## 1. Introduction

The **first problem** to solve is hardware failure. As soon as you start using many pieces of hardware, the chance that one will fail is fairly high. A common way of avoiding data loss is through replication, redundant copies. Hadoop Distributed Filesystem (**HDFS**).

The **second problem** is that most analysis tasks need to be able to combine the data in some way, and data read from one disk may need to be combined with data from any of the other 99 disks. **MapReduce** provides a programming model that abstracts the problem from disk reads and writes, transforming it into a computation over sets of keys and values. HDFS, MapReduce has built-in reliability.

For all its strengths, **MapReduce** is fundamentally a batch processing system, and is not suitable for interactive analysis. You can’t run a query and get results back in a few seconds or less. Queries typically take minutes or more, so it’s best for offline use, where there isn’t a human sitting in the processing loop waiting for results.

The first component to provide online access was HBase, a key-value store that uses HDFS for its underlying storage. **HBase** provides both online read/write access of individual rows and batch operations for reading and writing data in bulk, making it a good solution for building applications on.

The real enabler for new processing models in Hadoop was the introduction of **YARN** (which stands for Yet Another Resource Negotiator) in Hadoop 2. YARN is a cluster resource management system, which allows any distributed program (not just MapReduce) to run on data in a Hadoop cluster.

**Why can’t we use databases with lots of disks to do large-scale analysis? Why Hadoop?**

* The answer to these questions comes from another trend in disk drives: seek time is improving more slowly than transfer rate. **Seeking** is the process of moving the disk’s head to a particular place on the disk to read or write data. It characterizes the latency of a disk operation, whereas the transfer rate corresponds to a disk’s bandwidth.
* If the data access pattern is dominated by seeks, it will take longer to read or write large portions of the dataset than streaming through it, which operates at the transfer rate. On the other hand, for updating a small proportion of records in a database, a traditional BTree (the data structure used in relational databases, which is limited by the rate at which it can perform seeks) works well. For updating the majority of a database, a B-Tree is less efficient than MapReduce, which uses Sort/Merge to rebuild the database.
* MapReduce is a good fit for problems that need to analyze the whole dataset in a batch fashion, particularly for ad hoc analysis. An RDBMS is good for point queries or updates, where the dataset has been indexed to deliver low-latency retrieval and update times of a relatively small amount of data. MapReduce suits applications where the data is written once and read many times, whereas a relational database is good for datasets that are continually updated.
* However, the differences between relational databases and Hadoop systems are blurring. Relational databases have started incorporating some of the ideas from Hadoop, and from the other direction, Hadoop systems such as **Hive** are becoming more interactive (by moving away from MapReduce) and adding features like indexes and transactions that make them look more and more like traditional RDBMSs.
* Structured data is organized into entities that have a defined format, such as XML documents or database tables that conform to a particular predefined schema. This is the realm of the RDBMS. Hadoop works well on unstructured or semi-structured data because it is designed to interpret the data at processing time (so called schema-on-read). Ex: Web server log.
* Relational data is often normalized to retain its integrity and remove redundancy. Normalization poses problems for Hadoop processing because it makes reading a record a nonlocal operation, and one of the central assumptions that Hadoop makes is that it is possible to perform (high-speed) streaming reads and writes.
* MapReduce — and the other processing models in Hadoop — scales linearly with the size of the data. Data is partitioned, and the functional primitives (like map and reduce) can work in parallel on separate partitions. This means that if you double the size of the input data, a job will run twice as slowly. But if you also double the size of the cluster, a job will run as fast as the original one. This is not generally true of SQL queries.

The high-performance computing (HPC) and grid computing communities have been doing large-scale data processing for years. This works well for predominantly compute-intensive jobs, but it becomes a problem when nodes need to access larger data volumes (hundreds of gigabytes, the point at which Hadoop really starts to shine), since the network bandwidth is the bottleneck and compute nodes become idle. Hadoop tries to co-locate the data with the compute nodes, so data access is fast because it is local. This feature, known as **data locality**, is at the heart of data processing in Hadoop and is the reason for its good performance.

Coordinating the processes in a large-scale distributed computation is a challenge. The hardest aspect is gracefully handling partial failure — when you don’t know whether or not a remote process has failed — and still making progress with the overall computation. Distributed processing frameworks like MapReduce spare the programmer from having to think about failure, since the implementation detects failed tasks and reschedules replacements on machines that are healthy. MapReduce is able to do this because it is a **shared-nothing** architecture, meaning that tasks have no dependence on one other. So from the programmer’s point of view, the order in which the tasks run doesn’t matter.

In one well-publicized feat, the New York Times used Amazon’s EC2 compute cloud to crunch through 4 terabytes of scanned archives from the paper, converting them to PDFs for the Web. The processing took less than 24 hours to run using 100 machines, and the project probably wouldn’t have been embarked upon without the combination of Amazon’s pay-by-the-hour model (which allowed the NYT to access a large number of machines for a short period) and Hadoop’s easy-to-use parallel programming model.

In April 2008, Hadoop broke a world record to become the fastest system to sort an entire terabyte of data. Running on a 910-node cluster, Hadoop sorted 1 terabyte in 209 seconds (just under 3.5 minutes), beating the previous year’s winner of 297 seconds. In November of the same year, Google reported that its MapReduce implementation sorted 1 terabyte in 68 seconds.[17] Then, in April 2009, it was announced that a team at Yahoo! had used Hadoop to sort 1 terabyte in 62 seconds.

## 2. MapReduce

To speed up the processing, we need to run parts of the program in parallel.

First, dividing the work into equal-size pieces isn’t always easy or obvious. In this case, the file size for different years varies widely, so some processes will finish much earlier than others. Even if they pick up further work, the whole run is dominated by the longest file. A better approach, although one that requires more work, is to split the input into fixed-size chunks and assign each chunk to a process.

Second, combining the results from independent processes may require further processing.

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Each phase has key-value pairs as input and output, the types of which may be chosen by the programmer. The programmer also specifies two functions: the map function and the reduce function.

The Mapper class is a generic type, with four formal type parameters that specify the input key, input value, output key, and output value types of the map function.

public class MaxTemperature**Mapper** extends Mapper<LongWritable, Text, Text, IntWritable>

The map() method also provides an instance of Context to write the output to.

public void **map**(LongWritable key, Text **value**, Context context) throws IOException, InterruptedException;

Again, four formal type parameters are used to specify the input and output types, this time for the reduce function.

public class MaxTemperature**Reducer** extends Reducer<Text, IntWritable, Text, IntWritable> {

The input types of the reduce function must match the output types of the map function:

public void **reduce**(Text key, Iterable<IntWritable> **values**, Context context) throws IOException, InterruptedException

### Data Flow

A **MapReduce job** is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks. The tasks are scheduled using YARN and run on nodes in the cluster. If a task fails, it will be automatically rescheduled to run on a different node.

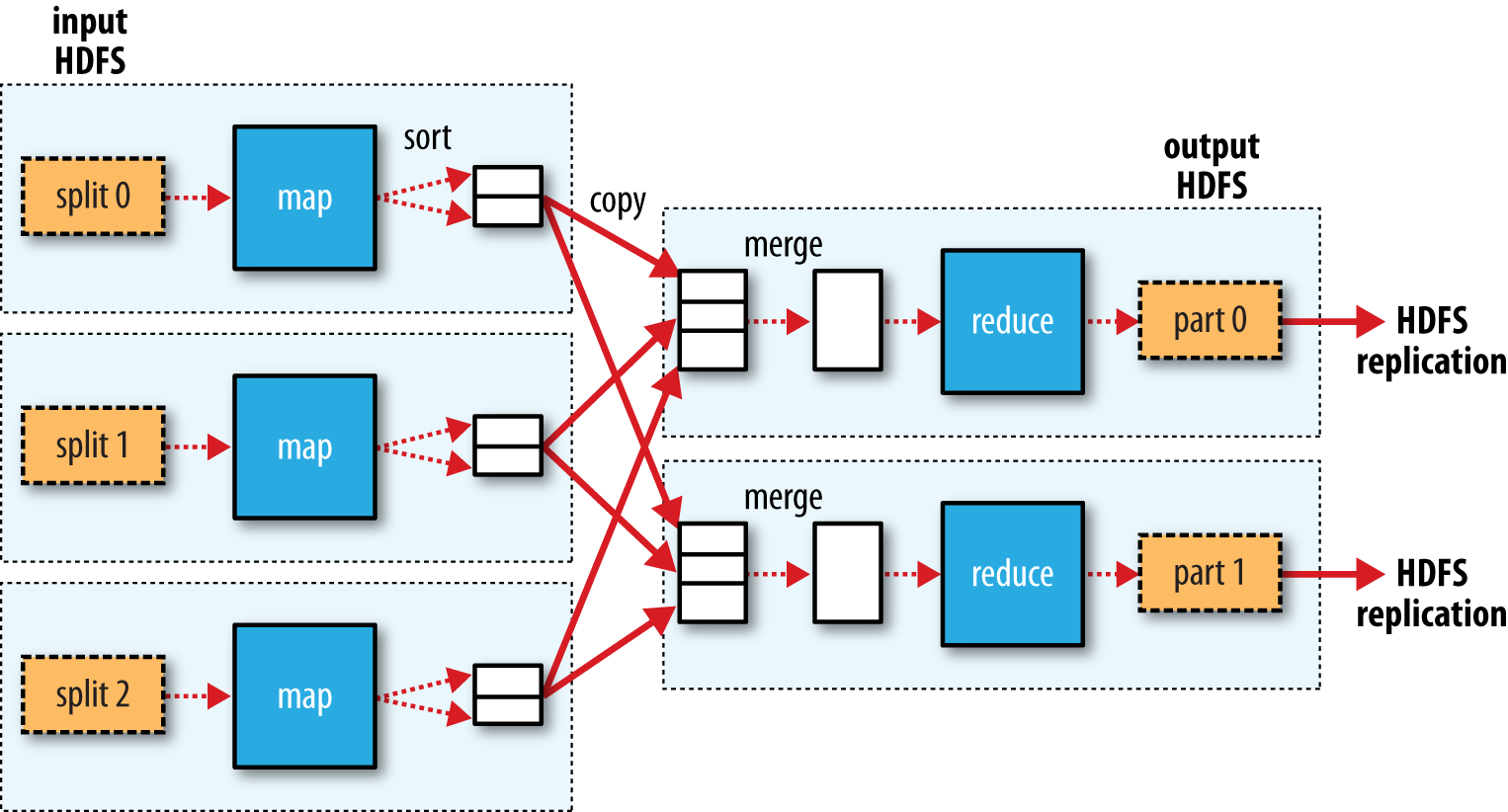
Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user-defined map function for each record in the split.

Hadoop does its best to run the map task on a node where the input data resides in HDFS, because it doesn’t use valuable cluster bandwidth. This is called the **data locality** optimization.

Map tasks write their output to the local disk, not to HDFS. Why is this? Map output is intermediate output: it’s processed by reduce tasks to produce the final output, and once the job is complete, the map output can be thrown away.

Reduce tasks don’t have the advantage of data locality; the input to a single reduce task is normally the output from all mappers. Map outputs must be transferred across the network to the node where the reduce task is running.

The number of reduce tasks is not governed by the size of the input, but instead is specified independently. When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as “the shuffle,” as each reduce task is fed by many map tasks.



Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a **combiner function** to be run on the map output, and the combiner function’s output forms the input to the reduce function.

More succinctly, we may express the function calls on the temperature values in this case as follows: Map [➜](https://www.alt-codes.net/arrow_alt_codes.php) Combiner [➜](https://www.alt-codes.net/arrow_alt_codes.php) Reducer

max(0, 20, 10, 25, 15) [➜](https://www.alt-codes.net/arrow_alt_codes.php) max(max(0, 20, 10), max(25, 15)) [➜](https://www.alt-codes.net/arrow_alt_codes.php) max(20, 25) = 25

However, not all functions possess this property. the combiner function is defined using the Reducer class.

**Hadoop Streaming** uses Unix standard streams as the interface between Hadoop and your program, so you can use any language that can read standard input and write to standard output to write your MapReduce program.

It’s worth drawing out a design difference between Streaming and the Java MapReduce API. The Java API is geared toward processing your map function one record at a time. The framework calls the map() method on your Mapper for each record in the input, whereas with Streaming the map program can decide how to process the input.

We can now simulate the whole MapReduce pipeline with a Unix pipeline with ruby scripts:

**% cat sample.txt | max\_temp\_map.rb | sort | max\_temp\_reduce.rb**

**1949 111**

**1950 22**

**Sample.txt**

1950 +0000

1950 +0022

1950 -0011

1949 +0111

1949 +0078

**max\_temp\_map.rb**

#!/usr/bin/env ruby

STDIN.each\_line do |line|

val = line

year, temp, q = val[15,4], val[87,5], val[92,1]

puts "#{year}\t#{temp}" if (temp != "+9999" && q =~ /[01459]/)

End

**max\_temp\_reduce.rb**

#!/usr/bin/env ruby

last\_key, max\_val = nil, -1000000

STDIN.each\_line do |line|

key, val = line.split("\t")

if last\_key && last\_key != key

puts "#{last\_key}\t#{max\_val}"

last\_key, max\_val = key, val.to\_i

else

last\_key, max\_val = key, [max\_val, val.to\_i].max

end

end

puts "#{last\_key}\t#{max\_val}" if last\_key

## 3. The Hadoop Distributed Filesystem (HDFS)

When a dataset outgrows the storage capacity of a single physical machine, it becomes necessary to partition it across a number of separate machines. Filesystems that manage the storage across a network of machines are called distributed filesystems. Since they are network based, all the complications of network programming kick in, thus making distributed filesystems more complex than regular disk filesystems. For example, one of the biggest challenges is **making the filesystem tolerate node failure without suffering data loss**.

**HDFS** is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware. Hardware failure is the norm rather than the exception.

Disk blocks are normally 512 bytes. HDFS, too, has the concept of a **block**, but it is a much larger unit—128 MB by default. Like in a filesystem for a single disk, files in HDFS are broken into block-sized chunks, which are stored as independent units. Unlike a filesystem for a single disk, a file in HDFS that is smaller than a single block does not occupy a full block’s worth of underlying storage. (For example, a 1 MB file stored with a block size of 128 MB uses 1 MB of disk space, not 128 MB.) When unqualified, the term “block” refers to a block in HDFS.

Time it takes to transfer the data from the disk can be significantly longer than the time to seek to the start of the block. A quick calculation shows that if the seek time is around 10 ms and the transfer rate is 100 MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100 MB. The default is actually 128 MB, although many HDFS installations use larger block sizes.

**Namenodes and Datanodes**

An HDFS cluster has two types of nodes operating in a master−worker pattern: a namenode (the master) and a number of datanodes (workers). The NameNode and DataNode are pieces of software designed to run on commodity machines. The **namenode** manages the filesystem namespace. It maintains the filesystem tree and the metadata for all the files and directories in the tree. This information is stored persistently on the local disk in the form of two files: the namespace image and the edit log. The namenode also knows the datanodes on which all the blocks for a given file are located; however, it does not store block locations persistently, because this information is reconstructed from datanodes when the system starts.

**Datanodes** are the workhorses of the filesystem. They store and retrieve blocks when they are told to (by clients or the namenode), and they report back to the namenode periodically with lists of blocks that they are storing.

Without the namenode, the filesystem cannot be used. It is important to make the namenode **resilient to failure**. Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems. It is also possible to run a **secondary namenode**, which despite its name does not act as a namenode. Its main role is to periodically merge the namespace image with the edit log to prevent the edit log from becoming too large.

Normally a datanode reads blocks from disk, but for frequently accessed files the blocks may be explicitly cached in the datanode’s memory, in an off-heap **block cache**. A small lookup table is a good candidate for caching.

The namenode keeps a reference to every file and block in the filesystem in memory, which means that on very large clusters with many files, memory becomes the limiting factor for scaling. **HDFS federation**, introduced in the 2.x release series, allows a cluster to scale by adding namenodes, each of which manages a portion of the filesystem namespace. For example, one namenode might manage all the files rooted under /user, say, and a second namenode might handle files under /share.

hadoop dfs -mkdir /foodir

hadoop fs -ls

hadoop fs -rm -R /foodir

hadoop dfs -cat /foodir/myfile.txt

**HDFS High Availability**

The namenode is still a single point of failure (SPOF). If it did fail, all clients—including MapReduce jobs—would be unable to read, write, or list files, because the namenode is the sole repository of the metadata and the file-to-block mapping.

To recover from a failed namenode in this situation, an administrator starts a new primary namenode with one of the filesystem metadata replicas and configures datanodes and clients to use this new namenode. The new namenode is not able to serve requests until it has (i) loaded its namespace image into memory, (ii) replayed its edit log, and (iii) received enough block reports from the datanodes to leave safe mode. On large clusters with many files and blocks, the time it takes for a namenode to start from cold can be 30 minutes or more.

Hadoop 2 remedied this situation by adding support for HDFS high availability (HA). In this implementation, there are a pair of namenodes in an active-standby configuration. All blocks in a file except the last block are the same size

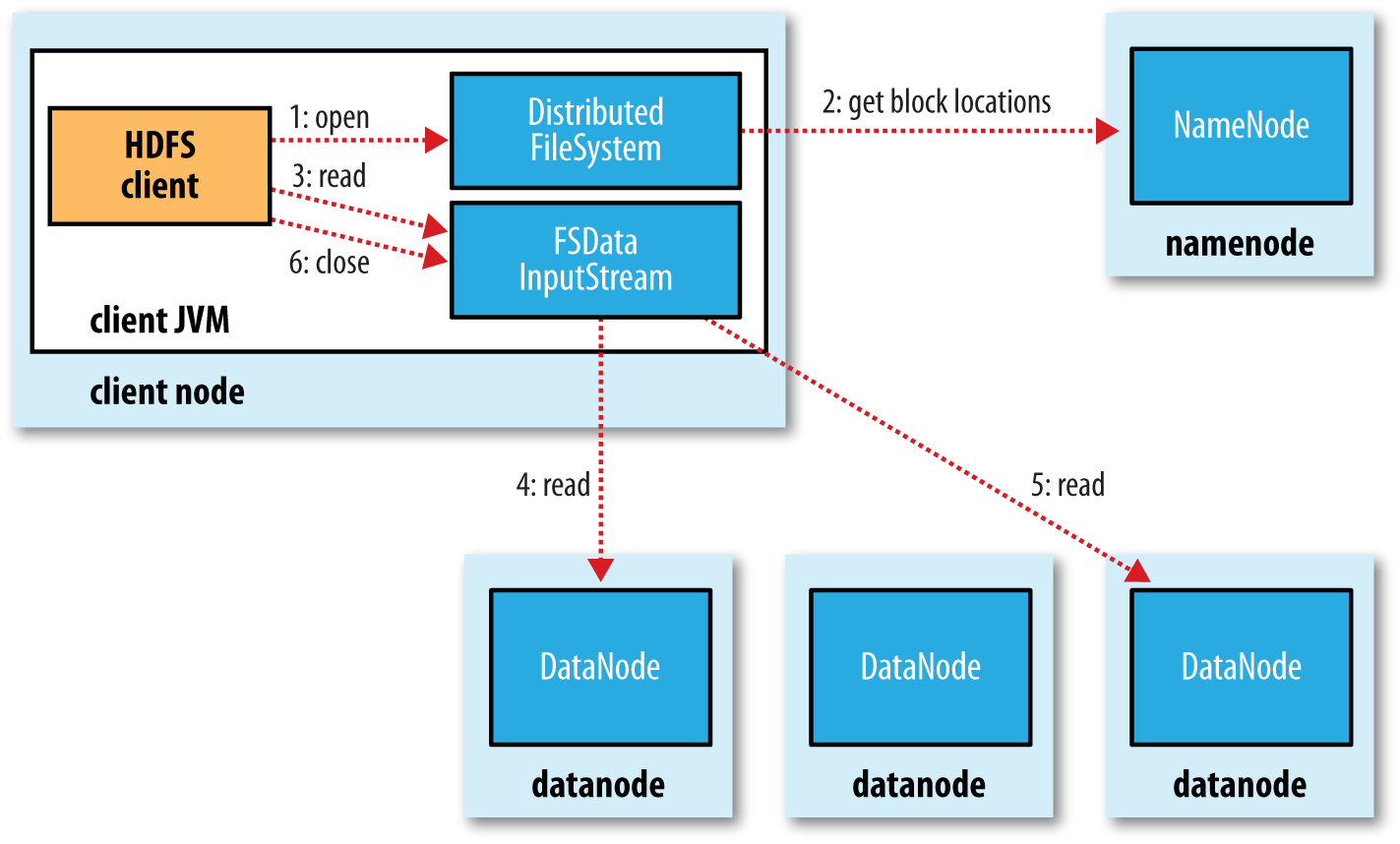
There are two choices for the highly available shared storage: an NFS filer, or a **quorum journal manager (QJM)**. The QJM is a dedicated HDFS implementation, designed for the sole purpose of providing a highly available edit log, and is the recommended choice for most HDFS installations. The QJM runs as a group of journal nodes, and each edit must be written to a majority of the journal nodes. Typically, there are three journal nodes, so the system can tolerate the loss of one of them.

**Failover and fencing**: The transition from the active namenode to the standby is managed by a new entity in the system called the failover controller. The HA implementation goes to great lengths to ensure that the previously active namenode is prevented from doing any damage and causing corruption—a method known as fencing.

**Client failover** is handled transparently by the client library. The simplest implementation uses client-side configuration to control failover. The HDFS URI uses a logical hostname that is mapped to a pair of namenode addresses (in the configuration file), and the client library tries each namenode address until the operation succeeds.

**Data Flow**

**Anatomy of a File Read**

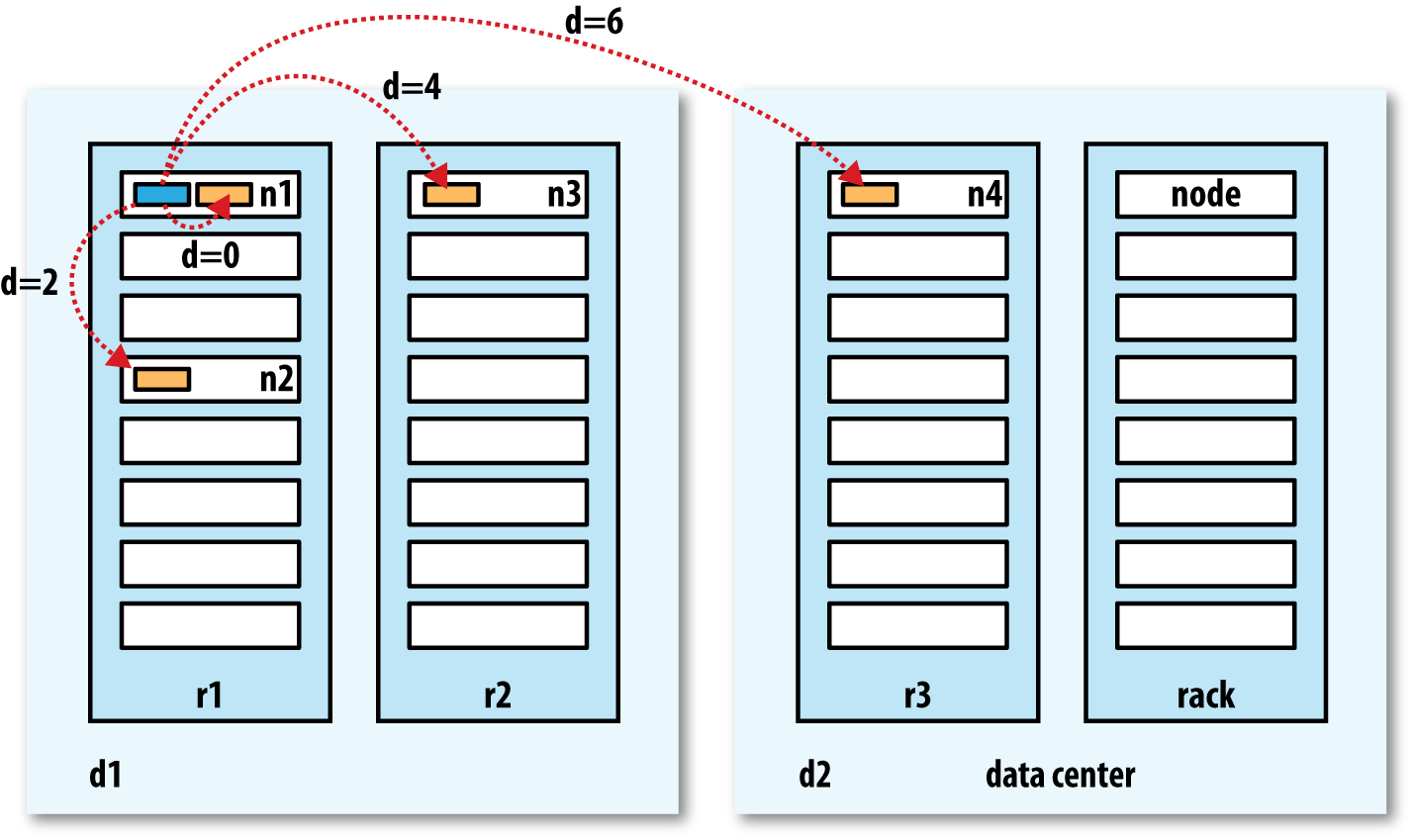


The client opens the file it wishes to read by calling open() on the FileSystem object, which for HDFS is an instance of DistributedFileSystem (Figure). DistributedFileSystem calls the namenode, using remote procedure calls (RPCs), to determine the locations of the first few blocks in the file (step 2). For each block, the namenode returns the addresses of the datanodes that have a copy of that block. Furthermore, the datanodes are sorted according to their proximity to the client (according to the topology of the cluster’s network. If the client is itself a datanode (in the case of a MapReduce task, for instance), the client will read from the local datanode if that datanode hosts a copy of the block.

Data is streamed from the datanode back to the client, which calls read() repeatedly on the stream (step 4). When the end of the block is reached, DFSInputStream will close the connection to the datanode, then find the best datanode for the next block (step 5). This happens transparently to the client, which from its point of view is just reading a continuous stream.

One important aspect of this design is that the client contacts datanodes directly to retrieve data and is guided by the namenode to the best datanode for each block. This design allows HDFS to scale to a large number of concurrent clients because the data traffic is spread across all the datanodes in the cluster. Meanwhile, the namenode merely has to service block location requests (which it stores in memory, making them very efficient) and does not, for example, serve data, which would quickly become a bottleneck as the number of clients grew.

**Network Topology**



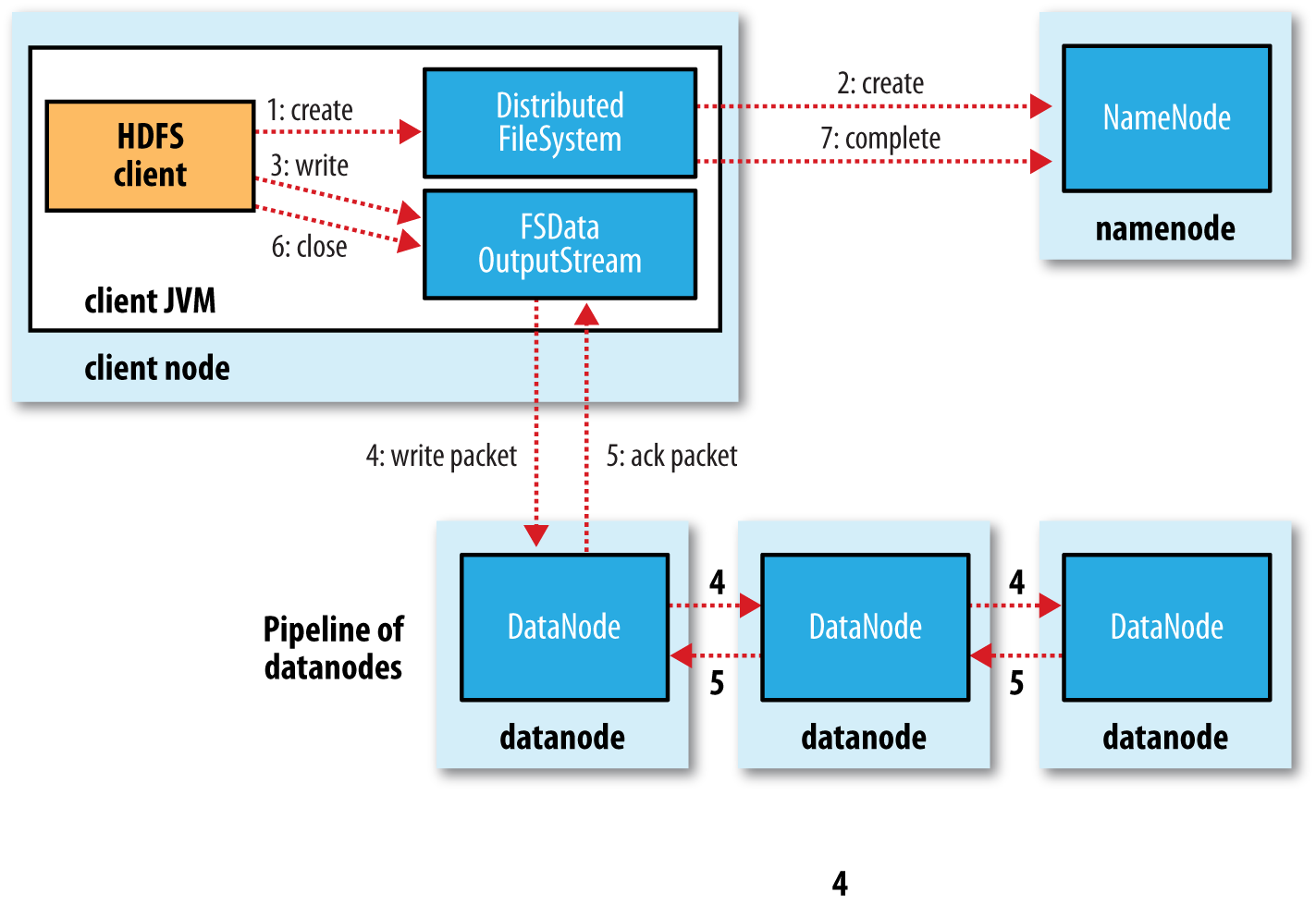
• distance(/d1/r1/n1, /d1/r1/n1) = 0 (processes on the same node)

• distance(/d1/r1/n1, /d1/r1/n2) = 2 (different nodes on the same rack)

• distance(/d1/r1/n1, /d1/r2/n3) = 4 (nodes on different racks in the same data center)

• distance(/d1/r1/n1, /d2/r3/n4) = 6 (nodes in different data centers)

**Anatomy of a File Write**



As long as dfs.namenode.replication.min replicas (which defaults to 1) are written, the write will succeed, and the block will be asynchronously replicated across the cluster until its target replication factor is reached (dfs.replication, which defaults to 3).

**Replica Placement**

How does the namenode choose which datanodes to store replicas on? There’s a tradeoff between reliability and write bandwidth and read bandwidth here. For example, placing all replicas on a single node incurs the lowest write bandwidth penalty (since the replication pipeline runs on a single node), but this offers no real redundancy (if the node fails, the data for that block is lost). Also, the read bandwidth is high for off-rack reads. At the other extreme, placing replicas in different data centers may maximize redundancy, but at the cost of bandwidth. Even in the same data center (which is what all Hadoop clusters to date have run in), there are a variety of possible placement strategies.

## 4. YARN

Apache **YARN** (Yet Another Resource Negotiator) is Hadoop’s cluster resource management system.

YARN provides its core services via two types of long-running daemon: a **resource manager** (one per cluster) to manage the use of resources across the cluster, and **node managers** running on all the nodes in the cluster to launch and monitor containers. A container executes an application-specific process with a constrained set of resources (memory, CPU, and so on). Depending on how YARN is configured a container may be a Unix process or a Linux cgroup.

To run an application on YARN, a client contacts the resource manager and asks it to run an application **master process**. The resource manager then finds a node manager that can launch the application master in a container. Precisely what the application master does once it is running depends on the application. It could simply run a computation in the container it is running in and return the result to the client. Or it could request more containers from the resource managers and use them to run a distributed computation. The latter is what the MapReduce YARN application does.

YARN itself does not provide any way for the parts of the application (client, master, process) to communicate with one another. Most nontrivial YARN applications use some form of remote communication (such as Hadoop’s RPC layer) to pass status updates and results back to the client, but these are specific to the application.

Locality is critical in ensuring that distributed data processing algorithms use the cluster bandwidth efficiently,2 so YARN allows an application to specify locality constraints for the containers it is requesting. Locality constraints can be used to request a container on a specific node or rack, or anywhere on the cluster (off-rack).

A YARN application can make resource requests at any time while it is running. For example, an application can make all of its requests up front, or it can take a more dynamic approach.

**Spark** takes the first approach, starting a fixed number of executors on the cluster. **MapReduce**, on the other hand, has two phases: the map task containers are requested up front, but the reduce task containers are not started until later.

Three schedulers are available in YARN: the **FIFO**, **Capacity** (a separate dedicated queue allows the small job to start as soon as it is submitted), and **Fair** (dynamically balance resources between all running jobs) Schedulers.

When a job is submitted to an empty queue on a busy cluster, the job cannot start until resources free up from jobs that are already running on the cluster. To make the time taken for a job to start more predictable, the Fair Scheduler supports preemption. **Preemption** allows the scheduler to kill containers for queues that are running with more than their fair share of resources so that the resources can be allocated to a queue that is under its fair share. Note that preemption reduces overall cluster efficiency, since the terminated containers need to be reexecuted.

All the YARN schedulers try to honor locality requests. On a busy cluster, if an application requests a particular node, there is a good chance that other containers are running on it at the time of the request. The obvious course of action is to immediately loosen the locality requirement and allocate a container on the same rack. However, it has been observed in practice that waiting a short time (no more than a few seconds) can dramatically increase the chances of being allocated a container on the requested node, and therefore increase the efficiency of the cluster. This feature is called **delay scheduling**, and it is supported by both the Capacity Scheduler and the Fair Scheduler.

## 5. Hadoop I/0

The volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high. A commonly used error-detecting code is **CRC-32** (32-bit cyclic redundancy check), which computes a 32-bit integer checksum for input of any size. HDFS transparently checksums all data written to it and by default verifies checksums when reading data.

In addition to block verification on client reads, each datanode runs a DataBlockScan ner in a background thread that periodically verifies all the blocks stored on the datanode. This is to guard against corruption due to “**bit rot**” in the physical storage media.

Because HDFS stores replicas of blocks, it can “**heal**” corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica.

**File compression** brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop. All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings. Use a compression format that **supports splitting**, such as **bzip2**. Formats like gzip does not support splitting. Or, Split the file into chunks in the application, and compress each chunk separately using any supported compression format.

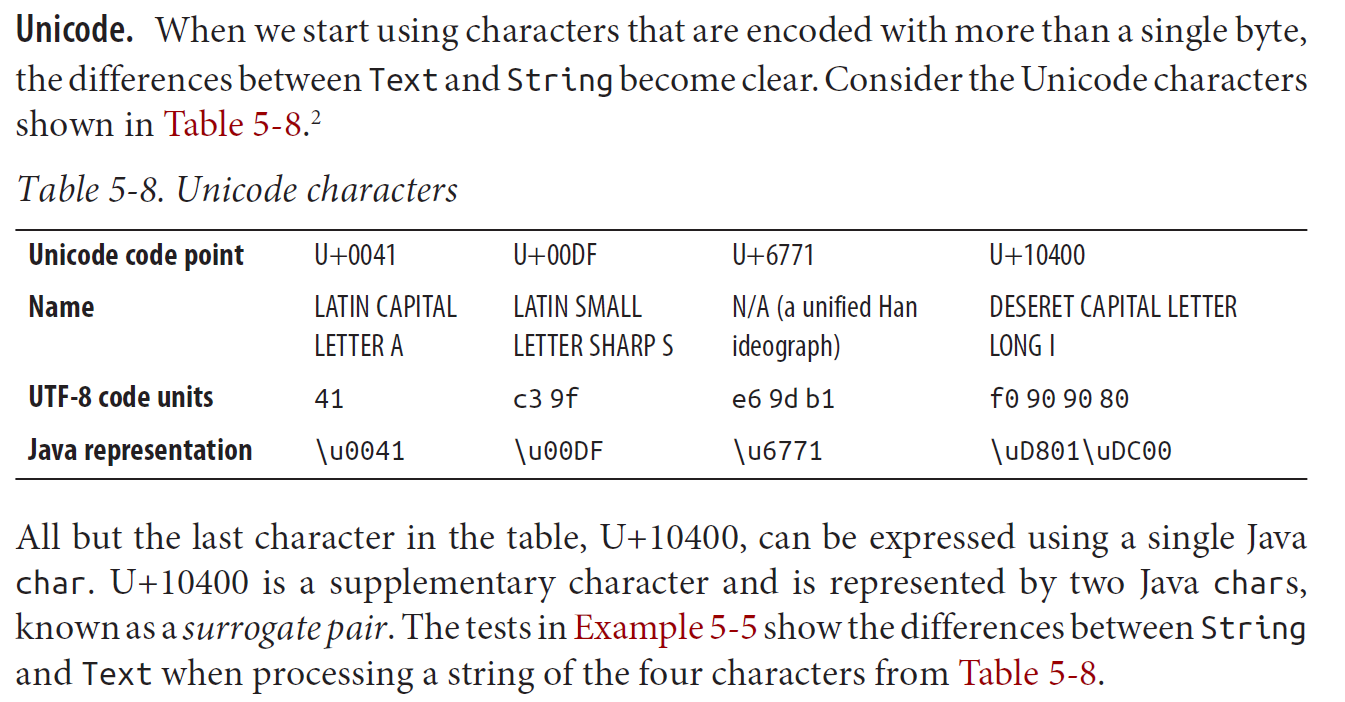
**Serialization** is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage. **Deserialization** is the reverse process of turning a byte stream back into a series of structured objects.

The **Writable** interface defines two methods—one for writing its state to a DataOut put binary stream and one for reading its state from a DataInput binary stream.

Ex: **IntWritable** implements the WritableComparable interface, which is just a subinterface of the Writable and java.lang.Comparable interfaces.

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another. One optimization that Hadoop provides is the **RawComparator** extension of Java’s Comparator.

**Text** is a Writable for UTF-8 sequences. It can be thought of as the Writable equivalent of java.lang.String.



First character is of length 1 byte, 2nd 2 bytes, 3rd 3 bytes and last 4 bytes.

## 6. MapReduce

**The client classpath**

The user’s client-side classpath set by hadoop jar <jar> is made up of:

* The job JAR file
* Any JAR files in the lib directory of the job JAR file, and the classes directory (if present)
* The classpath defined by HADOOP\_CLASSPATH, if set Incidentally, this explains why you have to set HADOOP\_CLASSPATH to point to dependent classes and libraries if you are running using the local job runner without a job JAR (hadoop CLASSNAME).

**The task classpath**

On a cluster (and this includes pseudodistributed mode), map and reduce tasks run in separate JVMs, and their classpaths are not controlled by HADOOP\_CLASSPATH. HADOOP\_CLASSPATH is a client-side setting and only sets the classpath for the driver JVM, which submits the job.

Instead, the user’s task classpath is comprised of the following:

* The job JAR file
* Any JAR files contained in the lib directory of the job JAR file, and the classes directory (if present)
* Any files added to the distributed cache using the -libjars option, or the addFileToClassPath() method on DistributedCache (old API), or Job (new API)

**Packaging dependencies**

Given these different ways of controlling what is on the client and task classpaths, there are corresponding options for including library dependencies for a job:

* Unpack the libraries and repackage them in the job JAR.
* Package the libraries in the lib directory of the job JAR.
* Keep the libraries separate from the job JAR, and add them to the client classpath via HADOOP\_CLASSPATH and to the task classpath via -libjars.

**Tuning a Job**

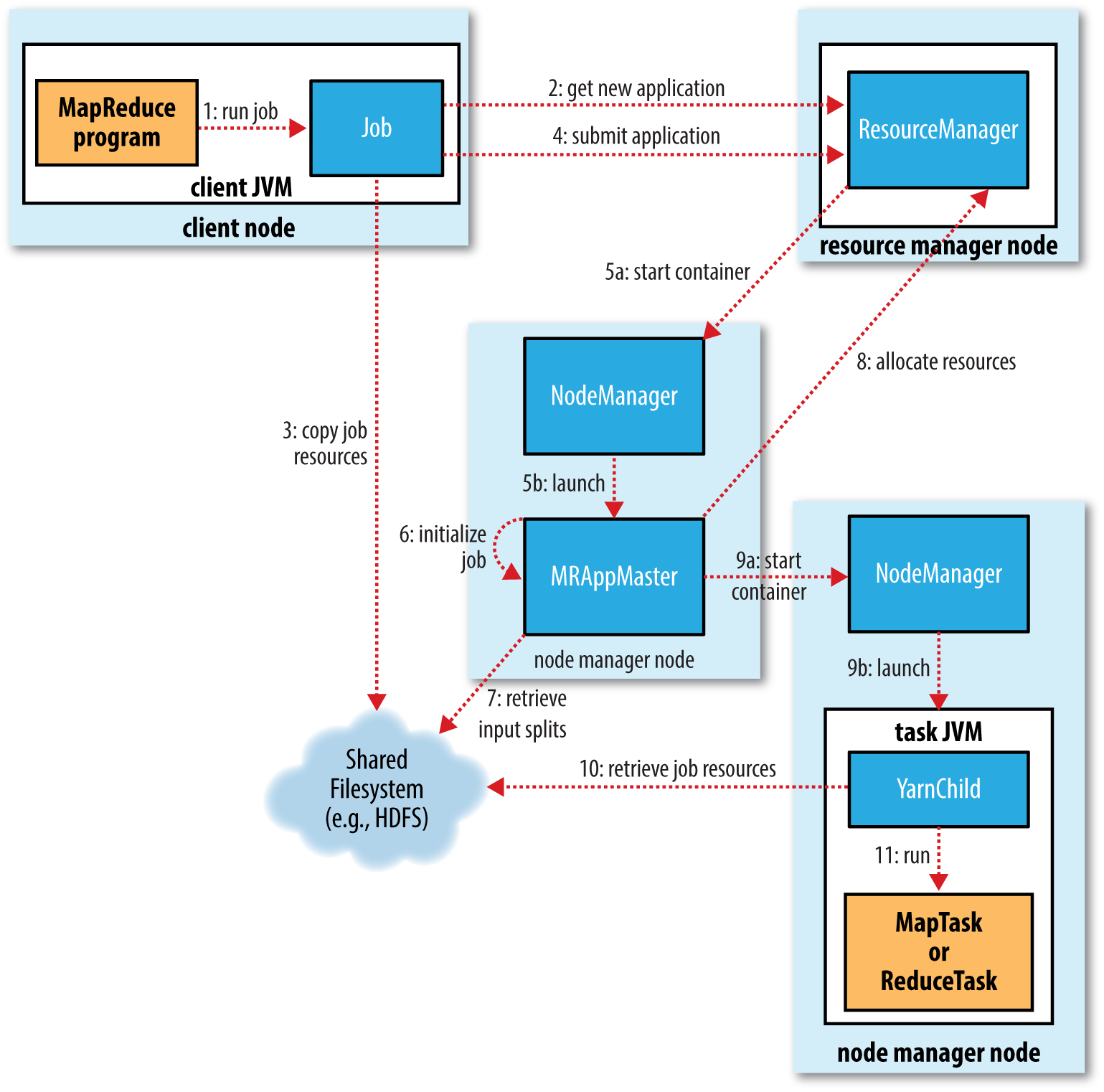
* Number of mappers: How long are your mappers running for? If they are only running for a few seconds on average, you should see whether there’s a way to have fewer mappers and make them all run longer—a minute or so, as a rule of thumb. The extent to which this is possible depends on the input format you are using.
* Number of reducers: Check that you are using more than a single reducer. Reduce tasks should run for five minutes or so and produce at least a block’s worth of data, as a rule of thumb.
* Combiners: Check whether your job can take advantage of a combiner to reduce the amount of data passing through the shuffle.
* Intermediate compression: Job execution time can almost always benefit from enabling map output compression.
* Custom serialization: If you are using your own custom Writable objects or custom comparators, make sure you have implemented RawComparator.
* Shuffle tweaks: The MapReduce shuffle exposes around a dozen tuning parameters for memory management, which may help you wring out the last bit of performance.

## 7. How MapReduce Works

**Anatomy of a MapReduce Job Run**

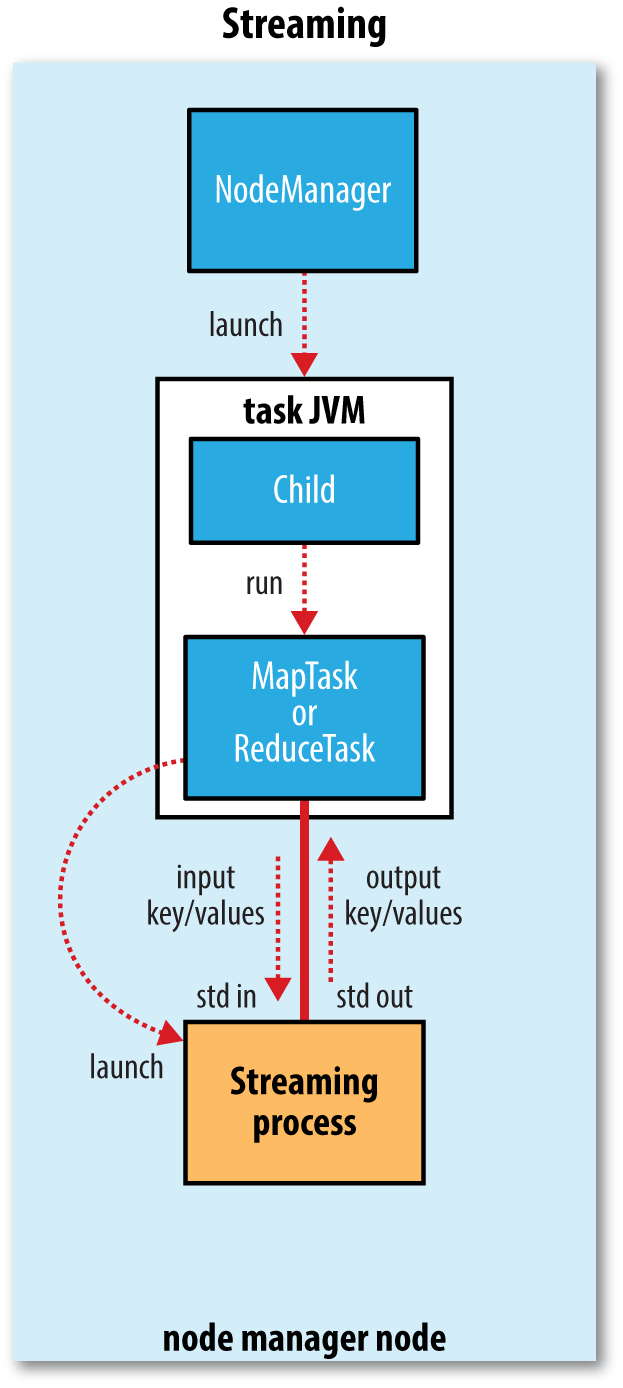
You can run a MapReduce job with a single method call: submit() on a Job object. You can also call waitForCompletion(), which submits the job if it hasn’t been submitted already, then waits for it to finish)

* The client, which submits the MapReduce job.
* The YARN resource manager, which coordinates the allocation of compute resources on the cluster.
* The YARN node managers, which launch and monitor the compute containers on machines in the cluster.
* The MapReduce application master, which coordinates the tasks running the Map‐ Reduce job. The application master and the MapReduce tasks run in containers that are scheduled by the resource manager and managed by the node managers.
* The distributed filesystem, which is used for sharing job files between the other entities.



**Streaming**

Streaming runs special map and reduce tasks for the purpose of launching the usersupplied executable and communicating with it. The Streaming task communicates with the process (which may be written in any language) using standard input and output streams. During execution of the task, the Java process passes input key-value pairs to the external process, which runs it through the user-defined map or reduce function and passes the output key-value pairs back to the Java process. From the node manager’s point of view, it is as if the child process ran the map or reduce code itself.

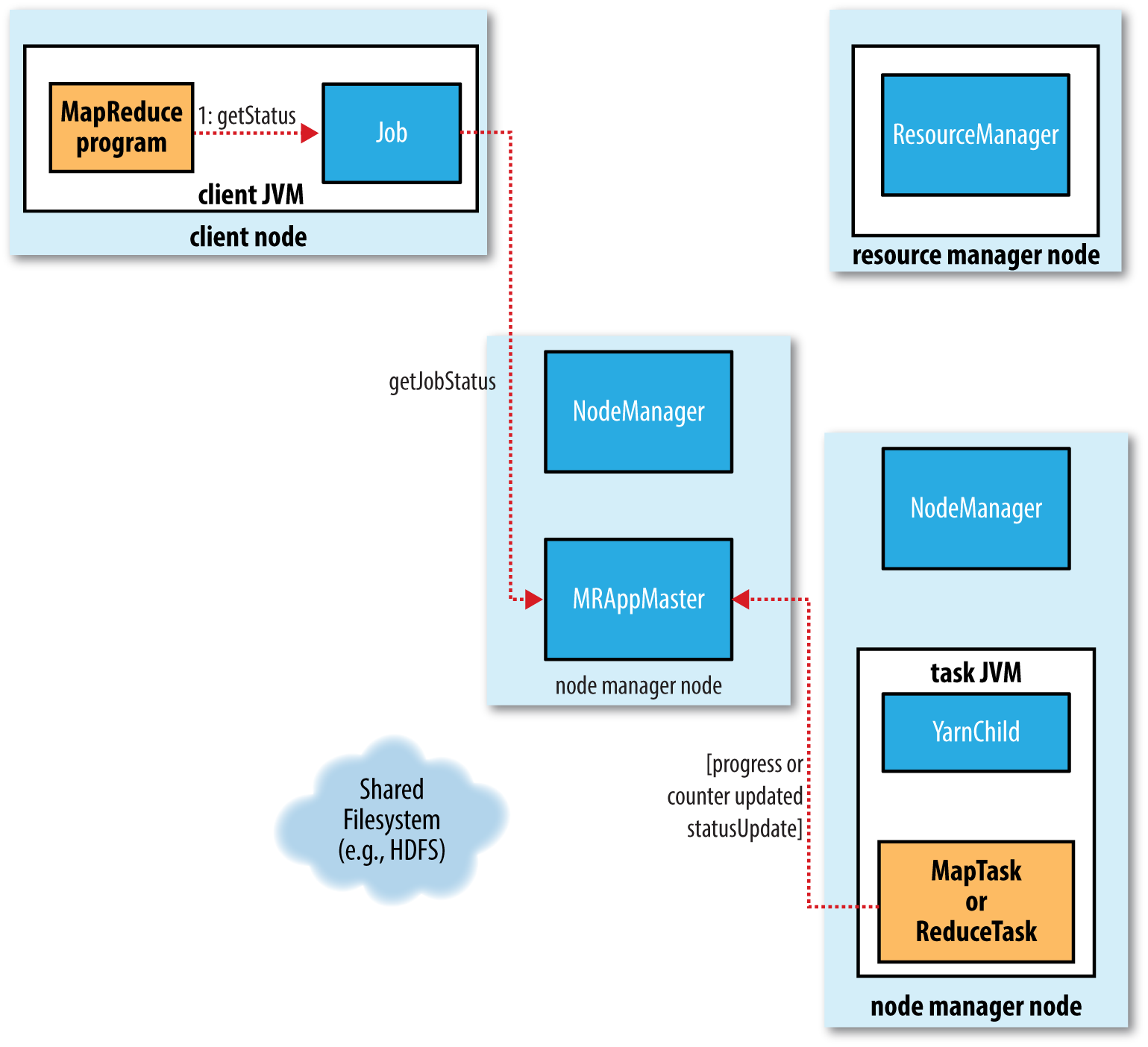


**Progress and Status Updates**

MapReduce jobs are long-running batch jobs, taking anything from tens of seconds to hours to run.

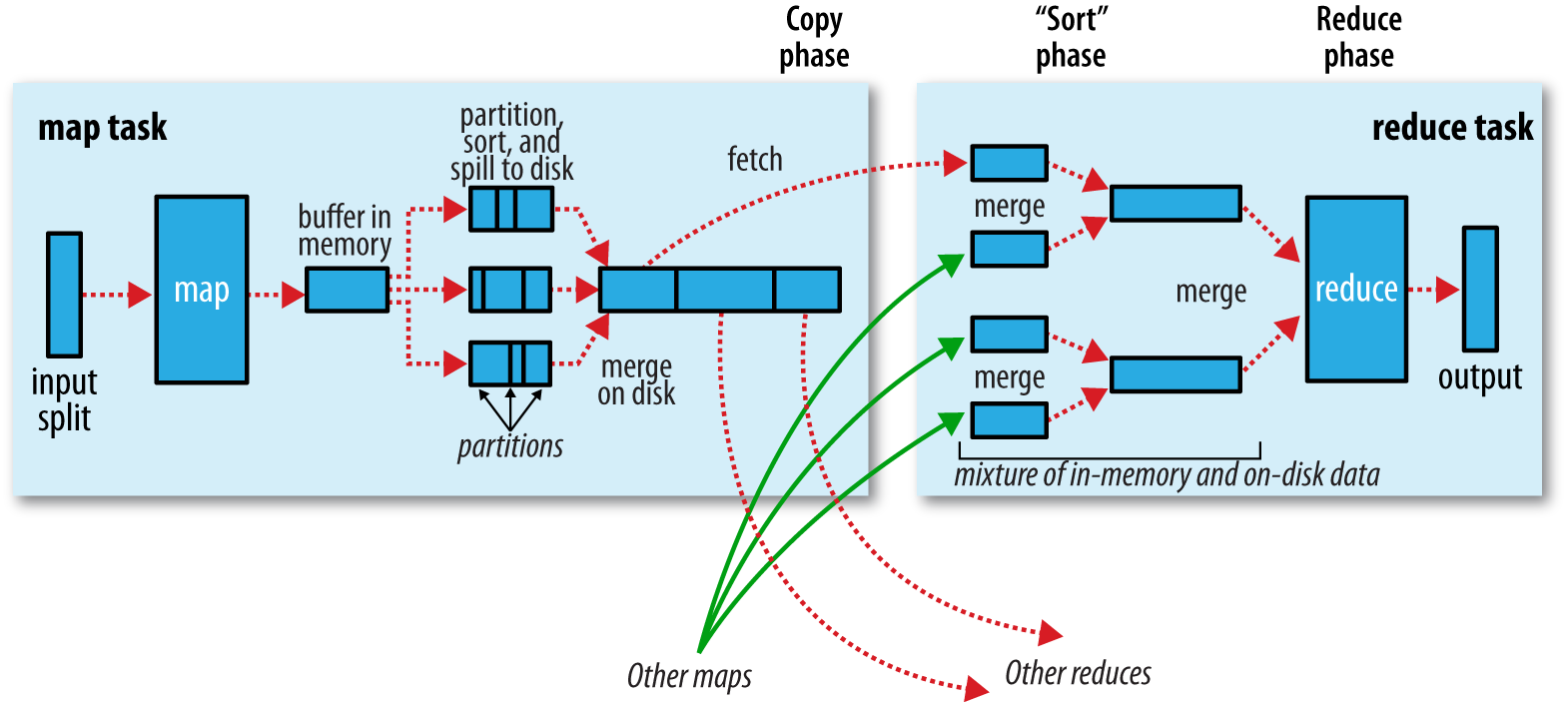
For map tasks, this is the proportion of the input that has been processed. For reduce tasks, it’s a little more complex, but the system can still estimate the proportion of the reduce input processed.

As the map or reduce task runs, the child process communicates with its parent application master through the **umbilical interface**. The task reports its progress and status (including counters) back to its application master, which has an aggregate view of the job, every three seconds over the umbilical interface.



**Shuffle and Sort**

MapReduce makes the guarantee that the input to every reducer is sorted by key. The process by which the system performs the sort—and transfers the map outputs to the reducers as inputs—is known as the **shuffle**.



**Speculative Execution**

The MapReduce model is to break jobs into tasks and run the tasks in parallel to make the overall job execution time smaller than it would be if the tasks ran sequentially. This makes the job execution time sensitive to slow-running tasks, as it takes only one slow task to make the whole job take significantly longer than it would have done otherwise. When a job consists of hundreds or thousands of tasks, the possibility of a few straggling tasks is very real.

Tasks may be slow for various reasons, including hardware degradation or software misconfiguration, but the causes may be hard to detect because the tasks still complete successfully, albeit after a longer time than expected. Hadoop doesn’t try to diagnose and fix slow-running tasks; instead, it tries to detect when a task is running slower than expected and launches another equivalent task as a backup. This is termed **speculative execution** of tasks. When a task completes successfully, any duplicate tasks that are running are killed since they are no longer needed.

## 8. MapReduce Types and Formats

MapReduce has a simple model of data processing: inputs and outputs for the map and reduce functions are key-value pairs.

The map and reduce functions in Hadoop MapReduce have the following general form:

**map: (K1, V1) → list(K2, V2)**

**combiner: (K2, list(V2)) → list(K3, V3)**

**reduce: (K2, list(V2)) → list(K3, V3)**

public class **Mapper**<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

public class **Context** extends MapContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

// ...

}

protected void **map**(KEYIN key, VALUEIN value,

Context context) throws IOException, InterruptedException {

// ...

}

}

public class **Reducer**<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

public class **Context** extends ReducerContext<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

// ...

}

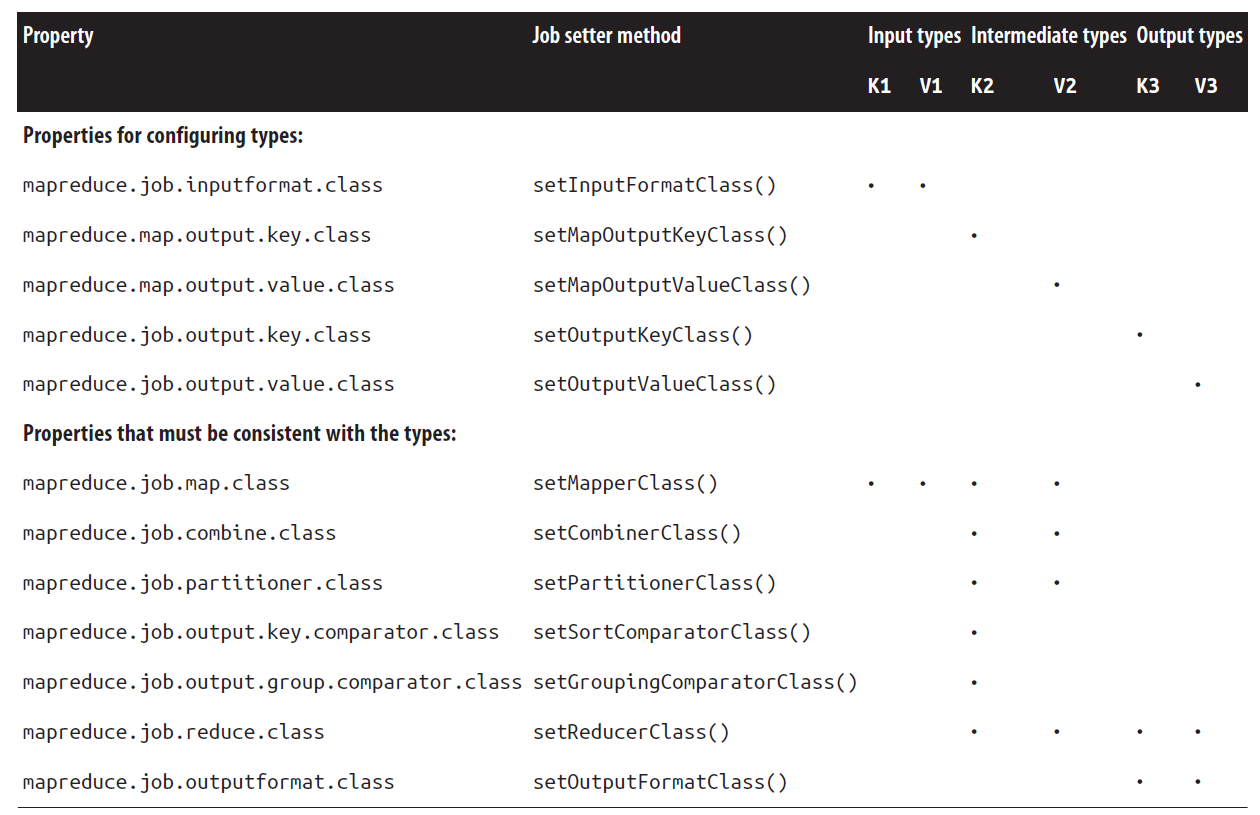
protected void **reduce**(KEYIN key, Iterable<VALUEIN> values,

Context context) throws IOException, InterruptedException {

// ...

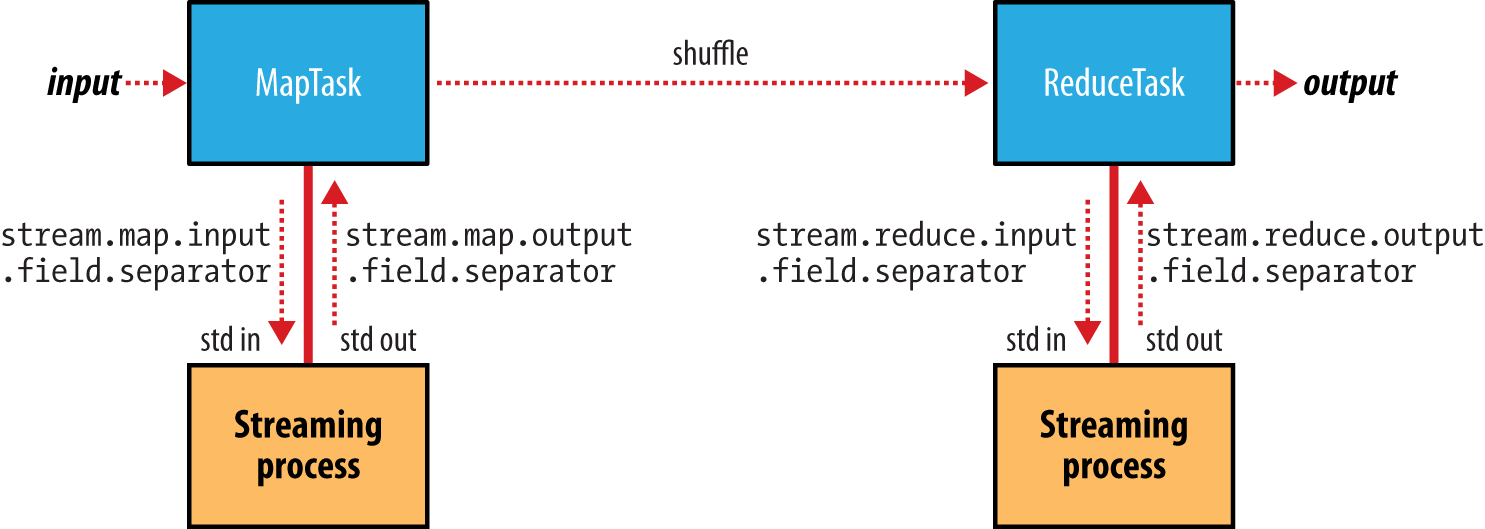
}

}



By default, there is a single reducer, and therefore a single partition; the action of the partitioner is irrelevant in this case since everything goes into one partition. However, it is important to understand the behavior of HashPartitioner when you have more than one reduce task. Assuming the key’s hash function is a good one, the records will be allocated evenly across reduce tasks, with all records that share the same key being processed by the same reduce task.

A non-Java Streaming application can control the separator that is used when a key-value pair is turned into a series of bytes and sent to the map or reduce process over standard input.



**Input Formats**

An **input split** is a chunk of the input that is processed by a single map. Each map processes a single split. Each split is divided into records, and the map processes each record—a key-value pair—in turn.

Hadoop works better with a small number of large files than a large number of small files. One reason for this is that FileInputFormat generates splits in such a way that each split is all or part of a single file. One technique for avoiding the many small files case is to merge small files into larger files by using a sequence file.

Although the input to a MapReduce job may consist of multiple input files (constructed by a combination of file globs, filters, and plain paths), all of the input is interpreted by a single InputFormat and a single Mapper. What often happens, however, is that the data format evolves over time, so you have to write your mapper to cope with all of your legacy formats.

## 9 MapReduce Features

**Counters**

* Task counters: map input records, map output records, map output bytes similarly for combine, reduce, etc.
* Job counters: Launched map tasks, launched reduce tasks, similarly failed, killed tasks etc.
* User defined counters: context.getCounter(Temperature.MALFORMED).increment(1);

**Controlling Sort Order**

The sort order for keys is controlled by a RawComparator, which is found as follows:

1. If the property mapreduce.job.output.key.comparator.class is set, either explicitly or by calling setSortComparatorClass() on Job, then an instance of that class is used. (In the old API, the equivalent method is setOutputKeyComparator Class() on JobConf.)
2. Otherwise, keys must be a subclass of WritableComparable, and the registered comparator for the key class is used.
3. If there is no registered comparator, then a RawComparator is used. The RawCompa rator deserializes the byte streams being compared into objects and delegates to the WritableComparable’s compareTo() method.

**Secondary Sort**

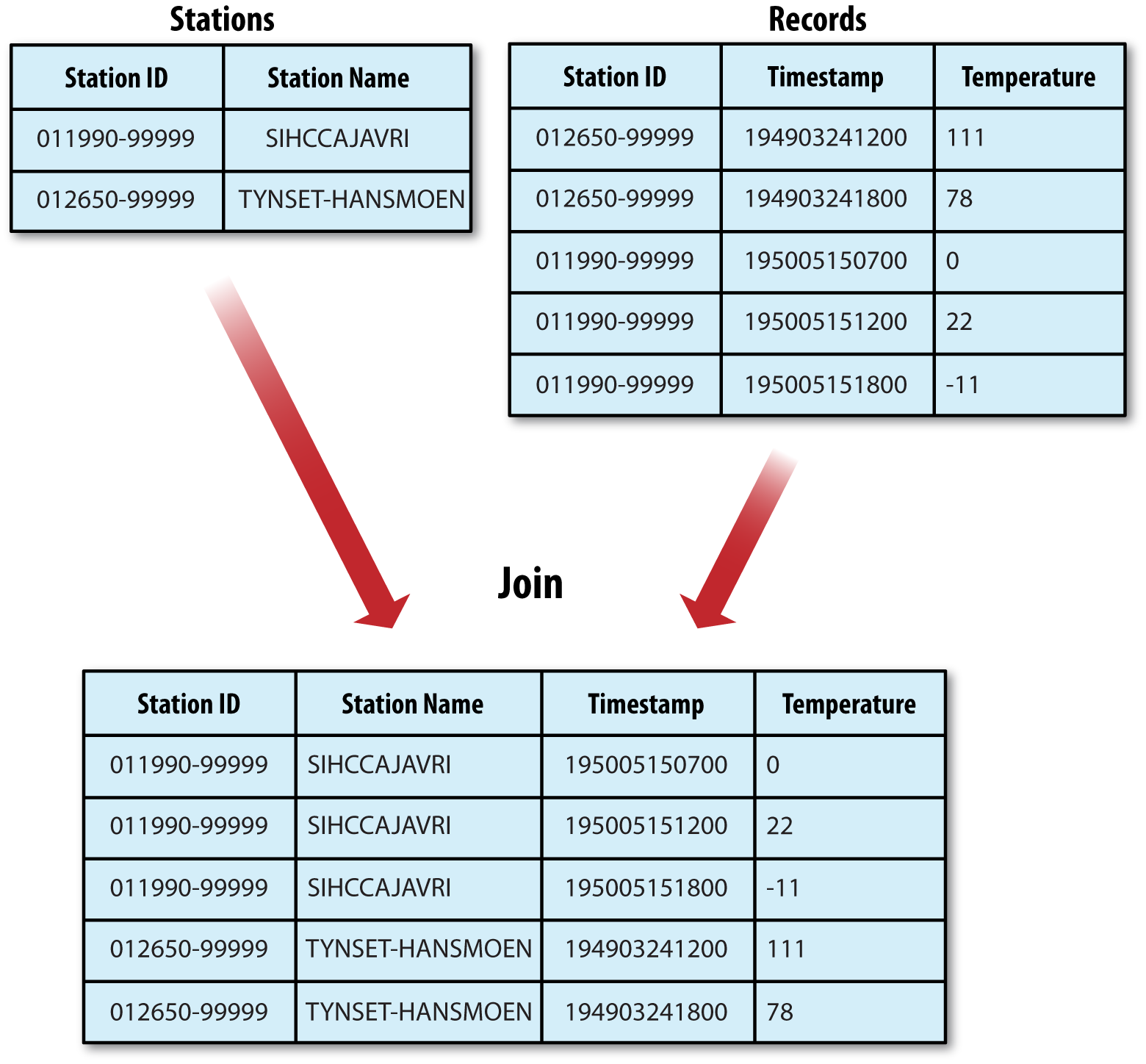
The MapReduce framework sorts the records by key before they reach the reducers. For any particular key, however, the values are not sorted. The order in which the values appear is not even stable from one run to the next, because they come from different map tasks, which may finish at different times from run to run. Generally speaking, most MapReduce programs are written so as not to depend on the order in which the values appear to the reduce function. However, it is possible to impose an order on the values by sorting and grouping the keys in a particular way.

The recipe here to get the effect of sorting by value:

* Make the key a composite of the natural key and the natural value.
* The sort comparator should order by the composite key (i.e., the natural key and natural value).
* The partitioner and grouping comparator for the composite key should consider only the natural key for partitioning and grouping.

**Joins**

MapReduce can perform joins between large datasets, but writing the code to do joins from scratch is fairly involved. Rather than writing MapReduce programs, you might consider using a higher-level framework such as Pig, Hive, Cascading, Cruc, or Spark, in which join operations are a core part of the implementation.



**Distributed Cache**

Rather than serializing side data in the job configuration, it is preferable to distribute datasets using Hadoop’s distributed cache mechanism. This provides a service for copying files and archives to the task nodes in time for the tasks to use them when they run. To save network bandwidth, files are normally copied to any particular node once per job.

## 10 Hadoop Operations

**Cluster Specification**

Hadoop is designed to run on commodity hardware. That means that you are not tied to expensive, proprietary offerings from a single vendor; rather, you can choose standardized, commonly available hardware from any of a large range of vendors to build your cluster.

“Commodity” does not mean “low-end.” Low-end machines often have cheap components, which have higher failure rates than more expensive (but still commodity-class) machines. When you are operating tens, hundreds, or thousands of machines, cheap components turn out to be a false economy, as the higher failure rate incurs a greater maintenance cost.

Although the hardware specification for your cluster will assuredly be different, Hadoop is designed to use multiple cores and disks, so it will be able to take full advantage of more powerful hardware.

The **namenode has high memory requirements**, as it holds file and block metadata for the entire namespace in memory. The secondary namenode, although idle most of the time, has a comparable memory footprint to the primary when it creates a checkpoint.

The aggregate bandwidth between nodes on the same rack is much greater than that between nodes on different racks. To get maximum performance out of Hadoop, it is important to configure Hadoop so that it knows the topology of your network. If your cluster runs on a single rack, then there is nothing more to do, since this is the default. However, for multirack clusters, you need to map nodes to racks. This allows Hadoop to prefer within-rack transfers (where there is more bandwidth available) to off-rack transfers when placing MapReduce tasks on nodes. HDFS will also be able to place replicas more intelligently to trade off performance and resilience.

**Cluster Setup**

It’s good practice to create dedicated Unix user accounts to separate the Hadoop processes from each other, and from other services running on the same machine. The HDFS, MapReduce, and YARN services are usually run as separate users, named hdfs, mapred, and yarn, respectively.

The Hadoop control scripts (but not the daemons) rely on SSH to perform cluster-wide operations. To work seamlessly, SSH needs to be set up to allow passwordless login for the hdfs and yarn users from machines in the cluster. The simplest way to achieve this is to generate a public/private key pair and place it in an NFS location that is shared across the cluster.

Formatting the HDFS Filesystem Before it can be used, a brand-new HDFS installation needs to be formatted. The formatting process creates an empty filesystem by creating the storage directories and the initial versions of the namenode’s persistent data structures. Datanodes are not involved in the initial formatting process, since the namenode manages all of the filesystem’s metadata, and datanodes can join or leave the cluster dynamically.

**hdfs namenode –format**

Hadoop comes with scripts for running commands and starting and stopping daemons across the whole cluster. To use these scripts, you need to tell Hadoop which machines are in the cluster. There is a file for this purpose, called slaves, which contains a list of the machine hostnames or IP addresses, one per line. It resides in Hadoop’s configuration directory.

**start-dfs.sh**

* Starts a namenode on each machine returned by executing hdfs getconf -namenodes
* Starts a datanode on each machine listed in the slaves file
* Starts a secondary namenode on each machine returned by executing hdfs getconf -secondarynamenodes

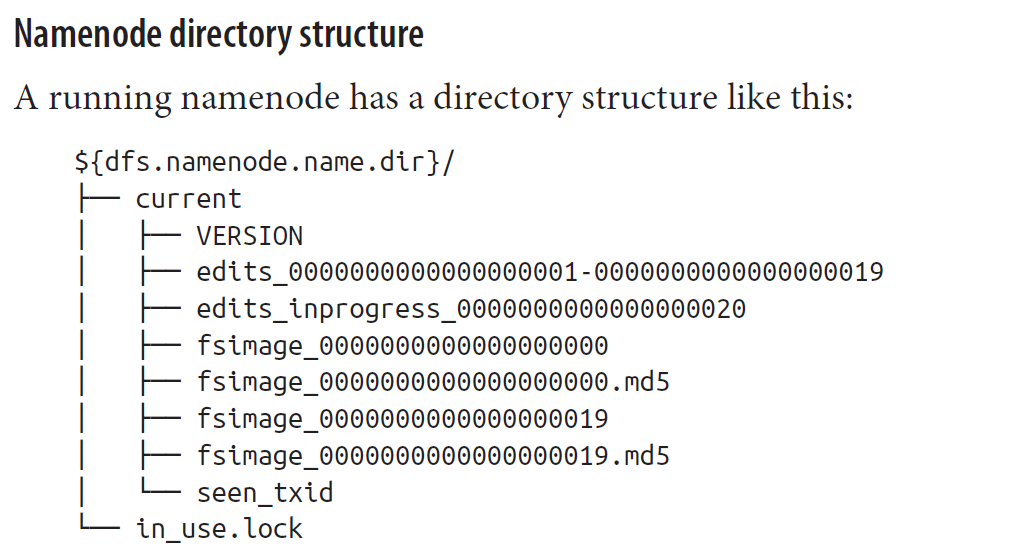
The YARN daemons are started in a similar way,

**start-yarn.sh**

Finally, there is only one MapReduce daemon—the job history server, which is started as follows, as the mapred user:

**mr-jobhistory-daemon.sh start historyserver**

## 11 Administering Hadoop



When a filesystem client performs a write operation (such as creating or moving a file), the transaction is first recorded in the **edit log**. The namenode also has an in-memory representation of the filesystem metadata, which it updates after the edit log has been modified. The in-memory metadata is used to serve read requests.

Conceptually the edit log is a single entity, but it is represented as a number of files on disk. Each file is called a segment, and has the prefix edits and a suffix that indicates the transaction IDs contained in it. Only one file is open for writes at any one time (edits\_inprogress\_0000000000000000020 in the preceding example), and it is flushed and synced after every transaction before a success code is returned to the client. For namenodes that write to multiple directories, the write must be flushed and synced to every copy before returning successfully. This ensures that no transaction is lost due to machine failure.

Each **fsimage** file is a complete persistent checkpoint of the filesystem metadata. (The suffix indicates the last transaction in the image.) However, it is not updated for every filesystem write operation, because writing out the fsimage file, which can grow to be gigabytes in size, would be very slow. This does not compromise resilience because if the namenode fails, then the latest state of its metadata can be reconstructed by loading the latest fsimage from disk into memory, and then applying each of the transactions from the relevant point onward in the edit log. In fact, this is precisely what the namenode does when it starts up.

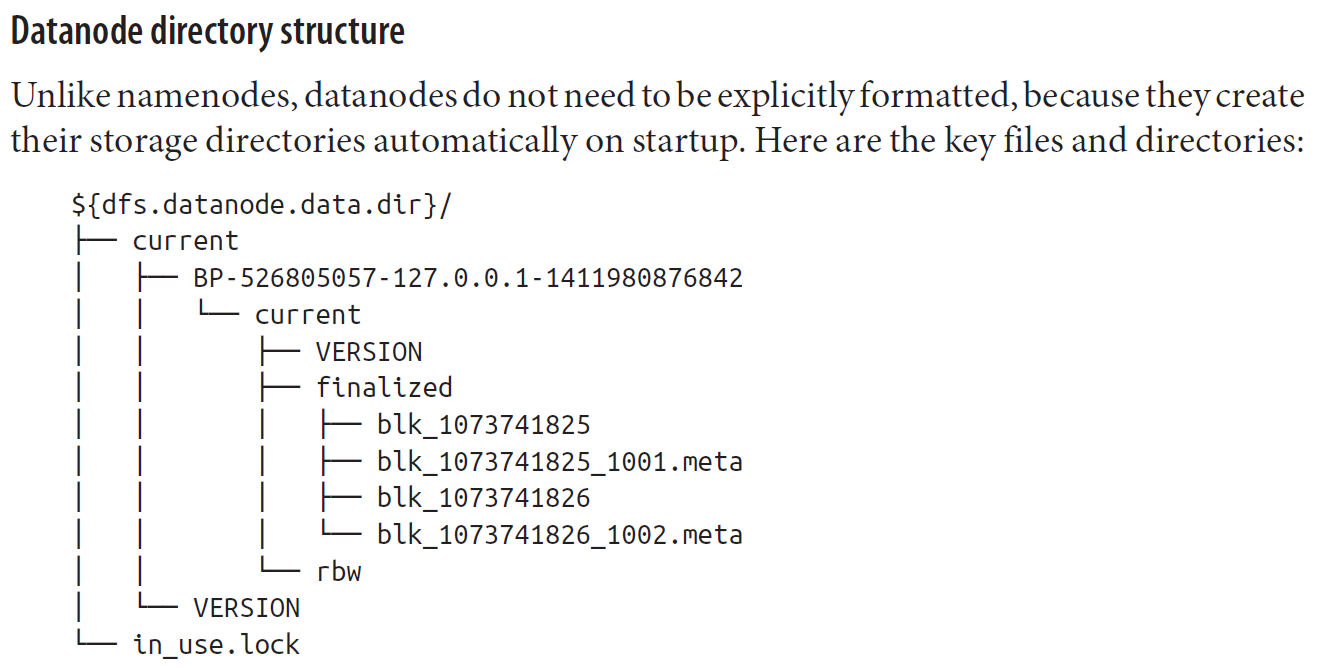
Each fsimage file contains a serialized form of all the directory and file inodes in the filesystem. Each inode is an internal representation of a file or directory’s metadata and contains such information as the file’s replication level, modification and access times, access permissions, block size, and the blocks the file is made up of. For directories, the modification time, permissions, and quota metadata are stored.

An fsimage file does not record the datanodes on which the blocks are stored. Instead, the namenode keeps this mapping in memory, which it constructs by asking the datanodes for their block lists when they join the cluster and periodically afterward to ensure the namenode’s block mapping is up to date.

As described, the edit log would grow without bound (even if it was spread across several physical edits files). Though this state of affairs would have no impact on the system while the namenode is running, if the namenode were restarted, it would take a long time to apply each of the transactions in its (very long) edit log. During this time, the filesystem would be offline, which is generally undesirable.

The solution is to run the secondary namenode, whose purpose is to produce checkpoints of the primary’s in-memory filesystem metadata.1 The checkpointing process proceeds as follows (and is shown schematically in Figure 11-1 for the edit log and image files shown earlier):

1. The secondary asks the primary to roll its in-progress edits file, so new edits go to a new file. The primary also updates the seen\_txid file in all its storage directories.
2. The secondary retrieves the latest fsimage and edits files from the primary (using HTTP GET).
3. The secondary loads fsimage into memory, applies each transaction from edits, then creates a new merged fsimage file.
4. The secondary sends the new fsimage back to the primary (using HTTP PUT), and the primary saves it as a temporary .ckpt file.
5. The primary renames the temporary fsimage file to make it available.



**Safe Mode**

When the namenode starts, the first thing it does is load its image file (fsimage) into memory and apply the edits from the edit log. Once it has reconstructed a consistent in-memory image of the filesystem metadata, it creates a new fsimage file (effectively doing the checkpoint itself, without recourse to the secondary namenode) and an empty edit log. During this process, the namenode is running in safe mode, which means that it offers only a read-only view of the filesystem to clients.

**New Nodes**

Datanodes that are permitted to connect to the namenode are specified in a file whose name is specified by the dfs.hosts property. The file resides on the namenode’s local filesystem, and it contains a line for each datanode, specified by network address (as reported by the datanode; you can see what this is by looking at the namenode’s web UI). If you need to specify multiple network addresses for a datanode, put them on one line, separated by whitespace.

Similarly, node managers that may connect to the resource manager are specified in a file whose name is specified by the yarn.resourcemanager.nodes.include-path property. In most cases, there is one shared file, referred to as the include file, that both dfs.hosts and yarn.resourcemanager.nodes.include-path refer to, since nodes in the cluster run both datanode and node manager daemons.

## 12 Avro

Apache **Avro** is a language-neutral data serialization system.

Avro has rich schema resolution capabilities. Within certain carefully defined constraints, the schema used to read data need not be identical to the schema that was used to write the data. This is the mechanism by which Avro supports schema evolution.

## 13 Parquet

Apache **Parquet** is a columnar storage format that can efficiently store nested data.

Columnar formats are attractive since they enable greater efficiency, in terms of both file size and query performance. File sizes are usually smaller than row-oriented equivalents since in a columnar format the values from one column are stored next to each other, which usually allows a very efficient encoding. A column storing a timestamp, for example, can be encoded by storing the first value and the differences between subsequent values (which tend to be small due to temporal locality: records from around the same time are stored next to each other). Query performance is improved too since a query engine can skip over columns that are not needed to answer a query.

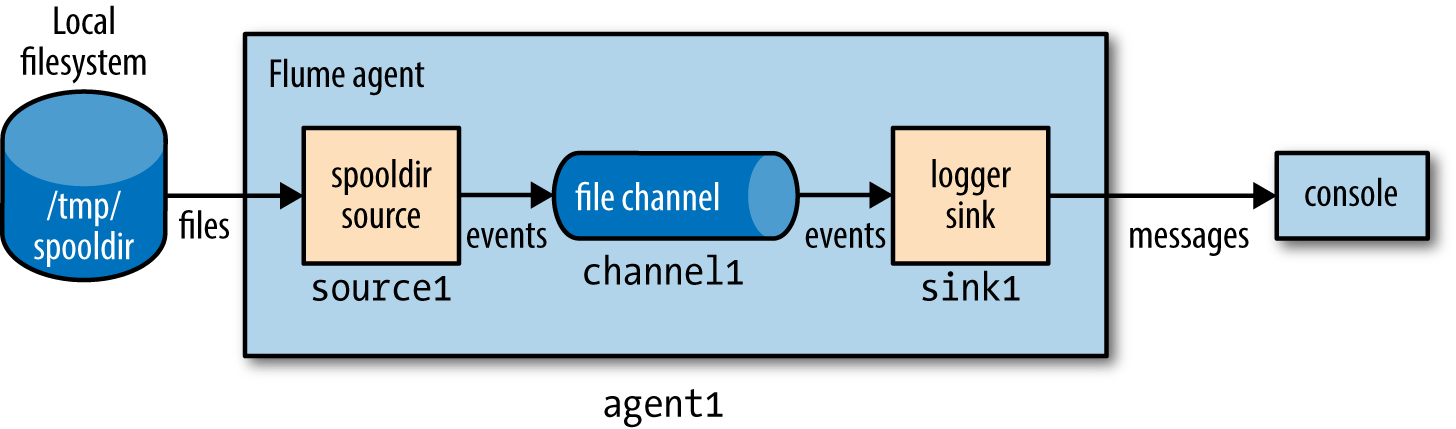
A key strength of Parquet is its ability to store data that has a deeply nested structure in true columnar fashion. This is important since schemas with several levels of nesting are common in real-world systems.

## 14 Flume

**Flume** is designed for high-volume ingestion into Hadoop of event-based data.

To use Flume, we need to run a Flume agent, which is a long-lived Java process that runs sources and sinks, connected by channels. A source in Flume produces events and delivers them to the channel, which stores the events until they are forwarded to the sink.

Flume is mainly a configuration exercise in wiring the pieces together.



Flume uses separate **transactions** to guarantee delivery from the source to the channel and from the channel to the sink.

**Fan out** is the term for delivering events from one source to multiple channels, so they reach multiple sinks. For

## 15 Sqoop

Apache **Sqoop** is an open source tool that allows users to extract data from a structured data store into Hadoop for further processing.

Sqoop is organized as a set of tools or commands.

Sqoop has an extension framework that makes it possible to import data from—and export data to—any external storage system that has bulk data transfer capabilities. A **Sqoop connector** is a modular component that uses this framework to enable Sqoop imports and exports. Sqoop ships with connectors for working with a range of popular databases, including MySQL, PostgreSQL, Oracle, SQL Server, DB2, and Netezza.

## 15 Pig

**Pig** is a scripting language for exploring large datasets.

**Script**

Pig can run a script file that contains Pig commands. For example, pig script.pig runs the commands in the local file script.pig. Alternatively, for very short scripts, you can use the -e option to run a script specified as a string on the command line.

**Grunt**

Grunt is an interactive shell for running Pig commands. Grunt is started when no file is specified for Pig to run and the -e option is not used. It is also possible to run Pig scripts from within Grunt using run and exec.

**Embedded**

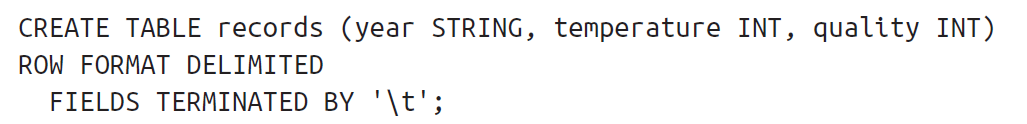
You can run Pig programs from Java using the PigServer class, much like you can use JDBC to run SQL programs from Java. For programmatic access to Grunt, use PigRunner.

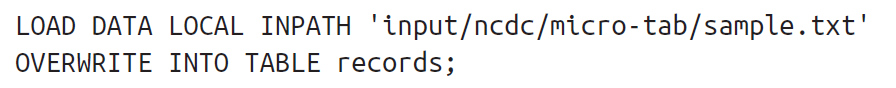
## 15 Hive

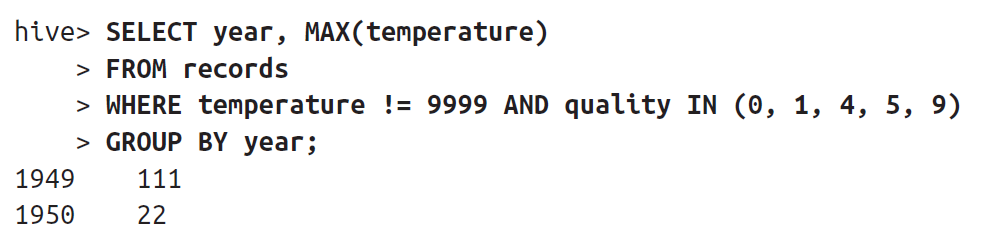
**Hive** was created to make it possible to run queries on the huge volumes of data.

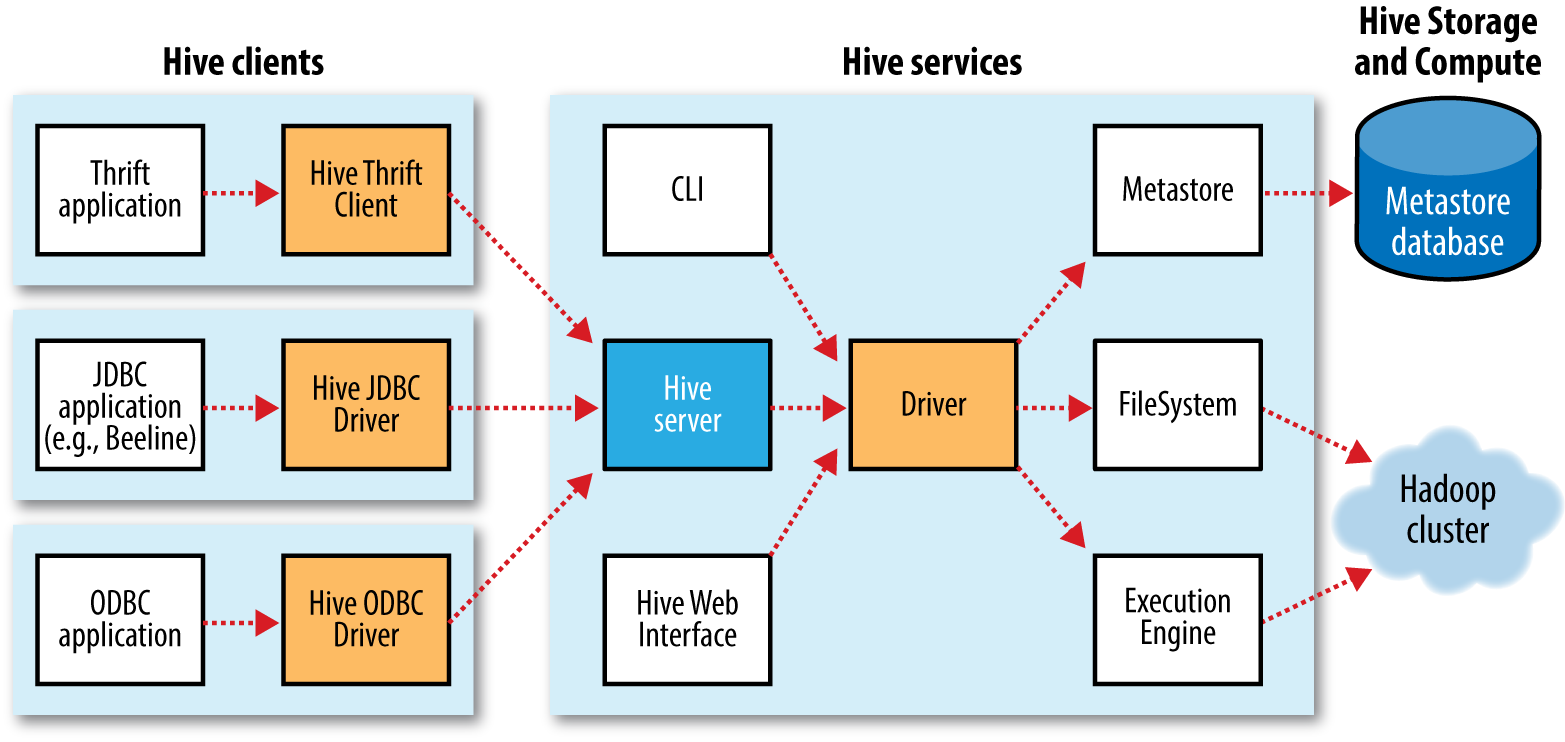
The **Hive Shell** is the primary way that we will interact with Hive, by issuing commands in HiveQL. **HiveQL** is Hive’s query language, a dialect of SQL. It is heavily influenced by MySQL, so if you are familiar with MySQL, you should feel at home using Hive.

**An Example**









The biggest limitation is that Hive does not provide record-level update, insert, nor delete. Hive doesn’t provide crucial features required for OLTP, Online Transaction Processing. It’s closer to being an OLAP tool, Online Analytic Processing.

**Hadoop-MapReduce**

MapReduce is a computing model that decomposes large data manipulation jobs into individual tasks that can be executed in parallel across a cluster of servers. The results of the tasks can be joined together to compute the final results.

The term MapReduce comes from the two fundamental data-transformation operations used, map and reduce. A map operation converts the elements of a collection from one form to another. In this case, input key-value pairs are converted to zero-to-many output key-value pairs, where the input and output keys might be completely different and the input and output values might be completely different.

In MapReduce, all the key-pairs for a given key are sent to the same reduce operation. Specifically, the key and a collection of the values are passed to the reducer. The goal of “reduction” is to convert the collection to a value, such as summing or averaging a collection of numbers, or to another collection. A final key-value pair is emitted by the reducer. Again, the input versus output keys and values may be different. Note that if the job requires no reduction step, then it can be skipped.

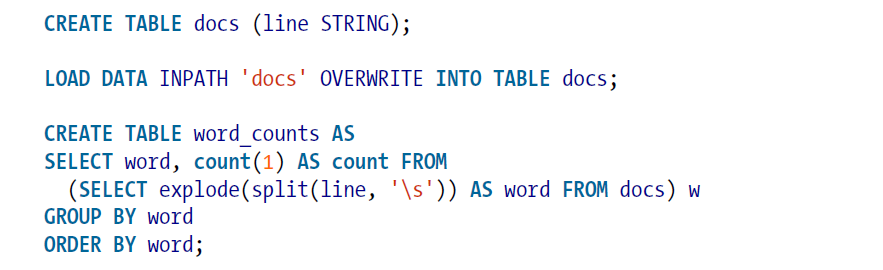
An implementation infrastructure like the one provided by Hadoop handles most of the chores required to make jobs run successfully. For example, Hadoop determines how to decompose the submitted job into individual map and reduce tasks to run, it schedules those tasks given the available resources, it decides where to send a particular task in the cluster (usually where the corresponding data is located, when possible, to minimize network overhead), it monitors each task to ensure successful completion, and it restarts tasks that fail.

The Hadoop Distributed Filesystem, HDFS, or a similar distributed filesystem, manages data across the cluster. Each block is replicated several times (three copies is the usual default), so that no single hard drive or server failure results in data loss. Also, because the goal is to optimize the processing of very large data sets, HDFS and similar filesystems use very large block sizes, typically 64 MB or multiples thereof. Such large blocks can be stored contiguously on hard drives so they can be written and read with minimal seeking of the drive heads, thereby maximizing write and read performance.

A unique feature of Hive, compared to most databases, is that it provides great flexibility in how data is encoded in files. Most databases take total control of the data, both how it is persisted to disk and its life cycle. By letting you control all these aspects, Hive makes it easier to manage and process data with a variety of tools.



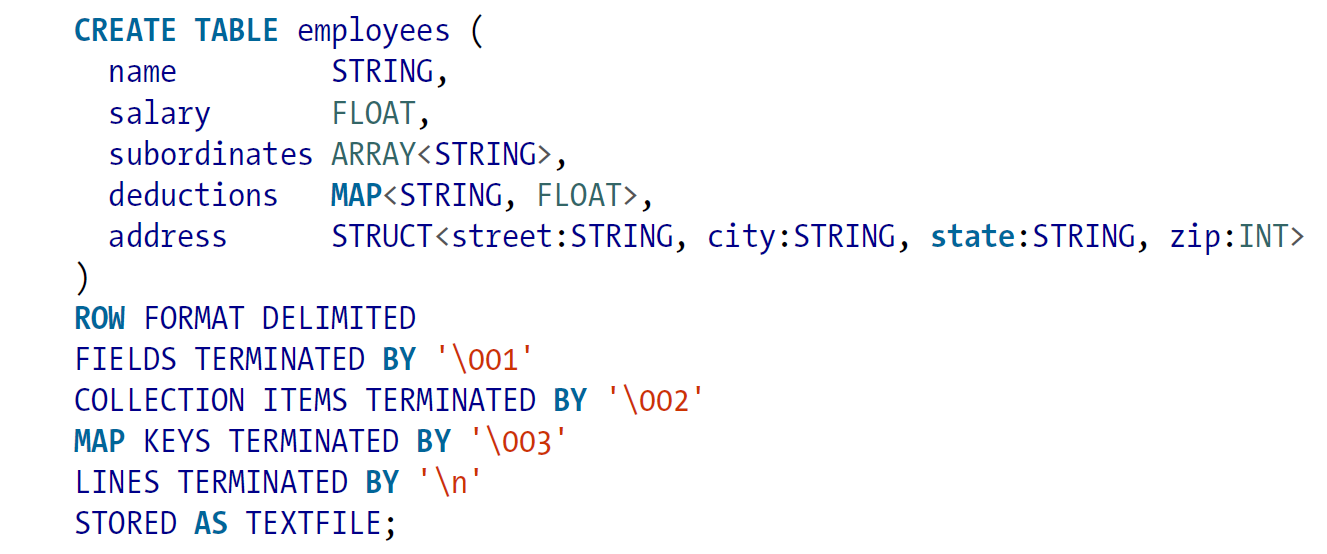
Same code written in HiveQL.



**Data Types**

Hive supports many of the primitive data types you find in relational databases, as well as three collection data types that are rarely found in relational databases

|  |  |  |
| --- | --- | --- |
| TINYINT | 1 byte |  |
| SMALLINT | 2 byte |  |
| INT | 4 byte |  |
| BIGINT | 8 byte |  |
| BOOLEAN | True or false |  |
| FLOAT | Single precision |  |
| DOUBLE | Double precision |  |
| STRING | Sequence of characters. |  |
| TIMESTAMP | Integer, float, or string. | 1327882394 (Unix epoch seconds)  1327882394.123456789 (Unix epoch seconds plus nanoseconds)  '2012-02-03 12:34:56.123456789' (java.sql.Timestamp) |
| BINARY | Array of bytes |  |
| STRUCT | Analogous to a C struct | struct('John', 'Doe') |
| MAP | A collection of key-value tuples | map('first', 'John', 'last', 'Doe')  'first'→'John' and 'last'→'Doe' |
| Array | Sequences of the same type | array('John', 'Doe') |



hive> CREATE DATABASE financials;

hive> CREATE DATABASE IF NOT EXISTS financials;

hive> DESCRIBE DATABASE financials;

hive> SHOW DATABASES;

hive> SHOW DATABASES LIKE 'h.\*';

hive> CREATE DATABASE financials LOCATION '/my/preferred/directory';

hive> CREATE DATABASE financials COMMENT 'Holds all financial tables';

hive> CREATE DATABASE financials WITH DBPROPERTIES ('creator' = 'Mark Moneybags', 'date' = '2012-01-02');

hive> set hive.cli.print.current.db=true;

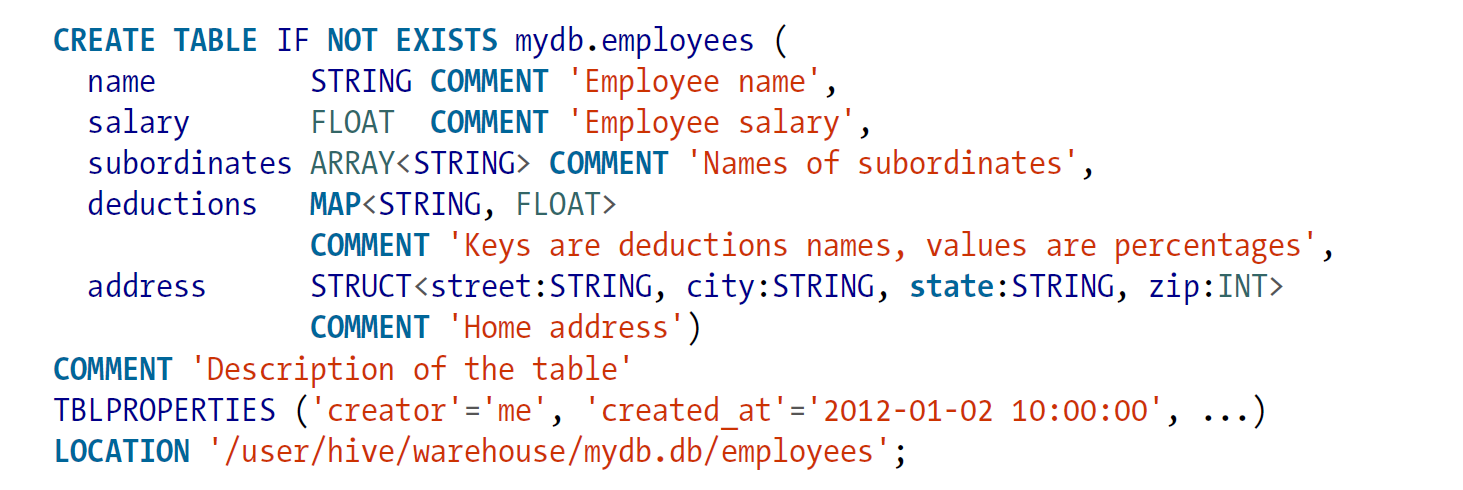
hive> DROP DATABASE IF EXISTS financials;

hive> DROP DATABASE IF EXISTS financials CASCADE;

hive> ALTER DATABASE financials SET DBPROPERTIES ('edited-by' = 'Joe Dba');

hive> CREATE TABLE IF NOT EXISTS mydb.employees2 LIKE mydb.employees;

hive> DESCRIBE EXTENDED mydb.employees;



The **EXTERNAL** keyword tells Hive this table is external and the LOCATION … clause is required to tell Hive where it’s located. Because it’s external, Hive does not assume it owns the data. Therefore, dropping the table does not delete the data, although the metadata for the table will be deleted.

The general notion of partitioning data is an old one. It can take many forms, but often it’s used for distributing load horizontally, moving data physically closer to its most frequent users, and other purposes. Hive has the notion of **partitioned tables**. Our HR people often run queries with WHERE clauses that restrict the results to a particular country or to a particular first-level subdivision (e.g., state in the United States).

Hive will create subdirectories reflecting the partitioning structure.

.../employees/country=CA/state=AB

.../employees/country=CA/state=BC

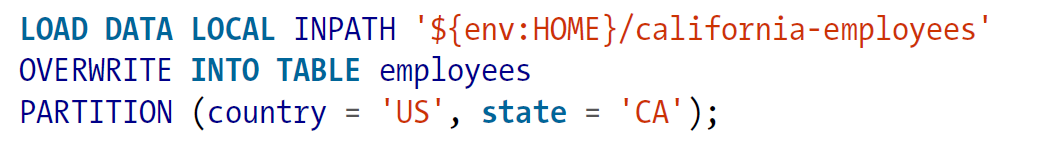
...

.../employees/country=US/state=AL

For very large data sets, partitioning can dramatically improve query performance, but only if the partitioning scheme reflects common range filtering. However, a query across all partitions could trigger an enormous MapReduce job if the table data and number of partitions are large.

We frequently use external partitioned tables because of the many benefits they provide, such as logical data management, performant queries, etc.

**Import:** Hive has no row-level insert, update, and delete operations, the only way to put data into any table is to use one of the “bulk” load operations. Or you can just write files in the correct directories by other means.



**Export:**

