# **Hadoop. The Definitive Guide. Summary**

[Chapter 1. Meet Hadoop 2](#_Toc525743545)

[Comparison with other systems 2](#_Toc525743546)

[Chapter 2. MapReduce 3](#_Toc525743547)

[Hadoop Streaming 5](#_Toc525743548)

[Chapter 3. The Hadoop Distributed Filesystem 5](#_Toc525743549)

[HDFS Concepts 5](#_Toc525743550)

[Hadoop Filesystems 7](#_Toc525743551)

[Data Flow 7](#_Toc525743552)

[Chapter 19. Spark 8](#_Toc525743553)

[Spark Applications, Jobs, Stages, and Tasks 8](#_Toc525743554)

[Resilient Distributed Datasets 9](#_Toc525743555)

[Executors and Cluster Managers 10](#_Toc525743556)

## Chapter 1. Meet Hadoop

Data! There is more and more data, that is good news. Bad news is that we are struggling to store and analyse it. The problem is simple, the capacity of hard drives does not increase as fast as the rate at which data volume has grown. Parallel disks is the solution. That brings other problems though: hardware failure (more disks 🡪 more chance of failure) and correctly combining data from multiple disks. The solution: Hadoop.

MapReduce is batch query processor and able to run an ad hoc query against a whole dataset and get the results in a reasonable time. For all its strengths, it is not suitable for interactive analysis. Hadoop however has evolved beyond batch processing. Hadoop is usually referred to a large ecosystem of projects, not just HDFS and MapReduce.

The first component to provide online access was HBase, a key-value store that uses HDFS for its underlying storage.

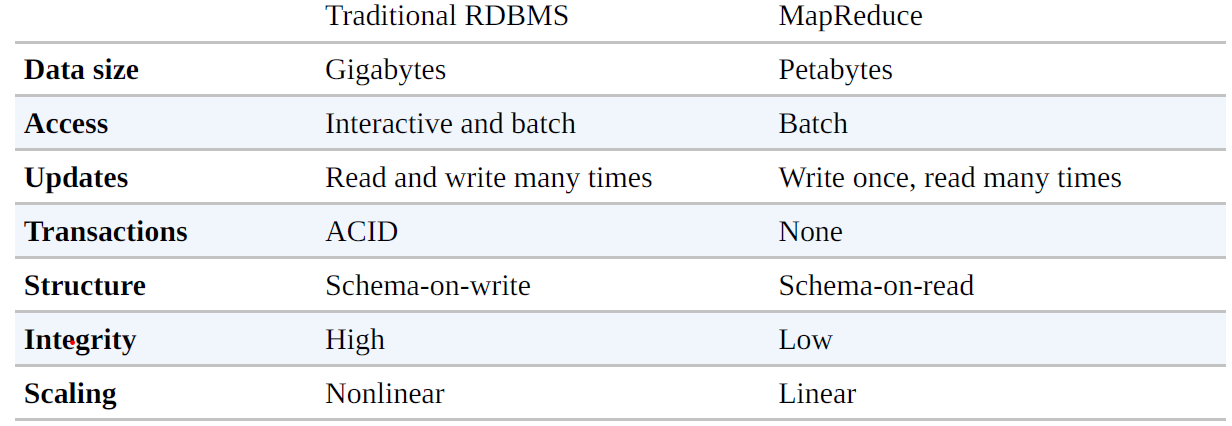
The real enabler for new processing models in Hadoop was the introduction of YARN. Which is a cluster resource management system, which allows any distributed program (not just MapReduce) to run on data in a Hadoop cluster. A couple of different processing patterns that now work with Hadoop:

* Interactive SQL
* Iterative processing
* Stream processing
* Search

### Comparison with other systems

**Relational Database Management Systems**

An RDBMS is good for point queries or updates, MapReduce is good for problem that need to analyse the whole dataset in a batch fashion.



The differences between the two are blurring. RDBMS are incorporating some ideas from Hadoop, whereas Hadoop incorporates systems like Hive to become more interactive.

Another difference is the level of structure in the datasets. Structured data for RDBMS, semi-structured and unstructured for Hadoop.

**Grid Computing**

High-performance computing works well for predominantly compute-intensive jobs, but becomes a problem when there are larger data volumes. Hadoop tries to co-locate the data with the compute node, so data access is fast because it is local. This feature, *data locality*, is at the heart of data processing in Hadoop and is the reason for its good performance.

High-performance computing with a Message Passing Interface (MPI) gives great control to programmers, but it requires that they explicitly handle the mechanics of the data flow, manage their own checkpointing and recovery.

**Volunteer Computing**

Volunteer computing projects work by breaking the problems they are trying to solve into chunks called *work units*, which are sent to computer around the world. This seems similar to MapReduce, but there are significant differences. It is very CPU-intensive, making it suitable for running on a large number of computers across the world. This also makes it a highly variable connection through the Internet in terms of speeds and no data locality. MapReduce is designed to run these jobs on trusted, dedicated hardware, with very high bandwidth and with data locality.

## Chapter 2. MapReduce

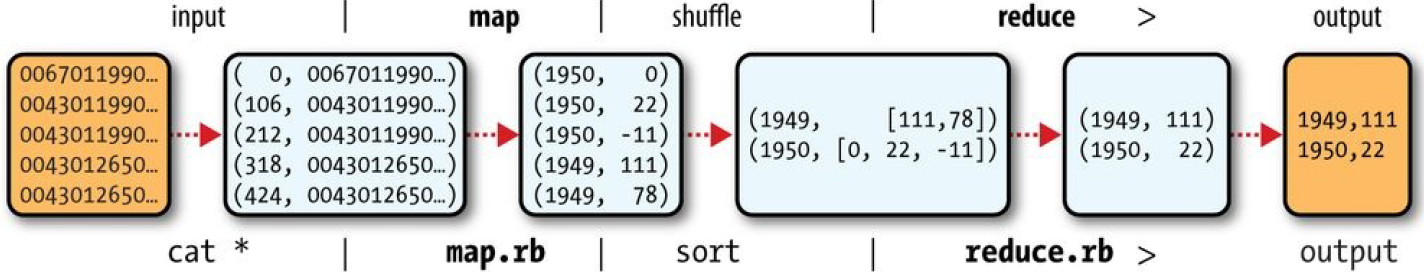
MapReduce is a programming model for data processing. The model is simple, yet not too simple to express useful programs in. MapReduce programs are inherently parallel, thus putting very large-scale data analysis into the hand of anyone with enough machines at their disposal.

Issues in dividing the work of a processing job which Hadoop solves:

1. Divide the work into equal-size pieces
2. Combining the results from independent processes
3. Limited by the process capacity of a single machine

MapReduce works by breaking the processing into two phases: the map phase and the reduce phase. Both have key-value pairs as input and output. The programmer has to define the key-value pairs and two functions: the map function and the reduce function.

A MapReduce dataflow would look like this:



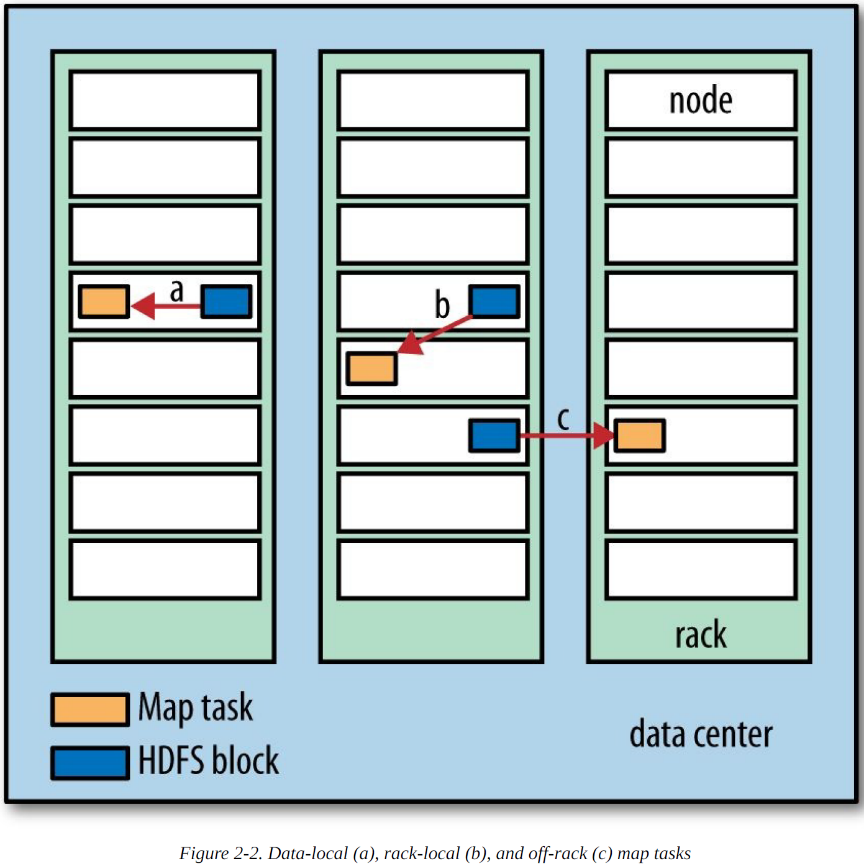
Hadoop provides its own set of basic Java types that are optimized for network serialization.

Terminology:

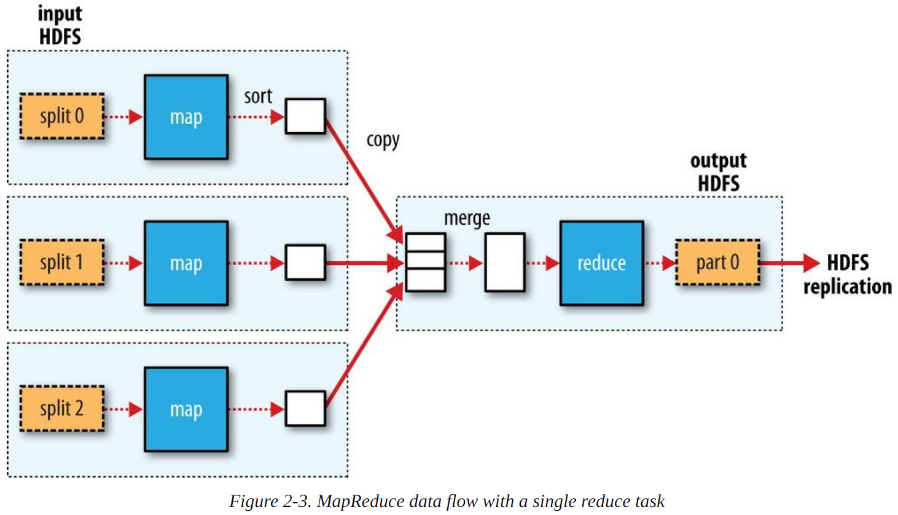
* A MapReduce *job* is a unit of work that the client wants to be performed.
  + Consists of input data, the MapReduce program, and configuration information.
* Hadoop runs the *job* by dividing it into *tasks.*
* There are two types of *tasks*, *map tasks* and *reduce tasks*.
* The *tasks* are scheduled using YARN and run on *nodes* in the *cluster*.

Hadoop divides the input file to a job into fixed-sized pieces called *splits*. Hadoop creates one map task for each split. This map task runs the user-defined map function for each *record* in the split.

Many small splits mean that when these are processed in parallel, you have a fast loading time. But if the splits are too small, the overhead of managing the splits and map task creation dominates the total job execution time.

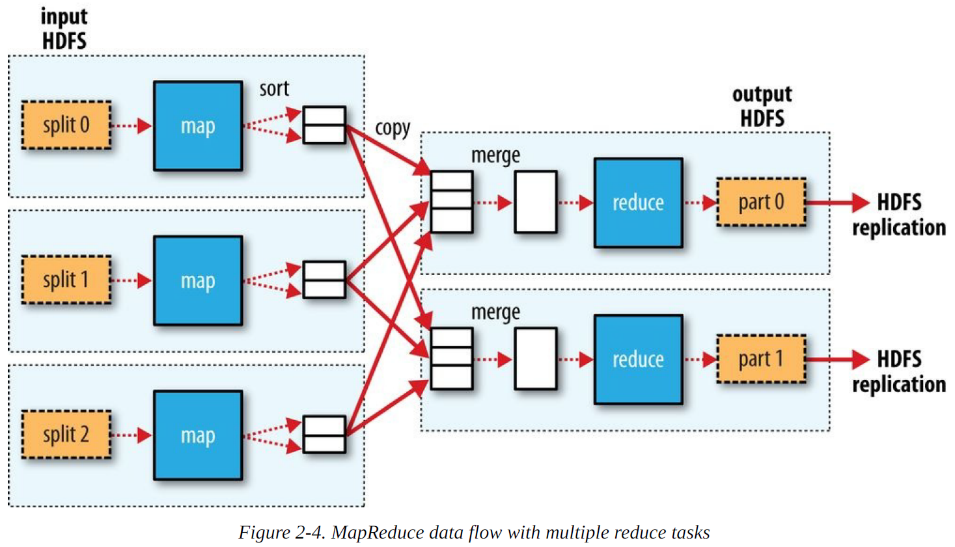
Hadoop tries to run the map task on a node where the input resides in HDFS, this way it doesn’t use valuable cluster bandwidth. This is called *data locality optimisation.* If all the nodes hosting the HDFS block replicas for a map task’s input are running other map tasks, the job scheduler will look for a free map slot on a node in the same rack as one of the blocks. If even this is not possible, an off-rack node is used, which is called an inter-rack network transfer.

Reduce tasks don’t have the advantage of data locality; the input to a single reduce task is normally the output from all mappers 🡪 transferred across the network to the node where the reduce task is. The output of the reduce is normally stored in HDFS for reliability. Thus, writing the reduce output does consume network bandwidth.



Number of reduce tasks is specified independently.

When there are multiple reducers, the map tasks *partition* their output. The number of partitions is the number of reduce tasks



The following picture shows the shuffling, as each reduce task is fed by many map tasks. Tuning the shuffle can have a big impact on job execution time.

It is also possible to have 0 reduce tasks.

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

Not all functions can be used as a combiner function. When you combine maximum values, the maximum value would still be the maximum value after the reduce task. When mean’s are calculated in the combiner function, the mean after the reduce task will be different.

### Hadoop Streaming

Hadoop provides an API to MapReduce that allows you to write your map and reduce functions in languages other than Java. *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. This is naturally suited for text processing.

## Chapter 3. The Hadoop Distributed Filesystem

Filesystems that manage the storage across a network of machines are called *distributed filesystems*. They are network based, thus all the complications of network programming kick in. Hadoop comes with a distributed filesystem called *Hadoop Distributed Filesystem* (HDFS).

HDFS is a filesystem designed for storing very large files with streaming data access patterns, running on clusters of commodity hardware.

* Very large files: petabytes.
* Streaming data access: write-once, read-many-times.
* Commodity hardware: no expensive, highly reliable hardware, just a lot of it. HDFS carries on working when one of the many commodity hardware fails.

HDFS does not work well for:

* Low-latency (delay) data access: not able to quickly access the data.
* Lots of small files: namenodes holds filesystem metadata in memory, the limit of number of files is limited by the amount of memory on the namenode. Millions of files is feasible, billions not yet.
* Multiple writers, arbitrary file modifications: no support for multiple writers or for modifications at arbitrary offsets in the file.

### HDFS Concepts

**Blocks**

Any disk has a block size, which is the minimum amount of data that it can read or write. HDFS also has blocks, but much larger than a regular disk, 128MB by default. But 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 storage capacity. 1MB on 128MB block in HDFS is 1MB disk space. 1MB on 128MB block in single disk is 128MB disk space. They are larger because that way the cost of seeks is minimized.

Having a block abstraction for a distributed filesystem brings several benefits:

1. A file can be larger than any single disk in the network.
2. Making the unit of abstraction a block rather than a file simplifies the storage subsystem. (Blocks are a fixed size, so easy to calculate how many can be stored on a given disk).
3. Blocks fit well with replication for providing fault tolerance and availability. Every block is typically three times replicated to insure against failures.

**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 manages the metadata for all the files and directories in the tree. Therefore, knowing the datanodes on which all the blocks for a given files are located. But does not choose the block locations.

Datanodes are the workhorses, they store and retrieve blocks when they are told to (by clients or the namenode). Without the namenode, the filesystem cannot be used. Since the namenode only knows the location of the blocks on the different datanodes. Hadoop provides two mechanisms to prevent the namenode to fail:

* Back up the files with the metadata, like the logs.
* Run a *secondary namenode*, which is an exact copy of the namenode. Does have lag because it does not always have the most recent logs, so there is always data loss.
* These can be combined

**Block Caching**

Normally a datanode reads blocks from disk. But frequently accessed files (and therefore its blocks) can be cached in the datanode’s memory, in an off-heap *block cache*. Way higher read performance for these cached blocks. Namenode gets instruction for which files to cache by a *cache directive* to a *cache pool*.

**HDFS Federation**

If there is a very large cluster with many files, the memory of the namenode becomes the limiting factor (has to keep track of all files). HDFS Federation adds namenodes, each of which manage a portion of the filesystem namespace. Under federation each namenode manages a *namespace volume*, existing of metadata, and a *block pool*, containing all the blocks for the files in the namespace.

**HDFS High Availability**

The combination of replicating namenode metadata and using the secondary namenode prevents data loss. But does not provide high availability of the filesystem. The namenode is still a single point of failure. When one fails, there is still a long recovery time. This situation is remedied with HDFS high availability. Here, there are namenodes in an active-standby configuration. A few architectural changes are needed to allows this to happen:

* Share the edit log
* Datanodes send block reports to both namenodes
* Clients must be configured to handle namenode failover

There are two choices for the highly available shared storage: an NFS files, or a *quorum journal manager* (QJM). QJM is a dedicated HDFS implementation, with a sole purpose of providing a highly available edit log.

**Failover and fencing**

The transition from the active namenode to the standby is manage by the *failover controller*. Default that is ZooKeeper, making sure there is only one namenode active. A *graceful failover* is one that is initiated manually, in case of routine maintenance for example. When an ungraceful failover happen, it can be the case that the previously active namenode is still running, and still think it is the active namenode. It is important that that namenode is not able to do any damage and cause corrupt files, which is called *fencing*. QJM handles this well, only one namenode is allowed to write to the edit log. For NFS files, there is a stronger fencing method needed.

### Hadoop Filesystems

|  |  |  |
| --- | --- | --- |
| **Filesystem** | **URI scheme** | **Description** |
| Local | file | Locally connected disk |
| HDFS | hdfs | Hadoop’s distributed filesystem |
| WebHDFS | webhdfs | Read/write access authentication over HTTP to HDFS |
| SecureWebHDFS | swebhdfs | Same as above, but with HTTPS |
| HAR | har | Filesystem layered on another filesystem for archiving files |
| View | viewfs | Client-side mount table for other Hadoop filesystems |
| FTP | ftp | Filesystem backed by an FTP-server |
| S3 | s3a | Filesystem backed by Amazon S3 |
| Azure | wasb | Filesystem backed by Microsoft Azure |
| Swift | swift | Filesystem backed by OpenStack Swift |

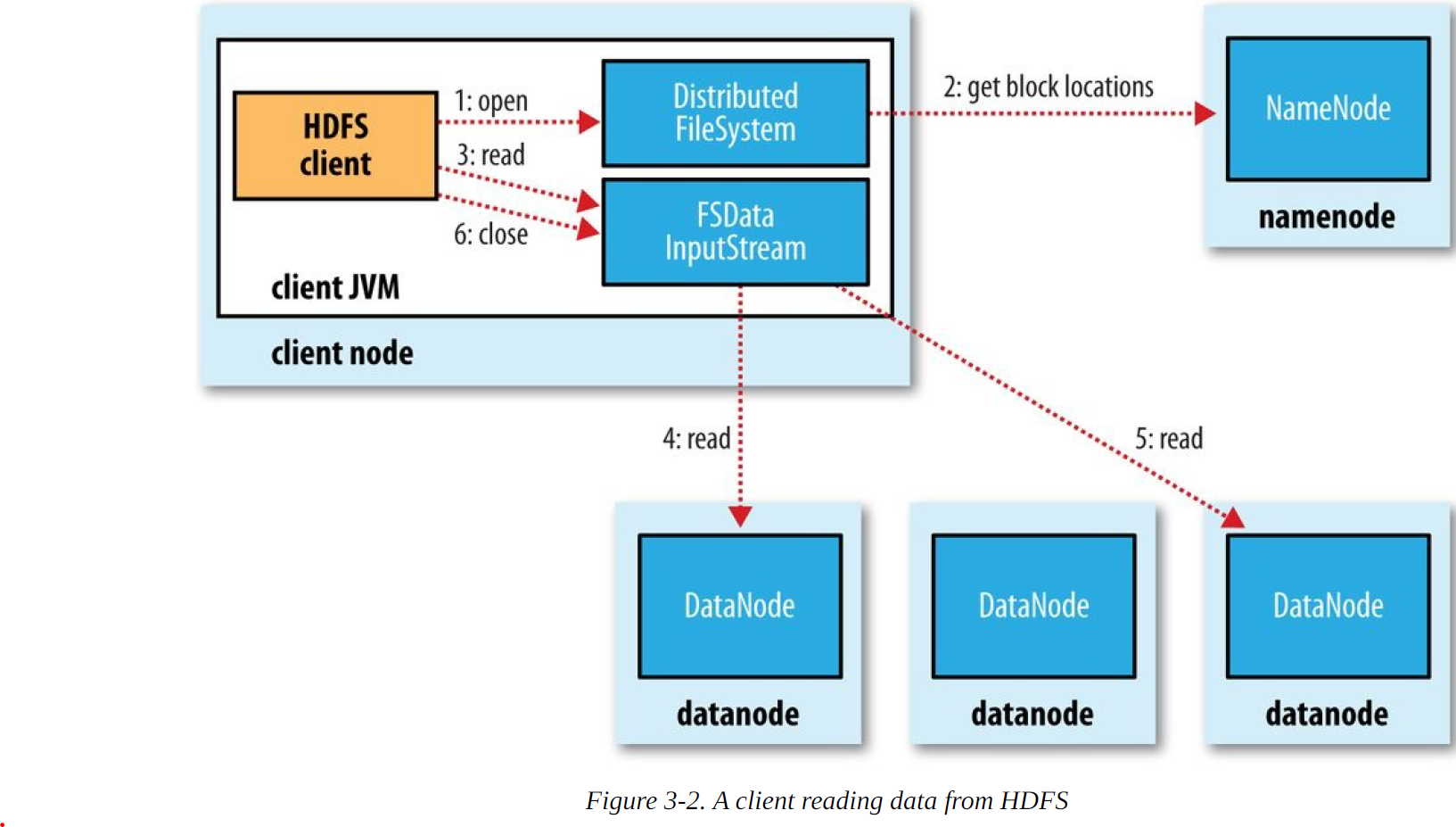
**Interfaces**

Hadoop is written in Java, so most Hadoop filesystem interactions are mediated through the Java API. Other interfaces are:

* HTTP
* C
* NFS
* FUSE

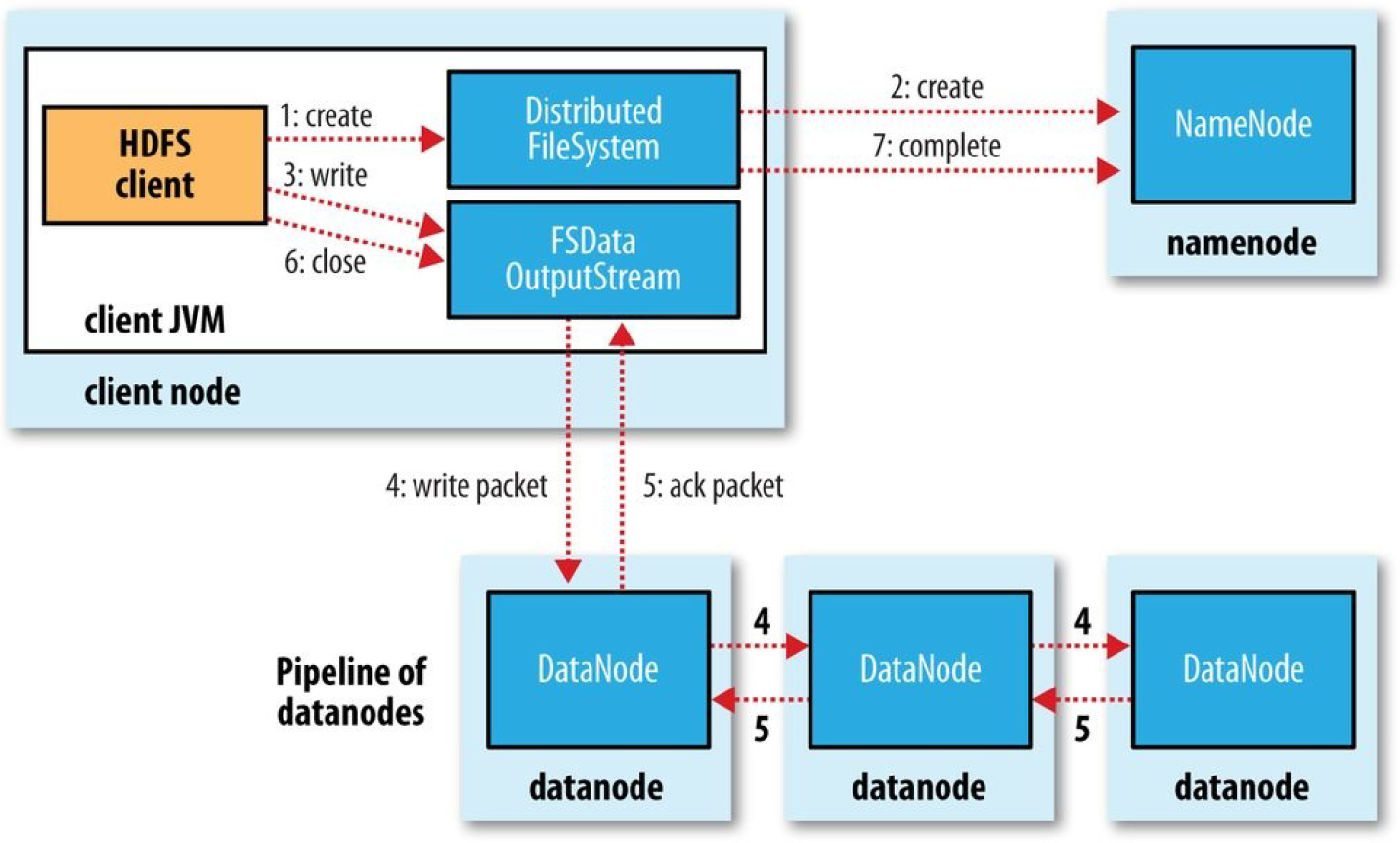
### Data Flow

**Anatomy of a File Read**



Bandwidth is dependent on: Processes on the same node, Different nodes on the same rack, Nodes on different racks in the same data center, Nodes in different data centers.

**Anatomy of File Write**



Here the replication level is three, therefore a pipeline of 3 datanodes.

A coherency model for a filesystem describes the data visibility of reads and writes for a file.

When copying data into HDFS, it’s important to consider cluster balance. HDFS works best when the file blocks are evenly spread across the cluster.

## Chapter 19. Spark

Apache Spark is a cluster computing framework for large-scale data processing. Does not use MapReduce as an execution engine but its own. Spark is closely integrated with Hadoop, it can run on YARN and works with Hadoop file formats like HDFS.

Spark is best known for its ability to keep large working datasets in memory *between jobs*. Iterative algorithms and interactive analysis benefit greatly from this. Even if you don’t need the in-memory caching, Spark is still better. Because of its DAG engine and its user experience. Spark’s DAG engine can process arbitrary pipelines of operators and translate them into a single job for the user. User experience wise: APIs in three languages: Scala, Java, and Python.

Spark proves to be a good platform for analytics tools, it includes the modules for machine learning, graph processing, stream processing, and SQL.

## Spark Applications, Jobs, Stages, and Tasks

Like MapReduce, Spark has the concept of a *job*. A Spark job is more general than a MapReduce job, it is made up of an arbitrary directed acyclic graph (DAG) of *stages*, each of which is roughly equivalent to a map or reduce phase in MapReduce.

Stages are split into *tasks* and are run in parallel on partitions of an RDD spread across the cluster, same as with tasks in MapReduce.

A job always runs in the context of an *application* that serves to group RDDs and shared variables. One application can rune more than one job.

## Resilient Distributed Datasets

RDDs are at the heart of every Spark program. Resilient because Spark can automatically reconstruct a lost partition by recomputing it from the RDDs that is was computed from.

There are three ways to create an RDD: 1) From an in-memory collection of objects, 2) Using a dataset from external storage (like HDFS), and 3) Transforming an existing RDD.

Spark provides two categories of operations on RDD: *transformations* and *actions*. A transformation generates a new RDD from an existing one, while an action triggers a computation on an RDD and does something with the results, returning them to the user, or saving them to external storage. Actions have an immediate effect, transformations are lazy (they don’t perform any work until an action is performed on the transformed RDD). One way of telling if an operation is a transformation or an action is by looking at its return type: if the return type is RDD, then it’s a transformation, otherwise an action.

**Aggregation transformations**

The three main transformations for aggregating RDDs of pairs by their keys are reduceByKey(), foldByKey(), and aggregateByKey(). The simplest is reduceByKey().

**Serialization**

There are two aspects of serialization to consider in Spark: serialization of data and serialization of functions.

**Shared Variables**

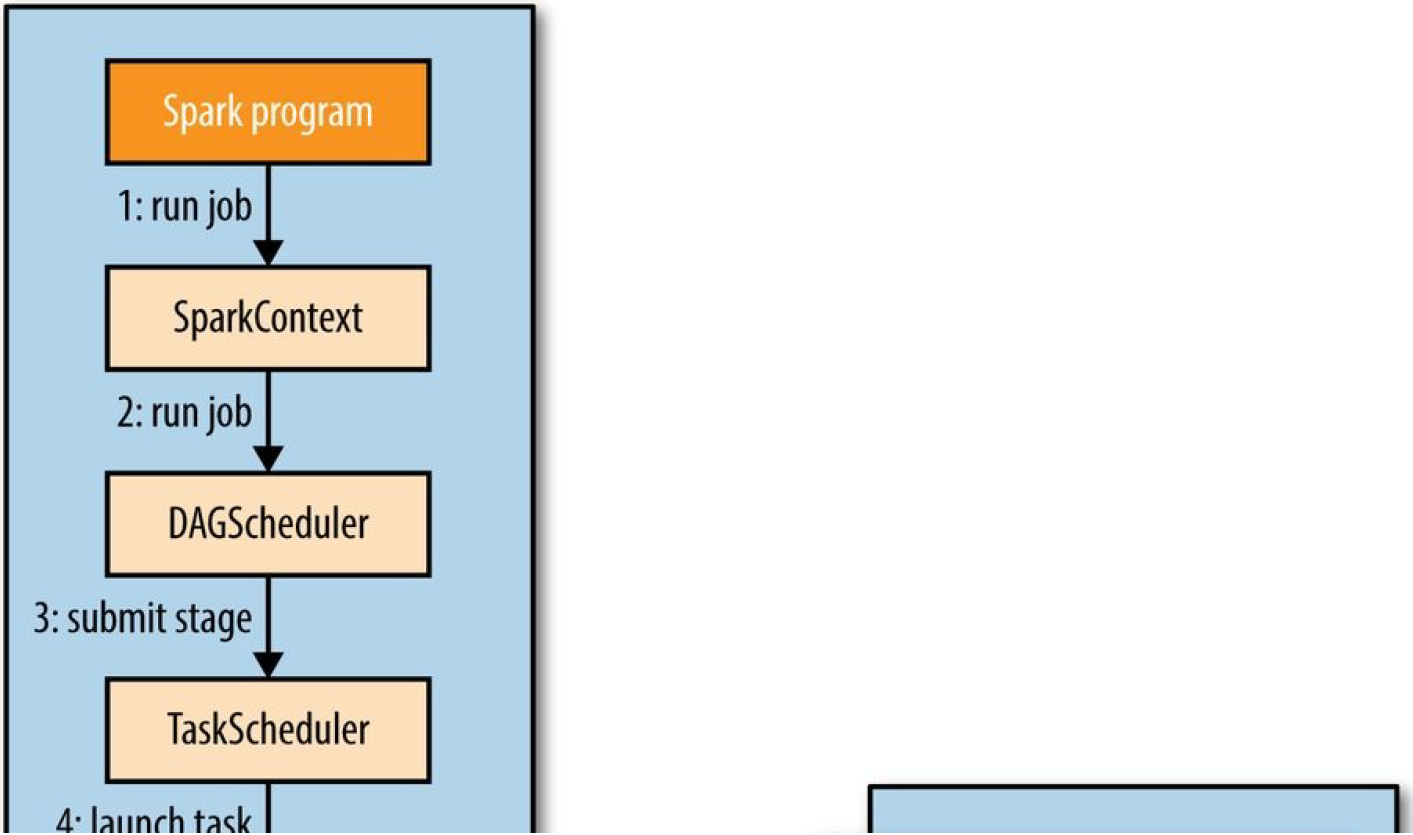
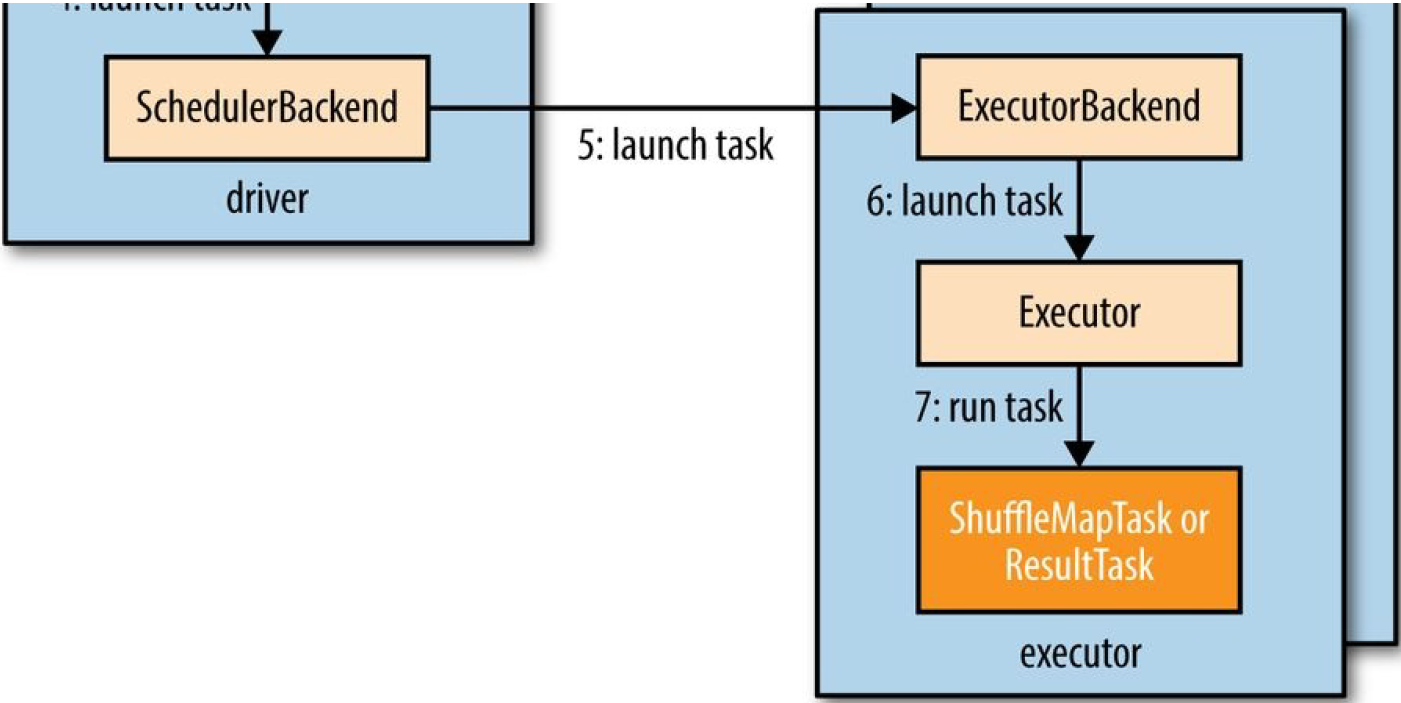
Broadcast variable is serialized and sent to each executor, where it is cached so that later tasks can access it if needed.

An accumulator is a shared variable that tasks can only add to, like counters in MapReduce.

**Anatomy of a Spark Job Run**

At the highest level, there are two independent entities: the *driver*, which hosts the application (SparkContext) and schedules tasks for a job; and the *executors*, which are exclusive to the application, run for the duration of the application, and execute the application’s tasks.

1. Action performed on an RDD, cause runJob() to be called on SparkContext,
2. Pass the call on onto the scheduler, scheduler is made up of two parts, a DAG scheduler that breaks down the job into a DAG of stages, and a task scheduler that is responsible for submitting the tasks from each stage to the cluster
   1. A job can be broken up into stages, there are two types: *shuffle map tasks* and *result tasks*
3. Once the DAG scheduler has constructed the complete DAG of stages, it submits each stage’s set of tasks to the task scheduler
4. The task scheduler uses its list of executors that are running for the application and constructs a mapping of tasks to executors. Next, the task scheduler assigns tasks to executors that have free cores. Assigned tasks are launched through a scheduler backend.
5. The scheduler backend sends a remote launch task message to the executor backend.
6. The executor backend tells the executor to run the task
7. The executor runs the task



### Executors and Cluster Managers

Spark relies on executors to run the tasks that make up a Spark job. Managing the lifecycle of executors is the responsibility of the *cluster manager*. Variety of cluster managers:

* Local, single executor running in the same JVM as the driver.
* Standalone, simple distributed implementation that runs a single Spark master and one or more workers.
* Mesos, general-purpose cluster resource manager that allows fine-grained sharing of resources
* YARN, resource manager used in Hadoop.

Mesos and YARN are superior to the standalone manager since they take into account the resource needs of other applications running on the cluster and enforce a scheduling policy across all of them.

**Spark on YARN**

Spark on YARN provides the tightest integration with other Hadoop components and is the most convenient way to use Spark when you have an existing Hadoop cluster. Spark offers two deploy modes:

1. YARN client mode: the driver runs in the client
   1. Required for programs that have any interactive component. Also useful when building Spark programs, since any debugging output is immediately visible.
2. YARN cluster mode: the driver runs on the cluster in the YARN application master
   1. Appropriate for production jobs, since the entire application runs on the cluster, which makes it easier to retain logfiles.