in memory improves performance, such as iterative algorithms, which are common in machine learning, and interactive data mining. Spark differs from classic MapReduce in two important ways. First, Spark holds intermediate results in memory, rather than writing them to disk. Second, Spark supports more than just MapReduce functions; that is, it greatly expands the set of possible analyses that can be executed over HDFS data stores. It also provides APIs in Scala, Java, and Python.

Since 2013, Spark has been running on production YARN clusters at Yahoo!. The advantage of porting and running Spark on top of YARN is the common resource management and a single underlying file system.

Apache Storm

Traditional MapReduce jobs are expected to eventually finish, but Apache Storm continuously processes messages until it is stopped. This framework is designed to process unbounded streams of data in real time. It can be used in any programming language. The basic Storm use-cases include real-time analytics, online machine learning, continuous computation, distributed RPC (remote procedure calls), ETL (extract, transform, and load), and more. Storm provides fast performance, is scalable, is fault tolerant, and provides processing guarantees. It works directly under YARN and takes advantage of the common data and resource management substrate.

Apache REEF: Retainable Evaluator Execution Framework

YARN's flexibility sometimes requires significant effort on the part of application implementers. The steps involved in writing a custom application on YARN include building your own ApplicationMaster, performing client and container management, and handling aspects of fault tolerance, execution flow, coordination, and other concerns. The REEF project by Microsoft recognizes this challenge and factors out several components that are common to many applications, such as storage management, data caching, fault detection, and checkpoints. Framework designers can build their applications on top of REEF more easily than they can build those same applications directly on YARN, and can reuse these common services/libraries. REEF's design makes it suitable for both MapReduce and DAG-like executions as well as iterative and interactive computations.

Hamster: Hadoop and MPI on the Same Cluster

The Message Passing Interface (MPI) is widely used in high-performance computing (HPC). MPI is primarily a set of optimized message-passing library calls for C, C++, and Fortran that operate over popular server interconnects such as Ethernet and InfiniBand. Because users have full control over their YARN containers, there is no reason why MPI applications cannot run within a Hadoop cluster. The Hamster effort is a work-in-progress that provides a good discussion of the issues involved in mapping MPI to a YARN cluster.

Apache Flink: Scalable Batch and Stream Data Processing

Apache Flink is a platform for efficient, distributed, general-purpose data processing. It features powerful programming abstractions in Java and Scala, a high-performance run time, and automatic program optimization. It also offers native support for iterations, incremental iterations, and programs consisting of large DAGs of operations.

Flink is primarily a stream-processing framework that can look like a batch-processing environment. The immediate benefit from this approach is the ability to use the same algorithms for both streaming and batch modes (exactly as is done in Apache Spark). However, Flink can provide low-latency similar to that found in Apache Storm, but which is not available in Apache Spark.

In addition, Flink has its own memory management system, separate from Java's garbage collector. By managing memory explicitly, Flink almost eliminates the memory spikes often seen on Spark clusters.

Apache Slider: Dynamic Application Management

Apache Slider (incubating) is a YARN application to deploy existing distributed applications on YARN, monitor them, and make them larger or smaller as desired in real time.

Applications can be stopped and then started; the distribution of the deployed application across the YARN cluster is persistent and allows for best-effort placement close to the previous locations. Applications that remember the previous placement of data (such as HBase) can exhibit fast startup times by capitalizing on this feature.

YARN monitors the health of "YARN containers" that are hosting parts of the deployed applications. If a container fails, the Slider manager is notified. Slider then requests a new replacement container from the YARN ResourceManager. Some of Slider's other features include user creation of on-demand applications, the ability to stop and restart applications as needed (preemption), and the ability to expand or reduce the number of application containers as needed. The Slider tool is a Java command-line application.