UNIT – III Notes

Big Data Analytics

Introduction Hadoop: Big Data – Apache Hadoop & Hadoop Eco System – Moving Data in and out of Hadoop – Understanding inputs and outputs of MapReduce - Data Serialization.

Apache Hadoop and Hadoop Echo system

What is Hadoop?

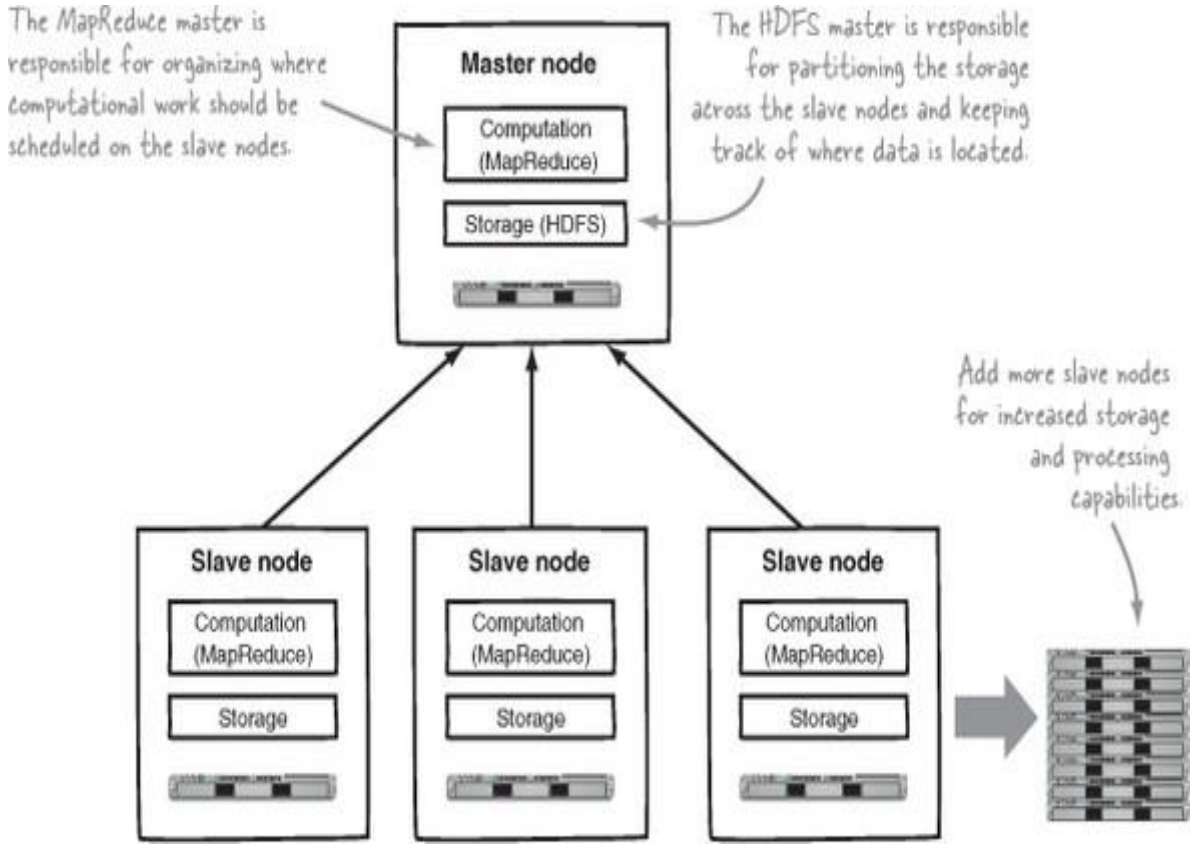
Hadoop is a platform that provides both distributed storage and computational capabilities. Hadoop was first conceived to fix a scalability issue that existed in Nutch, an open source crawler and search engine.

At the time Google had published papers that described its novel distributed filesystem, the Google File System (GFS), and Map-Reduce, a computational framework for parallel processing.

The successful implementation of these papers' concepts in Nutch resulted in its split into two separate projects, the second of which became Hadoop, a first-class Apache project.

The Nutch project, and by extension Hadoop, was led by Doug Cutting and Mike Cafarella.

Figure High-level Hadoop architecture



Core Hadoop components

To understand Hadoop's architecture we'll start by looking at the basics of HDFS.

HDFS

HDFS is the storage component of Hadoop. It's a distributed filesystem that's modeled after the Google File System (GFS) paper

HDFS replicates files for a configured number of times, is tolerant of both software and hardware failure, and automatically re-replicates data blocks on nodes that have failed.

shows a logical representation of the components in HDFS: the Name-Node and the DataNode.

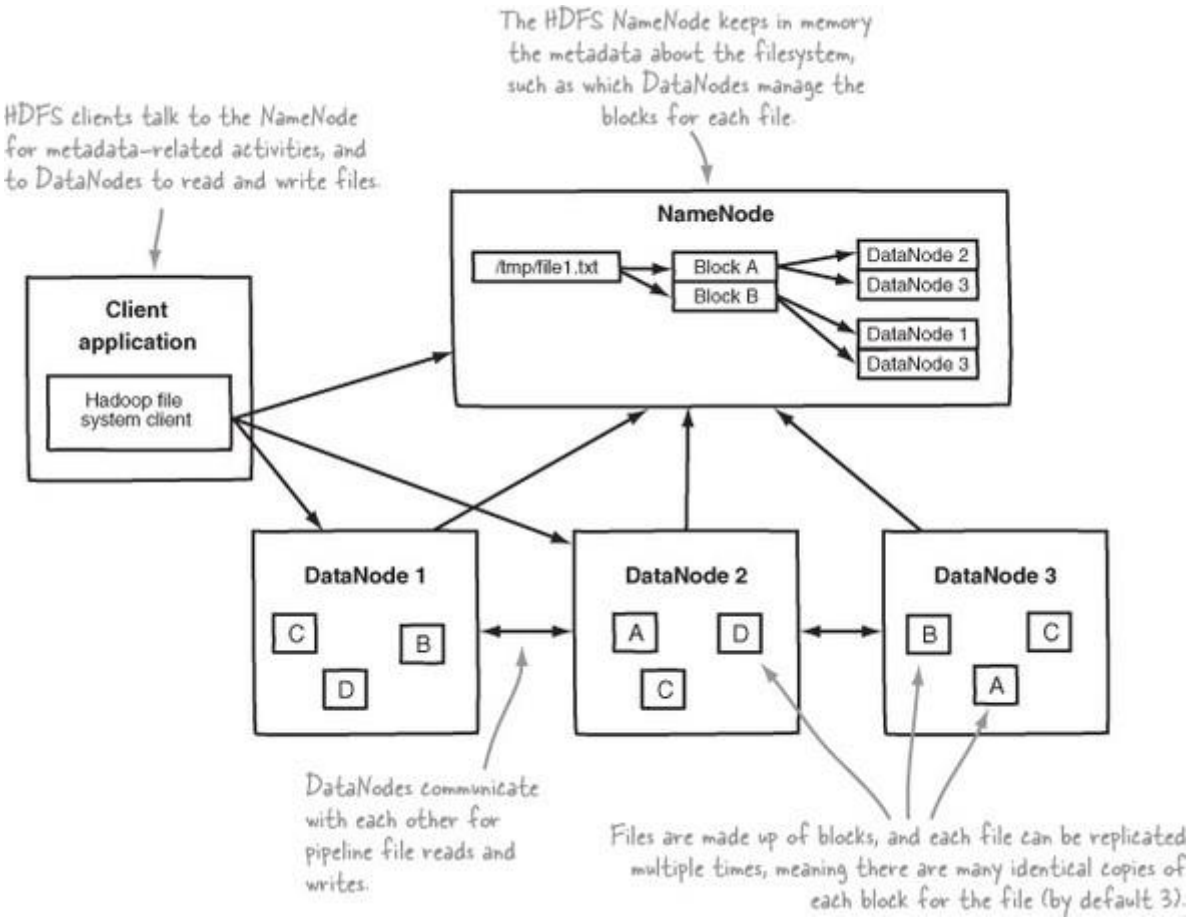
It also shows an application that's using the Hadoop filesystem library to access HDFS.

Mapreduce

MapReduce is a batch-based, distributed computing framework modeled after Google's paper on MapReduce.

It allows you to parallelize work over a large amount of raw data, such as combining web logs with relational data from an OLTP database to model how users interact with your website.

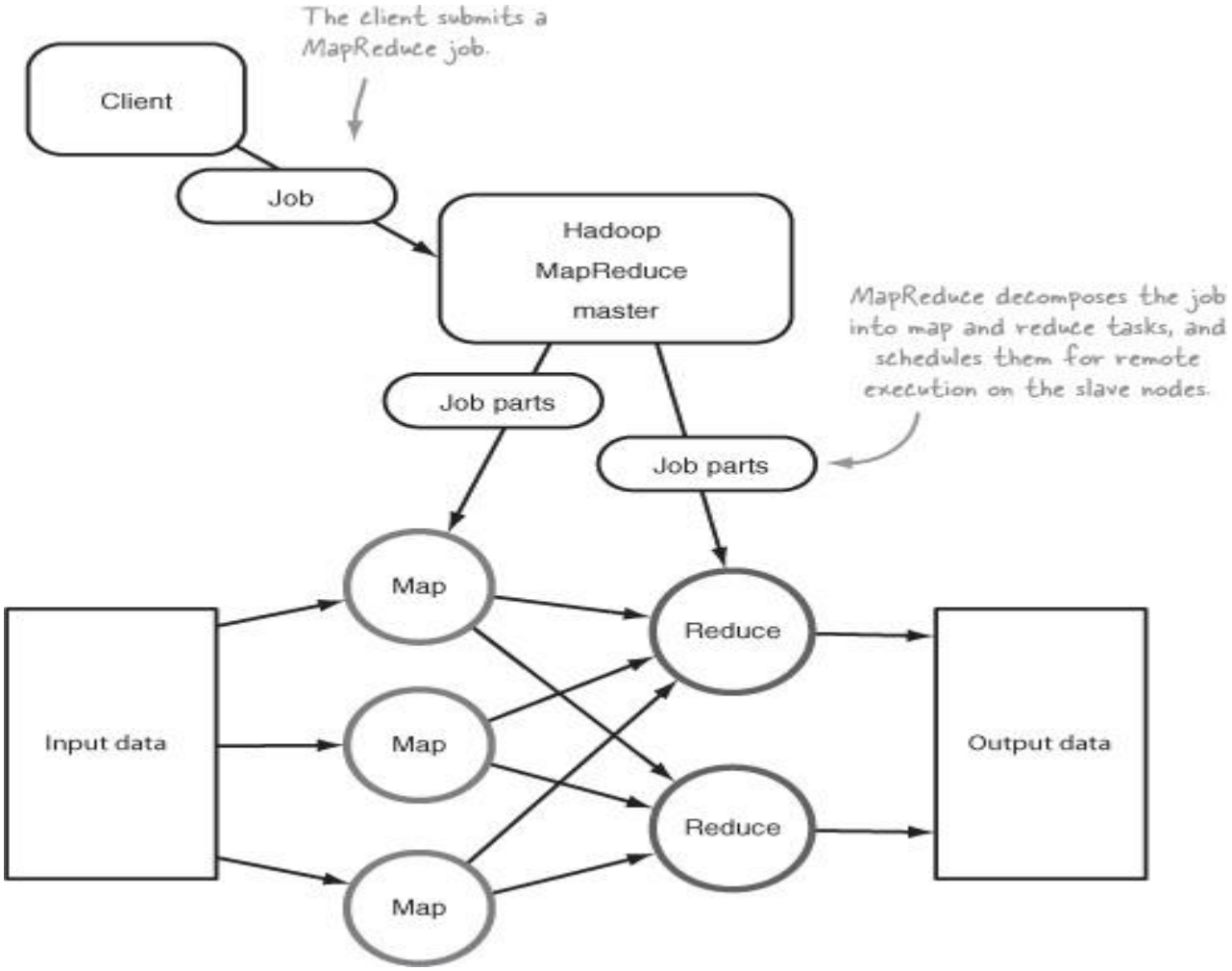
This type of work, which could take days or longer using conventional serial programming techniques, can be reduced down to minutes using MapReduce on a Hadoop cluster.



A client application submits a job to master node i.e name node.The text file is deployed on the 2 bolcks Block A and Block B.These blocks of data are placed on Data Nodes.On these data nodes the 3 copies of data are maintained.if there is hardware failure or data corruption can retrieve the data from the other nodes.

Figure 1.3. HDFS architecture shows an HDFS client communicating with the master NameNode and slave DataNodes.

A client submitting a job to MapReduce



The role of the programmer is to define map and reduce functions, where the map function outputs key/value tuples, which are processed by reduce functions to produce the final output.

MapReduce's shuffle and sort

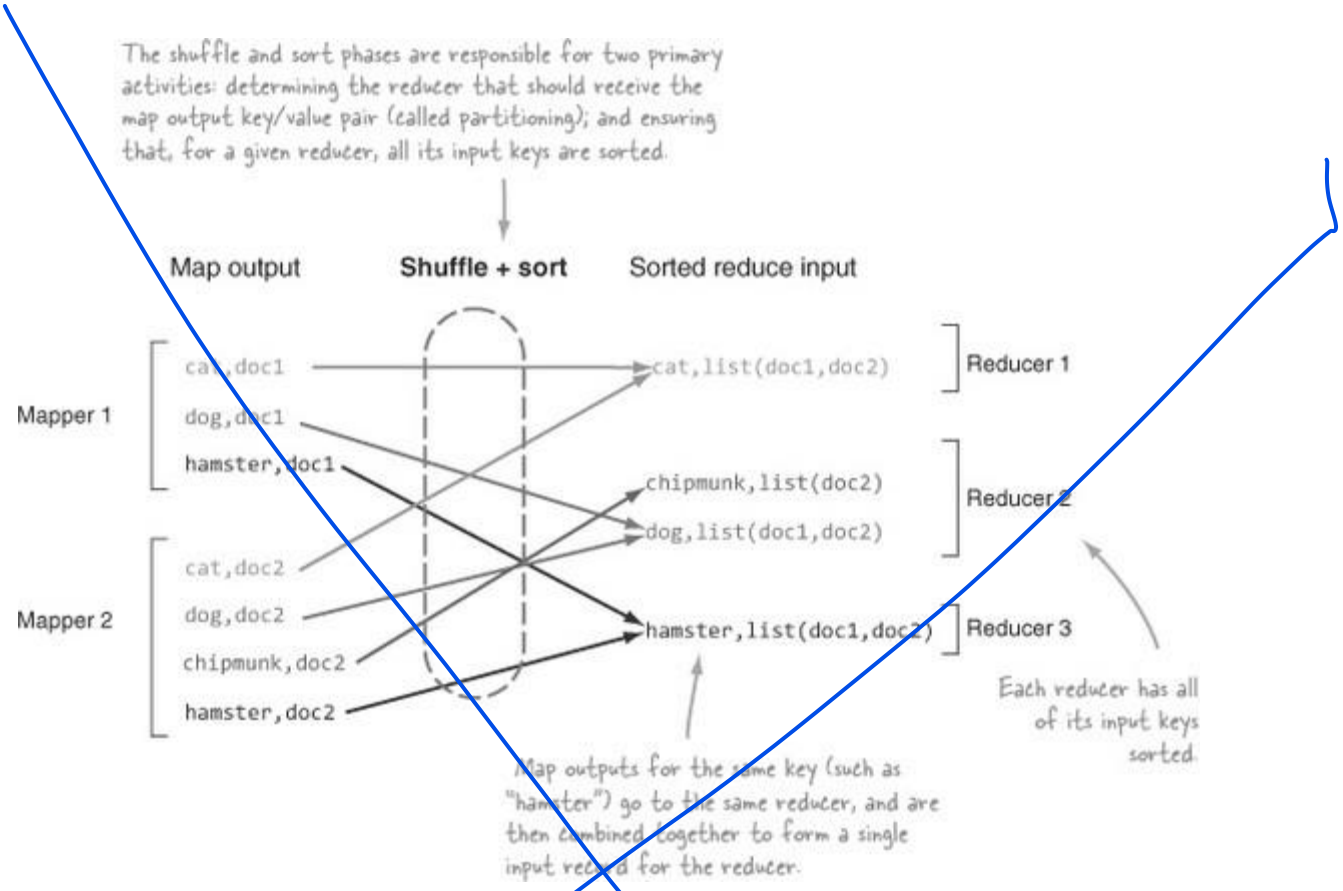


Figure A logical view of the reduce function

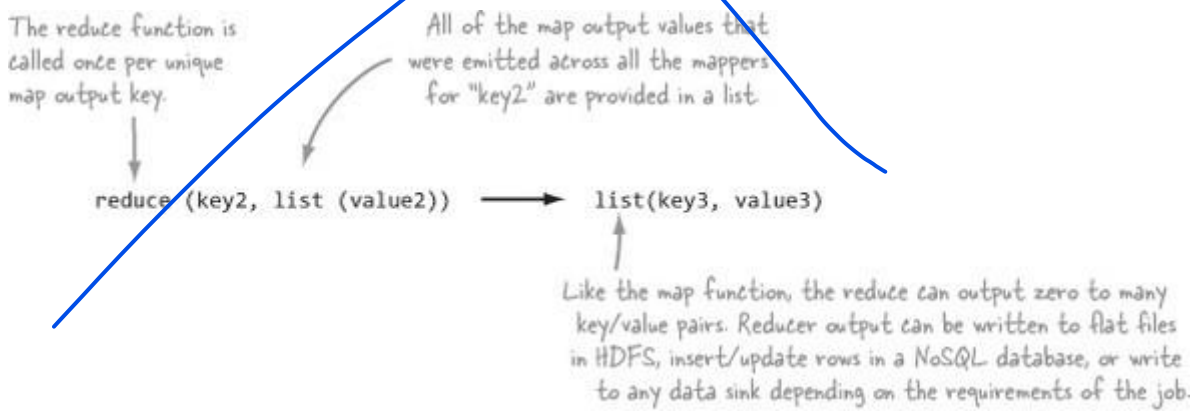
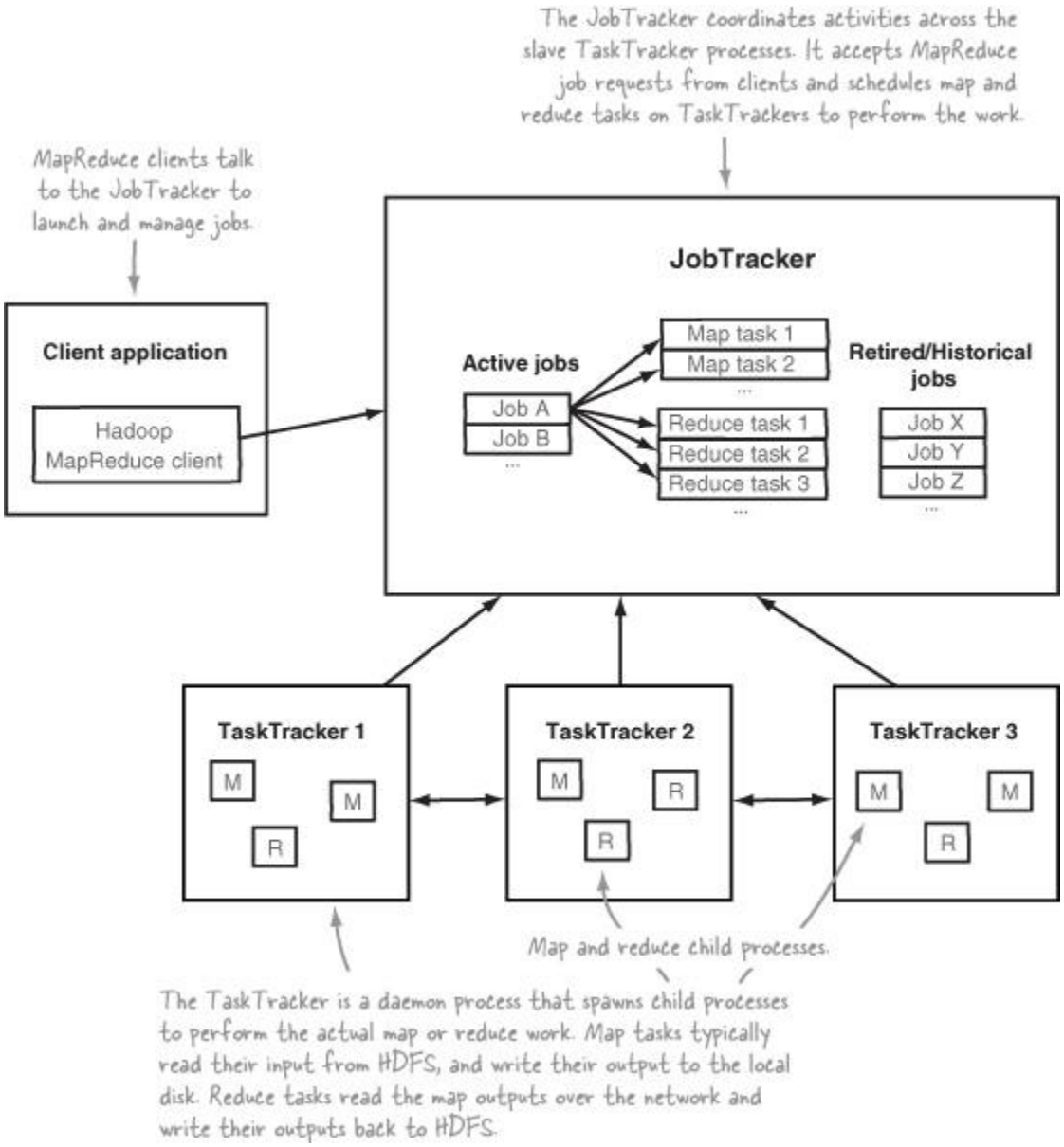


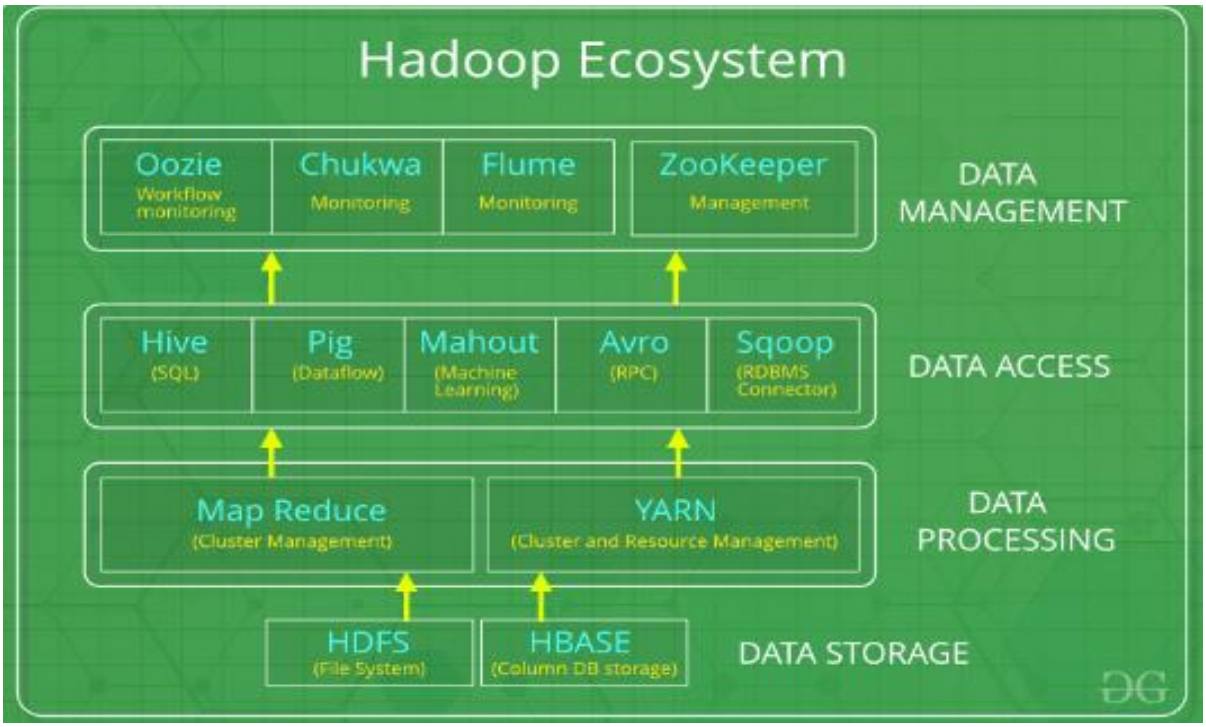
Figure MapReduce logical architecture



The Hadoop ecosystem

The Hadoop ecosystem is diverse and grows by the day. It's impossible to keep track of all of the various projects that interact with Hadoop in some form

Figure Hadoop and related technologies



Hadoop Ecosystem is a platform or a suite which provides various services to solve the big data problems. It includes Apache projects and various commercial tools and solutions.

There are *four major elements of Hadoop* i.e. **HDFS, MapReduce, YARN, and Hadoop Common**. Most of the tools or solutions are used to supplement or support these major elements.

All these tools work collectively to provide services such as absorption, analysis, storage and maintenance of data etc.

Following are the components that collectively form a Hadoop ecosystem:

- **HDFS:** Hadoop Distributed File System
- **YARN:** Yet Another Resource Negotiator
- **MapReduce:** Programming based Data Processing
- **Spark:** In-Memory data processing
- **PIG, HIVE:** Query based processing of data services
- **HBase:** NoSQL Database
- **Mahout, Spark MLLib:** Machine Learning algorithm libraries
- **Solar, Lucene:** Searching and Indexing
- **Zookeeper:** Managing cluster
- **Oozie:** Job Scheduling

HDFS:

- HDFS is the primary or major component of Hadoop ecosystem and is responsible for storing large data sets of structured or unstructured

data across various nodes and thereby maintaining the metadata in the form of log files.

- HDFS consists of two core components i.e.
    1. Name node
    2. Data Node
- Name Node is the prime node which contains metadata (data about data) requiring comparatively fewer resources than the data nodes that stores the actual data. These data nodes are commodity hardware in the distributed environment. Undoubtedly, making Hadoop cost effective.
- HDFS maintains all the coordination between the clusters and hardware, thus working at the heart of the system.

YARN:

- Yet Another Resource Negotiator, as the name implies, YARN is the one who helps to manage the resources across the clusters. In short, it performs scheduling and resource allocation for the Hadoop System.
- Consists of three major components i.e.
    1. Resource Manager
    2. Nodes Manager
    3. Application Manager
- Resource manager has the privilege of allocating resources for the applications in a system whereas Node managers work on the allocation of resources such as CPU, memory, bandwidth per machine and later on acknowledges the resource manager. Application manager works as an interface between the resource manager and node manager and performs negotiations as per the requirement of the two.

MapReduce:

- By making the use of distributed and parallel algorithms, MapReduce makes it possible to carry over the processing's logic and helps to write applications which transform big data sets into a manageable one.
- MapReduce makes the use of two functions i.e. Map() and Reduce() whose task is:
    1. *Map()* performs sorting and filtering of data and thereby organizing them in the form of group. Map generates a key-value pair based result which is later on processed by the Reduce() method.
    2. *Reduce()*, as the name suggests does the summarization by aggregating the mapped data. In simple, Reduce() takes the output generated by Map() as input and combines those tuples into smaller set of tuples.

PIG:

Pig was basically developed by Yahoo which works on a pig Latin language, which is Query based language similar to SQL.

- It is a platform for structuring the data flow, processing and analyzing huge data sets.
- Pig does the work of executing commands and in the background, all the activities of MapReduce are taken care of. After the processing, pig stores the result in HDFS.
- Pig Latin language is specially designed for this framework which runs on Pig Runtime. Just the way Java runs on the JVM.
- Pig helps to achieve ease of programming and optimization and hence is a major segment of the Hadoop Ecosystem.

HIVE:

- With the help of SQL methodology and interface, HIVE performs reading and writing of large data sets. However, its query language is called as HQL (Hive Query Language).
- It is highly scalable as it allows real-time processing and batch processing both. Also, all the SQL datatypes are supported by Hive thus, making the query processing easier.
- Similar to the Query Processing frameworks, HIVE too comes with two components: *JDBC Drivers* and *HIVE Command Line*.
- JDBC, along with ODBC drivers work on establishing the data storage permissions and connection whereas HIVE Command line helps in the processing of queries.

Mahout:

- Mahout, allows Machine Learnability to a system or application. Machine Learning, as the name suggests helps the system to develop itself based on some patterns, user/environmental interaction or on the basis of algorithms.
- It provides various libraries or functionalities such as collaborative filtering, clustering, and classification which are nothing but concepts of Machine learning. It allows invoking algorithms as per our need with the help of its own libraries.

ApacheSpark:

- It's a platform that handles all the process consumptive tasks like batch processing, interactive or iterative real-time processing, graph conversions, and visualization, etc.
- It consumes in memory resources hence, thus being faster than the prior in terms of optimization.
- Spark is best suited for real-time data whereas Hadoop is best suited for structured data or batch processing, hence both are used in most of the companies interchangeably.

ApacheHBase:

- It's a NoSQL database which supports all kinds of data and thus capable of handling anything of Hadoop Database. It provides capabilities of Google's BigTable, thus able to work on Big Data sets effectively.
- At times where we need to search or retrieve the occurrences of something small in a huge database, the request must be processed within a short quick span of time. At such times, HBase comes handy as it gives us a tolerant way of storing limited data

Other Components: Apart from all of these, there are some other components too that carry out a huge task in order to make Hadoop capable of processing large datasets. They are as follows:

- **Solr, Lucene:** These are the two services that perform the task of searching and indexing with the help of some java libraries, especially Lucene is based on Java which allows spell check mechanism, as well. However, Lucene is driven by Solr.
- **Zookeeper:** There was a huge issue of management of coordination and synchronization among the resources or the components of Hadoop which resulted in inconsistency, often. Zookeeper overcame all the problems by performing synchronization, inter-component based communication, grouping, and maintenance.
- **Oozie:** Oozie simply performs the task of a scheduler, thus scheduling jobs and binding them together as a single unit. There is two kinds of jobs .i.e Oozie workflow and Oozie coordinator jobs. Oozie workflow is the jobs that need to be executed in a sequentially ordered manner whereas Oozie Coordinator jobs are those that are triggered when some data or external stimulus is given to it.

Apache Avro

It is a row-oriented remote procedure call and data serialization framework developed within Apache's Hadoop project. It uses JSON for defining data types and protocols, and serializes data in a compact binary format.

Its primary use is in Apache Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services.

Avro uses a schema to structure the data that is being encoded. It has two different types of schema languages; one for human editing (Avro IDL) and another which is more machine-readable based on JSON.

Apache Sqoop

Apache Sqoop is part of the Hadoop ecosystem. Since a lot of the data had to be transferred from relational database systems onto Hadoop, there was a need for a dedicated tool to do this task fast. This is where Apache Sqoop came into the picture which is now extensively used for transferring data from **RDBMS files to the** Hadoop ecosystem for MapReduce processing and so on.

When it comes to transferring data, there is a certain set of requirements to be taken care of. It includes the following: Data has to have consistency; it should be prepared for provisioning the downstream pipeline, and the users should ensure the consumption of production system resources; among other things. The MapReduce application is not able to directly access the data that is residing in external relational databases. This method can expose the system to the risk of too much load generation from the cluster nodes.

Apache Chukwa

Apache Chukwa is an open source data collection system for monitoring large distributed systems.

Apache Chukwa is built on top of the Hadoop Distributed File System (HDFS) and Map/Reduce framework and inherits Hadoop's scalability and robustness.

Apache Chukwa also includes a flexible and powerful toolkit for displaying, monitoring and analyzing results to make the best use of the collected data.

Apache Flume

Apache Flume is an open-source tool for collecting, aggregating, and moving huge amounts of streaming data from the external web servers to the central store, say HDFS, HBase, etc. It is a highly available and reliable service which has tunable recovery mechanisms.

The main purpose of designing Apache Flume is to move streaming data generated by various applications to Hadoop Distributed FileSystem.

A company has millions of services that are running on multiple servers. Thus, produce lots of logs. In order to gain insights and understand customer behavior, they need to analyze these logs altogether.

In order to process logs, a company requires an extensible, scalable, and reliable distributed data collection service.

That service must be capable of performing the flow of unstructured data such as logs from source to the system where they will be processed (such as in Hadoop Distributed FileSystem). Flume is an open-source distributed data collection service used for transferring the data from source to destination.

It is a reliable, and highly available service for collecting, aggregating, and transferring huge amounts of logs into HDFS. It has a simple and flexible architecture.

Apache Flume is highly robust and fault-tolerant and has tunable reliability mechanisms for fail-over and recovery. It allows the collection of data collection in batch as well as in streaming mode.

Moving data in and out of Hadoop

Moving data in and out of Hadoop is as data ingress and egress, is the process by which data is transported from an external system into an internal system, and vice versa. Hadoop supports ingress and egress at a low level in HDFS and MapReduce.

Files can be moved in and out of HDFS, and data can be pulled from external data sources and pushed to external data sinks using MapReduce.

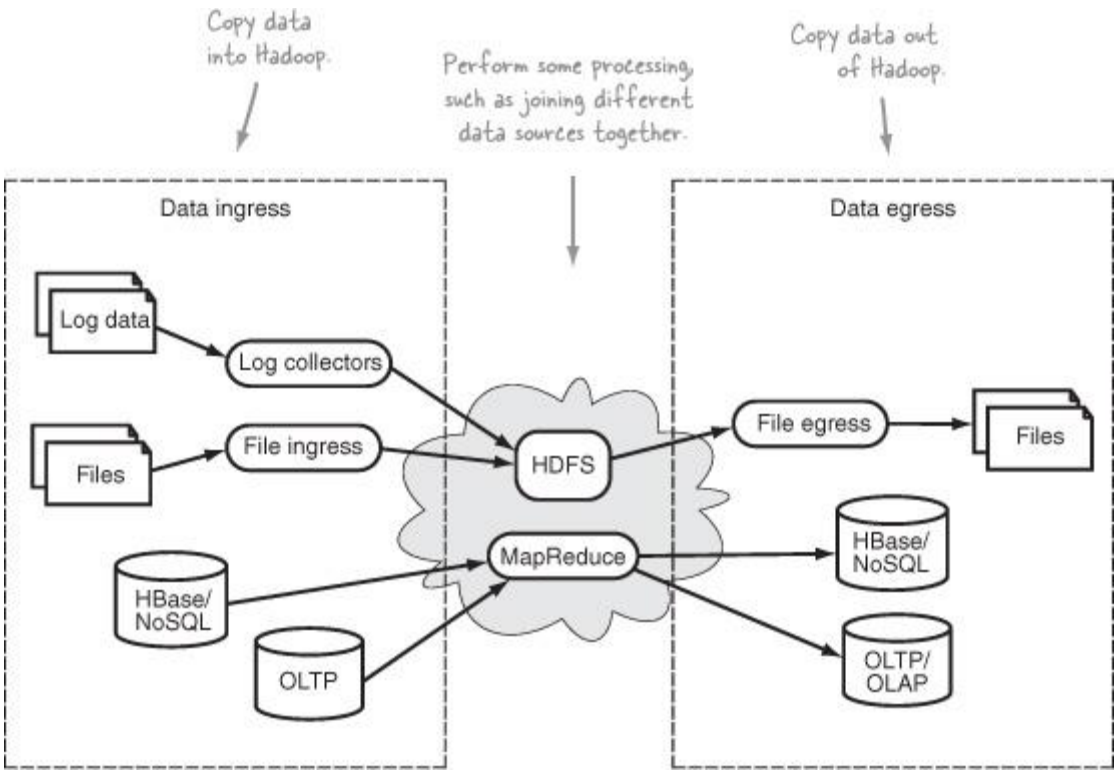


Figure :Hadoop data ingress and egress transports data to and from an external system to an internal one.

The fact that your data exists in various forms and locations throughout your environments complicates the process of ingress and egress.

How do you bring in data that's sitting in an OLTP (online transaction processing) database?

Or ingress log data that's being produced by tens of thousands of production servers? Or work with binary data sitting behind a firewall?

Further, how do you automate your data ingress and egress process so that your data is moved at regular intervals?

Automation is a critical part of the process, along with monitoring and data integrity responsibilities to ensure correct and safe transportation of data.

Key elements of ingress and egress

Idempotence

An idempotent operation produces the same result no matter how many times it's executed. In a relational database the inserts typically aren't idempotent, because executing them multiple times doesn't produce the same resulting database state.

Alternatively, updates often are idempotent, because they'll produce the same end result.

Any time data is being written idempotence should be a consideration, and data ingress and egress in Hadoop is no different.

How well do distributed log collection frameworks deal with data retransmissions?

How do you ensure idempotent behavior in a MapReduce job where multiple tasks are inserting into a database in parallel.

Aggregation

- The data aggregation process combines multiple data elements.
- In the context of data ingress this can be useful because moving large quantities of small files into HDFS potentially translates into NameNode memory woes, as well as slow MapReduce execution times.
- Having the ability to aggregate files or data together mitigates this problem, and is a feature to consider.

Data Format Transformation

- The data format transformation process converts one data format into another.
- Often your source data isn't in a format that's ideal for processing in tools such as Map-Reduce.
- If your source data is multiline XML or JSON form, for example, you may want to consider a preprocessing step.
- This would convert the data into a form that can be split, such as a JSON or an XML element per line, or convert it into a format such as Avro.

Recoverability

- Recoverability allows an ingress or egress tool to retry in the event of a failed operation.

- Because it's unlikely that any data source, sink, or Hadoop itself can be 100 percent available, it's important that an ingress or egress action be retried in the event of failure.

Correctness

- In the context of data transportation, checking for correctness is how you verify that no data corruption occurred as the data was in transit.

- When you work with heterogeneous systems such as Hadoop data ingress and egress tools, the fact that data is being transported across different hosts, networks, and protocols only increases the potential for problems during data transfer.

- Common methods for checking correctness of raw data such as storage devices include Cyclic Redundancy Checks (CRC), which are what HDFS uses internally to maintain block-level integrity.

Resource Consumption and Performance

Resource consumption and performance are measures of system resource utilization and system efficiency, respectively. Ingress and egress tools don't typically incur significant load (resource consumption) on a system, unless you have appreciable data volumes.

For performance, the questions to ask include whether the tool performs ingress and egress activities in parallel, and if so, what mechanisms it provides to tune the amount of parallelism.

For example, if your data source is a production database, don't use a large number of concurrent map tasks to import data.

Monitoring

Monitoring ensures that functions are performing as expected in automated systems.

For data ingress and egress, monitoring breaks down into two elements: ensuring that the process(es) involved in ingress and egress are alive, and validating that source and destination data are being produced as expected.

Moving data into Hadoop

There are two primary methods that can be used for moving data into Hadoop: writing external data at the HDFS level (a data push), or reading external data at the MapReduce level (more like a pull).

Reading data in MapReduce has advantages in the ease with which the operation can be parallelized and fault tolerant.

Low-Level Hadoop Ingress Mechanisms

These mechanisms include Hadoop's Java HDFS API, WebHDFS, the new Hadoop 0.23 REST API, and MapReduce.

Pushing log files into Hadoop

Log data has long been prevalent across all applications, but with Hadoop came the ability to process the large volumes of log data produced by production systems.

Various systems produce log data, from network devices and operating systems to web servers and applications.

These log files all offer the potential for valuable insights into how systems and applications operate as well as how they're used. What unifies log files is that they tend to be in text form and line-oriented, making them easy to process.

Comparing Flume, Chukwa, and Scribe

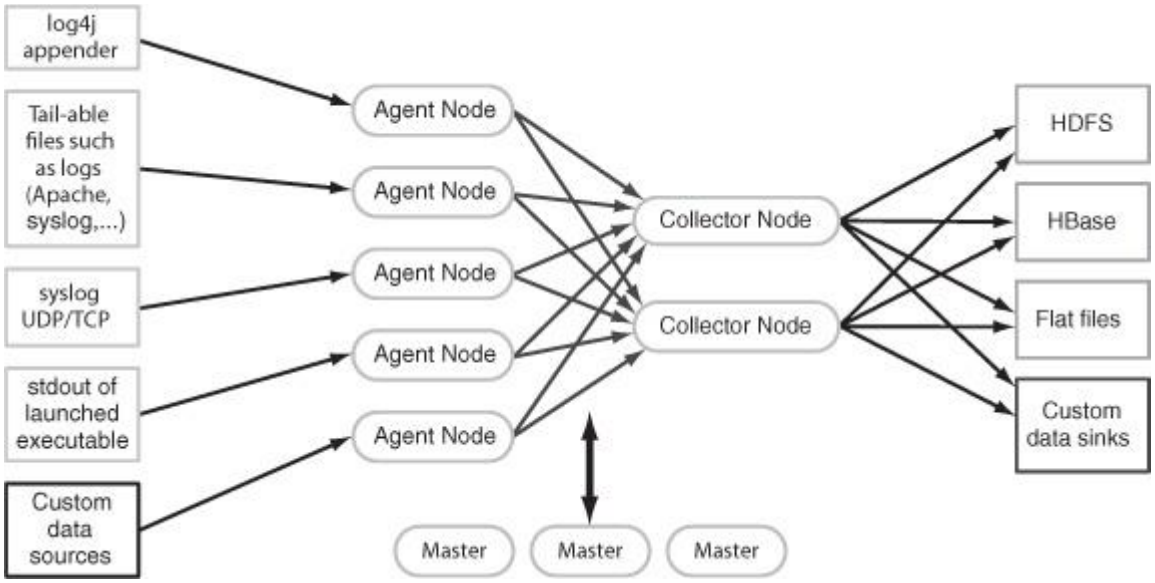
Flume, Chukwa, and Scribe are log collecting and distribution frameworks that have the capability to use HDFS as a data sink for that log data. It can be challenging to differentiate between them because they share the same features.

Flume

Apache Flume is a distributed system for collecting streaming data. It's an Apache project in incubator status, originally developed by Cloudera.

It offers various levels of reliability and transport delivery guarantees that can be tuned to your needs. It's highly customizable and supports a plugin architecture where you can add custom data sources and data sinks.

Flume architecture for collecting streaming data



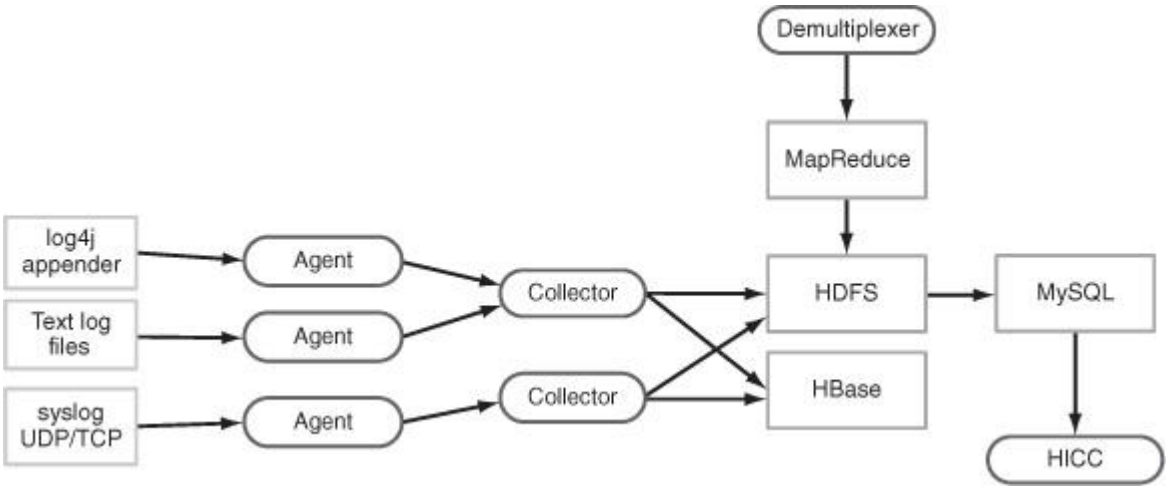
Chukwa

- Chukwa is an Apache subproject of Hadoop that also offers a large-scale mechanism to collect and store data in HDFS.

- And it's also in incubator status. Chukwa's reliability model supports two levels: end-to-end reliability, and fast-path delivery, which minimizes latencies.

- After writing data into HDFS Chukwa runs a MapReduce job to demultiplex the data into separate streams. Chukwa also offers a tool called Hadoop Infrastructure Care Center (HICC), which is a web interface for visualizing system performance.

Chukwa architecture for collecting and storing data in HDFS



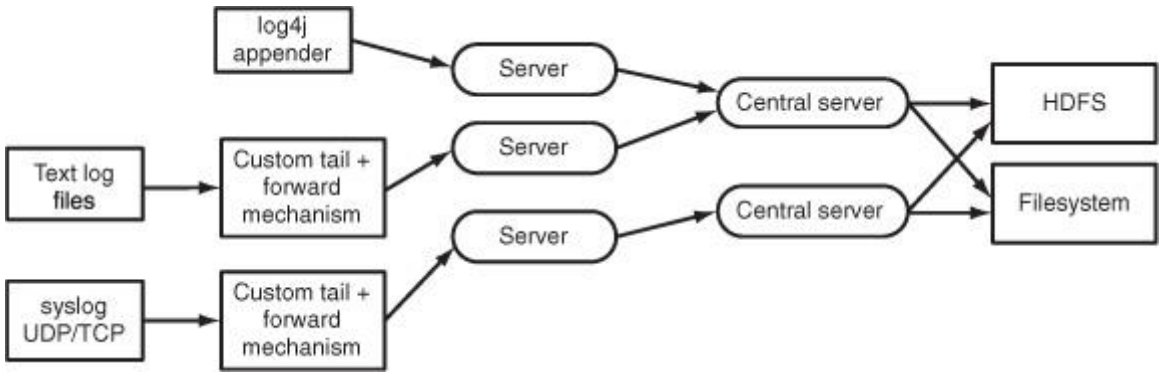
Scribe

Scribe is a rudimentary streaming log distribution service, developed and used heavily by Facebook.

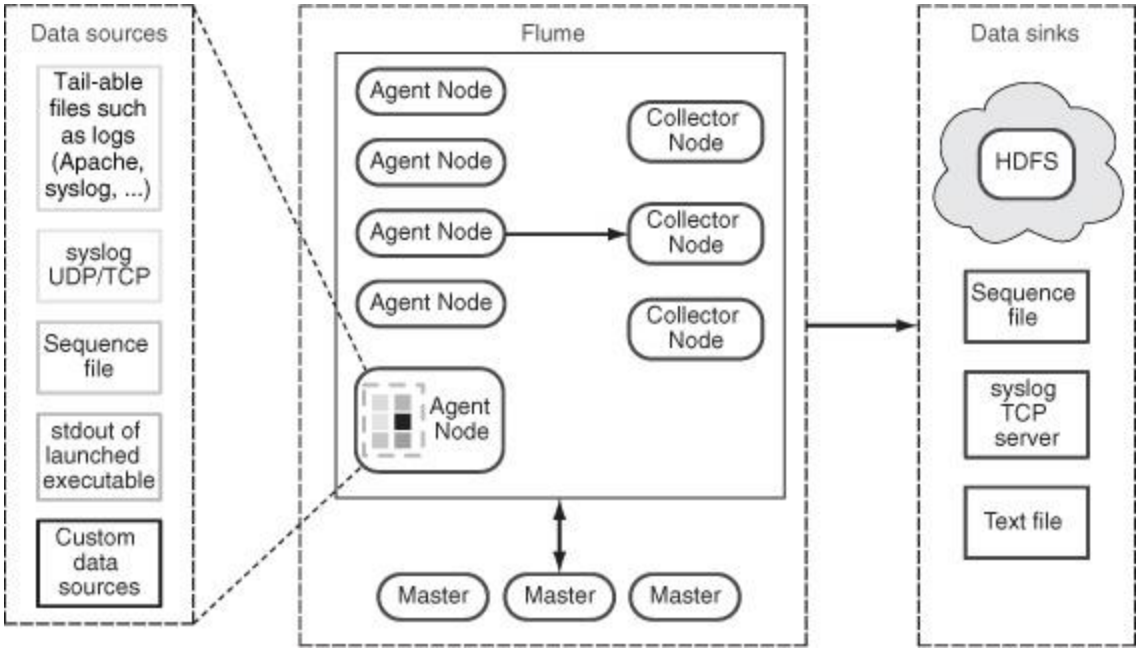
A scribe server that collects logs runs on every node and forwards them to a central Scribe server. Scribe supports multiple data sinks, including HDFS, regular filesystems, and NFS.

Scribe's reliability comes from a file-based mechanism where the server persists to a local disk in the event it can't reach the downstream server.

Scribe architecture also pushes log data into HDFS.



Pushing system log messages into HDFS with Flume



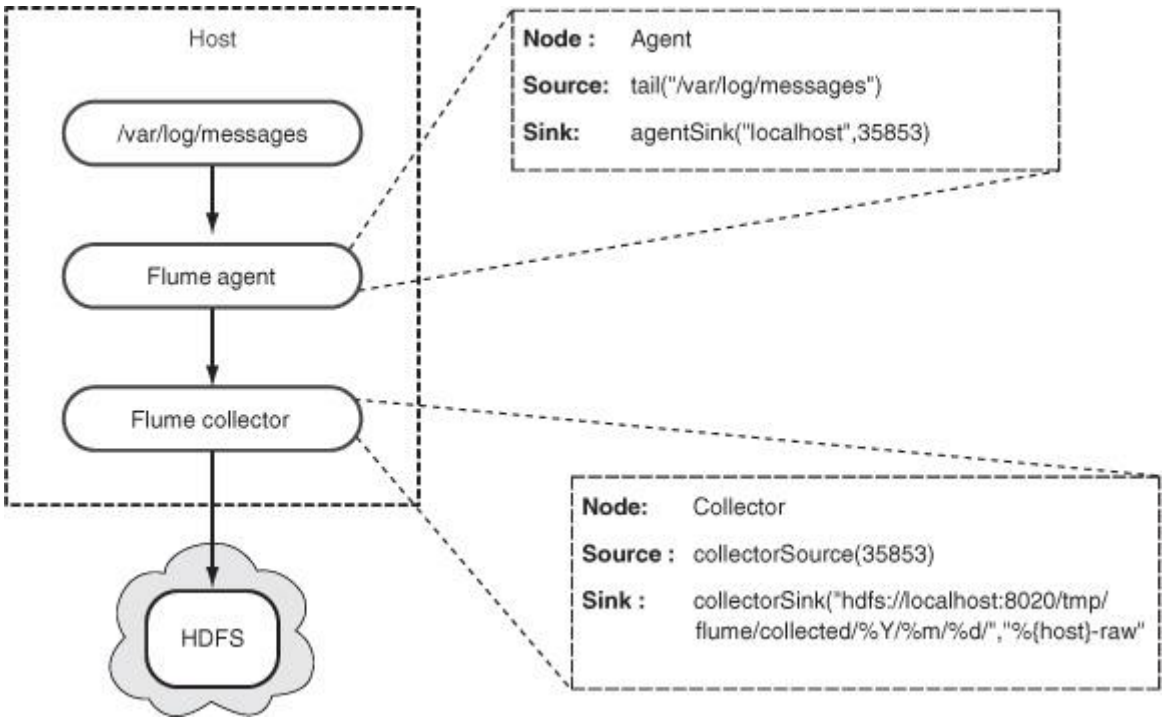
Example of Flume deployment for collecting streaming data

The primary components of the Flume are

- *Nodes*—Flume data paths that ferry data from a data source to a data sink. Agents and Collectors are simply Flume Nodes that are deployed in a way to efficiently and reliably work with a large number of data sources.
- *Agents*—Collect streaming data from the local host and forward it to the Collectors.
- *Collectors*—Aggregate data sent from the Agents and write that data into HDFS.
- *Masters*—Perform configuration management tasks and also help with reliable data flow.
-

Examples include application logs and Linux system logs, as well as nontext data that can be supported with custom data sources.

Data sinks are the destination of that data, which can be HDFS, flat files, and any data target that can be supported with custom data sinks.

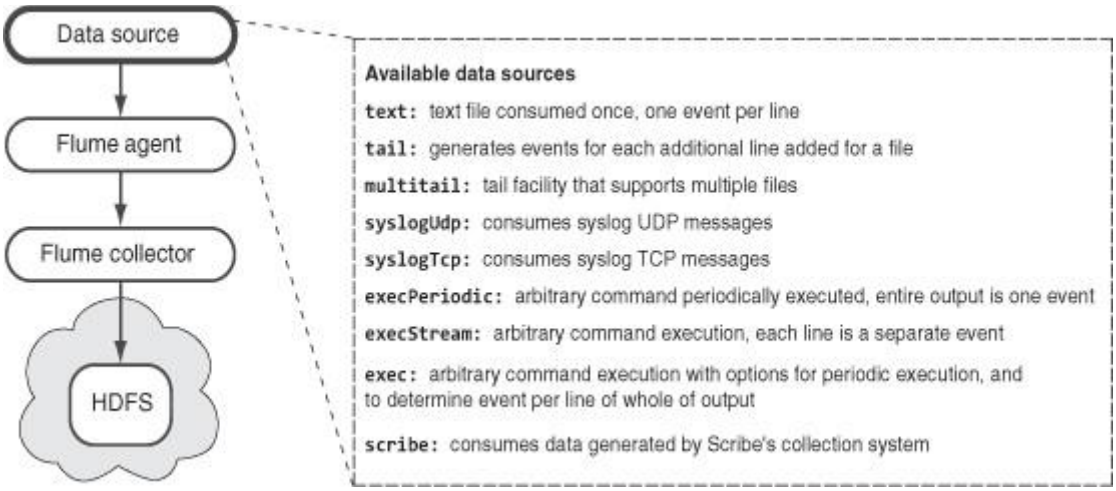


Data flow from /var/log/messages into HDFS

Flume Data Sources

A data source is required for both the Agent and Collector Nodes; it determines where they collect their data. The Agent Node's data source is your application or system data that you want to transfer to HDFS, and the Collector Node's data source is the Agent's data sink.

Figure Flume Agent Node data sources supported by the Agent Node



An automated mechanism to copy files into HDFS

Existing file transportation mechanisms such as Flume, Scribe, and Chukwa are geared towards supporting log files. What if you have different file formats for your files, such as semistructured or binary?

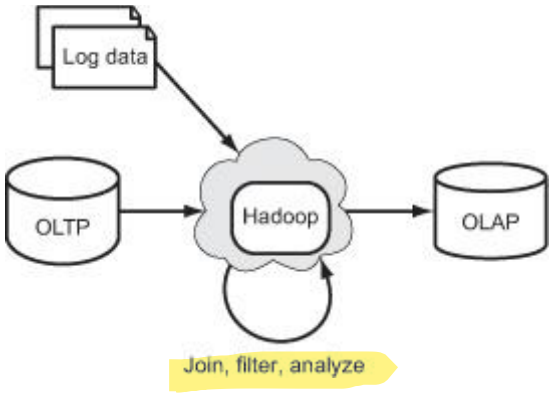
If the files were siloed in a way that the Hadoop slave nodes couldn't directly access, then you couldn't use Oozie to help with file ingress either.

Pulling data from databases

Most organizations' crucial data exists across a number of OLTP databases.

The data stored in these databases contains information about users, products, and a host of other useful items. If you want to analyze this data the traditional mechanism for doing so would be to periodically copy that data into a OLAP data warehouse.

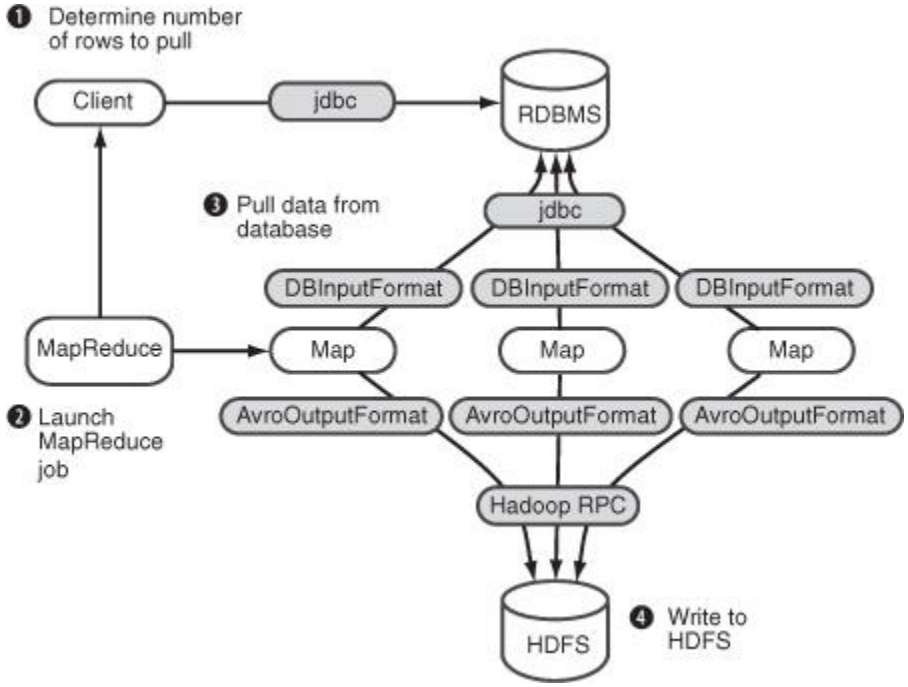
Using Hadoop for data ingress, joining, and egress to OLAP



Facebook is an example of an organization that has successfully utilized Hadoop and Hive as an OLAP platform to work with petabytes of data.

shows an architecture similar to that of Facebook's. This architecture also includes a feedback loop into the OLTP system, which can be used to push discoveries made in Hadoop, such as recommendations for users.

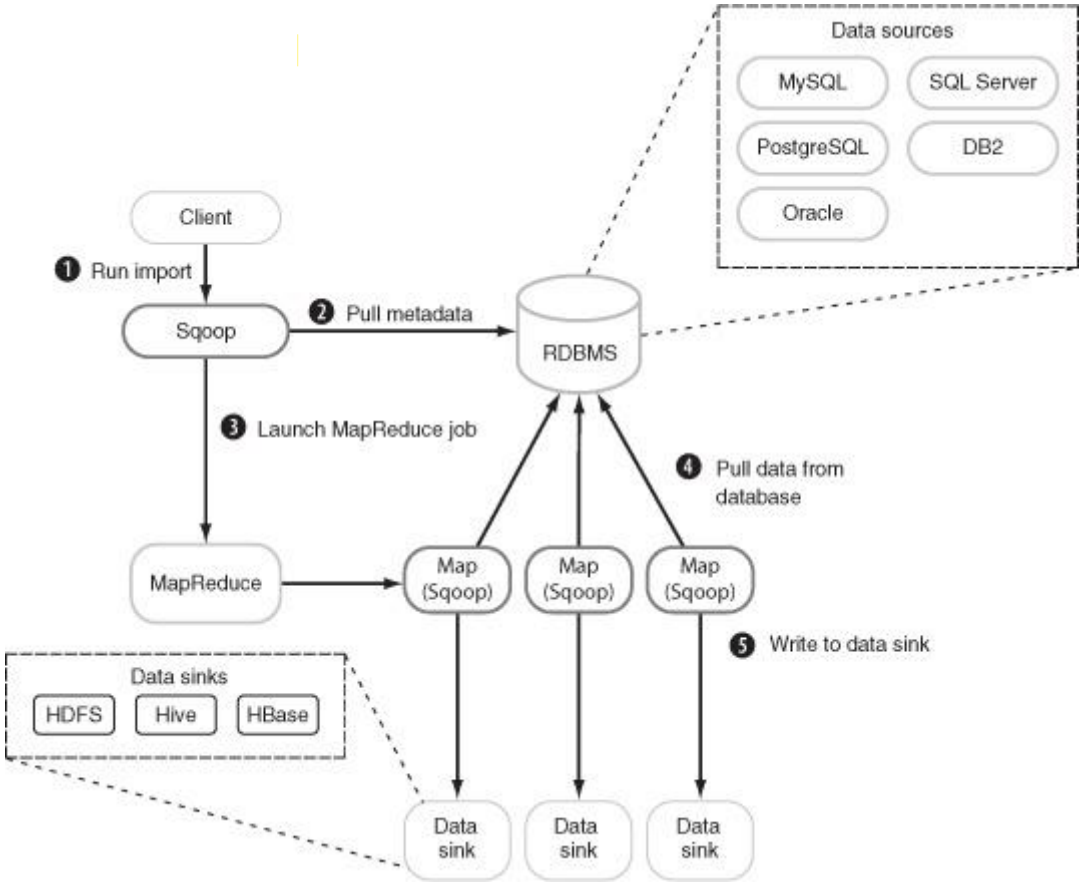
Example of MapReduce in four stages where `DBInputFormat` is used to pull data from a database



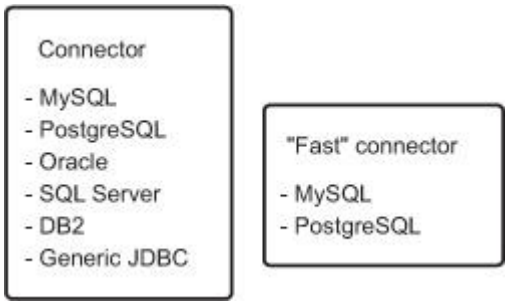
Using Sqoop to import data from MySQL

To use Sqoop as a simple mechanism to bring relational data into Hadoop clusters. We'll walk through the process of importing data from MySQL into Sqoop.

Figure Five-stage Sqoop import overview: connecting to the data source and using MapReduce to write to a data sink

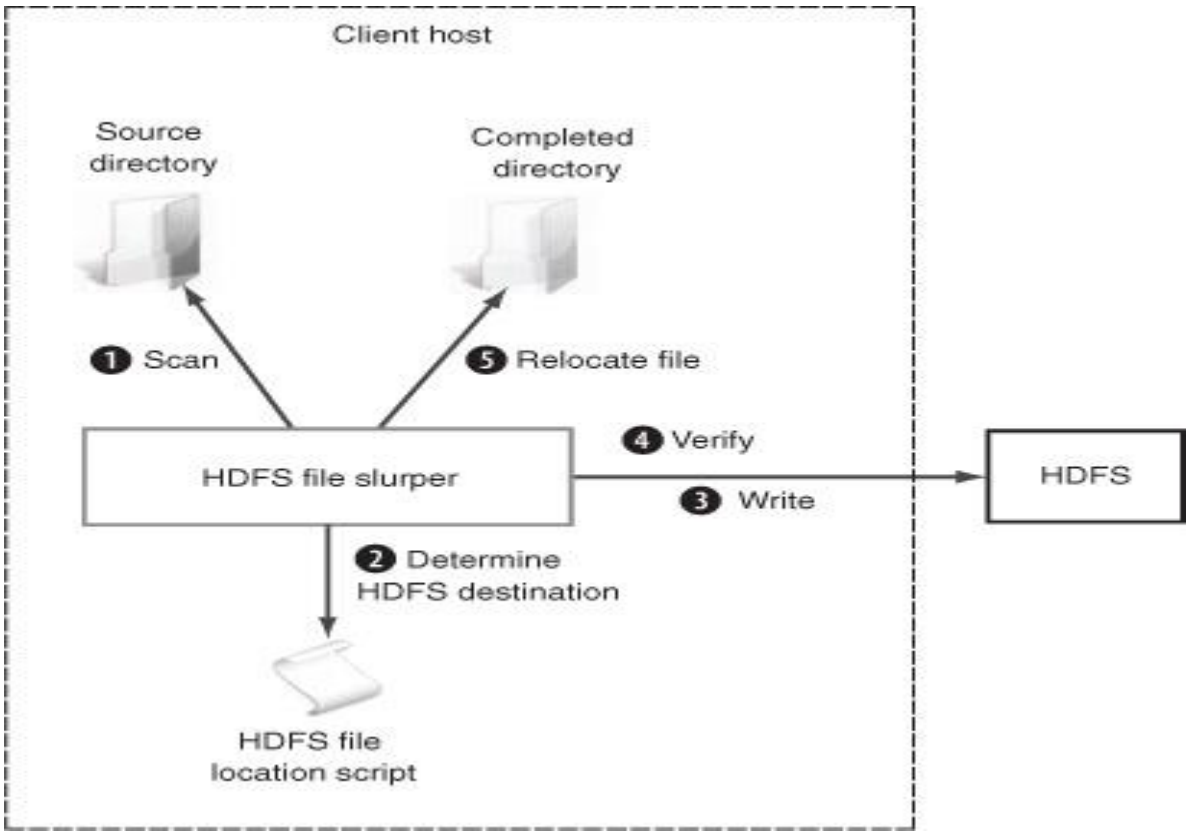


Sqoop connectors used to read and write to external systems



An automated mechanism to copy files into HDFS

Existing file transportation mechanisms such as Flume, Scribe, and Chukwa are geared towards supporting log files. What if you have different file formats for your files, such as semistructured or binary? If the files were siloed in a way that the Hadoop slave nodes couldn't directly access, then you couldn't use Oozie to help with file ingress either.



  - How do you effectively partition your writes to HDFS so that you don't lump everything into a single directory?
  - How do you determine that your data is ready in HDFS for processing in order to avoid reading files that are mid-copy?
  - How do you automate regular execution of your utility?
- The first step is to download and build the code. The following assumes that you have git, Java, and version 3.0 or newer of Maven installed locally:

```
$ git clone git://github.com/alexholmes/hdfs-file-slurper.git
$ cd hdfs-file-slurper/
$ mvn package
```

- Next you'll need to untar the tarball that the build created under /usr/local:

```
$ sudo tar -xzf target/hdfs-slurper-<version>-package.tar.gz \
  -C /usr/local/

$ sudo ln -s /usr/local/hdfs-slurper-<version>
/usr/local/hdfs-slurper
```

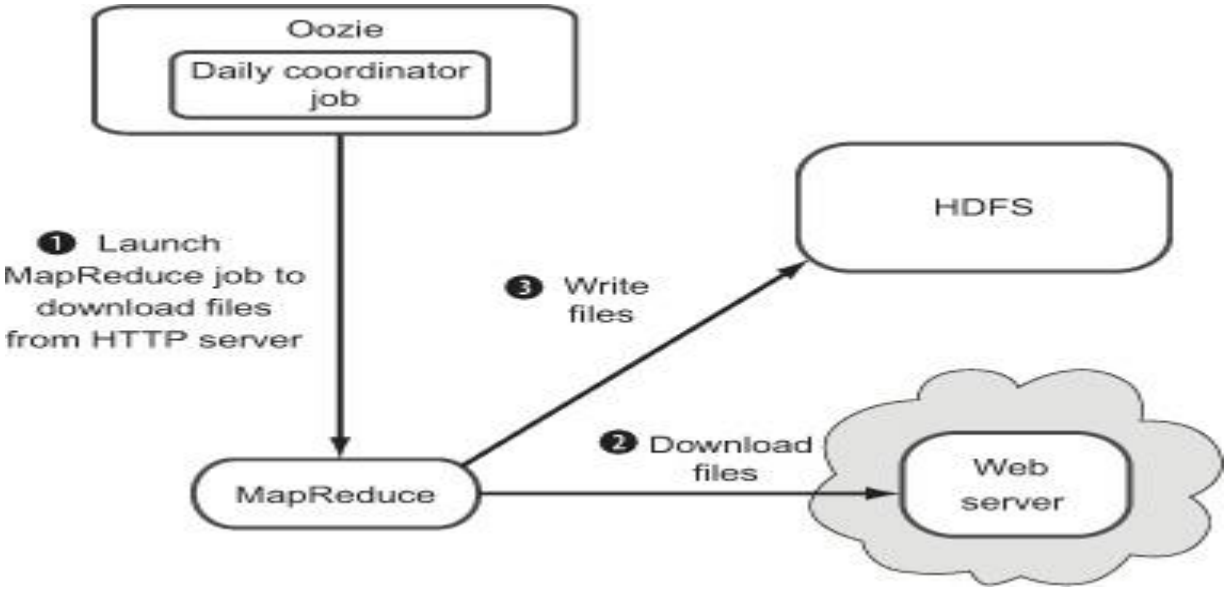
Scheduling regular ingress activities with Oozie

If your data is sitting on a filesystem, web server, or any other system accessible from your Hadoop cluster, you'll need a way to periodically pull that data into

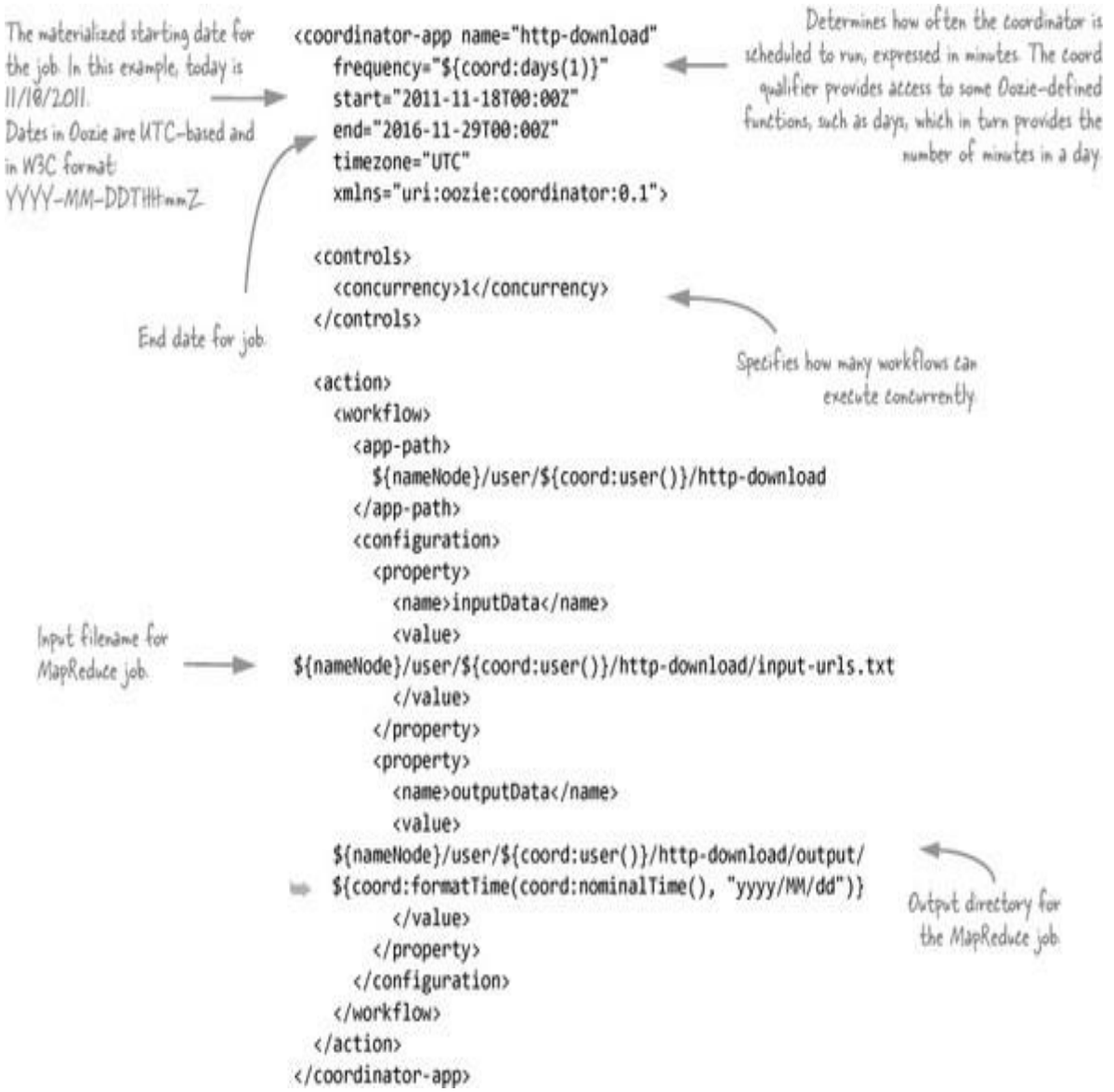
Hadoop. While tools exist to help with pushing log files and pulling from databases (which we'll cover in this chapter), if you need to interface with some other system, it's likely you'll need to handle the data ingress process yourself.

There are two parts to this data ingress process: the first is how you import data from another system into Hadoop, and the second is how you regularly schedule the data transfer.

Oozie can be used to ingress data into HDFS, and can also be used to execute post-ingress activities such as launching a MapReduce job to process the ingressed data. An Apache project, Oozie started life inside Yahoo. It's a Hadoop workflow engine that manages data processing activities. For our scenario Oozie has a coordinator engine that can start workflows based on data and time triggers.



You'll use Oozie's data triggering capabilities to kick off a MapReduce job every 24 hours. Oozie has the notion of a coordinator job, which can launch a workflow at fixed intervals. The first step is to look at the coordinator XML configuration file. This file is used by Oozie's Coordination Engine to determine when it should kick off a workflow. Oozie uses the JSP expression language to perform parameterization, as you'll see in the following code. Create a file called coordinator.xml with the content shown in the next listing.

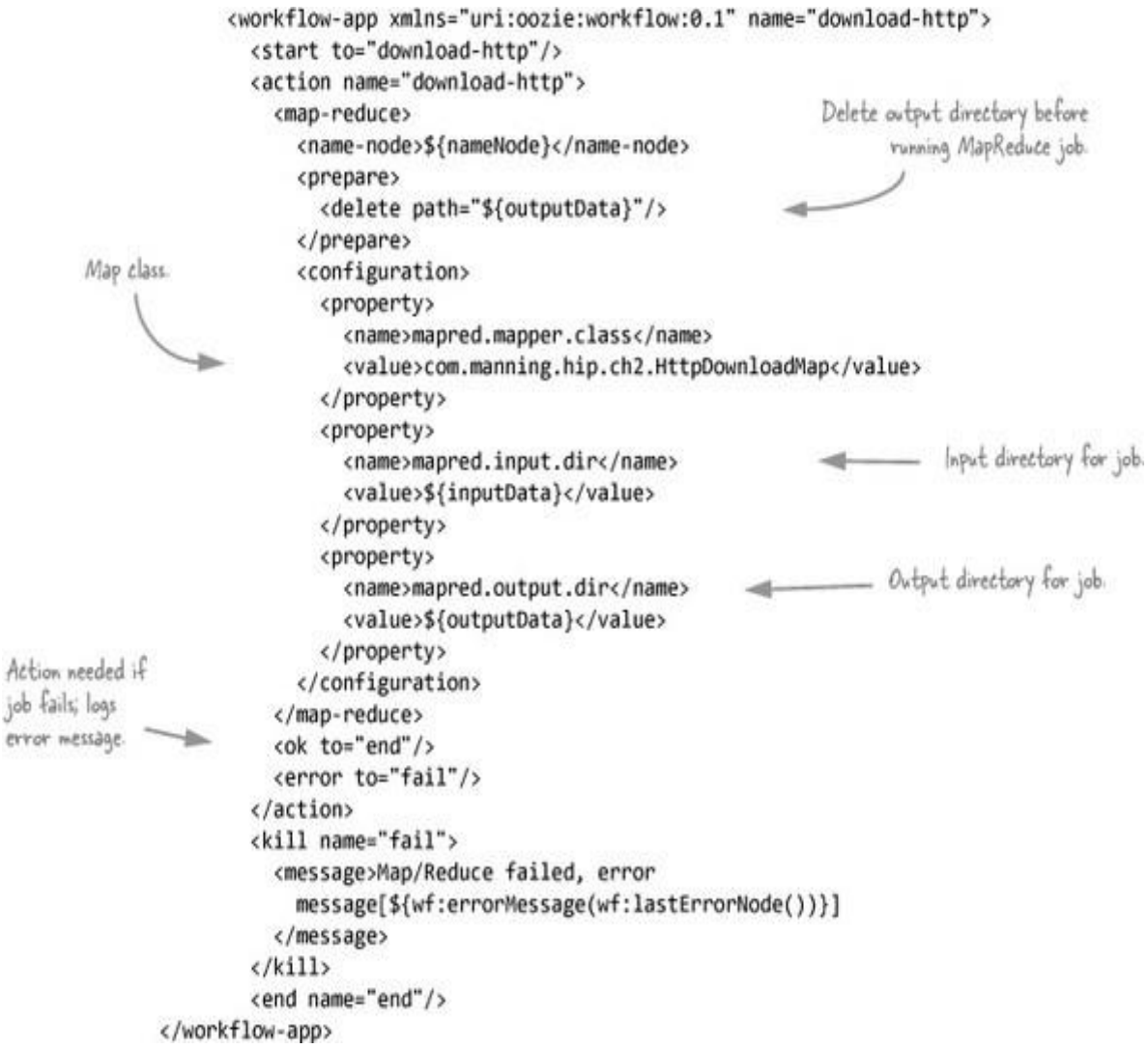



```
<coordinator-app name="http-download"
    frequency="${coord:days(1)}"
    start="2011-11-18T00:00Z"
    end="2016-11-29T00:00Z"
    timezone="UTC"
    xmlns="uri:oozie:coordinator:0.1">

  <controls>
    <concurrency>1</concurrency>
  </controls>

  <action>
    <workflow>
      <app-path>
        ${nameNode}/user/${coord:user()}/http-download
      </app-path>
      <configuration>
        <property>
          <name>inputData</name>
          <value>
${nameNode}/user/${coord:user()}/http-download/input-urls.txt
          </value>
        </property>
        <property>
          <name>outputData</name>
          <value>
    ${nameNode}/user/${coord:user()}/http-download/output/
    ${coord:formatTime(coord:nominalTime(), "yyyy/MM/dd")}
          </value>
        </property>
      </configuration>
    </workflow>
  </action>
</coordinator-app>
```

Defining the past workflow using Oozie's coordinator




```
<workflow-app xmlns="uri:oozie:workflow:0.1" name="download-http">
  <start to="download-http"/>
  <action name="download-http">
    <map-reduce>
      <name-node>${nameNode}</name-node>
      <prepare>
        <delete path="${outputData}"/>
      </prepare>
      <configuration>
        <property>
          <name>mapred.mapper.class</name>
          <value>com.manning.hip.ch2.HttpDownloadMap</value>
        </property>
        <property>
          <name>mapred.input.dir</name>
          <value>${inputData}</value>
        </property>
        <property>
          <name>mapred.output.dir</name>
          <value>${outputData}</value>
        </property>
      </configuration>
    </map-reduce>
    <ok to="end"/>
    <error to="fail"/>
  </action>
  <kill name="fail">
    <message>Map/Reduce failed, error
      message[${wf:errorMessage(wf:lastErrorNode())}]
    </message>
  </kill>
  <end name="end"/>
</workflow-app>
```

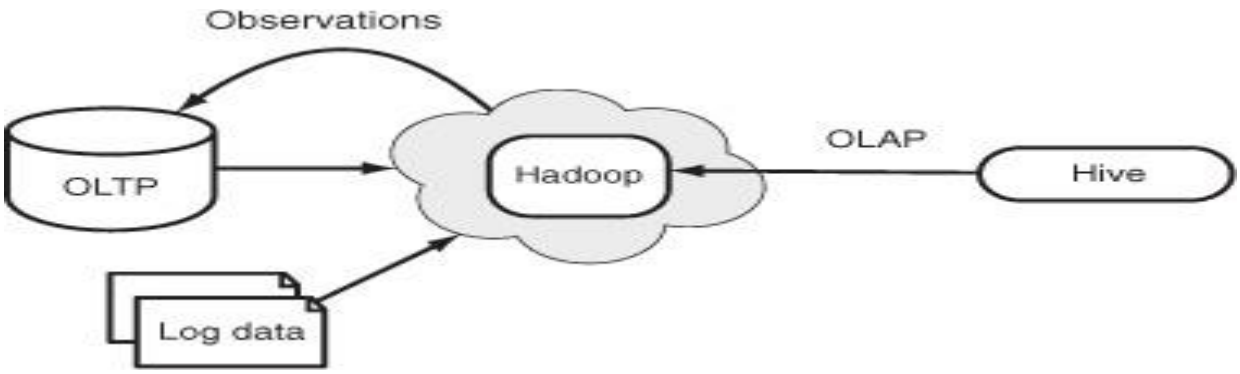
Pulling data from databases

Most organizations' crucial data exists across a number of OLTP databases. The data stored in these databases contains information about users, products, and a host of other useful items. If you want to analyze this data the traditional mechanism for doing so would be to periodically copy that data into a OLAP data warehouse.

Using Hadoop for data ingress, joining, and egress to OLAP

Facebook is an example of an organization that has successfully utilized Hadoop and Hive as an OLAP platform to work with petabytes of data.

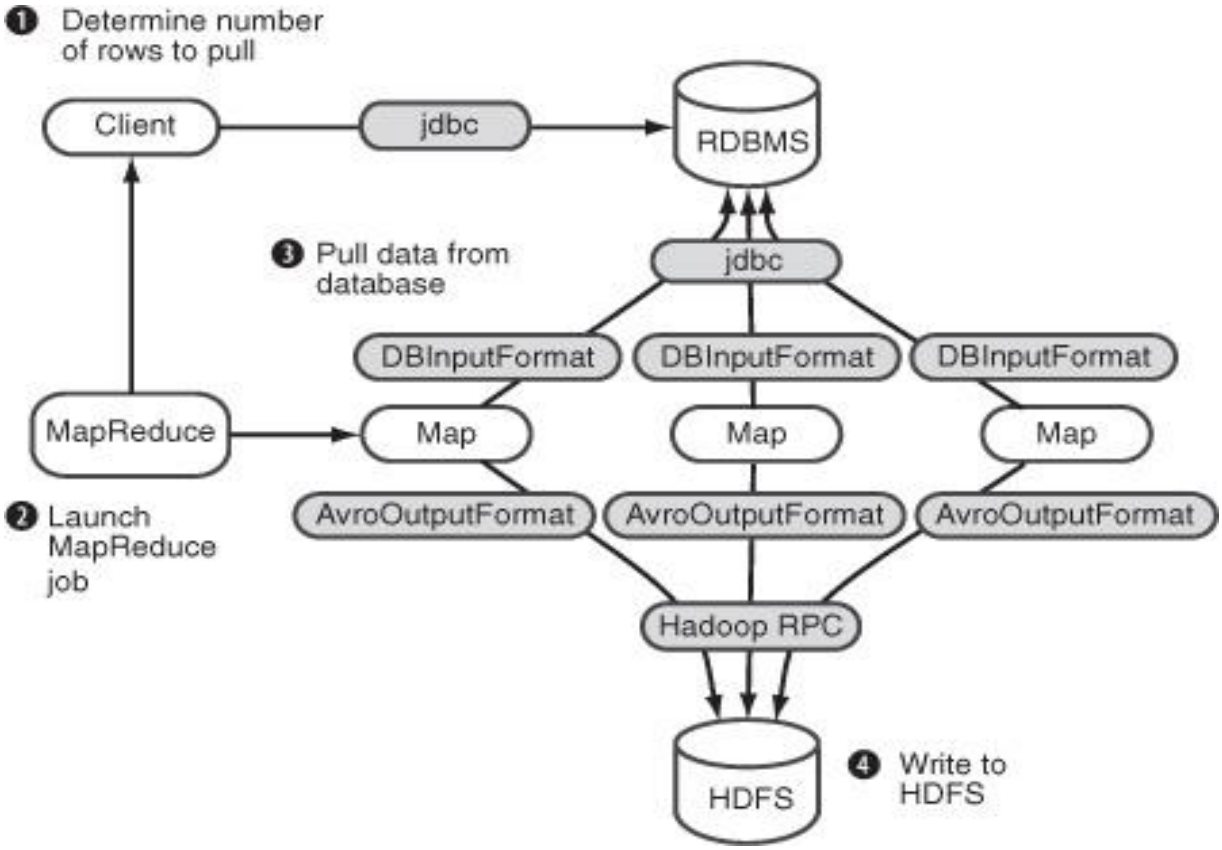
Using Hadoop for OLAP and feedback to OLTP systems



In either of the usage models shown in the previous figures, you need a way to bring relational data into Hadoop, and to also export it into relational databases. In the next techniques we'll cover two mechanisms you can use for database ingress. The first uses some built-in MapReduce classes, and the second provides an easy-to-use tool that removes the need for you to write your own code.

Database ingress with MapReduce

Example of MapReduce in four stages where `DBInputFormat` is used to pull data from a database



The details on the `Writable` interface and an implementation of the `Writable` called `StockPriceWritable`, which represents the stock data. The `DBInputFormat` class requires a bean representation of the table being imported, which implements both the `Writable` and `DBWritable` interfaces. Because you'll also need to implement the `DBWritable` interface, you'll do so by extending the `StockPriceWritable` class, as shown in the following code.

HBase

Our final foray into the area of moving data into Hadoop is a look at HBase. HBase is a real-time, column-oriented database, and is often either co-located on the same hardware that serves as your Hadoop cluster, or is in close proximity to a Hadoop cluster. Being able to work with HBase data directly in MapReduce, or push it into HDFS, is one of the huge advantages when picking HBase as a solution.

Two techniques in this section, the first focusing on how to import HBase data into HDFS, and the second on how to use HBase as a data source for a MapReduce job.

HBase ingress into HDFS

What if you had some customer data sitting in HBase that you wanted to leverage in MapReduce in conjunction with data in HDFS? You could write a MapReduce job which takes as input the HDFS dataset and pulls data directly from HBase in your map or reduce code. But in some cases it may be more useful to take a dump of the data in HBase into HDFS directly, especially if you plan to utilize that data in multiple MapReduce jobs and the HBase data is immutable, or changes infrequently.

How do you get HBase data into HDFS?

HBase includes an `Export` class that can be used to import HBase data into HDFS in SequenceFile format. This technique also walks through code that can be used to read the imported HBase data.

```
$ hbase shell



hbase(main):012:0> list

stocks_example

1 row(s) in 0.0100 seconds



hbase(main):007:0> scan 'stocks_example'

ROW              COLUMN+CELL

AAPL2000-01-03   column=details:stockAvro,
timestamp=1322315975123,...

AAPL2001-01-02   column=details:stockAvro, timestamp=1322315975123,
```

Moving data out of Hadoop

After data has been brought into Hadoop it will likely be joined with other datasets to produce some results. At this point either that result data will stay in HDFS for future access, or it will be pushed out of Hadoop. An example of this scenario would be one where you pulled some data from an OLTP database, performed some machine learning activities on that data, and then copied the results back into the OLTP database for use by your production systems.

In this section we'll cover how to automate moving regular files from HDFS to a local filesystem. We'll also look at data egress to relational databases and HBase. To start off we'll look at how to copy data out of Hadoop using the HDFS Slurper.

Egress to a local filesystem

we looked at two mechanisms to move semistructured and binary data into HDFS, the HDFS File Slurper open source project, and Oozie to trigger a data ingress workflow. The challenge to using a local filesystem for egress (and ingress for that matter) is that map and reduce tasks running on clusters won't have access to the filesystem on a specific server. You need to leverage one of the following three broad options for moving data from HDFS to a filesystem:

1. Host a proxy tier on a server, such as a web server, which you would then write to using MapReduce.
2. Write to the local filesystem in MapReduce and then as a postprocessing step trigger a script on the remote server to move that data.
3. Run a process on the remote server to pull data from HDFS directly.

The third option is the preferred approach because it's the simplest and most efficient, and as such is the focus of this section. We'll look at how you can use the HDFS Slurper to automatically move files from HDFS out to a local filesystem.

Databases

Databases are usually the target of Hadoop data egress in one of two circumstances: either when you move data back into production databases to be used by production systems, or when you move data into OLAP databases to perform business intelligence and analytics functions.

Using Sqoop to export data to MySQL

we'll use Apache Sqoop to export data from Hadoop to a MySQL database. Sqoop is a tool that simplifies database imports and exports. Sqoop is covered in detail in technique 5.

We'll walk through the process of exporting data from HDFS to Sqoop. We'll also cover methods using the regular connector, as well as how to do bulk imports using the fast connector.

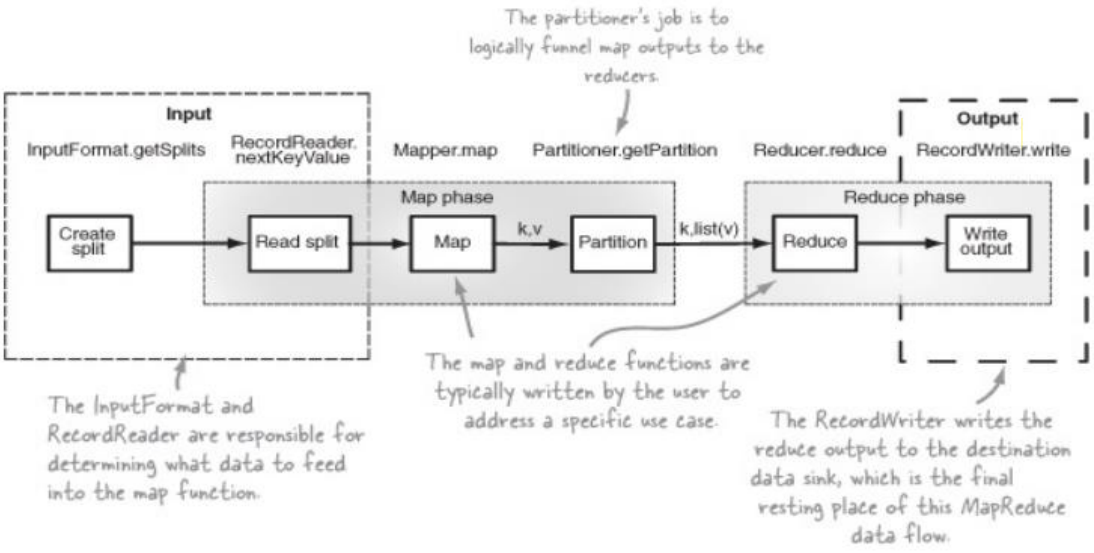
Using Sqoop to export data to MySQL

Hadoop excels at performing operations at scales which defeat most relational databases, so it's common to extract OLTP data into HDFS, perform some analysis, and then export it back out to a database.

3.Understanding inputs and outputs of Map reduce

Your data might be XML files sitting behind a number of FTP servers, text log files sitting on a central web server, or Lucene indexes[1] in HDFS. How does MapReduce support reading and writing to these different serialization structures across the various storage mechanisms? You'll need to know the answer in order to support a specific serialization format.

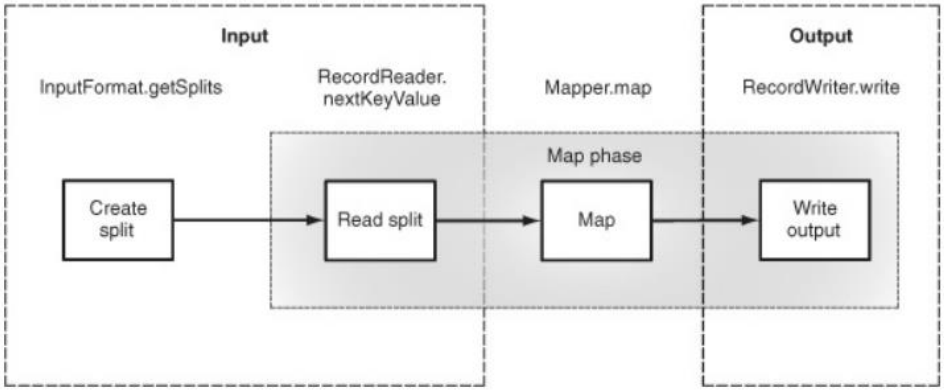
Figure 3.1. High-level input and output actors in MapReduce



This diagram shows the high-level data flows through MapReduce and identifies the actors responsible for various parts of the flow. On the input side you see that some work (*Create split*) is performed outside of the map phase, and other

work is performed as part of the map phase (*Read split*). All of the output work is performed in the reduce phase (*Write output*).

Figure 3.2. Input and output actors in MapReduce with no reducers



The above diagram shows the same flow with a map-only job. In a map-only job the MapReduce framework still uses the `OutputFormat` and `RecordWriter` classes to write the outputs directly to the data sink.

Let's walk through the data flow and describe the responsibilities of the various actors. As we do this, we'll also look at the relevant code from the built-in `TextInput-Format` and `TextOutputFormat` classes to better understand the concepts. The `Text-InputFormat` and `TextOutputFormat` classes read and write line-oriented text files.

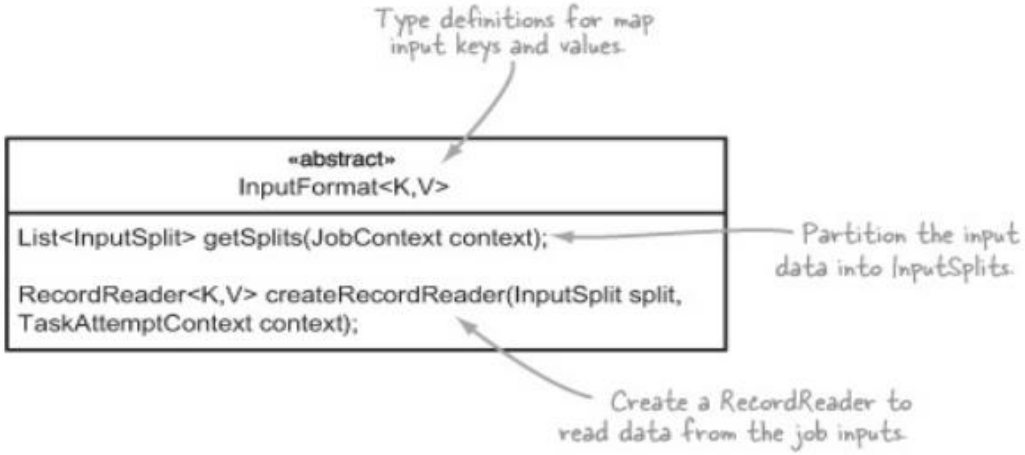
Data input

The two classes that support data input in MapReduce are `InputFormat` and `Record-Reader`. The `InputFormat` class is consulted to determine how the input data should be partitioned for the map tasks, and the `RecordReader` performs the reading of data from the inputs.

Inputformat

Every job in MapReduce must define its inputs according to contracts specified in the `InputFormat` abstract class. `InputFormat` implementers must fulfill three contracts: first, they describe type information for map input keys and values; next, they specify how the input data should be partitioned; and finally, they indicate the `RecordReader` instance that should read the data from source.

The below diagram shows the `InputFormat` class and how these three contracts are defined.

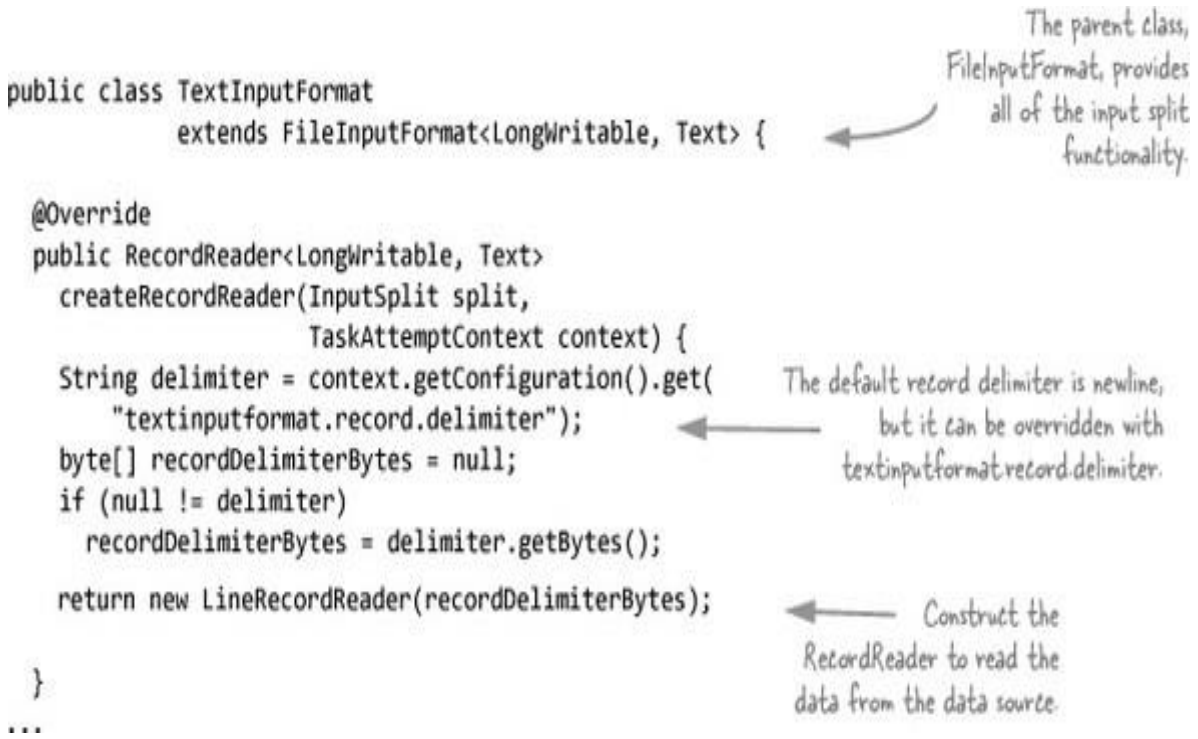


Arguably the most crucial contract is that of determining how to divide the input data. In MapReduce nomenclature these divisions are referred to as *input splits*. The input splits directly impact the map parallelism because each split is processed by a single map task.

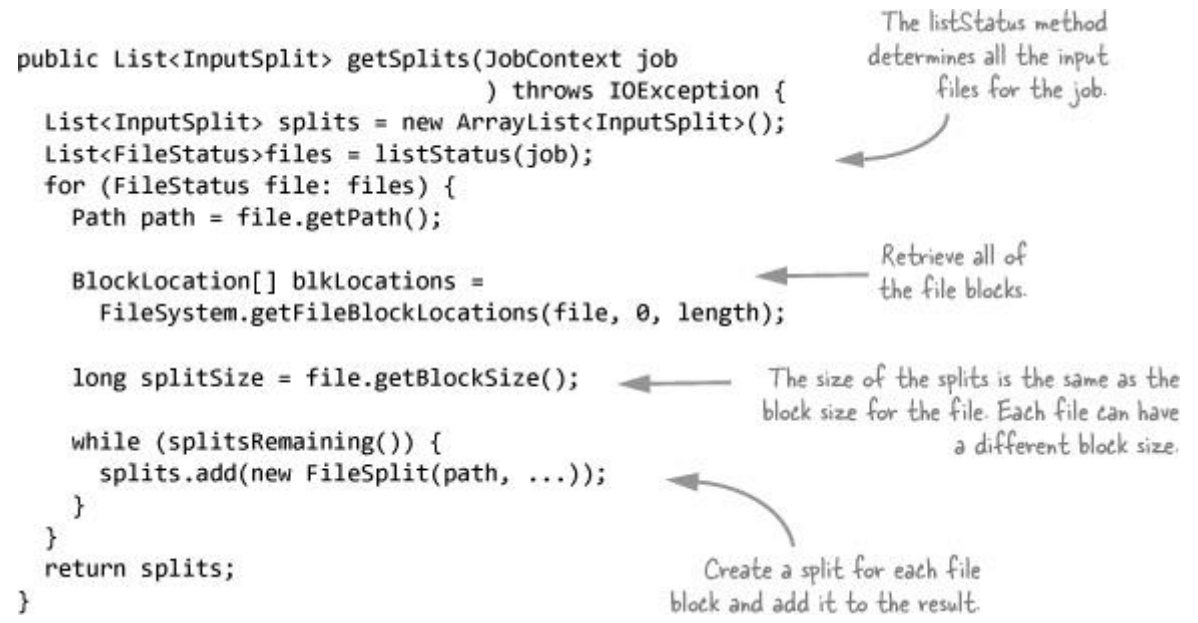
Working with an `InputFormat` that is unable to create multiple input splits over a single data source (such as a file) will result in a slow map phase because the file will be processed sequentially.

The `TextInputFormat` class provides an implementation of the `InputFormat` class's `createRecordReader` method but delegates the calculation of input splits to its parent class, `FileInputFormat`.

The following code shows the relevant parts of the `TextInputFormat` class:



The code in `FileInputFormat` to determine the input splits is a little more complicated. A simplified form of the code is shown in the following example to portray the main elements of this method:



The following code is an example of how you specify the `InputFormat` to use for a MapReduce job:

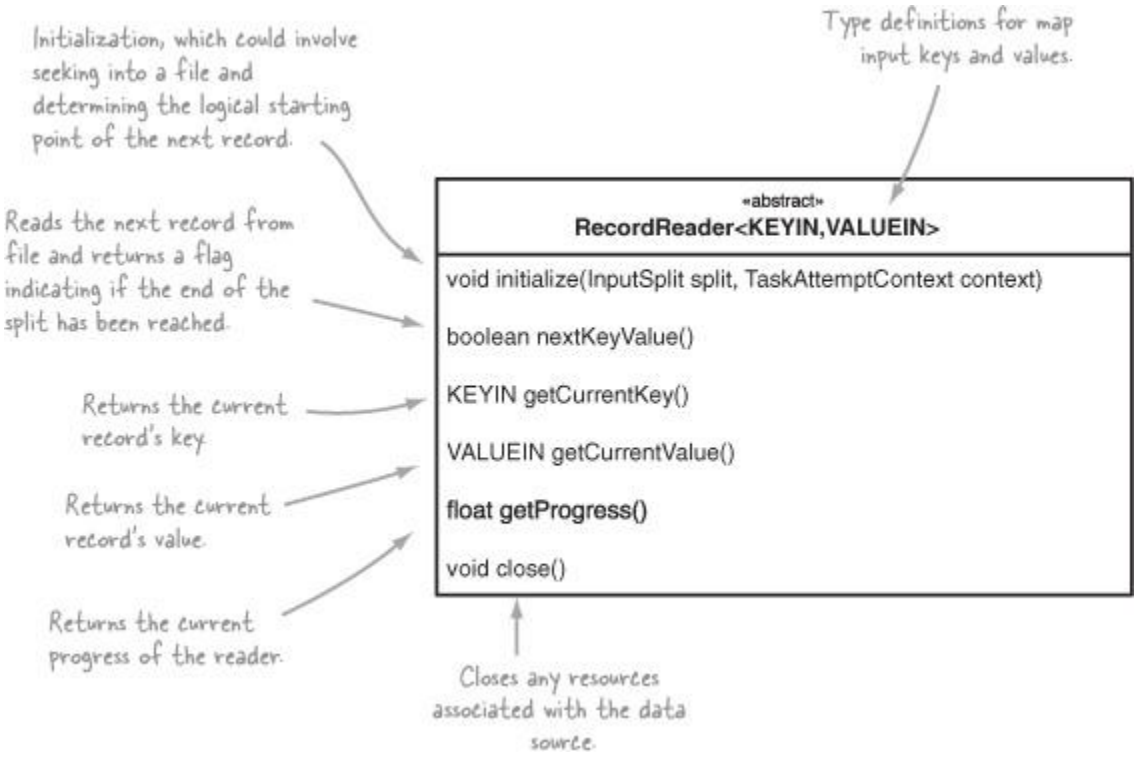
```
job.setInputFormatClass(TextInputFormat.class);
```

Recordreader

The `RecordReader` class is used by MapReduce in the map tasks to read data from an input split and provide each record in the form of a key/value pair for use by mappers.

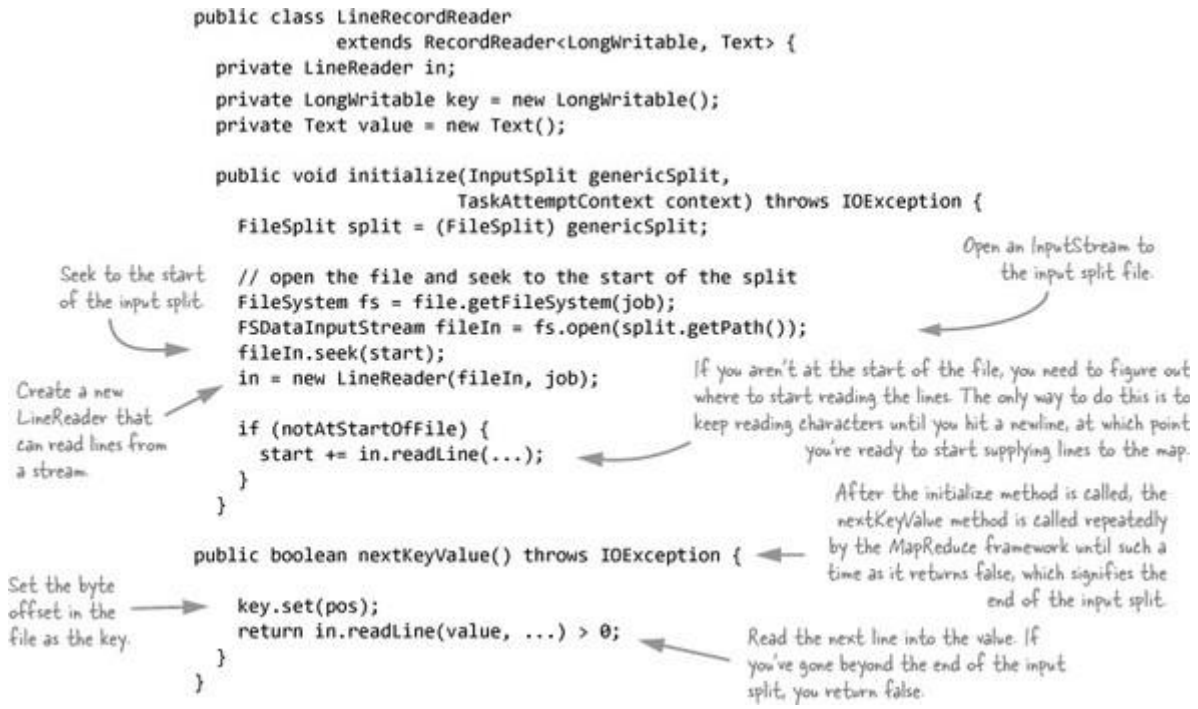
A task is commonly created for each input split, and each task has a single `RecordReader` that's responsible for reading the data for that input split. Figure 3.4 shows the abstract methods you must implement.

Figure 3.4. The annotated RecordReader class and its abstract methods



As shown in a previous section, the `TextInputFormat` class created a `LineRecordReader` to read records from the input splits.
The `LineRecordReader` directly extends the `RecordReader` class and leverages the `LineReader` class to read lines from the input split.
The `LineRecordReader` uses the byte offset in the file for the map key and the contents of the line for the map value.

I have included a simplified version of the `LineRecordReader` in the following example



Because the `LineReader` class is easy, we'll skip that code. The next step will be a look at how MapReduce supports data outputs.

Data output

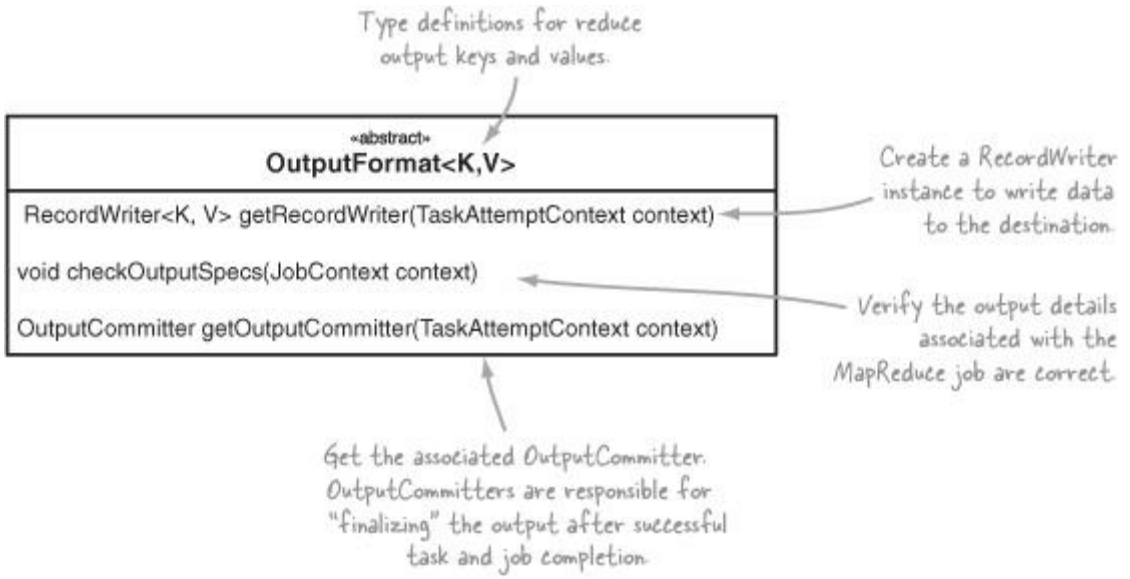
MapReduce uses a similar process for supporting output data as it does for input data.

Two classes must exist, an `OutputFormat` and a `RecordWriter`. The `OutputFormat` performs some basic validation of the data sink properties, and the `RecordWriter` writes each reducer output to the data sink.

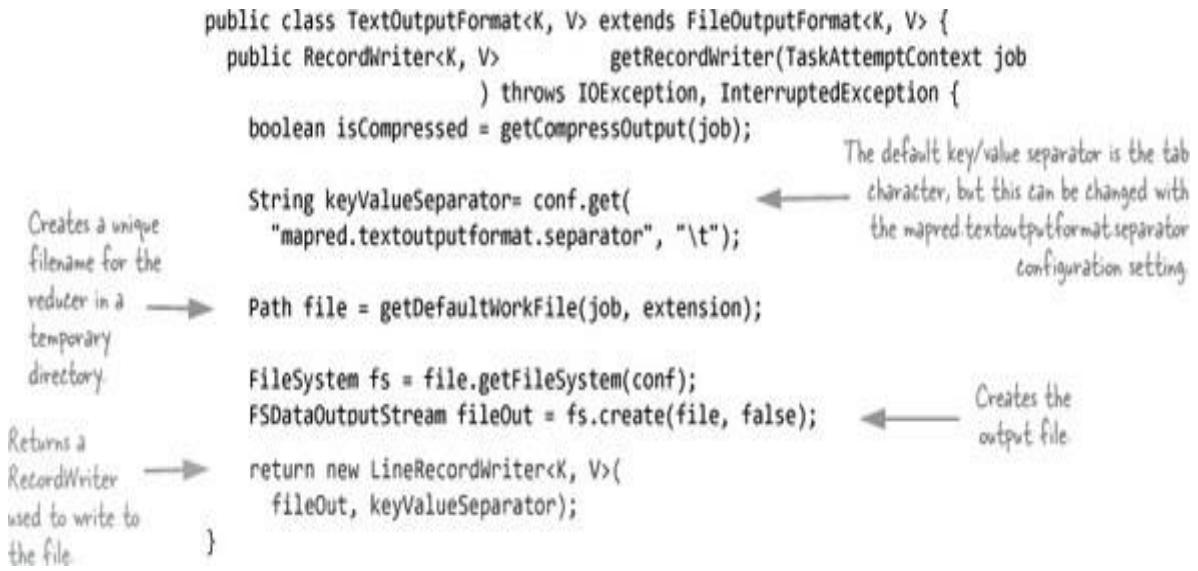
Outputformat

Much like the `InputFormat` class, the `OutputFormat` class, defines the contracts that implementers must fulfill, including checking the information related to the job output, providing a `RecordWriter`, and specifying an output committer, which allows writes to be staged and then made "permanent" upon task and/or job success.

Figure 3.5. The annotated OutputFormat class



Just like the `TextInputFormat`, the `TextOutputFormat` also extends a base class, `File-OutputFormat`, which takes care of some complicated logistics such as output committing, For now let's take a look at the work the `TextOutputFormat` is performing.



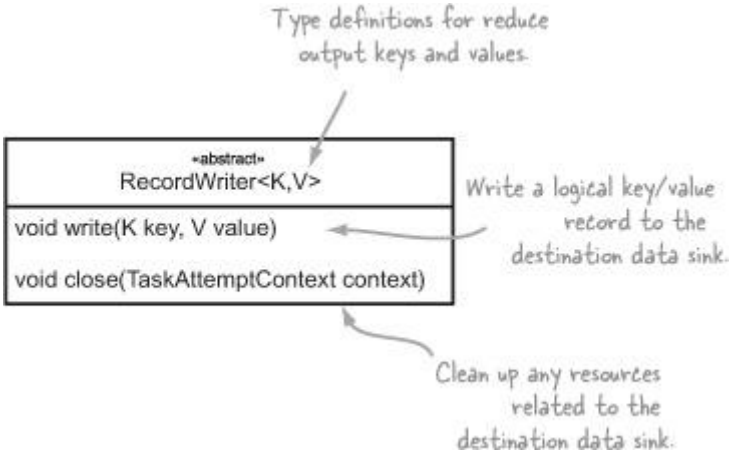
The following code is an example of how you specify the `OutputFormat` that should be used for a MapReduce job:

```
job.setOutputFormatClass(TextOutputFormat.class);
```

Recordwriter

You'll use the `RecordWriter` to write the reducer outputs to the destination data sink. It's a simple class, as figure 3.6 illustrates.

Figure The annotated RecordWriter class overview



The `TextOutputFormat` returned a `LineRecordWriter` object (`LineRecordWriter` is an inner class of `TextOutputFormat`) to perform the writing to file. A simplified version of that class (source at http://goo.gl/8ab7Z) is shown in the following example:

```
protected static class LineRecordWriter<K, V>   extends RecordWriter<K, V> {

  protected DataOutputStream out;

  public synchronized void write(K key, V value)
    throws IOException {

    writeObject(key);
    out.write(keyValueSeparator);
    writeObject(value);
    out.write(newline);
  }

  private void writeObject(Object o) throws IOException {
    out.write(o);
  }
```

Write out the key, separator, value, and newline.

Write out the Object to the output stream.

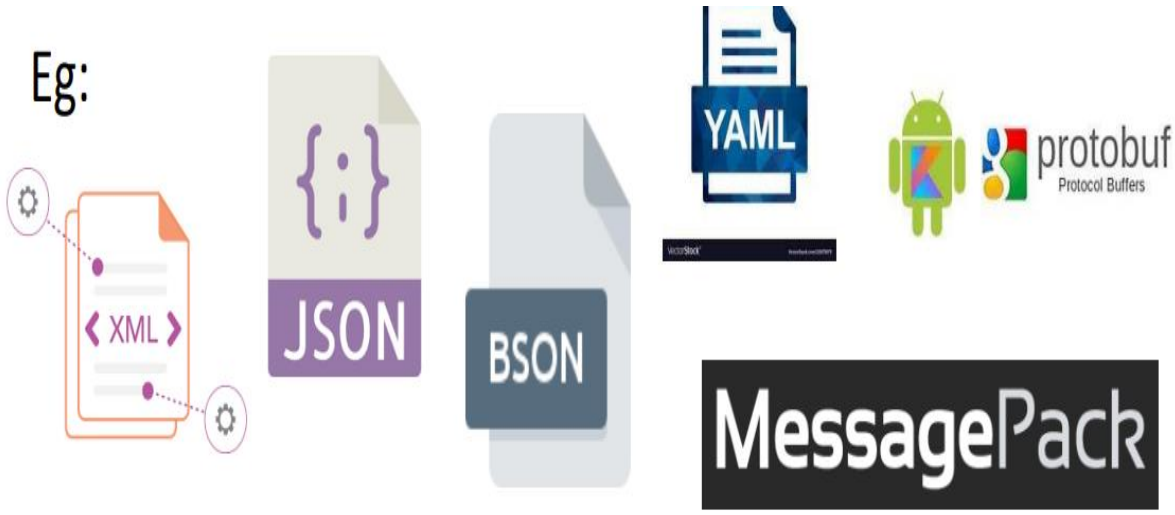
While on the map side it's the `InputFormat` that determines how many map tasks are executed, on the reducer side the number of tasks is solely based on

the value for `mapred.reduce.tasks` set by the client (or if it isn't set, the value is picked up from mapred-site.xml, or mapred-default.xml if it doesn't exist in the site file).

Now that you know what's involved in working with input and output data in MapReduce, it's time to apply that knowledge to solving some common data serialization problems. Your first step in this data serialization journey is to learn how to work with file formats such as XML.

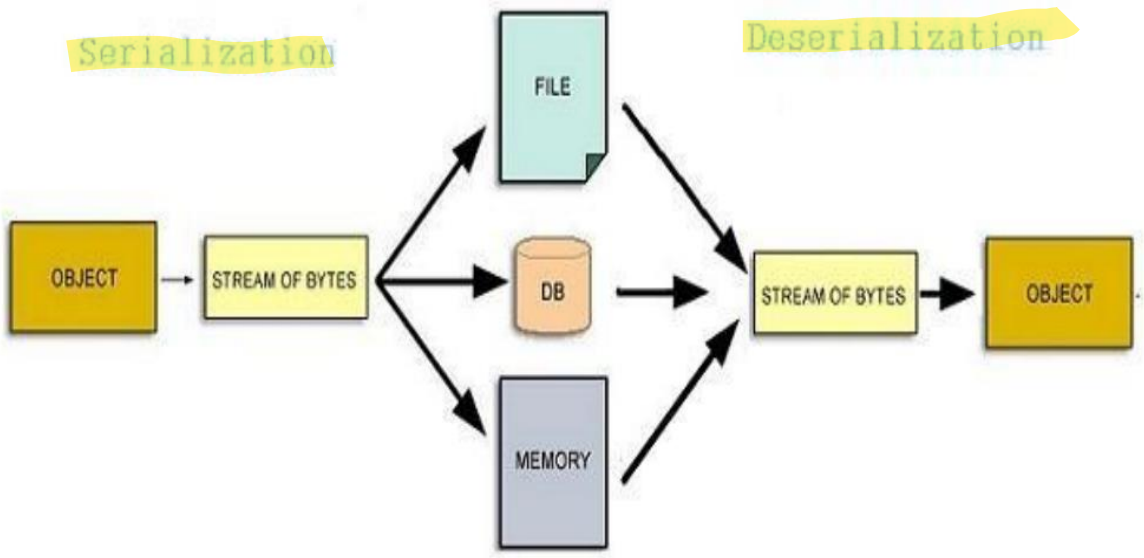
Data Serialization

Data serialization is the process of converting data objects present in complex data structures into a byte stream for storage,transfer and distribution purposes on physical devices. Once the serialized data is transmitted the reverse process of creating objects from the byte sequence called deserialization.



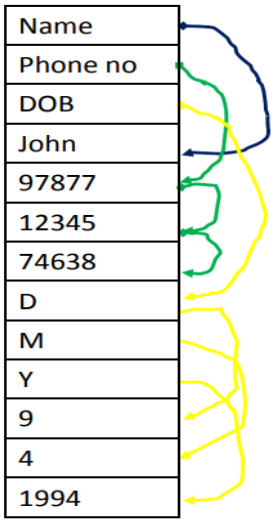
Computer data is generally organized in data structures such as arrays, tables, trees, classes. When data structures need to be stored or transmitted to another location, such as across a network, they are serialized.

Serialization becomes complex for nested data structures and object references.



COMPLICATED STRUCTURE

Name: John
Phone no: 97877
12345
74638
DOB: D: 9
M: 4
Y: 1994



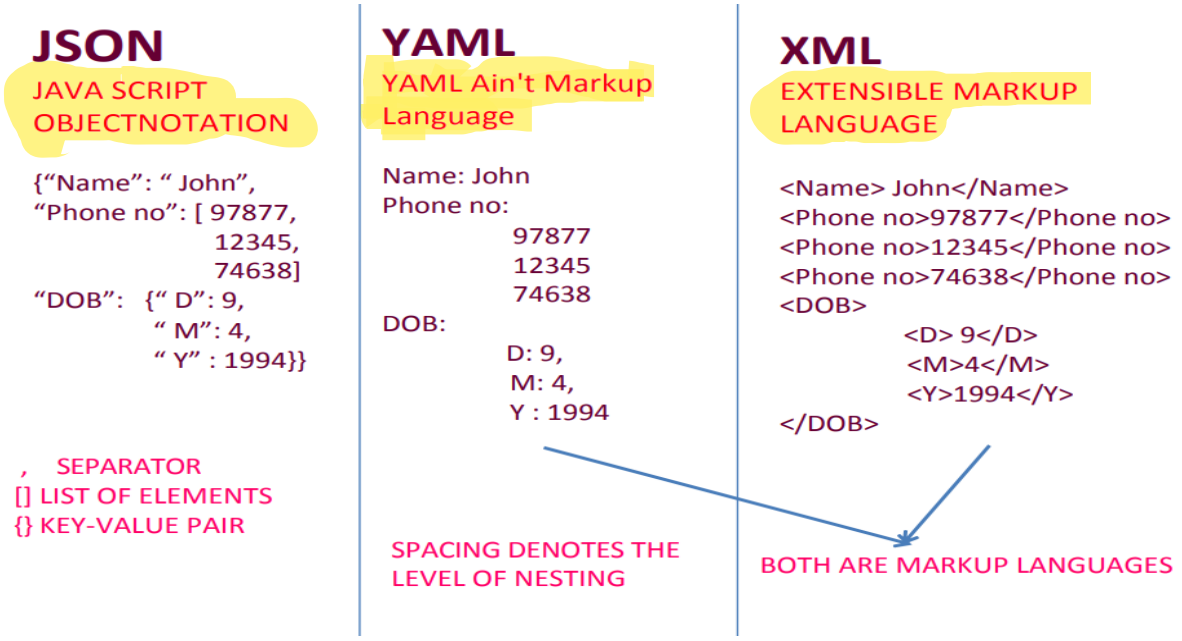
SERIALIZED FORMAT

{“Name”: “ John”,
“Phone no”: [97877,
12345,
74638]
“DOB”: {“ D”: 9,
“ M”: 4,
“ Y” : 1994}}

JSON

IN ONE LINE

{“Name”: “John”,“Phone no”: [97877,12345,74638”],“DOB”: {“ D”: 9,“ M”: 4,“ Y” : 1994}}



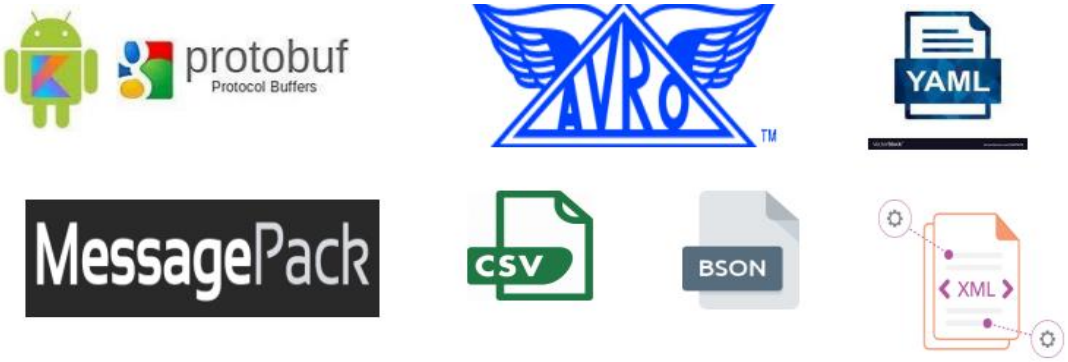
APPLICATIONS OF DATA SERIALIZATION

- Serialization allows a program to save the state of an object and recreate it when needed.
- Persisting data onto files – happens mostly in language-neutral formats such as CSV or XML. However, most languages allow objects to be serialized directly into binary using APIs
- Storing data into Databases – when program objects are converted into byte streams and then stored into DBs, such as in Java JDBC.
- Transferring data through the network – such as web applications and mobile apps passing on objects from clientto server and vice versa.
- Sharing data in a Distributed Object Model – When programs written in different languages need to share object data over a distributed network .However, SOAP, REST and other web services have replaced these applications now.

POTENTIAL RISK DUE TO SERIALIZATION

- It may allow a malicious party with access to the serialization byte stream to read private data, create objects with illegal or dangerous state, or obtain references to the private fields of deserialized objects. Workarounds are tedious, not guaranteed.

- Open formats too have their security issues.
- XML might be tampered using external entities like macrosor unverified schema files.
- JSON data is vulnerable to attack when directly passed to a JavaScript engine due to features like JSONP requests.



DATA SERIALIZATION FORMATS

Serialization Formats in Hadoop

- XML
- CSV
- YAML
- JSON
- BSON
- MessagePack
- Thrift
- Protocol buffers
- Avro

TEXT-BASED DATA SERIALIZATION FORMATS

XML (Extensible Markup Language) :

- Nested textual format. Human-readable and editable.
- Schema based validation.
- Used in metadata applications, web services data transfer, web publishing.

CSV (Comma-Separated Values) :

- Table structure with delimiters.
- Human-readable textual data.
- Opens as spreadsheet or plaintext.
- CSV file is the most commonly used data file format.
- Easy to read, Easy to parse, Easy to export data from an RDBMS table.

It has three major drawbacks when used for HDFS.

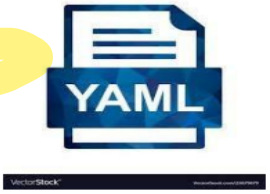
1. All lines in a CSV file is a record, therefore, we should not include any headers or footers. In other word, CSV file cannot be stored in HDFS with any meta data.
2. CSV file has very limited support for schema evolution. Because the fields for each record are ordered, we are not able to change the orders. We can only append new fields to the end of each line.
3. It does not support block compression which many other file formats support. The whole file has to be compressed and decompressed for reading, adding a significant read performance cost to the files.

JSON (JavaScript Object Notation) :

- Short syntax textual format with limited data types.
- Human-readable. Derived from JavaScript data formats.
- No need of a separate parser (like XML) since they map to JavaScript objects. No direct support for DATE data type. All data is dynamically processed
- It is in text format that stores meta data with the data, so it fully supports schema evolution and also spiltable.
- We can easily add or remove attributes for each datum. However, because it's text file, it doesn't support block compression.

YAML Ain't Markup Language :

- It is a data serialization language which is designed to be human -friendly and works well with other programming languages for everyday tasks.
- Superset of JSON
- Supports complex data types. Maps easily to native data structures.



Binary JSON:

- It is a binary-encoded serialization of JSON-like documents.
- MongoDB uses BSON ,when storing documents in collections
- It deals with attribute-value pairs like JSON. Includes datetime, bytearray and other data types not present in JSON.

- It is designed for data to be transparently converted from/to JSON.
- Support rich set of data structures
- It create schema based annotation
- Primary use is network communication

•It is Created by Google
•It is Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data
•Protocol buffers currently support generated code in Java, Python, Objective-C, and C++.

MessagePack

AVRO

- Apache **Avro** is a language-neutral data
- serialization system, developed by **Doug Cutting.** the father of Hadoop.
- It also called a schema-based serialization technique.

FEATURES

- Avro uses JSON format to declare the data structures. Presently, it supports languages such as Java, C, C++, C#, Python, and Ruby.
- Avro creates binary structured format that is both **compressible** and **splitable**. Hence it can be efficiently used as the input to Hadoop MapReduce jobs.
- Avro provides **rich data structures**.

- Avro **schemas** defined in **JSON**, facilitate implementation in the languages that already have JSON libraries.
- Avro creates a self-describing file named *Avro Data File*, in which it stores data along with its schema in the metadata section.
- Avro is also used in Remote Procedure Calls (RPCs).

Thrift and **Protocol Buffers** are the most competent libraries with Avro. Avro differs from these frameworks in the following ways

- Avro supports both dynamic and static types as per the requirement. Protocol Buffers and Thrift use Interface Definition Languages (IDLs) to specify schemas and their types. These IDLs are used to generate code for serialization and deserialization.
- Avro is built in the Hadoop ecosystem. Thrift and Protocol Buffers are not built in Hadoop ecosystem.

PERFORMANCE CHARACTERISTICS

- Speed – Binary formats are faster than textual formats. A late entrant, **protobuf reports the best times**. JSON is preferable due to readability and being schema-less.
- Data size – This refers to the physical space in bytes post serialization. For small data, compressed JSON data occupies more space compared to binary formats like protobuf. Generally, **binary formats always occupy less space**.
- Usability – Human readable formats like JSON are naturally preferred over binary formats. For editing data, YAML is good. Schema definition is easy in protobuf, with in-built tools.
- Compatibility-Extensibility – JSON is a closed format. XML is average with schema versioning. Backward compatibility (extending schemas) is best handled by protobuf.